Guide to the VAXlab Laboratory I/O Routines

Order Number: AA-KN99C-TE

February 1990

This document describes the VAXlab Laboratory I/O routines. It provides an overview of laboratory I/O concepts and presents detailed reference information about the laboratory I/O routines you use to initiate, control, and terminate I/O to and from VAXlab I/O devices.

Revision/Update Information:	This is a revised document.
Operating System and Version:	VMS Version 5.2
Software Version:	VAXlab Software Library Version 1.4

digital equipment corporation
maynard, massachusetts

First Printing, December 1987
Revised, August 1988
Revised, February 1990

The 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	MicroVAX	VAXstation
DECnet	Q-bus	VMS
DRB32	VAX	VT
LN03	VAXcluster	
LN03 PLUS	VAXlab	

digital™

This document was prepared with VAX DOCUMENT, Version 1.2.

Contents

FIGURES

TABLES

Preface

Intended Audience

The *Guide to the VAXlab Laboratory I/O Routines* is intended for use by scientists and engineers working in a laboratory environment performing real-time data acquisition experiments.

You can use this document initially as a training guide for learning the basic components of the Laboratory I/O (LIO) application routines. Later, you can use it as a reference guide to look up specific information about the LIO application routines, such as how to use an optional parameter.

This guide assumes a basic understanding of computer concepts and an extensive knowledge of laboratory data acquisition and experiment control concepts.

Document Structure

The *Guide to the VAXlab Laboratory I/O Routines* provides a comprehensive overview of the LIO facility, and explains how to do the following with VAXlab devices:

- Initiate I/O
- Control I/O
- Terminate I/O

The document is structured in the following way:

Chapter Number	Contents
Chapter 1	Presents an overview of the laboratory I/O concepts and device capabilities you should be familiar with before you begin writing programs using the VAXlab laboratory I/O routines.
Chapter 2	Describes the I/O devices supported by VAXlab and gives instructions for setting up each device using the LIO software.
Chapter 3	Provides reference descriptions of the LIO routines, including the routine call syntax, argument descriptions, and device-specific considerations.
Chapter 4	Provides reference descriptions of the parameters you use to set up VAXlab laboratory I/O devices and to display device-specific characteristics.
Chapter 5	Describes the error handling method supported by the LIO facility and explains the error messages and suggested recovery procedures.
Chapter 6	Describes the online sample programs shipped with your VAXlab system.
Appendix A	Explains the clock triggering modes of the ADQ32.
Appendix B	Explains how to connect the connect-to-interrupt driver to the AXV11-C device. This appendix also includes instructions for reloading and reconnecting the QIO driver to the AXV11-C device.

Associated Documents

In addition to this guide, the VAXlab documentation set includes the following guides:

- *Getting Started with VAXlab* is your introduction to the VAXlab system and application software. This document describes the optional hardware you can configure in a VAXlab system, the VAXlab software, and the related software you need to use with your VAXlab system, such as DEC GKS and a high-level programming language. This document also presents guidelines for developing application programs with VAXlab and considerations specific to programming languages, such as declaring variables and data types.
- The *Guide to the VAXlab Interactive Data Acquisition Tool* describes how to communicate with VAXlab through the Interactive Data Acquisition Tool (IDAT) to establish parameters for data acquisition and to initiate, control, obtain, analyze, and plot real-time data.
- The *Guide to the VAXlab Laboratory Graphics Package* describes how to specify plotting attributes and how to plot real-time data or data produced by calculations in two dimensions, three dimensions, and two-dimensional contours from a three-dimensional view.
- The *Guide to the VAXlab Signal-Processing Routines* describes how to use the signal-processing routines to perform Fourier transforms, correlation functions, and filtering of data.
- The *Installation Guide* details how to install the VAXlab software.
- The *Master Index* contains index entries from all the documents in the VAXlab documentation set.

The following is a list of associated software documents to reference for additional information about programming concepts and techniques not covered in this guide:

- The *DEC GKS Reference Manual, Volumes I and II* provide detailed information about advanced graphics programming concepts and techniques.
- The *VAX Realtime User's Guide* describes those features of VAX systems that pertain to real-time applications in scientific and industrial settings. If you are unfamiliar with VAX systems, read this guide before you begin using the VAXlab system.

The following is a list of associated hardware documents to reference for additional information:

- *AAF01 User's Manual* [1]
- *ADF01 User's Manual*[1]
- *ADQ32 A/D Converter Module User's Guide*
- *ADV11-D/AAV11-D Analog Modules User's Guide*
- *AMF01 User's Manual*[1]
- *ASF01 User's Manual*[1]
- *AXV11-C/KWV11-C Analog Module and Real-Time Clock Module User's Guide*
- *DRB32 Technical Manual*
- *DRQ11-C Alternate Buffer DMA Interface*[1]
- *DRQ3B Parallel DMA I/O Module User's Guide*
- *DRV11-J Parallel Line Interface User's Guide*
- *DRV11-WA General Purpose DMA Interface User's Guide*
- *IEU11-A/IEQ11-A User's Guide*
- *IEZ11 Hardware Installation Guide*
- *IEZ11 Software Installation Guide*
- *IEZ11 VMS Class Driver User's Guide*
- *Industrial I/O Modules for Q-Bus* (IAV11-A, IAV11-AA, IAV11-B, IAV11-C, IAV11-CA, IDV11-A, IDV11-B, IDV11-C, IDV11-D) [2]
- *Universal Data Interface Panel Reference Card*

[1] This device is available only in Europe.

[2] These devices are available only in Europe.

Conventions

The *Guide to the VAXlab Laboratory I/O Routines* uses the following documentation conventions:

Convention	Meaning
Italics	Phrases appearing in italics reference an associated document.
Bold	A boldface word or phrase indicates one of the following: • Emphasis on an important concept or word • Discussion of a routine argument in text • Referencing of a subsection within a routine or parameter reference description
Ellipses	Vertical ellipses indicate that portions of a display or programming example are excluded for presentation purposes.
[Brackets]	Square brackets generally enclose optional parameters or arguments in routine lines. However, square brackets used in the context of a Pascal program or program segment are required programming syntax.

Chapter 1

Laboratory I/O Interfaces and Operations

This chapter provides an overview of LIO, describes the LIO interfaces and operations supported by VAXlab, and describes special features specific to some of the VAXlab I/O devices.

1.1 Overview of LIO

The VAXlab Laboratory I/O routines are used to control real-time I/O devices. You use these routines as program modules, linking them with your main program.

Although you can use any programming language to call these routines, information in this guide and sample programs are written for five languages:

- Ada
- BASIC
- C
- FORTRAN
- Pascal

I/O can be done synchronously or asynchronously with the LIO routines. By default, all devices use asynchronous I/O.

You can set up a device to use synchronous I/O. However, when you want that device to operate asynchronously again, you must set up the device for asynchronous I/O.

There are only eight LIO routines. They can be grouped into four pairs:

- ATTACH/DETACH logically connect or disconnect a device. They also build or delete all the internal queues and data structures required. Your program calls the ATTACH routine before it makes any other routine call to a device.
- SET/SHOW set or return the status of various device characteristics such as the following:
  - Channel lists
  - Triggering mode
  - Clock rate
  - Gains
  - Synchronous or asynchronous transfers

  Any device characteristic that can be set can also be shown. In addition, some devices have characteristics that cannot be set but can be shown.
- READ/WRITE transfer data to or from a device. These routines are used only for synchronous I/O. With synchronous I/O, the program waits for the I/O to complete before continuing execution.
- ENQUEUE/DEQUEUE put data buffers on a queue and remove them from the queue when data transfer is complete. They permit the continuous transfer of high speed data by simultaneously transferring data among multiple devices. These routines are used only for asynchronous I/O.

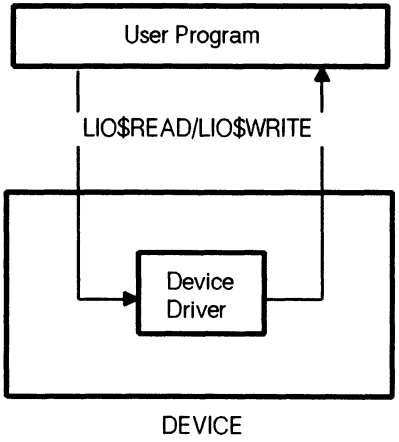
1.2 Synchronous I/O

Synchronous I/O enables a user program to transfer data to or from a device with one routine call. The routine call blocks the program until the I/O operation completes. The device does not continue to transfer data while the program is preparing for the next I/O operation.

You use the LIO$READ and LIO$WRITE routines, described in Chapter 3, to perform synchronous I/O.

Figure 1–1 shows the synchronous I/O process.

Figure 1–1: Synchronous I/O Device Model



The synchronous I/O interface is recommended for applications that examine or modify the data between I/O operations, such as single-point or slow data acquisition and closed loop control. High-speed applications requiring only one buffer of data to complete I/O operations can also benefit from using synchronous I/O routine calls.

1.3 Asynchronous I/O

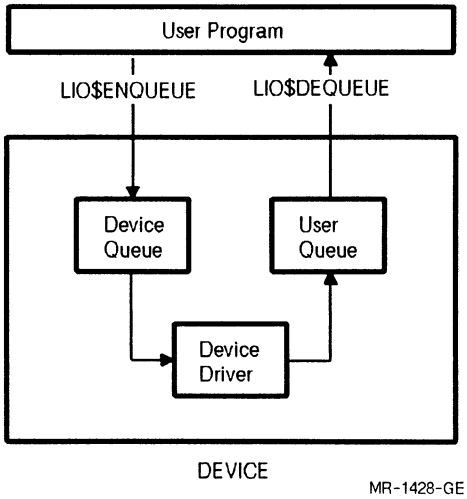
Asynchronous I/O enables a user program to queue several values or arrays of data to be transferred. A program is not blocked for I/O and continues execution during I/O operations. This enables I/O operations to continue on one or more devices simultaneously.

Each device that is set up to use the asynchronous I/O interface has a **device queue** and a **user queue**. An asynchronous I/O routine call places a buffer in the device queue to send it to the device. The device processes the buffer and places the buffer in the user queue to return it to the program.

You use the LIO$ENQUEUE routine and either the LIO$DEQUEUE routine or one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms, to perform asynchronous I/O. LIO$ENQUEUE and LIO$DEQUEUE are described in Chapter 3.

Figure 1-2 illustrates the asynchronous I/O process.

Figure 1-2: Asynchronous I/O Device Model



The asynchronous I/O interface is recommended for high-speed communications, data acquisition, and open loop control where I/O operations are continuous.

1.4 I/O Operations Supported by VAXlab

VAXlab devices can handle I/O operations in three ways:

- Queued I/O (QIO) to a VMS device driver
- Polled I/O
- I/O through a VMS connect-to-interrupt (CTI) handler

Not all devices, however, can handle all three methods.

Table 1–1 lists the devices and summarizes the I/O interfaces and I/O operations supported for each device.

The three methods are explained in the following pages.

Table 1–1: I/O Interfaces and Operations Summary

I/O Device	Synchronous Operations (Read/Write)	Asynchronous Operations (Enqueue/Dequeue)
AAF01[1,2]	QIO	QIO
AAV11-D[3]	Polled, QIO	QIO
ADF01[1,2]	QIO	QIO
ADQ32[2]	QIO	QIO
ADV11-D[3]	Polled, QIO	QIO
AXV11-C	QIO, CTI	QIO
DRB32	QIO	QIO
DRB32W	QIO	QIO
DRQ11-C[1,2]	QIO	QIO
DRQ3B[2]	QIO	QIO
DRV11-J	Polled, QIO	QIO
DRV11-WA[2]	QIO	QIO
IAV11-A[1]	QIO	QIO
IAV11-AA[1]	QIO	QIO
IAV11-B[1]	QIO	QIO
IAV11-C[1]	QIO	QIO
IAV11-CA[1]	QIO	QIO
IDV11-A[1]	QIO	QIO
IDV11-B[1]	QIO	QIO
IDV11-C[1]	QIO	QIO
IDV11-D[1]	QIO	QIO
IEEE-488	QIO	QIO
KWV11-C	Polled, QIO	QIO
Preston[2]	QIO	QIO
Simpact RTC01[4]	Polled, QIO	QIO
Disk files	QIO	QIO

[1]This device is available only in Europe.

[2]This device uses QIOs to implement direct memory access.

[3]This device is capable of direct memory access only when set to use QIOs.

[4]The VMS SYSGEN utility refers to the Simpact RTC01 as a KWB32.

Table 1–1 (Cont.): I/O Interfaces and Operations Summary

I/O Device	Synchronous Operations (Read/Write)	Asynchronous Operations (Enqueue/Dequeue)
Memory queue	Read-only[5]	Transfer Display-only
Real-time plotting	Write-only	N/A
Serial line	QIO	QIO

[5]The synchronous I/O interface is only supported for interprocess read-only when a memory queue device is set up to copy data buffers displayed by a memory queue device in another process.

1.4.1 QIOs to a VMS Device Driver

Most laboratory I/O devices can be attached to perform QIOs to a VMS device driver. (QIO is a VMS term for queued input/output using the SYS$QIO system service routine.)

QIOs are best used to perform continuous I/O using asynchronous I/O routine calls and multiple buffers, or to perform I/O through direct memory access (DMA) driven devices.

Keep in mind the following restrictions when you attach a device to use QIOs:

- Certain I/O devices can use DMA to transfer data. Because of the system overhead associated with each QIO call, QIO is best used when moving large amounts of data in large buffers with few input or output calls, depending on the device type (A/D or D/A) or the device direction (input or output).

  If you set up a device to perform continuous DMA, other restrictions specific to the device may apply. See Section 1.6.3.2, Continuous DMA, for more information.
- The QIO driver must be connected to the device. This is relevant only to AXV11-C users if you have previously used the AXV11-C for CTI I/O and connected the CTI driver to the device. See Section 1.4.3, Connect-to-Interrupt I/O, and Appendix B for more information.

1.4.2 Polled I/O

Some laboratory I/O devices can be attached to perform polled, or memory-mapped, I/O.

During polled I/O, a synchronous routine call maps directly to the I/O page of the system and reads from or writes to the control and status register of the device. This provides the least software overhead between the user program and the I/O device.

Polled I/O is often used for control loops where the CPU is dedicated to obtaining a sample, processing it, producing a result, obtaining another sample, and so on. During the actual I/O (read/write) operations, the CPU can only maintain the control loop; it cannot perform any other processing tasks.

To prevent another program from preempting the I/O polling loop, set up your programs to run at real-time priority (16 or higher). (You must have the ALTPRI privilege to increase process priorities.) Doing so, however, has a severe impact on multiuser systems because it locks out other programs during the I/O call.

Hardware interrupts have a higher priority than the memory mapped I/O polling loop. In this case, nothing else should be happening on the system—no file I/O, terminal I/O, or network I/O—to ensure the fastest response time.

The following restrictions apply when you attach a device to use memory-mapped (polled) I/O:

- Only the synchronous I/O interface is supported.
- The software cannot use a device's direct memory access (DMA) feature, so the maximum transfer rate is limited.

1.4.3 Connect-to-Interrupt I/O

A laboratory I/O device can be attached to perform connect-to-interrupt, or interrupt-driven, I/O. During interrupt-driven I/O, a user program and the LIO interrupt service routine communicate with a minimum of operating system overhead to provide fast interrupt servicing.

NOTE

The AXV11-C is the only hardware device that supports connect-to-interrupt I/O.

The following restrictions apply when you attach the AXV11-C to use CTI I/O:

- The connect-to-interrupt driver, CONINTERR, must be connected to the AXV11-C. See Appendix B for information about connecting the driver.
- Only the synchronous I/O interface is supported.
- The software cannot use a device's direct memory access (DMA) feature.
- The data buffer must be allocated when the device is set up, and must be large enough to contain the connect-to-interrupt overhead (approximately 250 bytes) in addition to the data.
- The data buffer size is limited to 65,536 bytes minus the connect-to-interrupt overhead (approximately 250 bytes).

1.5 Asynchronous I/O Buffer-Handling Mechanisms

The LIO facility supports three mechanisms for user programs to retrieve completed asynchronous I/O buffers from a device:

- Using the LIO$DEQUEUE routine to return completed buffers to the main program. See Section 1.5.1, Buffer Dequeueing.
- Forwarding completed buffers to another device in a forwarding loop. When a buffer transaction completes, the device forwards, or passes, the buffer to another device. See Section 1.5.2, Buffer Forwarding.
- Supplying an asynchronous system trap (AST) routine to receive completed buffers. When a buffer transaction completes, the device calls the AST routine and passes the buffer to it. The AST routine receives the buffer and performs whatever tasks it is written to perform, such as processing the data contained in the buffer and requeueing the buffer to the device for another transaction. An AST routine is used instead of the LIO$DEQUEUE routine. See Section 1.5.3, Asynchronous System Traps (ASTs).

1.5.1 Buffer Dequeueing

A user program can determine when an asynchronous I/O buffer transaction is complete and a buffer is ready to be dequeued by doing one of the following:

- **Polling the device.** The main program makes successive calls to the LIO$DEQUEUE routine until the buffer transaction is complete. The LIO$DEQUEUE routine call returns the LIO$_EMPTYQ error until the buffer transaction completes. The LIO$DEQUEUE routine returns a success status when the buffer transaction completes. The buffer is available for return to the main program.
- **Waiting for the buffer.** To wait for a buffer transaction to complete, the main program must do both of the following:
  1. Set up the device or supply the buffer with an event flag. Supply the buffer with an event flag using the *event_flag* argument of the LIO$ENQUEUE routine when you enqueue the buffer to the device.
  2. Specify a nonzero **wait** argument in the LIO$DEQUEUE routine. When the main program makes a call to the LIO$DEQUEUE routine, the routine waits for a buffer to become available on the device's user queue.

  The nature of your application program determines whether polling a device or waiting for a completed buffer is more appropriate.

  The online sample program LIO_BUF_INX.FOR is a VSL application program that uses the asynchronous I/O interface and single-buffer DMA with buffer indexing to read analog-to-digital values from an A/D device. In this program, the LIO$DEQUEUE routine call includes a nonzero **wait** argument. The LIO$ENQUEUE routine calls supply each buffer with a unique event flag so that the LIO$DEQUEUE routine can wait for the buffers.

1.5.2 Buffer Forwarding

When devices are set up to use the asynchronous I/O interface, a main program can set up these devices to forward completed buffers to another device. When the first device in the forwarding loop completes a buffer, it enqueues the buffer to the second device in the forwarding loop, and so on. Typically, buffer forwarding is used by application programs that move data from device to device. Examples are moving data from an A/D converter to a disk file, or from a disk file to a D/A converter.

You can use buffer forwarding to link together any number of devices. All devices in the loop must be set up to use the asynchronous I/O interface. If the last device in the loop forwards buffers to the first device in the loop, the data flow runs continuously until the forwarding completes. A forwarding loop completes when one of the following conditions occurs:

- A device or file reaches a stop condition, such as a full output file.
- An error, such as a data overrun on an A/D converter, occurs on a device.

When a device is set up for buffer forwarding, it must also be set up with a device event flag (LIO$K_DEVICE_EF). The device sets the device event flag when it completes a buffer.

You can also use the AST routine LIO$K_AST_RTN to receive notice of error conditions.

When a device reaches a stop condition, it refuses to accept any more forwarded buffers. If another device in the loop attempts to forward (enqueue) additional buffers to the device that completed the forwarding loop, the LIO$ENQUEUE routine returns an error. The device attempting to forward the buffers responds to the error by making the buffers available on its user queue. You can use the LIO$DEQUEUE routine to receive the buffers.

If you set up all devices in the forwarding loop with AST routines, the buffers are passed to the AST routine instead of being placed on the device's user queue.

The online sample program LIO_BUF_FWD.FOR is a VSL application routine that uses the asynchronous I/O interface and single-buffer DMA with buffer forwarding to read analog-to-digital values from the ADV11-D device and forward the buffers to a disk file. The A/D device used in this routine is set up with a device event flag (LIO$K_DEVICE_EF) that the A/D device sets when there is a buffer available on its user queue.

1.5.3 Asynchronous System Traps (ASTs)

An asynchronous system trap (AST) is a VAX/VMS mechanism for providing a software interrupt when an external event occurs.

When the external event occurs, the VMS operating system interrupts execution of the current process and calls a procedure that you supply. This procedure is called an AST handler or an AST routine.

The interrupt mechanism is called an **asynchronous** system trap because the interrupt occurs out of sequence with respect to process execution.

The AST interrupt transfers control to the AST routine that services the event. This AST routine can call other procedures, including library procedures. When the AST routine finishes servicing the event, control returns to the calling program.

Within the context of VSL application programs, an AST routine is a normal subroutine that you supply to a device as the value of the LIO$K_AST_RTN parameter when you set up the device in your main program. All devices to which you supply an AST routine must be set up to use the asynchronous I/O interface.

Typically, a main program sets up an AST routine to receive completed buffers from a device for processing. When a device finishes a buffer, it calls the AST routine and passes the buffer to it. Instead of waiting for the LIO$DEQUEUE routine call to return the buffer to the main program, the AST routine processes the buffer.

An AST routine is useful when data needs to be processed as soon as it is available and the main program cannot sit idle waiting for it. For example:

- **When more than one device must be kept running continuously.** The main program cannot call the LIO$DEQUEUE routine to wait for the buffer on any device because one of the devices might reach a stop condition before the others. Using AST routines to receive buffers makes a call to the LIO$DEQUEUE routine unnecessary. All completed buffers are passed to the AST routine. The AST routine must use the LIO$ENQUEUE routine call to enqueue the buffers to the device again.
- **When one step in a buffer-forwarding loop requires that the buffers be processed.** Instead of forwarding the buffer from one device to the next device in the loop, you supply the first device with an AST routine. The AST routine receives the completed buffer from the first device, processes the buffer, and enqueues the buffer to the next device in the forwarding loop.
- **When a buffer needs to be checked immediately for error conditions and subsequently requires time-consuming processing.** The AST routine receives the completed buffer from the device, checks the buffer for error conditions, and enqueues the buffer to the memory queue device. The main program can then dequeue the buffer from the memory queue device and perform the time-consuming processing of the buffer.

The following sample FORTRAN program segment shows the arguments you need to define when you set up an AST routine. These argument declarations are used by an AST routine that receives completed buffers from the ADV11-D device.

```
  SUBROUTINE ADV_AST(status, device_id, buffer, buffer_length,
1            data_length, buffer_index, device_specific)
  INTEGER status             ! Returns the status of the I/O operation
  INTEGER device_id          ! Specifies the LIO-assigned device ID
  INTEGER*2 buffer           ! The actual buffer, NOT its address
  INTEGER buffer_length      ! The length of the buffer, in bytes
  INTEGER data_length        ! The length of the data in the buffer, in bytes
  INTEGER buffer_index       ! The buffer index, if supplied in LIO$ENQUEUE
  INTEGER device_specific    ! Device-specific argument (not used by ADV11-D)
```

Keep in mind the following when you use an AST routine:

- **An AST routine is used instead of a call to the LIO$DEQUEUE routine.** Since the AST routine is called whenever a device completes a buffer, neither the main program nor the AST routine uses the LIO$DEQUEUE routine.
- **If an AST routine does I/O to other devices, use the asynchronous I/O interface to minimize its execution time.** This means you should avoid using normal terminal I/O or file I/O for your programming language, such as the FORTRAN READ, WRITE, TYPE, ACCEPT, and PRINT statements; the VSL synchronous routine calls (LIO$READ and LIO$WRITE); and the LIO$DEQUEUE routine with a nonzero **wait** argument.

See Chapter 3 in *Getting Started with VAXlab* for information about how to write an AST routine to receive completed buffers from a device using VAX Ada, VAX BASIC, VAX C, VAX FORTRAN, and VAX Pascal.

The following online sample programs show the use of AST routines:

LIO_ADV_AST.BAS
LIO_ADV_AST.C
LIO_ADV_AST.FOR
LIO_ADV_AST.PAS

AST routines and their execution can also be tied to a particular event such as an overflow on a real-time clock, the setting of a bit on a parallel board, or the assertion of a service request on an IEEE-488 bus. Online sample program LIO_KWV_AST.FOR is a VSL application routine that uses the asynchronous I/O interface and an event AST routine to show the KWV11-C or Simpact RTC01 clock module's ability to call an AST routine on every clock tick.

1.6 I/O Device-Specific Interfacing

The LIO facility supports the following device-specific interfaces:

- First-in/first-out buffers (FIFOs)
- Handshaking
- Direct memory access (DMA)

FIFOs are supported for the following devices:

ADQ32
DRQ3B
Preston
Simpact RTC01

See Section 1.6.1, First-In/First-Out Buffers, before you write VSL application programs that use these devices.

Handshaking is supported for the following devices:

DRQ3B
DRV11-J
DRV11-WA

See Section 1.6.2, Handshaking, before you write VSL application programs that use these devices.

The following types of direct memory access are supported:

- Single-buffer DMA

  This interface is supported for the following devices:

  AAF01[1]
  AAV11-D
  ADF01[1]
  ADQ32
  ADV11-D
  DRB32
  DRQ3B
  DRQ11-C[1]
  Preston

[1] This device is available only in Europe.

See Section 1.6.3.1, Single-Buffer DMA, before you write VSL application programs that use these devices.

- Continuous DMA

  This feature is supported by the following devices:

  AAF01[1]
  ADF01[1]
  AAV11-D
  ADV11-D
  DRQ11-C[1]

  See Section 1.6.3.2, Continuous DMA, before you write VSL application programs that use these devices.

- Alternate-buffer DMA

  This feature is supported by the following devices:

  AAF01[1]
  ADF01[1]
  DRQ11-C[1]

  See Section 1.6.3.3, Alternate-Buffer DMA, before you write VSL application programs that use these devices.

- Double-buffer DMA

  This feature is supported by the following devices:

  ADQ32
  DRB32
  DRQ3B

  See Section 1.6.3.4, Double-Buffer DMA, before you write VSL application programs that use these devices.

The following sections describe each of these device-specific interfaces.

[1] This device is available only in Europe.

1.6.1 First-In/First-Out Buffers

The ADQ32, DRQ3B, Simpact RTC01, and Preston devices support first-in/first-out (FIFO) buffers that are built into these options.

If the load on your system bus is such that the data transfer cannot be made from the device to memory before another piece of data is available, then you lose data. FIFOs act as on-board memory for storing data while the device is unable to transfer data. This prevents new data from overwriting the data not yet transferred to memory.

It is possible to fill up the FIFO if the system cannot read the data and put it into memory quickly enough. This condition is called a **FIFO overrun**. The data rate of the data coming into the FIFO is faster than the data rate of the data being read out of the FIFO.

A similar but less common condition is called a **FIFO underrun**. This occurs when the system reads data out of the FIFO faster than data is coming into the FIFO. This condition is less likely to occur because the system usually checks to see if there is additional data in the FIFO before reading any data out of the FIFO.

The ADQ32 has one 512-word input FIFO. The DRQ3B has one 512-word FIFO for input and one 512-word FIFO for output. See Section 1.6.3.4, Double-Buffer DMA, for information about double buffering with the ADQ32 and DRQ3B.

The Simpact RTC01 has a 512-entry longword FIFO buffer.

Preston devices can be configured with a 1K to 64K FIFO buffer. The size of the FIFO buffer determines the time between LIO$ENQUEUE calls that the device can tolerate and still maintain continuous throughput. When the Preston is connected to the DRQ3B it will also use the pair of 512-word FIFOs.

1.6.2 Handshaking

This section explains the handshaking interfaces available when you use the DRQ3B, DRV11-J, and DRV11-WA devices.

1.6.2.1 The DRQ3B and Handshaking

The DRQ3B uses a two-line interlocked handshake to ensure data transfer between an external device and the DRQ3B.

The handshake consists of signals STROBE and ACKNOWLEDGE for input and DATA VALID and ACKNOWLEDGE for output that are found on the connectors for the DRQ3B. These connectors let you attach the DRQ3B to another device, such as the Preston GMAD A/D subsystem. See the *DRQ3B Parallel DMA I/O Module User's Guide* for more information.

For input on channel 0, STROBE is sent (assertion low) from the external device when the external device is asserting valid data. The ACKNOWLEDGE (ACK) signal is sent (assertion low) from the DRQ3B after the DRQ3B has successfully read the data value.

The sequence of events for input of data is as follows:

1. When STROBE is received low from the external device, the DRQ3B reads the data input lines and places the data value in the FIFO. The DRQ3B then asserts the ACK signal low, indicating it has received the data value.
2. When the external device sees ACK go low, it sets the STROBE signal high and prepares to transmit the next data word.
3. The DRQ3B then releases the ACK signal letting it go high. (If the FIFO is full, however, ACK is held low until a word is read out of the FIFO.) ACK high indicates that the DRQ3B is ready for the next data word.
4. The external device places valid data on the data lines and asserts STROBE low, starting the cycle over.

For output on channel 1, the process is similar, but the roles of the DRQ3B and external device are reversed. Two handshake signals, DATA VALID (DAV) and ACKNOWLEDGE (ACK), are used. DAV is sent (asserted low) from the DRQ3B to the external device and indicates that there is currently valid data on the data lines. ACK is received (asserted low) from the external device when the external device has successfully read the data value.

The sequence of events for output of data is as follows:

1. The DRQ3B places a data value from the FIFO onto the output data lines. It then asserts the DAV signal low, indicating that valid data is present.
2. The external device reads the data and asserts its ACK signal low.
3. When the DRQ3B receives the ACK signal, it releases the DAV line, letting it go high.
4. When the external device sees the DAV line go high, it releases the ACK signal, letting it go high.
5. The DRQ3B places the next data word on the output data lines and then asserts DAV low, starting the handshake cycle over again.

1.6.2.2 The DRV11-J and Handshaking

The DRV11-J device hardware can be physically jumpered for a two-wire handshake. The setting of Jumper W11 on the DRV11-J board determines whether the hardware is jumpered for a two-wire handshake. See the *DRV11-J Parallel Line Interface User's Guide* for information about how to jumper the board.

The value of the LIO$K_HANDSHAKE parameter determines whether handshaking is software-enabled for the device. **To transfer more than one data point per buffer, the physical hardware of the DRV11-J device must be jumpered appropriately and the LIO$HANDSHAKE parameter must be used to software-enable the device's handshaking feature.**

When the device is set up for a two-wire handshake, only the low 12 bits of port A are available for AST routines. See Section 2.3.4, DRV11-J Support, for more information.

1.6.2.3 The DRV11-WA and Handshaking

The DRV11-WA uses a two-wire handshake to synchronize data transfers. In addition, the DRV11-WA option supports several different transfer types. Because a complete description of both the two-wire handshake and data transfer types is beyond the scope of this guide, see the *DRV11-WA General Purpose DMA Interface User's Guide* for more information.

1.6.3 Direct Memory Access I/O

The AAV11-D and ADV11-D devices transfer data using single-buffer or continuous direct memory access (DMA) I/O. See Section 1.6.3.1, Single-Buffer DMA, and Section 1.6.3.2, Continuous DMA, for more information.

The AAF01[1], ADF01[1], and DRQ11-C[1] devices transfer data using single-buffer, continuous, or alternate-buffer DMA. See Section 1.6.3.1, Single-Buffer DMA, Section 1.6.3.2, Continuous DMA, and Section 1.6.3.3, Alternate-Buffer DMA, for more information.

The ADQ32, DRB32, and DRQ3B devices transfer data using single-buffer or double-buffer DMA. Double-buffer DMA means that the device can switch from the first buffer to the second buffer with no software intervention. See Section 1.6.3.4, Double-Buffer DMA, for more information.

1.6.3.1 Single-Buffer DMA

When a device is performing single-buffer DMA, information about each buffer must be written to the device by software before each buffer is transferred. This contrasts with double buffering, where information about the next buffer is written to the device while the current buffer is transferred.

Devices that perform DMA perform single-buffer DMA by default when they are set up to use the synchronous I/O interface. When a device is set for synchronous I/O, you use the LIO$READ and LIO$WRITE routines to transfer data to and from the device.

The routine calls stop your program until the I/O operation completes. The device does not continue to transfer data while the program is preparing for the next I/O operation.

Except for the ADQ32, the buffer must be word-aligned. Most high-level languages automatically word-align buffers on an even memory address. However, if you are programming in VAX MACRO, or if the buffer is not the first datum in a FORTRAN COMMON, you must word-align the buffer from within the program context.

[1] This device is available only in Europe.

For VAX MACRO, use the .EVEN directive before the buffer. For FORTRAN COMMON, make sure that there are an even number of any LOGICAL*1, BYTE, or INTEGER*1 variables in front of the buffer. CHARACTER*n variables in front of the buffer must be of an even length.

Consider the following when you use the ADQ32 for single-buffer DMA:

- The ADQ32 restarts the current triggering mode after each buffer. Data points are lost because of the software delay in setting up the next buffer. For example, if the A/D is set up to be externally triggered, external triggers are ignored during the setup of the next buffer.

Consider the following when you use an AAV11-D or ADV11-D device for single-buffer DMA:

- When the AAV11-D and the ADV11-D are set to do single-buffer DMA output and input, respectively, you must supply an additional 512 bytes in each buffer. The DMA does not stop cleanly at the end of the buffer. It stops some time up to 256 D/A or A/D values later.
- When the AAV11-D is set to use asynchronous output, the **data_length** argument of the LIO$DEQUEUE routine returns the actual number of data values written. When the AAV11-D is set to use synchronous output, the LIO$WRITE routine does not return this information.
- When the ADV11-D is set for asynchronous or synchronous input, the **data_length** argument of the LIO$DEQUEUE and LIO$READ routines returns the size of the buffer, including the additional number of bytes read.

Consider the following when you set the AAF01,[1] ADF01,[1] or DRQ11-C[1] for single-buffer DMA:

- You must supply the address of a dummy buffer for the second buffer. You must also supply a dummy buffer length greater than zero.

[1] This device is available only in Europe.

1.6.3.2 Continuous DMA

When the AAF01,[1] AAV11-D, ADF01,[1] ADV11-D, and DRQ11-C[1] are set to do continuous DMA I/O, the DMA hardware runs continuously instead of stopping at the end of each buffer. This allows these devices to run at top speed with no interruptions.

For the AAV11-D and the ADV11-D, the DMA can continue at top speed with no interruptions because it is confined to a 64K-byte block of memory that it wraps around in. This memory is divided into a minimum of three buffers or a maximum of 16 buffers. All the software has to do is to keep filling or emptying buffers as fast as the DMA empties or fills them.

For the AAF01,[1] ADF01,[1] and DRQ11-C,[1] the DMA can continue at top speed without interruptions because it is confined to a (maximum) 252K-byte block of memory that it wraps around in. This memory is divided into two buffers. All the software has to do is to keep filling or emptying buffers as fast as the DMA empties or fills them. Each of the two buffers making up the (maximum) 252K-byte block of memory contains a 4-byte header that is used to synchronize the user program with the continuous DMA transfer.

LIO-Specific Details:

To do continuous DMA I/O with the AAV11-D and ADV11-D, you must do the following:

- Attach to the device through the LIO$ATTACH routine specifying the value of the **io_type** argument as LIO$K_QIO. This sets the device to use QIO. QIO is the default for both devices.
- Set the device to use the asynchronous user interface. This is the default for both devices.
- Set the device to continuous DMA mode using the LIO$K_CONT parameter.

The 64K-byte block of memory must be divided into a minimum of three buffers. To do this you can create a two-dimensional array that takes up 64K bytes. For example, a 64K block of 4 buffers can be defined as a 4-by-8192 array of two-byte integers.

[1] This device is available only in Europe.

To do continuous DMA I/O with the AAF01,[1] ADF01,[1] and DRQ11-C,[1] you must do the following:

- Attach to the device through the LIO$ATTACH routine specifying the value of the **io_type** argument as LIO$K_QIO. This sets the device to use QIO. QIO is the default for all devices.
- Set the device to use the asynchronous user interface. This is the default for all devices.

The 252K-byte block of memory must be divided into two buffers. To do this you can create a two-dimensional array that takes up 252K bytes. For example, a 252K block of two buffers can be defined as a 2-by-129024 array of two-byte integers.

Program-Specific Details:

Before continuous DMA output can start for the AAV11-D and the ADV11-D, you must fill and then enqueue all the buffers to the AAV11-D. Before continuous DMA input can start, you must enqueue all the buffers to the ADV11-D. For both devices, you must enqueue the buffers in ascending order (incrementing the array index). When all the buffers are enqueued, you start the AAV11-D output and ADV11-D input through the LIO$SET_I routine specifying the LIO$K_START parameter.

You must assign each buffer a unique event flag. The first time you enqueue a buffer, the event flag is internally associated with that buffer. The event flag is then used for all subsequent enqueues of that buffer, whether or not you specify the event flag.

The buffers can be refilled or emptied by using any of the following LIO mechanisms:

- Dequeueing the buffers and enqueueing them again
- Using a buffer-completion AST routine which requeues them
- Setting the AAV11-D and the ADV11-D to forward the buffers to another device that fills them and forwards them back

The buffers must always be enqueued in the same order. A minimum of two buffers must be enqueued to the device at all times, or the I/O stops without returning a condition value. Subsequent LIO$DEQUEUE calls do not return a buffer and do not generate a condition value.

When you stop the continuous DMA, the buffer transfer in progress returns with a zero **data_length**. The LIO facility does not know how much data was in the buffer so it returns a zero.

Before continuous DMA output can start for the AAF01,[1] ADF01,[1] and DRQ11-C,[1] you must fill and enqueue both buffers to the AAF01 or DRQ11-C device. Before continuous DMA input can start, you must enqueue both buffers to the ADF01 or DRQ11-C device. For all three devices, the **device_specific** argument of the LIO$ENQUEUE routine supplies the buffers. Thus, only one LIO$ENQUEUE routine call is necessary to enqueue both buffers and to start the continuous DMA. The parameters passed by the **device_specific** argument for continuous DMA transfer must contain the following:

- A mask specifying block mode, start the DMA conversion, and, if desired, burst mode, for example:

  ```
  LIO$M_BLOCK!LIO$M_START_CONV!LIO$M_BURST
  ```

- The address of the first data buffer
- The length of the first data buffer, excluding the 4-byte header
- The address of the second data buffer
- The length of the second data buffer, excluding the 4-byte header
- A block count of 0, indicating continuous DMA

The device continues to operate in continuous DMA mode until one of the following occurs:

- The program terminates the transfer with an LIO$K_CANCEL request.
- An error condition occurs.

[1] This device is available only in Europe.

When transferring more than one buffer, the program must acknowledge completion of each buffer by clearing the buffer header. The buffer header contains the current block count number after each transfer. Synchronization with the driver is provided at the end of each buffer by all of the following:

- Setting the specified event flag
- Queueing the specified AST
- Setting the "buffermark" in the I/O status block

The **buffermark** is the block number (1 or 2) of the last block being transferred. It is updated at the completion of each block.

The buffers can then be refilled or emptied and the buffer header cleared by either:

- Waiting for the specified event flag
- Executing an AST routine

To stop continuous DMA data transfers, issue an LIO$K_CANCEL request and dequeue the buffers.

VMS-Specific Details:

You must page-align the 64K-block of memory required to perform continuous DMA transfers. To do this, you can place the 64K block into a PSECT and then use a linker options file to page-align it.

If you are programming in VAX C, you can put a block of memory in a PSECT by declaring an external array or structure of 64K-bytes, for example:

```
/*
page_align.c
*/
main()
{
extern  buffers[4][8*1024];
                /* this program does nothing */
}
```

VAX C creates a PSECT with the name of the array or structure.

If you are programming in VAX FORTRAN, you can put a block for memory in a PSECT by declaring an array of 64K bytes and placing the array in a named COMMON block, for example:

```
        PROGRAM page_align
        COMMON /buffers/ibuffs              !Put the data in a PSECT
        INTEGER*2 ibuffs(8*1024, 4)         !Four 8K-word buffers
C                       this program does nothing
        STOP
        END
```

VAX FORTRAN generates a PSECT with the name COMMON, which the linker can page-align.

Use a linker options file to page-align the PSECT. The linker options file must contain the following statement:

```
PSECT_ATTR = user_buff,PAGE
```

The following LINK command page-aligns the PSECT declared by the previous sample programs. For simplicity, this example reads the options from SYS$INPUT instead of from a separate linker options file.

```
$ LINK/MAP/FULL page_align,SYS$INPUT/OPT
PSECT_ATTR=buffers,PAGE
^Z
```

1.6.3.3 Alternate-Buffer DMA

Alternate-buffer DMA is the type of double-buffer DMA accomplished by the AAF01,[1] ADF01,[1] and DRQ11-C[1] devices. For alternate-buffer DMA, as with continuous DMA, a block of memory is allocated and divided into two buffers. The device alternates between the two buffers, retrieving data from the buffers, or filling the buffers with data. Each buffer has a maximum size of 126K bytes.

The only difference between continuous DMA and alternate-buffer DMA for these devices is that with alternate-buffer DMA, the block count parameter, specified through the **device_specific** argument of the LIO$ENQUEUE routine, contains the actual number of blocks to transfer. The device alternates between the two buffers until it exhausts the block count. Synchronization is accomplished in the same way.

[1] This device is available only in Europe.

1.6.3.4 Double-Buffer DMA

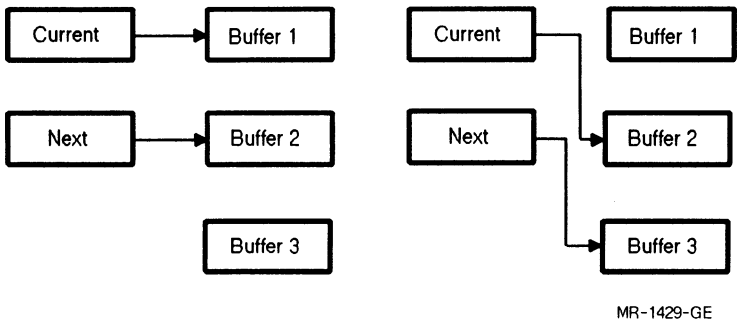
Double buffering allows the hardware to transfer one buffer, and then start transferring another buffer with no software intervention.

With single-buffer devices, after each buffer is filled or emptied, the software must set up the next buffer before the device can continue. The device does not continue to transfer data while the program is preparing for the next I/O operation.

Devices set for double-buffer DMA transfer have registers containing pointers to buffers, the CURRENT and NEXT pointers. See Figure 1–3 for the explanation below.

When the device completes a transfer using the CURRENT pointer to the current buffer (buffer 1), it generates an interrupt indicating that the transfer has completed. The device, however, does not have to wait for the interrupt to be serviced. Instead it uses the NEXT pointer to start the DMA transfer to or from the next buffer (buffer 2). Often the hardware is set up to copy the NEXT pointer to the CURRENT pointer. Then, the software sets up the NEXT pointer to point to the next transfer (buffer 3).

Figure 1–3: Double-Buffer DMA Pointer Sequence



This allows the data to be transferred continuously, without having to wait for the time it takes the software to set up the next buffer. Of course, the time it takes to transfer the first buffer must be greater than the time it takes to set up for the next buffer.

If the buffer setup time is reasonably well known, then the size of the buffer needed for double buffering can be easily computed by the following equation:

Buffer_Size = Software_Time * Data Transfer Rate

For example, assume that the buffer setup time is one millisecond. If the data transfer rate is 1 MHz (one million samples per second), then the equation becomes:

Buffer_Size = .001 sec * 1,000,000 samples/second
Buffer_Size = 1,000 samples

If each sample is one word, then you need a 1,000-word or 2,000-byte buffer.

If one millisecond is the least amount of time (best case) it takes to set up a buffer, but in actuality it may take up to one-tenth of a second (worst case) to set up a buffer, enter the worst case values into the equation, for example:[1]

Buffer_Size = .1 sec * 1,000,000 samples/second
Buffer_Size = 100,000 samples

Double-buffer DMA also allows you to queue multiple I/O requests before actually starting the I/O transfer. By queuing the I/O requests in advance, you minimize the amount of time it takes for the next buffer to be set up, because all the overhead associated with signaling the device about which buffers are going to be used is already done.

The ADQ32 device driver uses single-buffer transfers by default, but can be set for double-buffer transfers with the LIO$K_DBL_BUF parameter. The DRQ3B device driver double-buffers I/O requests whenever possible. Use the guidelines that follow to maximize the use of double-buffering with these devices.

In the last example, you can see that a buffer of 100,000 samples can accommodate the .1 second to get the next buffer ready. The ADQ32 and DRQ3B can handle a maximum buffer size of 32K words. By queuing ten 10,000-word buffers and holding them until all are queued, you can transfer 100,000 words and bypass the restriction of having a maximum buffer size of 32K words.

[1] ''Best'' case and ''worst'' case here are strictly dependent on an application's computational work requirements between I/O requests.

Note that the ADQ32 and DRQ3B device drivers take some time to set up the next buffer for an I/O transfer. The time it takes for this to occur varies based on system load, but should not be a problem if you use buffers of at least 8K words.

To queue 10 buffers to the DRQ3B or the ADQ32 before actually starting to transfer data, use the device-specific parameter LIO$M_HOLD_DMA with LIO$ENQUEUE for the first nine buffers. Then remove this parameter when you call LIO$ENQUEUE for the tenth buffer.

When the ADQ32 is set for double-buffer DMA transfers, the A/D takes advantage of its double-buffering capabilities and its on-board data FIFO buffer to keep data flowing continuously to succeeding buffers.

To do this, a user program should use a minimum of three buffers and must keep two buffers enqueued to the A/D at all times. If the A/D finishes a buffer and there are fewer than two more buffers enqueued to the device, the ADQ32 finishes the buffers it has and returns the last enqueued buffer with the LIO$_OVERRUN warning.

To prevent the A/D from terminating on the first buffer, enqueue a minimum of two buffers to the device using the LIO$ENQUEUE routine with LIO$M_HOLD_DMA as the value of the **device-specific** argument. This **device-specific** argument value inhibits the start of the DMA transfers until you enqueue a buffer without using the LIO$M_HOLD_DMA **device-specific** argument value. When the user program enqueues the last buffer in a double-buffering sequence, use the LIO$M_DONE_DBL_BUF value of the **device-specific** argument of the LIO$ENQUEUE routine. This prevents the device from returning the buffer with the LIO$_OVERRUN warning message.

Chapter 2

Laboratory I/O Device Support

This chapter describes the hardware devices and software pseudo-devices supported by VAXlab. The devices are listed alphabetically by category for ease of use. Each device support section presents an overview of the capabilities of the device, and instructions for attaching, setting up, and using the device.

This chapter is not intended to be read sequentially.

2.1 Real-Time Clock Devices

The KWV11-C and Simpact RTC01 are real-time clock devices you can use in the following ways:

- As a steady frequency source
- As a single-pulse source
- As a source of regular calls to an AST routine on the setting of event flags
- To count or time external events

The KWV11-C is compatible with the Q-bus.

The Simpact RTC01 is a native-mode device compatible with the VAXBI bus.

The primary differences between the two devices are:

- Bits of resolution—the KWV11-C has 16, the Simpact RTC01 has 32.
- Maximum speed—1 MHz for the KWV11-C, 10 MHz for the Simpact RTC01.
- The KWV11-C has a single-count register, while the Simpact RTC01 has a 512-entry longword FIFO buffer to store successive counts.

For more information about the KWV11-C, see the *AXV11-C/KWV11-C Analog Module and Real-Time Clock Module User's Guide.*

For more information about the RTC01, see the documentation from Simpact Associates, Inc.

2.1.1 Attaching the KWV11-C or Simpact RTC01

Attaching the KWV11-C or the Simpact RTC01 means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers, to the device.

You use the LIO$ATTACH routine to attach the KWV11-C or Simpact RTC01.

```
status = LIO$ATTACH (clock_id, 'KZA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **clock_id** argument returns the LIO-assigned device ID for the KWV11-C or Simpact RTC01 device. The KWV11-C or RTC01 is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification KZA0 specifies a KWV11-C (KZ) device, with controller letter A and unit number 0. (To specify an RTC01 (KB) device, use KBA0 as the device specification.) If you have only one KWV11-C or RTC01 device configured in your system, specifying the device type KZ or KB is sufficient.

The LIO$K_QIO constant value specifies the I/O type. The LIO facility also supports memory-mapped I/O (LIO$K_MAP) for the KWV11-C or RTC01 device.

NOTE

When you use the KWV11-C as the clock source for the AAV11-D, ADV11-D, and AXV11-C devices, attach both the clock and the I/O device with the same **io_type** argument.

2.1.2 Setting Up the KWV11-C or Simpact RTC01

Before you can begin using the KWV11-C or the Simpact RTC01 to trigger data transfers or time external events, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show KWV11-C or RTC01 device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–1: KWV11-C and Simpact RTC01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device to use asynchronous I/O.
LIO$K_CLK_RATE	Takes a specified frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_CLK_SRC	Sets the source frequency and divider for clock ticks and the source frequency for event timing.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on clock overflows or ST2 events.
LIO$K_EVENT_EF	Specifies the event flag to set on an external event or clock overflow.

Table 2–1 (Cont.): KWV11-C and Simpact RTC01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_FUNCTION	Specifies the function the clock is to perform.
LIO$K_START	Starts the device.
LIO$K_STOP	Stops the device.
LIO$K_SYNCH	Sets the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time in seconds before an I/O request is aborted.
LIO$K_TRIG	Sets the device trigger mode or source.
Simpact RTC01 only:	
LIO$K_COUNTER	Reads the count register of the Simpact RTC01.
LIO$K_INTERRUPT_LEVEL	Sets the level at which interrupts occur for the Simpact RTC01.
LIO$K_SCHMITT_TRIGGER	Sets the mode of operation for the two Schmitt triggers on the Simpact RTC01.

The function that the clock performs depends on how your program sets up the clock. The following sections describe how to use the parameters above.

2.1.3 Using the KWV11-C or Simpact RTC01 to Time External Events

This section describes how to use the KWV11-C or the Simpact RTC01 to time external events.

External events are negative (or positive) TTL transitions from a device or switch in your application. The polarity of the transition is set by a switch on the clock device UDIP panel. **The external event to be timed is connected to the Schmitt trigger 2 (ST2) input on the clock.**

To use the clock device to time external events, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the clock as described in Section 2.1.1, Attaching the KWV11-C or Simpact RTC01.
4. Set up the I/O interface. To time external events, you can use either the synchronous or asynchronous I/O interface.

```
status = LIO$SET_I (clock_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the clock function.

```
status = LIO$SET_I (clock_id, LIO$K_FUNCTION, 1, LIO$K_EVENT_REL)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine sets up the KWV11-C clock device to time the interval between pulses on the ST2, resetting the count to zero on each ST2 pulse.

   Note that LIO$K_EVENT_ABS could also be chosen. In this mode, the counter continues to run and is not reset to zero on each ST2 pulse.

6. Specify the clock source.

```
status = LIO$SET_I (clock_id, LIO$K_CLK_SRC, 1, 3)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies the clock source as the KWV11-C internal 10 kHz clock crystal. No clock divider is given, because the clock is being used to time external events.

NOTE

When using the KWV11-C or the Simpact RTC01 to time external events, you must use the LIO$K_CLK_SRC parameter to specify the clock source. The event-timing functions use only the clock source, and not the divider, to time external events. Do not use the LIO$K_CLK_RATE parameter because it sets both the clock source and a divider.

7. Set up the clock trigger mode.

```
status = LIO$SET_I (clock_id, LIO$K_TRIG, 1, LIO$K_IMMEDIATE)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This routine sets up the start condition. The clock does not start running at this time. It is started by the subsequent LIO$READ routine.

8. Read 10 clock pulses (20 bytes for the KWV11-C, 40 for the RTC01).

```
status = LIO$READ (clock_id, buffer, 20, data_length, )
         IF(.NOT. status) CALL LIB$SIGNAL(%VAL(status))
```

This routine starts the clock and reads 10 KWV11-C ST2 pulses. Each value in the buffer is the value of the clock counter when the ST2 pulse occurred. You can obtain the relative time between ST2 pulses by multiplying the number of clock ticks read by the source frequency selected.

The **device_specific** argument is not used with the KWV11-C or the RTC01 device.

Because this example sets the KWV11-C to use the synchronous I/O interface in step 4 of this procedure, the LIO$READ routine is used here to read the clock pulses. If you set the clock to use the asynchronous I/O interface, then you use the LIO$ENQUEUE and LIO$DEQUEUE routine calls to read the clock pulses.

9. Detach the I/O device and the clock.

```
status = LIO$DETACH (device_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$DETACH (clock_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_TIME_EVENT.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that uses the KWV11-C clock to time external events.

2.1.4 Using the KWV11-C to Trigger a Device

This section describes how to use the KWV11-C clock to trigger data transfers to and from the AAV11-D, ADV11-D, and AXV11-C devices. (You cannot use the Simpact RTC01 clock with these devices.) **To use the KWV11-C as the clock source for these devices, the clock overflow output must be externally wired to the device.**

To set up the clock to trigger a device, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the clock as described in Section 2.1.1.
4. Attach the I/O device being clocked as described in the device-specific support section in this chapter. Remember to attach the I/O device and the clock with the same **io_type** argument.
5. Set up the clock function.

```
status = LIO$SET_I (clock_id, LIO$K_FUNCTION, 1, LIO$K_REP_COUNT)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies the KWV11-C clock device as the clock source for the device. Remember that the clock must be wired to the device if it is to perform this function.

6. Set up the clock rate.

```
status = LIO$SET_R (clock_id, LIO$K_CLK_RATE, 1, 1000.0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies a clock rate of 1,000 Hertz.

   (LIO$K_CLK_SRC can also be used to specify a clock source and divider. LIO$K_CLK_SRC produces an exact known clock rate. LIO$K_CLK_RATE produces the best approximation of the specified rate.)

7. Set up the clock trigger mode.

```
status = LIO$SET_I (clock_id, LIO$K_TRIG, 1, LIO$K_IMMEDIATE)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

This routine sets up the start condition. The clock does not start running at this time. The routine call in step 10 of this procedure actually starts the clock.

(LIO$K_EXTERNAL can also be specified as the argument to LIO$K_TRIG. In this mode the clock starts on an external ST2 input.)

8. Set up the I/O device parameters, such as the I/O interface, A/D channels, and channel gains; and specify the I/O device's trigger mode to the clock.

```
.
.
.
status = LIO$SET_I (device_id, LIO$K_TRIG, 1, LIO$K_CLK_POINT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This routine sets the I/O device to output to one channel on each clock tick.

9. Enqueue a buffer to the I/O device.
10. Start the clock. If the clock trigger mode was specified as external, the clock starts when the external ST2 input occurs.

```
status = LIO$SET_I (clock_id, LIO$K_START, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Dequeue the completed buffer from the I/O device.
12. Stop the clock.

```
status = LIO$SET_I (clock_id, LIO$K_STOP, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

13. Process the buffer.
14. Detach the I/O device and the clock.

```
status = LIO$DETACH (device_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$DETACH (clock_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_ASYNCH_CLK_TRIG.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that uses the KWV11-C with the AXV11-C device for clocked asynchronous input.

2.1.5 Using the Simpact RTC01 to Count External Events

This section describes how to use the Simpact RTC01 to count external events.

External events are negative (or positive) TTL transitions from a device or switch in your application. The polarity of the transition is set by a switch on the clock device UDIP panel. **The external event to be timed is connected to the Schmitt trigger 2 input on the clock.**

To use the clock device to count external events, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the clock as described in Section 2.1.1, Attaching the KWV11-C or Simpact RTC01.
4. Set up the I/O interface.

```
status = LIO$SET_I (clock_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the clock trigger mode.

```
status = LIO$SET_I (clock_id, LIO$K_TRIG, 1, LIO$K_IMMEDIATE)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine sets up the software start condition. The LIO$K_START parameter starts the clock running.

6. Set up the clock source and rate.

```
status = LIO$SET_R (clock_id, LIO$K_CLK_RATE, 1, 1.0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine sets the clock rate as 1 Hz.

7. Set up the clock for repeat counting.

```
status = LIO$SET_I (clock_id, LIO$K_FUNCTION, 1, LIO$K_REP_COUNT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Start the clock.

```
status = LIO$SET_I (clock_id, LIO$K_START, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Read the counter on the clock.

```
status = LIO$SHOW (clock_id, LIO$K_COUNTER, showbuf(1), showlen)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
status = LIO$SHOW (clock_id, LIO$K_COUNTER, showbuf(2), showlen)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

These routines read the counter on the RTC01 twice in order to measure the net overhead of reading the clock. You will need to calculate the difference.

10. Detach the clock.

```
status = LIO$DETACH (clock_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_RTC01_COUNTER.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program showing the routines given above.

2.1.6 Using the KWV11-C to Avoid Trigger Slivering

The AAV11-D, ADV11-D, and AXV11-C I/O devices cannot be successfully started when a trigger signal is produced at approximately the same time that the LIO software attempts to enable the device. This condition is called **trigger slivering**. LIO returns the LIO$_IOERROR condition value when this happens, indicating that the hardware detected an error. If this condition value is returned, check the value of the **data_length** argument of the LIO$DEQUEUE or LIO$READ routines. If the value of the **data_length** argument is zero, trigger slivering is probably occurring.

To avoid trigger slivering, use one of the following solutions:

- If the trigger signal is produced by the KWV11-C clock device, there are two separate solutions, one for synchronous calls and one for asynchronous calls.
  - For synchronous calls, supply the clock ID as the optional parameter value to the trigger mode (LIO$K_TRIG) set parameter. The synchronous call (LIO$READ or LIO$WRITE) to the I/O device starts the device, and then starts the clock.

- For asynchronous calls, you must ensure that your program enqueues buffers to the I/O device before it starts the clock. If the I/O device is set for continuous DMA, the program must start the device (LIO$K_START) **before** it starts the clock.

- If the trigger signal is produced by an external trigger, the first external trigger signal must not occur before the device is enabled through LIO$ENQUEUE or LIO$READ, or LIO$SET_I when starting continuous DMA.

If the trigger signal is produced by an external trigger, it can be gated by the KWV11-C clock and handled according to the I/O interface (synchronous or asynchronous) that the device is set to use.

To set up the I/O device and the clock to gate an external trigger with the clock, do the following:

1. Attach the I/O device and the clock. If the I/O device is attached to use QIOs, you should attach the clock to use QIOs also. Otherwise, the overhead associated with the QIO to the device may cause the I/O device to start up after the clock starts up.
2. Set up the I/O device by:
   a. Specifying the I/O interface that the I/O device and the clock are to use.
   b. Setting the I/O device trigger source (LIO$K_TRIG) to be the clock instead of the external trigger.
3. Set up the clock by:
   a. Specifying the clock rate (LIO$K_CLK_RATE) as 1 MHz.
   b. Specifying the clock function (LIO$K_FUNCTION) as single count (LIO$K_SGL_COUNT).
   c. Specifying the clock trigger source (LIO$K_TRIG) as the external trigger.

NOTE

See the individual reference descriptions of the LIOK_TRIG, LIOK_CLK_RATE, and LIO$K_FUNCTION set parameters for the appropriate parameter values you must use.

Then, enable the clock as previously described for the synchronous and asynchronous calls.

2.2 Analog I/O Devices

This section describes the analog I/O devices supported by VAXlab.

2.2.1 AAF01 and ASF01 Support

The AAF01[1] is a 16-channel, high-speed, D/A converter subsystem. The interface is controlled by a programmable conversion rate and by a 1K-word Control Table. The conversion rate is controlled by an internal programmable clock or by an external clock signal supplied by the user. The internal clock rate is programmable in 100 nsec steps from 2.5 microseconds (300 kHz) to 400 microseconds (2.5 kHz).

The ASF01[1] is a 16-channel, simultaneous sample-and-hold (S/H) conditioning device for the AAF01 D/A conversion subsystem. With the ASF01, the AAF01 subsystem can perform conversions on all channels simultaneously. The AAF01 subsystem directly controls the ASF01's 16 sample-and-hold amplifiers.

The 1K-word Control Table determines the sequence of the data output. The Control Table must be maintained by the user program. The channel address and the operation mode for each conversion is contained in a control word. The operation, or control word, mode can have one of the following values:

Mode	Meaning
0	Conversion and increment Control Table Address (CTA) for next control word.
1	Conversion and go to control word 0.
2	Dummy cycle for current channel.
3	Same as mode 2. In addition, wait for sequence start pulse and go to next control word (must use SEQ CONT L input signal).

[1] This device is available only in Europe.

Mode	Meaning
4	Same as mode 2. In addition, deassert the SYSIN PROG L output signal during this conversion.
5	Same as modes 2 and 4. In addition, the Control Table starts at control word 0 after this conversion.
6	Same as modes 2 and 4. In addition, use the complement of the 12-bit Programmable Clock Register (PCR) as cycle time for this conversion.
7	Same as modes 2 and 4. In addition, assert the COUT L output signal during this conversion.

For more information about the AAF01, see the *AAF01 User's Manual.*

For more information about the ASF01, see the *ASF01 User's Manual.*

2.2.1.1 Attaching the AAF01

Attaching the AAF01 means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the AAF01.

```
status = LIO$ATTACH (aaf_id, 'UUA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **aaf_id** argument returns the LIO-assigned device ID for the AAF01 device. The AAF01 is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification UUA0 specifies an AAF01 (UU) device with controller letter A and unit number 0. If you have additional AAF01 devices, or if you have any number of ADF01 and/or DRQ11-C devices, or both, **you must attach each device with a unique controller letter.**

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the AAF01 device.

2.2.1.2 Setting Up the AAF01

Before you can begin data transfers using the AAF01, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show AAF01 device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-2: AAF01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ANA_OUT	Outputs a voltage value to one of the digital-to-analog channels on the AAF01 device.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_CANCEL	Cancels all pending I/O requests on the specified channel; used to stop continuous DMA.
LIO$K_CHANNEL	Specifies the D/A channel to use for output.
LIO$K_CLR_LBO	Clears the large buffer overflow condition on the AAF01 device.
LIO$K_COB	Reads or writes the Command Output (COUT) bit in the Command and Status Register (CSR) of the AAF01 device.
LIO$K_CTA	Reads or writes the Control Table Address (CTA) register of the AAF01 device.
LIO$K_CWT	Reads the Control Word Registers from, or writes the Control Word Registers to, the AAF01 device.
LIO$K_DATA_PATH	Selects the data path and channel number for the AAF01 device.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DRX_AST_RTN	Specifies a user-written AST routine to receive buffers when an AAF01 finishes processing them.
LIO$K_DRX_STAT	Returns the contents of the hardware registers of the DRQ11-C device.
LIO$K_ED_CTT	Enables or disables the Memory Transfer (MET) bit in the Command and Status Register (CSR) in the AAF01 device.

Table 2-2 (Cont.): AAF01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ED_ECE	Enables or disables the External Clock Enable (ECE) bit in the Command and Status Register (CSR) of the AAF01 device.
LIO$K_ED_SBE	Enables or disables the Sequence Break Enable (SBE) bit in the Command and Status Register (CSR) of the AAF01 device.
LIO$K_ERR_HANDLE	Specifies the way in which the AAF01 device handles errors.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on AAF01 unsolicited interrupts.
LIO$K_FUNCTION_BITS	Enables the setting of the four function bits in the DRQ11-C Status and Command Register (SCR).
LIO$K_PCR	Specifies the number of steps in the Programmable Clock Register (PCR) of the AAF01 device.
LIO$K_READ_STAT	Returns the status of the read-only bits in the Command and Status Register (CSR) of the AAF01 device.
LIO$K_RESET_AXF	Resets the AAF01 device.
LIO$K_RESET_DRX	Resets the DRQ11-C device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time in seconds before an I/O request is aborted.

2.2.1.3 Using the AAF01 for Synchronous Output

To set up the AAF01 device for synchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the AAF01 device as described in Section 2.2.1.1, Attaching the AAF01.

4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (aaf_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (aaf_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (aaf_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCTO, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (aaf_id, LIO$K_EVENT_AST, 3, aaf_ast_rtn,
1            aaf_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Set up the Control Table. This example sets up the control word mode to mode 0 for channels 0 through 14, and to mode 1 for channel 15. The **control_table(16)** array is a longword array with a dimension as large as the number of channels to sample, in this example 16. The **control_word_mode(8)** is a word array of length 8. The **control_word_mode(8)** is initialized to contain the following values: 0, 64, 128, 192, 256, 320, 384, and 448.

```
   INTEGER*4 control_table(16)
   INTEGER*2 control_word_mode(8)
   /0,64,128,192,256,320,384,448/

   DO 10 i = 1,15
      channel_number = i - 1
      control_table(i) = channel_number + control_word_mode(1)

10 CONTINUE

      control_table(16) = 15 + control_word_mode(2)
```

9. Load the Control Word Table, beginning at position 0 and ending at position 15. Begin loading the Control Word Table at location 0.

```
 val(1) = LIO$K_OUTPUT
 val(2) = %LOC(control_table)
 val(3) = 0
 val(4) = 15
 val(5) = 0

 status = LIO$SET_I (aaf_id, LIO$K_CWT, 5, val(1), val(2),
1            val(3), val(4), val(5))
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Load the Control Table Address register for the start of the conversion.

```
status = LIO$SET_I (aaf_id, LIO$K_CTA, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Set the speed by loading the Programmable Clock Register.

```
status = LIO$SET_I (aaf_id, LIO$K_PCR, 1, 100)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Use the LIO$WRITE routine to start the data transfer immediately. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer. The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes
4	Buffer address
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

To perform single word output, the source program looks as follows:

```
 device_specific(1) = LIO$M_WORD

 status = LIO$WRITE (aaf_id, buffer, buffer_length, data_length,
1            device_specific)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

To perform single buffer block output, the source program looks as follows:

```
device_specific(1) = LIO$M_BLOCK .OR. LIO$M_START_CONV
device_specific(2) = %LOC(buffer)
device_specific(3) = buffer_length
device_specific(4) = %LOC(dummy_buffer)
device_specific(5) = dummy_buffer_length
device_specific(6) = 1

status = LIO$WRITE (aaf_id, buffer, buffer_length, data_length,
1            device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

For word (LIO$M_WORD) output, the **buffer** argument is a word that contains the data to output. The **buffer_length** argument contains 2.

For block (LIO$M_BLOCK) or large-buffer (LIO$M_LARGE_BUF) output, **buffer** and **buffer_length** are dummy arguments. The actual required arguments are pointed to by the **device_specific** argument.

For single- or alternate-block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one subbuffer
e. Number of subbuffers to transfer

13. Detach the device.

```
status = LIO$DETACH (aaf_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.1.4 Using the AAF01 for Asynchronous Output

To set up the AAF01 device for asynchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the AAF01 device as described in Section 2.2.1.1, Attaching the AAF01.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (aaf_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (aaf_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (aaf_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCT0, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (aaf_id, LIO$K_EVENT_AST, 3, drq_ast_rtn,
1            drq_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Set up the Control Table. This example sets up the control word mode to mode 0 for channels 0 through 14, and to mode 1 for channel 15. The **control_table(16)** array is a longword array with a dimension as large as the number of channels to sample, in this example 16. The **control_word_mode(8)** is a word array of length 8. The **control_word_mode(8)** is initialized to contain the following values: 0, 64, 128, 192, 256, 320, 384, and 448.

```
   INTEGER*4 control_table(16)
   INTEGER*2 control_word_mode(8)
   /0,64,128,192,256,320,384,448/

   DO 10 i = 1,15
      channel_number = i - 1
      control_table(i) = channel_number + control_word_mode(1)

10 CONTINUE

      control_table(16) = 15 + control_word_mode(2)
```

9. Load the Control Word Table, beginning at position 0 and ending at position 15. Begin loading the Control Word Table at location 0.

```
val(1) = LIO$K_OUTPUT
val(2) = %LOC(control_table)
val(3) = 0
val(4) = 15
val(5) = 0

status = LIO$SET_I (aaf_id, LIO$K_CWT, 5, val(1), val(2),
1           val(3), val(4), val(5))
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Load the Control Table Address register for the start of the conversion.

```
status = LIO$SET_I (aaf_id, LIO$K_CTA, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Set the speed by loading the Programmable Clock Register.

```
status = LIO$SET_I (aaf_id, LIO$K_PCR, 1, 100)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Use the LIO$ENQUEUE routine to start the write request. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer. The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_INPUT or LIO$M_OUTPUT LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes
4	Buffer address or zero
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

To perform alternate-buffer block output, the source program looks as follows:

```
device_specific(1) = LIO$M_OUTPUT .OR. LIO$M_BLOCK .OR. LIO$M_START_CONV
device_specific(2) = %LOC(buffer_1)
device_specific(3) = buffer_1_length
device_specific(4) = %LOC(buffer_2)
device_specific(5) = buffer_2_length
device_specific(6) = 2

status = LIB$GET_EF(event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = SYS$CLREF(%VAL(event_flag))
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ENQUEUE (aaf_id, buffer, buffer_length, ,event_flag, ,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **buffer** and **buffer_length** arguments of the LIO$ENQUEUE routine are dummy arguments. The actual required arguments are pointed to by the **device_specific** argument.

For single- or alternate-block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one subbuffer
e. Number of subbuffers to transfer

13. Dequeue the buffer or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
14. Detach the device.

```
status = LIO$DETACH (aaf_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.2 AAV11-D Support

The AAV11-D is a two-channel 250-kHz D/A converter that supports direct memory access (DMA) I/O.

The LIO facility supports mapped output (for synchronous calls only) and QIO output. You can set four general purpose digital control lines on each output call.

You can use the KWV11-C real-time clock device as a steady frequency source to trigger data transfers to the AAV11-D. See Section 2.1, Real-Time Clock Devices, for more information.

2.2.2.1 Attaching the AAV11-D

Attaching the AAV11-D means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the AAV11-D.

```
status = LIO$ATTACH (aav_id, 'AYA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **aav_id** argument returns the LIO-assigned device ID for the AAV11-D device. The AAV11-D is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification AYA0 specifies an AAV11-D (AY) device with controller letter A and unit number 0. If you have only one AAV11-D device configured in your system, specifying the device type AY is sufficient.

The LIO$K_QIO value sets up the device to use QIOs. The LIO facility also supports memory-mapped I/O (LIO$K_MAP) for the device. If you do not specify the I/O type when you attach the AAV11-D device, by default it is attached to use QIOs.

2.2.2.2 Setting Up the AAV11-D

Before you can begin data transfers using the AAV11-D, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show AAV11-D device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–3: AAV11-D LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_CONT	Sets the device for continuous DMA mode.
LIO$K_DA_CHAN	Sets the AAV11-D D/A channels to use.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_N_DA_CHAN	Returns the number of device D/A channels in use.
LIO$K_SGL_BUF	Sets the device to stop DMA between buffers. Output is not continuous.
LIO$K_START	Starts the device when it is set up for continuous DMA transfers.
LIO$K_STOP	Stops the device when it is set up for continuous DMA transfers.

Table 2–3 (Cont.): AAV11-D LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time in seconds before an I/O request is aborted.
LIO$K_TRIG	Sets the device trigger mode or source.

2.2.2.3 Using the AAV11-D for Synchronous Output

To set up the AAV11-D device for synchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the AAV11-D device as described in Section 2.2.2.1, Attaching the AAV11-D. When the device is attached to use QIOs (LIO$K_QIO), it performs single-buffer DMA transfers. When the device is attached to use mapped I/O (LIO$K_MAP), it does not perform DMA transfers.

   When performing single-buffer DMA transfers, the data can overrun the end of the buffer up to 256 points. (The actual number of points varies each time.)

   Be sure to declare your data buffer (in step 2 of this procedure) to be at least 256 words longer than the buffer length your program passes to the LIO facility. Fill the overrun area with known values. Otherwise whatever happens to be there is output to the D/A if an overrun occurs.

   Acceptable values are zeros, copies of the last point in the buffer if one D/A channel is used, or copies of the last two points in the buffer if both D/A channels are used. Data overrun generally does not occur at low clock rates or at burst rates. The overrun area is not required when performing continuous DMA transfers.

   The minimum number of data points in the buffer must be twice the number of selected output channels. If your program needs to output one point to each selected D/A channel, attach the AAV11-D with memory-mapped (LIO$K_MAP) I/O.

4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (aav_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the digital-to-analog channel to use.

```
status = LIO$SET_I (aav_id, LIO$K_DA_CHAN, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies D/A channel 0.

6. Specify the device trigger mode.

```
status = LIO$SET_I (aav_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. This means that the data output begins as soon as the program executes the subsequent LIO$WRITE routine call and empties the buffer as fast as possible.

7. Output the buffer to the AAV11-D device. The single-buffer DMA transfer begins immediately on the LIO$WRITE routine call, and empties the buffer as fast as possible.

```
status = LIO$WRITE (aav_id, buffer, data_length, device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Detach the device.

```
status = LIO$DETACH (aav_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

If desired, you can use the **device_specific** argument of the LIO$WRITE routine to write the four digital control lines. Before outputting the buffer, the control lines are set with the complement of the value in the low four bits of the **device_specific** argument. When the buffer transaction completes, the bits are cleared.

The online sample program LIO_SGLBUF_DMA.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the synchronous I/O interface and single-buffer DMA to read 20 values from the ADV11-D device and then to write the values to the AAV11-D device.

2.2.2.4 Using the AAV11-D for Asynchronous Output

To set up the AAV11-D device for asynchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the AAV11-D device as described in Section 2.2.2.1, Attaching the AAV11-D. When the device is attached to use QIOs (LIO$K_QIO), it performs single-buffer DMA transfers by default. To set up the device to perform continuous DMA data transfers, specify continuous DMA mode in step 5 of this procedure.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (aav_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the DMA mode. To perform single-buffer DMA, completing this step is optional. (The AAV11-D device performs single-buffer DMA transfers, by default, when it is attached to use QIOs and the asynchronous I/O interface.) To perform continuous DMA, completing this step is required. Be sure to include the following routine line in your program.

```
status = LIO$SET_I (aav_id, LIO$K_CONT, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the digital-to-analog channel to use.

```
status = LIO$SET_I (aav_id, LIO$K_DA_CHAN, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies D/A channel 0.

7. Specify the device trigger mode.

```
status = LIO$SET_I (aav_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode.

   Immediate burst mode means that the data output begins as soon as the program executes the subsequent LIO$ENQUEUE routine call, and empties the buffer as fast as possible.

8. Enqueue the output buffer to the device. The single-buffer DMA data transfer starts immediately on the LIO$ENQUEUE routine call and empties the buffer as fast as possible.

```
 status = LIO$ENQUEUE (aav_id, buffer, buffer_length, , , ,
1            device_specific)
           IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You can use the **device_specific** argument of the LIO$ENQUEUE routine to write the four digital control lines. Before outputting the buffer, the control lines are set with the complement of the value in the low four bits of the **device_specific** argument. When the buffer transaction completes, the bits are cleared.

   See the description of the LIO$ENQUEUE routine in Chapter 3 for more information about using the **device_specific** argument.

9. Dequeue the buffer or use one of the other asynchronous I/O buffer handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.

10. Detach the device.

```
status = LIO$DETACH (aav_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.3 ADF01, AMF01, and ASF01 Support

The ADF01[1] is a high-speed, multichannel analog-to-digital converter subsystem.

The ADF01 has 16 single-ended input channels and eight differential input channels, and one output channel for the calibration of the input channels.

The AMF01[1] is an analog input multiplexer add-on option for the ADF01.

The AMF01 has 48 single-ended input channels or 24 differential input channels. The AMF01 option can extend the number of ADF01 channels to 64 single-ended input channels or 32 differential input channels.

[1] This device is available only in Europe.

The AMF01 has a 23-bit software programmable sequence timer that is controlled by a 1-MHz clock crystal. You can use the sequence timer for external timing of conversion sequences. The conversion rate is controlled by an internal programmable clock or by an external clock signal supplied by the user.

For each conversion, an entry in the ADF01 Control Table contains the channel number and channel gain used for the conversion. This entry also contains a code that signals which entry in the Control Table to use for the next conversion. You can use the Control Table to set up a large, fixed sequence of conversions.

The ASF01[1] is a 16-channel, simultaneous sample-and-hold (S/H) conditioning device for the ADF01 A/D conversion subsystem.

With the ASF01, the ADF01 subsystem can perform conversions on all channels simultaneously. The ADF01 subsystem directly controls the ASF01's 16 sample-and-hold amplifiers.

The 1K-word Control Table determines the sequence of the data input. The Control Table must be maintained by the user program. The channel address and the channel gain for each conversion is contained in a control word. The operation, or control word, mode can have one of the following values:

Mode	Meaning
0	Conversion and incrementing of Control Table Address (CTA) for next control word.
1	Conversion and go to control word 0.
2	Dummy cycle for current channel (conversion delay) and incrementing of Control Table Address (CTA).
3	Wait for sequence start pulse and increment Control Table Address (must use SEQ CONT L input signal).

Gain values of 1, 2, 5, 10, 20, 50, 100, and 200 can be applied to each channel.

For more information about the ADF01, see the *ADF01 User's Manual.*

[1] This device is available only in Europe.

For more information about the AMF01, see the *AMF01 User's Manual.*

For more information about the ASF01, see the *ASF01 User's Manual.*

2.2.3.1 Attaching the ADF01

Attaching the ADF01 means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device. Use the LIO$ATTACH routine to attach the ADF01.

```
status = LIO$ATTACH (adf_id, 'UUA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **adf_id** argument returns the LIO-assigned device ID for the ADF01 device. The ADF01 is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification UUA0 specifies an ADF01 (UU) device with controller letter A and unit number 0. If you have additional ADF01 devices, or if you have any number of AAF01 and/or DRQ11-C devices, or both, you must attach each device with a unique controller letter.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the ADF01 device.

2.2.3.2 Setting Up the ADF01

Before you can begin data transfers using the ADF01, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show ADF01 device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-4: ADF01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_BIN_DDR	Moves a complementary offset binary-coded output voltage into the DAC Data Register (DDR) of the ADF01 device.

Table 2–4 (Cont.): ADF01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_CANCEL	Cancels all pending I/O requests on the specified channel; used to stop continuous DMA.
LIO$K_CHANNEL	Specifies the D/A channel to use for output.
LIO$K_CLR_LBO	Clears the large buffer overflow condition on the ADF01 device.
LIO$K_COB	Reads or writes the Command Output (COUT) bit in the Command and Status Register (CSR) of the ADF01 device.
LIO$K_CTA	Reads or writes the Control Table Address (CTA) register of the ADF01 device.
LIO$K_CWT	Reads the Control Word Registers from, or writes the Control Word Registers to, the ADF01 device.
LIO$K_DATA_PATH	Selects the data path and channel number for the ADF01 device.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DRX_AST_RTN	Specifies a user-written AST routine to receive buffers when an ADF01 finishes processing them.
LIO$K_DRX_STAT	Returns the contents of the hardware registers of the DRQ11-C device.
LIO$K_ED_CTT	Enables or disables the Control Table Transfer (CTT) bit in the Command and Status Register (CSR) in the ADF01 device.
LIO$K_ED_ECE	Enables or disables the External Clock Enable (ECE) bit in the Command and Status Register (CSR) of the ADF01 device.
LIO$K_ED_SBE	Enables or disables the Sequence Break Enable (SBE) bit in the Command and Status Register (CSR) of the ADF01 device.
LIO$K_ERR_HANDLE	Specifies the way in which the ADF01 device handles errors.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on ADF01 unsolicited interrupts.

Table 2–4 (Cont.): ADF01 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_FUNCTION_BITS	Enables the setting of the four function bits in the DRQ11-C Status and Command Register (SCR).
LIO$K_PCR	Specifies the number of steps in the Programmable Clock Register (PCR) of the ADF01 device.
LIO$K_READ_STAT	Returns the status of the read-only bits in the Command and Status Register (CSR) of the ADF01 device.
LIO$K_RESET_AXF	Resets the ADF01 device.
LIO$K_RESET_DRX	Resets the DRQ11-C device.
LIO$K_STE	Clears the Sequence Timer Enable (STE) in the AMF01 Sequence Timer Register (ST1).
LIO$K_ST0_1	Writes to the 23-bit counter contained in the Sequence Timer Registers ST0 and ST1 of the AMF01 device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time in seconds before an I/O request is aborted.
LIO$K_VLT_DDR	Converts a voltage into its corresponding complementary binary-coded value and moves it to the DAC Data Register (DDR) of the ADF01 device.

2.2.3.3 Using the ADF01 for Synchronous Input

To set up the ADF01 device for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADF01 device as described in Section 2.2.3.1, Attaching the ADF01.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (adf_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (adf_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (adf_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCTO, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (adf_id, LIO$K_EVENT_AST, 3, adf_ast_rtn,
1            adf_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Set up the Control Table. This example sets up the control word mode to mode 0 for channels 0 through 14, and to mode 1 for channel 15. The channel gain is set to 0 for all channels.

   The **control_table(16)** array is a longword array with a dimension as large as the number of channels to sample, in this example 16. The **control_word_mode(4)** array is a word array of length 4. The **control_word_mode(4)** array is initialized to contain the following values: 0, 64, 128, and 192. The **gain_table(8)** array is a word array of length 8. The **gain_table(8)** array is initialized to contain the following values: 0, 512, 1024, 1536, 2048, 2560, 3072, and 3584. The **register_sub_address** argument is a word integer constant that has a value of 12288.

```
   INTEGER*4 control_table(16)
   INTEGER*2 control_word_mode(4) /0,64,128,192/
             ,gain_table(8)
              /0,512,1024,1536,2048,2560,3072,3584/
             ,register_sub_address /12288/

   DO 10 i = 1,15
      channel_number = i - 1
      control_table(i) = channel_number + control_word_mode(1) +
1                        gain_table(1) + register_sub_address

10 CONTINUE

      control_table(16) = 15 + control_word_mode(2) + gain_table(1) +
1                         register_sub_address
```

9. Load the Control Word Table, beginning at position 0 and ending at position 15. Begin loading the Control Word Table at location 0.

```
val(1) = LIO$K_OUTPUT
val(2) = %LOC(control_table)
val(3) = 0
val(4) = 15
val(5) = 0

status = LIO$SET_I (adf_id, LIO$K_CWT, 5, val(1), val(2),
1           val(3), val(4), val(5))
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Load the Control Table Address register for the start of the conversion.

```
status = LIO$SET_I (adf_id, LIO$K_CTA, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Set the speed by loading the Programmable Clock Register.

```
status = LIO$SET_I (adf_id, LIO$K_PCR, 1, 100)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Use the LIO$READ routine to start the data transfer immediately. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer.

    The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes
4	Buffer address
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

    To perform single word output, the source program looks as follows:

```
device_specific(1) = LIO$M_WORD

status = LIO$READ (adf_id, buffer, buffer_length, data_length,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

To perform single-buffer block input, the source program looks as follows:

```
device_specific(1) = LIO$M_BLOCK .OR. LIO$M_START_CONV
device_specific(2) = %LOC(buffer)
device_specific(3) = buffer_length
device_specific(4) = %LOC(dummy_buffer)
device_specific(5) = dummy_buffer_length
device_specific(6) = 1

status = LIO$READ (adf_id, buffer, buffer_length, data_length,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

For word (LIO$M_WORD) input, the **buffer** argument is a word that returns the input data. The **buffer_length** argument contains 2.

For block (LIO$M_BLOCK) or large-buffer (LIO$M_LARGE_BUF) input, **buffer** and **buffer_length** are dummy arguments. The required arguments are pointed to by the **device_specific** argument.

For single- or alternate-block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one subbuffer
e. Number of subbuffers to transfer

13. Detach the device.

```
status = LIO$DETACH (adf_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.3.4 Using the ADF01 for Asynchronous Input

To set up the ADF01 device for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADF01 device as described in Section 2.2.3.1, Attaching the ADF01.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (adf_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (adf_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (adf_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCT0, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (adf_id, LIO$K_EVENT_AST, 3, adf_ast_rtn,
1            adf_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Set up the Control Table. This example sets up the control word mode to mode 0 for channels 0 through 14, and to mode 1 for channel 15. The channel gain is set to 0 for all channels.

   The **control_table(16)** array is a longword array with a dimension as large as the number of channels to sample, in this example 16. The **control_word_mode(4)** array is a word array of length 4. The **control_word_mode(4)** array is initialized to contain the following values: 0, 64, 128, and 192. The **gain_table(8)** array is a word array of length 8. The **gain_table(8)** array is initialized to contain the following values: 0, 512, 1024, 1536, 2048, 2560, 3072, and 3584.

The **register_sub_address** argument is a word integer constant that has a value of 12288.

```
   INTEGER*4 control_table(16)
   INTEGER*2 control_word_mode(4) /0,64,128,192/
             ,gain_table(8)
              /0,512,1024,1536,2048,2560,3072,3584/
             ,register_sub_address /12288/

   DO 10 i = 1,15
      channel_number = i - 1
      control_table(i) = channel_number + control_word_mode(1) +
1                       gain_table(1) + register_sub_address

10 CONTINUE

      control_table(16) = 15 + control_word_mode(2) + gain_table(1) +
1                         register_sub_address
```

9. Load the Control Word Table, beginning at position 0 and ending at position 15. Begin loading the Control Word Table at location 0.

```
 val(1) = LIO$K_OUTPUT
 val(2) = %LOC(control_table)
 val(3) = 0
 val(4) = 15
 val(5) = 0

 status = LIO$SET_I (adf_id, LIO$K_CWT, 5, val(1), val(2),
1            val(3), val(4), val(5))
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Load the Control Table Address register for the start of the conversion.

```
status = LIO$SET_I (adf_id, LIO$K_CTA, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Set the speed by loading the Programmable Clock Register.

```
status = LIO$SET_I (adf_id, LIO$K_PCR, 1, 100)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Use the LIO$ENQUEUE routine to start the write request. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer.

The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_INPUT or LIO$M_OUTPUT LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes
4	Buffer address
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

To perform large-buffer input, the source program looks as follows:

```
device_specific(1) = LIO$M_INPUT .OR. LIO$M_LARGE_BUF .OR. LIO$M_START_CONV
device_specific(2) = %LOC(buffer)
device_specific(3) = buffer_length
device_specific(4) = 0
device_specific(5) = subbuffer_length
device_specific(6) = 10000

status = LIB$GET_EF(event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = SYS$CLREF(%VAL(event_flag))
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ENQUEUE (adf_id, buffer, buffer_length, ,event_flag, ,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

For block (LIO$M_BLOCK) or large-buffer (LIO$M_LARGE_BUF) input, **buffer** and **buffer_length** are dummy arguments. The required arguments are pointed to by the **device_specific** argument.

For single- or alternate-block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one subbuffer
e. Number of subbuffers to transfer

13. Dequeue the buffer or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
14. Detach the device.

```
status = LIO$DETACH (adf_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.4 ADQ32 Support

The ADQ32 device is a 200 kHz analog-to-digital (A/D) converter with 32 single-ended channels or 16 differential channels that supports direct memory access (DMA) I/O. The device's DMA architecture enables it to run multibuffered at 200 kHz continuously.

The ADQ32 device also contains five on-board counters. LIO uses combinations of the five counters to merge them into two clocks for actual use, the primary clock and the sweep clock.

The device supports a mixture of single-ended and differential channels, and can be set up with different gains on each channel. A flexible triggering scheme allows the A/D channels to be scanned in any order.

There are two basic triggering modes for the ADQ32:

- All points are triggered by the same source.
- Channel sweeps triggered by a source. Points within each channel sweep are trigger by another source.

The following variations on the basic triggering modes can be used to achieve a triggering scheme appropriate for your application:

- Using the LIO$K_TRIG parameter, choose the source to trigger points or sweeps.
  - Sources for points are: burst at top speed of the A/D (specify LIO$K_BURST as the first value), the A/D clock (specify LIO$K_AD_CLOCK as the first value), and the external trigger input (specify LIO$K_EXTERNAL as the first value).
  - Sources for sweeps are: the A/D clock (specify LIO$K_AD_CLOCK as the second value) and the external gate/trigger (specify LIO$K_EXTERNAL as the second value).

NOTE

The ADQ32 has two external inputs: the **external gate/trigger** input and the **external frequency** input. The LIO$K_EXTERNAL value generally refers to the external gate/trigger input. However, when you are specifying all points triggered by the same source (LIO$K_EXTERNAL, LIO$K_SAME, LIO$_SAME), and you are using the external gate/trigger input to gate the trigger (specified by the LIO$K_GATE parameter), then the LIO$K_EXTERNAL value refers to the external frequency input.

- Using the LIO$K_TRIG parameter, wait for the external gate/trigger to go low before starting the data collection (specify LIO$K_EXTERNAL, instead of LIO$K_SAME as the third value).
- Using the LIO$K_GATE parameter, gate the A/D on and off using the external gate/trigger input.
  - Level gating (LIO$K_LEVEL) means that the A/D runs while the external gate/trigger is high.
  - Edge gating (LIO$K_EDGE) means that succeeding low-going edges toggle the A/D from on to off, and from off to on.
  - Delayed edge gating (LIO$K_EDGE_DELAY) is the same as edge gating except that the acquisition is delayed one tick of the sweep clock. The delay time is 1/(sweep rate), where the sweep rate is set using the LIO$K_SWEEP_RATE parameter. For example, if the frequency of the sweep clock is 5 Hz, then the gate is delayed 1/5 of a second.

See Appendix A for more information about ADQ32 trigger sources and external gating.

See the *ADQ32 A/D Converter Module User's Guide* for more information about the ADQ32.

2.2.4.1 Attaching the ADQ32

Attaching the ADQ32 device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the ADQ32 device.

```
status = LIO$ATTACH (adq_id, 'AWA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **adq_id** argument returns the LIO-assigned device ID for the ADQ32 device. The ADQ32 is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification AWA0 specifies an ADQ32 (AW) device with controller letter A and unit number 0. If you have only one ADQ32 device configured in your system, specifying the device type AW is sufficient.

The LIO$K_QIO value sets up the device to use QIOs.

2.2.4.2 Setting Up the ADQ32

Before you can begin data transfers with the ADQ32, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show ADQ32 device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–5: ADQ32 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AD_CHAN	Sets the ADQ32 A/D channels.
LIO$K_AD_DIFFERENTIAL	Specifies whether to use single-ended or differential input for each channel set up with the LIO$K_AD_CHAN parameter.
LIO$K_AD_GAIN	Sets the ADQ32 A/D channel gains.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_BUFF_SIZE	Sets the maximum size, in bytes, of the asynchronous buffers to use.
LIO$K_CLK_RATE	For the primary clock, takes a specified frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_DBL_BUF	Enables double-buffer DMA data transfers.
LIO$K_DIAG_CHAN	Enables or disables the diagnostic inputs to ADQ32 channels 0, 1, and 2.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_GATE	Specifies the type of external gating used with the ADQ32 device.
LIO$K_N_AD_CHAN	Returns the number of device A/D channels.
LIO$K_SGL_BUF	Enables single-buffer DMA data transfers.
LIO$K_SWEEP_RATE	For the sweep rate clock, takes an ideal frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TRIG	Sets the device trigger mode or source.

2.2.4.3 Using the ADQ32 for Synchronous Input

To use the ADQ32 for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADQ32 device as described in Section 2.2.4.1, Attaching the ADQ32.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (adq_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the A/D channels to use.

```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 3, 0, 1, 2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the channel gains.

```
status = LIO$SET_I (adq_id, LIO$K_AD_GAIN, 3, 1, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Set up the device trigger mode.

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1            LIO$K_SAME, LIO$K_SAME)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. This means that the data input begins as soon as the program executes the subsequent LIO$READ routine call and fills the buffer as fast as possible.

8. Specify the buffer size (in bytes).

```
status = LIO$SET_I (adq_id, LIO$K_BUFF_SIZE, 1, 24)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies a 24-byte (12-word) buffer. The maximum allowable value for this parameter is a 64K-byte (32768-word) buffer.

9. The ADQ32 starts the data transfer immediately on the LIO$READ routine call and fills the buffer as fast as possible.

```
status = LIO$READ (adq_id, buffer, 24, data_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. You can process, store, or print out the data at this step.
11. Detach the device.

```
status = LIO$DETACH (adq_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.4.4 Using the ADQ32 for Asynchronous Input

To use the ADQ32 for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADQ32 device as described in Section 2.2.4.1, Attaching the ADQ32.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (adq_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the A/D channels to use.

```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 3, 0, 1, 2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the channel gains.

```
status = LIO$SET_I (adq_id, LIO$K_AD_GAIN, 3, 1, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Set up the device trigger mode.

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1            LIO$K_SAME, LIO$K_SAME)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. This means that the data input begins as soon as the program executes the subsequent LIO$ENQUEUE routine call and fills the buffer as fast as possible.

8. Specify the buffer size (in bytes).

```
status = LIO$SET_I (adq_id, LIO$K_BUFF_SIZE, 1, 24)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies a 24-byte (12-word) buffer. The maximum allowable value for this parameter is a 64K-byte (32768-word) buffer.

9. The ADQ32 starts the data transfer immediately on the LIO$ENQUEUE routine call and fills the buffer as fast as possible.

```
status = LIO$ENQUEUE (adq_id, buffer, 24, data_length, , , , )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Dequeue the buffer or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
11. Detach the device.

```
status = LIO$DETACH (adq_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.5 ADV11-D Support

The ADV11-D is a 50-kHz analog-to-digital (A/D) converter with programmable gain that supports direct memory access (DMA) I/O. You can set the ADV11-D with jumpers to either 16 single-ended channels or 8 differential channels. The LIO facility supports memory-mapped input (for synchronous calls only) and QIO input.

You can use the KWV11-C real-time clock device as a steady frequency source for the ADV11-D. See Section 2.1, Real-Time Clock Devices, for more information.

2.2.5.1 Attaching the ADV11-D

Attaching the ADV11-D means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the ADV11-D.

```
status = LIO$ATTACH (adv_id, 'AZA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **adv_id** argument returns the LIO-assigned device ID for the ADV11-D device. The ADV11-D is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification AZA0 specifies an ADV11-D (AZ) device with controller letter A and unit number 0. If you have only one ADV11-D device configured in your system, specifying the device type AZ is sufficient.

The LIO$K_QIO value sets up the device to use QIOs. The LIO facility also supports memory-mapped I/O (LIO$K_MAP) for the device. If you do not specify the I/O type when you attach the ADV11-D device, by default it is attached to use QIOs.

2.2.5.2 Setting Up the ADV11-D

Before you can begin data transfers using the ADV11-D, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show ADV11-D device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-6: ADV11-D LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AD_CHAN	Sets the ADV11-D A/D channels.
LIO$K_AD_GAIN	Sets the ADV11-D A/D channel gains.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_CONT	Sets the device for continuous DMA mode.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_N_AD_CHAN	Returns the number of device A/D channels.
LIO$K_SGL_BUF	Enables single-buffer DMA data transfers.
LIO$K_START	Starts the device.
LIO$K_STOP	Stops the device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TRIG	Sets the device trigger mode or source.

2.2.5.3 Using the ADV11-D for Synchronous Input

To set up the ADV11-D device for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADV11-D device as described in Section 2.2.5.1, Attaching the ADV11-D. When the device is attached to use QIOs (LIO$K_QIO), it performs single-buffer DMA transfers by default. When the device is attached to use mapped I/O (LIO$K_MAP), it does not perform DMA transfers.

NOTE

When performing single-buffer DMA transfers, the data can overrun the end of the buffer up to 256 points. (The actual number of points varies each time.) Be sure to declare your data buffer (in step 2 of this procedure) to be at least 256 words longer than the buffer length your program passes to the LIO facility. Data overrun generally does not occur at low clock rates or at burst rates. The overrun area is not required when performing continuous DMA transfers.

4. Set up the device to use synchronous I/O.

```
status = LIO$SET_I (adv_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the analog-to-digital channels to use.

```
status = LIO$SET_I (adv_id, LIO$K_AD_CHAN, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies A/D channel 0.

6. Specify the channel gain.

```
status = LIO$SET_I (adv_id, LIO$K_AD_GAIN 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies a channel gain of 1.

7. Specify the device trigger mode.

```
status = LIO$SET_I (adv_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. This means that the data input begins as soon as the program executes the subsequent LIO$READ routine call and fills the buffer as fast as possible.

8. Read a buffer from the device. The single-buffer DMA data transfer starts immediately on the LIO$READ routine call and fills the buffer as fast as possible.

```
status = LIO$READ (adv_id, buffer, data_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Process the buffer.
10. Detach the device.

```
status = LIO$DETACH (adv_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_SGLBUF_DMA.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the synchronous I/O interface and single-buffer DMA to read 20 values from the ADV11-D device and then to write the values to the AAV11-D device.

2.2.5.4 Using the ADV11-D for Asynchronous Input

To set up the ADV11-D device for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the ADV11-D device as described in Section 2.2.5.1, Attaching the ADV11-D. When the ADV11-D is attached to use QIOs (LIO$K_QIO), it performs single-buffer DMA transfers by default. To set up the device to perform continuous DMA data transfers, specify continuous DMA mode in step 5 of this procedure.
4. Set up the device to use asynchronous I/O.

```
status = LIO$SET_I (adv_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the DMA mode. To perform single-buffer DMA, completing this step is optional. (The ADV11-D device performs single-buffer DMA transfers by default when it is attached to use QIOs and the asynchronous I/O interface.) To perform continuous DMA, completing this step is required.

   Be sure to include the following routine line in your program.

```
status = LIO$SET_I (adv_id, LIO$K_CONT, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the analog-to-digital channels to use.

```
status = LIO$SET_I (adv_id, LIO$K_AD_CHAN, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies A/D channel 0.

7. Specify the channel gain.

```
status = LIO$SET_I (adv_id, LIO$K_AD_GAIN 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies a channel gain of 1.

8. Specify the device trigger mode.

```
status = LIO$SET_I (adv_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. This means that the data input begins as soon as the program executes the subsequent LIO$ENQUEUE routine call and fills the buffer as fast as possible.

9. Enqueue a buffer to the device. The single-buffer DMA data transfer starts immediately on the LIO$ENQUEUE routine call and fills the buffer as fast as possible.

```
status = LIO$ENQUEUE (adv_id, buffer, buffer_length, , , , , )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   If you are performing continuous DMA (see step 5), enqueue all buffers to be used. Data transfer does not start in continuous DMA until the LIO$K_START parameter executes. (If the trigger mode was LIO$K_EXT_BURST, data collection starts when the external trigger occurs.)

10. Dequeue the buffer or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
11. Detach the device.

```
status = LIO$DETACH (adv_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_CONT_DMA.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the asynchronous I/O interface and continuous DMA to read values from the ADV11-D device.

2.2.6 AXV11-C Support

The AXV11-C is a combination device with one 16-channel analog-to-digital (A/D) converter with programmable gain, and two digital-to-analog (D/A) converters. You can set the AXV11-C A/D with jumpers for 16 single-ended channels or 8 differential channels.

You can use the KWV11-C real-time clock device as a steady frequency source. See Section 2.1, Real-Time Clock Devices, for more information.

For more information about the AXV11-C, see the *AXV11-C/KWV11-C Analog Module and Real-Time Clock Module User's Guide.*

2.2.6.1 Attaching the AXV11-C

Attaching the AXV11-C means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the AXV11-C.

```
status = LIO$ATTACH (axv_id, 'AXA0', LIO$K_CTI)
         IF(.NOT. status) CALL LIB$SIGNAL(%VAL(status))
```

The **axv_id** argument returns the LIO-assigned device ID for the AXV11-C device. The AXV11-C is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification AXA0 specifies an AXV11-C (AX) device with controller letter A and unit number 0. If you have only one AXV11-C device configured in your system, specifying the device type AX is sufficient.

The LIO$K_CTI value sets up the device to use connect-to-interrupt (CTI) I/O. The AXV11-C is the only LIO device that supports CTI I/O.

NOTE

Before you can use CTI I/O with the AXV11-C, you must connect the CTI driver to the device. See Appendix B for instructions about connecting the CTI driver to the AXV11-C.

The LIO facility also supports QIOs (LIO$K_QIO) and memory-mapped (LIO$K_MAP) I/O for the device. If you do not specify the I/O type when you attach the AXV11-C device, by default it is attached to use QIOs.

2.2.6.2 Setting Up the AXV11-C

Before you can begin data transfers using the AXV11-C, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show AXV11-C device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–7: AXV11-C LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AD_CHAN	Sets the AXV11-C A/D channels.
LIO$K_AD_GAIN	Sets the AXV11-C A/D channel gains.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_CTI_BUF	Attaches the device with connect-to-interrupt I/O.
LIO$K_CTI_OVERHD	Returns the size (in bytes) of the connect-to-interrupt overhead.
LIO$K_DA_CHAN	Sets the AXV11-C D/A channels.

Table 2–7 (Cont.): AXV11-C LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_N_AD_CHAN	Returns the number of device A/D channels.
LIO$K_N_DA_CHAN	Returns the number of device D/A channels.
LIO$K_SYNCH	Sets up the device for synchronous I/O when attached for QIO only.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.
LIO$K_TRIG	Sets the device trigger mode or source.

2.2.6.3 Using the AXV11-C for Synchronous Input

To set up the AXV11-C device for synchronous connect-to-interrupt input, do the following:

1. Include the symbolic definition file appropriate for the programming language you are using.
2. Declare the variables and the data types of the variables you are using in your program.
3. Attach the AXV11-C device as described in Section 2.2.6.1, Attaching the AXV11-C.
4. Get an event flag for this process. This example uses the VMS Run-Time Library routine LIB$GET_EF to obtain a free VMS event flag.

```
status = LIB$GET_EF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the connect-to-interrupt buffer.

```
 status = LIO$SET_I (axv_id, LIO$K_CTI_BUF, 3, buffer, buffer_length,
1            event_flag)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Set up the device to use synchronous I/O.

```
status = LIO$SET_I (axv_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify the analog-to-digital channels to use.

```
status = LIO$SET_I (axv_id, LIO$K_AD_CHAN, 5, 0, 1, 2, 3, 4)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Specify the channel gains.

```
status = LIO$SET_I (axv_id, LIO$K_AD_GAIN, 5, 1, 1, 1, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Specify the device trigger mode.

```
status = LIO$SET_I (axv_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies immediate burst mode. Immediate burst mode means that the input begins as soon as the program executes the subsequent LIO$WRITE routine call and fills the buffer as fast as possible.

10. Read a buffer from the device. The data transfer starts immediately on the LIO$READ routine call and fills the buffer as fast as possible.

```
status = LIO$READ (axv_id, buffer, buffer_length, data_length,)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

    Buffer sizes must be 64K bytes or smaller for input.

The online sample program LIO_AXV_CTI.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to read A/D values from the AXV11-C using connect-to-interrupt I/O. See the following online sample programs for more information about using the AXV11-C with mapped I/O and with QIOs:

- LIO_AXV_MAPPED.BAS
- LIO_AXV_MAPPED.C
- LIO_AXV_MAPPED.FOR
- LIO_AXV_MAPPED.PAS
- LIO_AXV_QIO.FOR

2.2.6.4 Using the AXV11-C for Asynchronous Input

To set up the AXV11-C device for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the AXV11-C device as described in Section 2.2.6.1, Attaching the AXV11-C.
4. Set the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (axv_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the analog-to-digital channels to use.

```
status = LIO$SET_I (axv_id, LIO$K_AD_CHAN, 5, 0, 1, 2, 3, 4)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the channel gains.

```
status = LIO$SET_I (axv_id, LIO$K_AD_GAIN, 5, 1, 1, 1, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify the device trigger mode.

```
status = LIO$SET_I (axv_id, LIO$K_TRIG, 1, LIO$K_CLK_POINT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   The LIO$K_CLK_POINT value sets the A/D converter to sample one channel on each external trigger. If you specify several channels for use, the next channel in the series is sampled on each successive external trigger. When the A/D converter reaches the end of the channel list, it begins sampling again at the first channel in the list.

8. Get an event flag for the buffer. This example uses the VMS Run-Time Library routine LIB$GET_EF to obtain a free VMS event flag.

```
status = LIB$GET_EF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Enqueue the buffer to the AXV11-C A/D.

```
 status = LIO$ENQUEUE (axv_id, buffer, buffer_length, , event_flag, ,
1            LIO$K_INPUT)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

In this routine, the LIO$K_INPUT value of the **device_specific** argument signals the AXV11-C that the buffer is an input buffer and is to be enqueued to the A/D converter. Buffer sizes must be 64K bytes or smaller for input.

The **event_flag** argument is required if the device is attached with QIO and you want to wait for the buffer transaction to complete by calling the LIO$DEQUEUE routine with a nonzero **wait** argument. This is done in the following step.

10. Dequeue the buffer, specifying a nonzero **wait** argument. The LIO$DEQUEUE routine call waits for the input buffer transaction to complete before dequeueing the buffer.

```
 status = LIO$DEQUEUE (axv_id, buffer, buffer_length, data_length,
1            1, , )
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Detach the device.

```
status = LIO$DETACH (axv_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_ASYNCH_CLK_TRIG.FOR is a complete VAX FORTRAN program that shows how to read A/D values from the AXV11-C device using asynchronous I/O. This program also shows how to use the KWV11-C real-time clock device to externally trigger the data transfer.

2.2.7 DRQ11-C Support

The DRQ11-C[1] is a double-buffer DMA interface for continuous high-speed data exchange between the Q-bus and either the user's external device or another Q-bus.

For more information about the DRQ11-C, see the *DRQ11-C Alternate Buffer DMA Interface.*

[1] This device is available only in Europe.

2.2.7.1 Attaching the DRQ11-C

Attaching the DRQ11-C means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the DRQ11-C.

```
status = LIO$ATTACH (drc_id, 'UUA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drc_id** argument returns the LIO-assigned device ID for the DRQ11-C device. The DRQ11-C is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification UUA0 specifies a DRQ11-C (UU) device with controller letter A and unit number 0. If you have additional DRQ11-C devices, or if you have any number of AAF01 or ADF01 devices, or both, **you must attach each device with a unique controller letter.**

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the DRQ11-C device.

2.2.7.2 Setting Up the DRQ11-C

Before you can begin data transfers using the DRQ11-C, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRQ11-C device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-8: DRQ11-C LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_CANCEL	Cancels all pending I/O requests on the specified channel; used to stop continuous DMA.
LIO$K_CLR_LBO	Clears the large buffer overflow condition on the DRQ11-C device.
LIO$K_DATA_PATH	Selects the data path and channel number for the DRQ11-C device.

Table 2–8 (Cont.): DRQ11-C LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DRX_AST_RTN	Specifies a user-written AST routine to receive buffers when a DRQ11-C finishes processing them.
LIO$K_DRX_STAT	Returns the contents of the hardware registers of the DRQ11-C device.
LIO$K_ERR_HANDLE	Specifies the way in which the DRQ11-C device handles errors.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on DRQ11-C unsolicited interrupts.
LIO$K_FUNCTION_BITS	Enables the setting of the four function bits in the DRQ11-C Status and Command Register (SCR).
LIO$K_RESET_DRX	Resets the DRQ11-C device.
LIO$K_STAT_BITS	Reads the status bits (STAT0 - STAT3) in the Status and Command Register (SCR) of the DRQ11-C device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.

2.2.7.3 Using the DRQ11-C for Synchronous I/O

To set up the DRQ11-C device for synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the DRQ11-C device as described in Section 2.2.7.1, Attaching the DRQ11-C.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (drc_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (drc_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (drc_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCTO, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (drc_id, LIO$K_EVENT_AST, 3, drc_ast_rtn,
1            drc_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Use the LIO$READ or LIO$WRITE routine to start the data transfer immediately. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer. The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes
4	Buffer address or zero
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

To perform single word input, the source program looks as follows:

```
 device_specific(1) = LIO$M_WORD

 status = LIO$READ (drc_id, buffer, buffer_length, data_length,
1            device_specific)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

To perform single-buffer block output, the source program looks as follows:

```
device_specific(1) = LIO$M_BLOCK .OR. LIO$M_START_CONV
device_specific(2) = %REF(buffer)
device_specific(3) = buffer_length
device_specific(4) = %REF(dummy_buffer)
device_specific(5) = dummy_buffer_length
device_specific(6) = 1

status = LIO$WRITE (drc_id, buffer, buffer_length, data_length,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

For word (LIO$M_WORD) I/O, the **buffer** argument is a word that specifies where the input data is to be stored, or that contains the data to be output. The **buffer_length** argument contains 2.

For block (LIO$M_BLOCK) or large-buffer (LIO$M_LARGE_BUF) I/O, **buffer** and **buffer_length** are dummy arguments. The actual required arguments are pointed to by the **device_specific** argument.

For single- or alternate-block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one sub-buffer
e. Number of sub-buffers to transfer

9. Detach the device.

```
status = LIO$DETACH (drc_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.7.4 Using the DRQ11-C for Asynchronous I/O

To set up the DRQ11-C device for asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the DRQ11-C device as described in Section 2.2.7.1, Attaching the DRQ11-C.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (drc_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the device for direct data path.

```
status = LIO$SET_I (drc_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Reset the DRQ11-C DMA interface and clear the FNCT0 bit.

```
status = LIO$SET_I (drc_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCT0, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Connect to unsolicited interrupts and cancel any previous I/O request.

```
 status = LIO$SET_I (drc_id, LIO$K_EVENT_AST, 3, drc_ast_rtn,
1            drc_ast_param, LIO$K_CANCEL)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Use the LIO$ENQUEUE routine to start the read or write request. The **device_specific** argument is an array of longwords of length six that you use to specify control information about a data transfer. The following table shows the values of **device_specific**.

Index	Value
1	LIO$M_INPUT or LIO$M_OUTPUT LIO$M_WORD or LIO$M_BLOCK or LIO$M_LARGE_BUF LIO$M_START_CONV LIO$M_BURST
2	Buffer address
3	Buffer size, in bytes

Index	Value
4	Buffer address or zero
5	Buffer or subbuffer size, in bytes
6	Number of buffers or subbuffers to transfer

To perform alternate-buffer block input, the source program looks as follows:

```
device_specific(1) = LIO$M_INPUT .OR. LIO$M_BLOCK .OR. LIO$M_START_CONV
device_specific(2) = %REF(buffer_1)
device_specific(3) = buffer_1_length
device_specific(4) = %REF(buffer_2)
device_specific(5) = buffer_2_length
device_specific(6) = 2

status = LIB$GET_EF(event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = SYS$CLREF(%VAL(event_flag))
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ENQUEUE (drc_id, buffer, buffer_length, ,event_flag, ,
1           device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **buffer** and **buffer_length** arguments of the LIO$ENQUEUE routine are dummy arguments. The required arguments are pointed to by the **device_specific** argument.

For single- or alternate-buffer block I/O, the **device_specific** argument contains the:

a. Address of the first data buffer
b. Size of the first data buffer
c. Address of the second data buffer
d. Size of the second data buffer
e. Number of buffers to transfer

For large buffer I/O, the **device_specific** argument contains the:

a. Address of the large buffer
b. Size of the large buffer
c. Zero
d. Size of one subbuffer
e. Number of subbuffers to transfer

9. Dequeue the buffer or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
10. Detach the device.

```
status = LIO$DETACH (drc_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.8 Preston Support

The Preston device is an analog-to-digital (A/D) converter that supports arbitrary channel lists of up to 1024 channels with no programmable gain.

NOTE

In this guide, "Preston" refers to an A/D converter from Preston's GM or EM series.

The Preston contains an internal real-time clock capable of generating sampling rates in the range of 160 Hz to 1 MHz. It also contains an external start input and, optionally, an external clock input.

The Preston device is interfaced to a VAXlab system through one of three ways:

- DRQ3B parallel device
- DRV11-WA parallel device
- DRB32W, a DR11W-compatible port for the VAXBI bus

The LIO application routines treat each Preston interface as a separate device.

The maximum transfer rate from the Preston to memory using the DRQ3B interface is 1 MHz.

The maximum transfer rate from the Preston to memory using the DRV11-WA interface is 250 kHz.

For more information about the Preston, see the documentation from Preston Scientific.

2.2.8.1 Attaching the Preston

Attaching a Preston device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach a Preston device.

```
! To attach for DRQ3B interface:
status = LIO$ATTACH (preston_id, 'PFA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

! To attach for DRV11-WA or DRB32W interface:
status = LIO$ATTACH (preston_id, 'PGA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **preston_id** argument returns the LIO-assigned device ID for the device. The Preston is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification PFA0 specifies a Preston device interfaced to a VAXlab through the DRQ3B device, with controller letter A and unit number 0.

The device specification PGA0 specifies a Preston device interfaced to a VAXlab through the DRV11-WA or the DRB32W device, with controller letter A and unit number 0.

If you have only one Preston device configured in your system, specifying the device type PF or PG is sufficient.

NOTE

The Preston (DRV11-WA) device behaves identically to the Preston (DRB32W) device. When you attach a PG device on a VAXBI machine, the LIO facility automatically uses the DRB32W device as the Preston interface. You must install the UQW device driver on your system before you attempt to attach a Preston (DRB32W) device. The LIO$ATTACH routine fails if the device driver is not installed.

The LIO$K_QIO value specifies the I/O type. This is the only I/O type supported for use with Preston devices.

2.2.8.2 Setting Up the Preston

You can use the Preston (DRQ3B) device without specifying any LIO$SET or LIO$SHOW parameters if the default mode of operation is sufficient for your application. The default parameters were selected based on the simplest case of single channel data acquisition. The default operating parameters for this device are:

- A/D channel 0
- 500 kHz sampling rate
- Immediate start, clocked sweep
- Synchronous I/O interface

Before you begin data transfers with a Preston device, you may want to set up certain device characteristics different from the default operating parameters supplied.

The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show Preston device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–9: Preston LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AD_CHAN	Specifies the Preston A/D channels to use.
LIO$K_ADD_AD_CHAN	Adds a single channel to the Preston A/D converter channel list.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_BUFF_SIZE[1]	Sets the maximum buffer size for a data transfer, in bytes.
LIO$K_BURST_DIV	Specifies the divisor of the Preston's internal burst rate clock.
LIO$K_BURST_RATE	Specifies the rate of the Preston's internal burst rate clock.

[1]DRQ3B interface only

Table 2-9 (Cont.): Preston LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_CLK_BASE	Specifies the base crystal frequency of the Preston's internal clock.
LIO$K_CLK_DIV	Specifies the sampling rate of the Preston's internal clock.
LIO$K_CLK_RATE	Takes an ideal frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_CONT[1]	Sets the device for continuous DMA mode.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_INIT_AD_CHAN	Initalizes or clears the existing Preston A/D channel list.
LIO$K_N_AD_CHAN	Returns the number of device A/D channels.
LIO$K_SGL_BUF[1]	Sets the device to stop DMA between buffers. Output is not continuous.
LIO$K_START[1]	Starts the device running when it is set for continuous DMA mode.
LIO$K_STOP[1]	Stops the device when it is running in continuous DMA mode.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT[2]	Sets the length of time (in seconds) before an I/O request is aborted.
LIO$K_TRIG	Sets the device trigger mode or source.
LIO$K_UPDATE	Updates the Preston device to the current set-up specifications.

[1]DRQ3B interface only

[2]DRB32W and DRV11-WA interfaces only

2.2.8.3 Using the Preston for Synchronous Input

To set up a Preston device for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the Preston device as described in Section 2.2.8.1, Attaching the Preston.
4. Set up the A/D channel using the LIO$K_AD_CHAN parameter, or the LIO$K_INIT_AD_CHAN and LIO$K_ADD_AD_CHAN parameters.
5. Specify the A/D sampling rate using the LIO$K_CLK_RATE or LIO$K_CLK_DIV parameter.
6. Specify the device trigger source or mode using the LIO$K_TRIG parameter.
7. Set the device to use the synchronous I/O interface. This step is optional but is included in this procedure for clarity.

```
status = LIO$SET_I (preston_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. If you are setting up a Preston (DRQ3B), specify the buffer size (in bytes).

```
status = LIO$SET_I (preston_id, LIO$K_BUFF_SIZE, 1, 16384)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies an 8K-word (16,384-byte) buffer. The maximum allowable value for this parameter is a 32K-word (65,536-byte) buffer.

9. Update the Preston device using the LIO$K_UPDATE parameter. The parameter values you specify for the other parameters do not take effect until your program executes LIO$K_UPDATE.

```
status = LIO$SET_I (preston_id, LIO$K_UPDATE, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Read buffers of data using the LIO$READ routine.
11. Detach the device.

```
status = LIO$DETACH (preston_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.2.8.4 Using the Preston for Asynchronous Input

To use the Preston device for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the Preston device as described in Section 2.2.8.1, Attaching the Preston.
4. Set up the A/D channel using the LIO$K_AD_CHAN parameter, or the LIO$K_INIT_AD_CHAN and LIO$K_ADD_AD_CHAN parameters.
5. Specify the A/D sampling rate using the LIO$K_CLK_RATE or LIO$K_CLK_DIV parameter.
6. Specify the device trigger source or mode using the LIO$K_TRIG parameter.
7. Set the device to use the asynchronous I/O interface. This step is optional but is included in this procedure for clarity.

```
status = LIO$SET_I (preston_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Specify an AST routine (LIO$K_AST_RTN), a device event flag (LIO$K_DEVICE_EF), or buffer forwarding (LIO$K_FORWARD) as a synchronization mechanism to handle completed buffers.
9. If you are setting up a Preston (DRQ3B), specify the buffer size (in bytes).

```
status = LIO$SET_I (preston_id, LIO$K_BUFF_SIZE, 1, 16384)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies an 8K-word (16,384-byte) buffer. The maximum allowable value for this parameter is a 32K-word (65,536-byte) buffer.

10. Update the Preston device using the LIO$K_UPDATE parameter. The parameter values you specify for the other parameters do not take effect until your program executes the following routine.

```
status = LIO$SET_I (preston_id, LIO$K_UPDATE, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Initiate I/O requests by using the LIO$ENQUEUE routine to enqueue buffers to the device.

12. Use the LIO$DEQUEUE routine or one of the synchronization methods specified in step 8 of this procedure to retrieve the data buffers specified by the LIO$ENQUEUE routine.
13. Detach the device.

```
status = LIO$DETACH (preston_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.3 Digital I/O Devices

This section describes the digital I/O devices supported by VAXlab.

2.3.1 DRB32 Support

The DRB32 is a 32-bit parallel I/O port for the VAXBI bus. It operates in one direction (input or output) at a time.

The DRB32 contains eight status lines (input) and eight control lines (output) for sensing and controlling external hardware.

The DRB32 data path is 32 bits (longword) wide and it can be configured for 16-bit (word) and 8-bit (byte) transfers. It can also be configured to generate one parity bit per byte for applications where data integrity is critical.

The DRB32 supports half-duplex block mode DMA transfers between VAX memory and a user device using two sets of registers for continuous transfers.

The speed of the DRB32 depends on system activity and configuration. The peak transfer rate is 6 MB/second, which assumes immediate response by the VAXBI. On processors where the BI is not the memory bus, data rates may be slower. Using multiple DRB32 devices also reduces the throughput of each individual device.

The DRB32 supports buffer sizes up to a maximum of 960K bytes. The maximum number of buffers in continuous DMA mode is 16. To maintain continuous throughput, buffers must be enqueued in pairs to ensure double buffering. The DRB32 automatically double buffers when two or more DMA requests are outstanding. Performance is higher with larger buffers since the overhead of loading the DMA registers is significantly reduced.

The DRB32 performs octaword transfers. To achieve the most efficient operation from the device, align buffers on octaword boundaries, specifying the size as an integral number of octawords. If buffers are not aligned on octaword boundaries, the DRB32 generates masked octaword instructions which are slower.

For more information about the DRB32, see the *DRB32 Technical Manual.*

2.3.1.1 Attaching the DRB32

Attaching the DRB32 assigns a VMS I/O channel to the device and initializes LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the DRB32.

```
status = LIO$ATTACH (drb_id, 'UQA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drb_id** argument returns the LIO-assigned device ID for the DRB32 device. The DRB32 is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification UQA0 specifies a DRB32 (UQ) device, with controller letter A and unit number 0. If you have only one DRB32 configured in your system, specifying the device type UQ is sufficient.

The LIO$K_QIO value is the I/O type. This is the only supported I/O type for the DRB32 device.

2.3.1.2 Setting Up the DRB32

Before you can begin data transfers with the DRB32, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRB32 device characteristics. See Chapter 4 guide for reference descriptions of the parameters listed in this table.

Table 2–10: DRB32 LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_CTRL_AST	Assigns a user-written AST routine to be called when an external device writes data to the DRB32's input control port.
LIO$K_CTRL_PORT	Sets the output control port on the DRB32 device.
LIO$K_DATA	Performs an output operation to the parallel data path without using DMA.
LIO$K_DATA_WIDTH	Specifies the width of the parallel data path.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DIRECTION	Sets the direction (input or output) of the device.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_LOCK_BUFFER	Specifies buffers to be locked before beginning DMA transfers.
LIO$K_LOOP_BACK	Enables or disables loopback mode.
LIO$K_PARITY	Enables or disables the device to accept parity from external devices.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.
LIO$K_UNLOCK_BUFFER	Unlocks buffers previously locked with LIO$K_LOCK_BUFFER.

2.3.1.3 Using the DRB32 for Synchronous I/O

Using the DRB32 for synchronous I/O is the simplest way to use the device. Results are optimal when continuous performance is not required. You can still read or write data up to 960K bytes in a single transfer at the maximum speed of the device. However, synchronous I/O tends to make the overhead associated with each I/O operation more significant. Your program must wait until one transfer is complete before starting the next one. When set to use synchronous I/O, the DRB32 does not use double buffering.

To use the DRB32 for synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Declare or allocate a buffer up to 960K bytes in size. Be sure to specify the data type of the buffer in agreement with the value of the LIO$K_DATA_WIDTH parameter (see step 6). For example, if the data path is set to a longword (32 bits), the data buffer should be declared as a longword array. If the data path is set to a word (16 bits), the data buffer should be declared as a word array, and so on.

   ```
   INTEGER*2 buffer(4096)
   ```

4. Attach the DRB32 device as described in Section 2.3.1.1, Attaching the DRB32.
5. Set the width of the data path. To specify a 16-bit or 8-bit wide data path, you must set this parameter to the appropriate value. If this parameter is not set, the data path width defaults to 32 bits wide.

   ```
   status = LIO$SET_I (drb_id, LIO$K_DATA_WIDTH, 1, LIO$K_WORD)
            IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
   ```

   The width of the data path must be in agreement with the data type of the buffer you declared or allocated in step 3 of this procedure.

   Setting the data path width is optional. However, many external devices are only 8 or 16 bits wide. Setting the data path width to be the same as the size of the external device is memory efficient. This saves DMA transfers because the data buffer does not contain unused bytes that require I/O if the data path is wider than necessary.

6. Start the data transfer. Use the LIO$READ routine for input. Use the LIO$WRITE routine for output.

```
ctrl_bits = 5   !decimal 5 = binary 101
status =  LIO$WRITE (drb_id, buffer, 4096, ctrl_bits)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   Optionally, you can specify the **device_specific** argument to change the state of the output control lines. This may be necessary to place the external device in the proper mode for the data transfer. The **device_specific** argument is a longword value. The low order byte of this value contains the bit pattern you want to write to the output control port. This byte is written to the output control port before the data transfer begins.

   See the descriptions of the LIO$READ and LIO$WRITE routines in Chapter 3 for specific details about using these routines.

7. Detach the device.

```
status = LIO$DETACH (drb_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.3.1.4 Using the DRB32 for Asynchronous I/O

Using the DRB32 for asynchronous I/O enables your program to issue multiple I/O requests without waiting for the first request to complete. The driver double buffers DMA requests, thus reducing the overhead associated with reloading the DMA registers.

An asynchronous DMA request is issued by calling the LIO$ENQUEUE routine. This routine inserts an I/O request in the device's DMA queue. Each request is serviced in the order in which it is queued.

The DRB32 contains two sets of DMA registers. When two or more DMA requests are pending, the DRB32 loads both sets of DMA registers. When the first request completes, the device simply switches DMA registers in hardware and begins processing the next request. This improves continuous throughput by minimizing the delay between buffers.

To use the DRB32 for asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.

3. Declare or allocate buffers up to 960K bytes in size each. Declare as many buffers as you need for your application. (Declaring an even number of buffers helps you make the most efficient use of the DRB32's double buffering feature.)

   For maximum performance, make sure your buffers are quadword-aligned, and that their sizes are integral numbers of quadwords. The DRB32 moves quadwords on the VAXBI bus. Smaller buffer alignments cause less efficient operations.

```
INTEGER*4  buffer_address(4096)
```

4. Attach the DRB32 device as described in Section 2.3.1.1, Attaching the DRB32.
5. Set the DRB32 to perform input or output.

```
status = LIO$SET_I (drb_id, LIO$K_DIRECTION 1, LIO$K_INPUT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Set the width of the data path. To specify a 16-bit or 8-bit wide data path, you must set this parameter to the appropriate value. If this parameter is not set, the data path width defaults to 32 bits.

```
status = LIO$SET_I (drb_id, LIO$K_DATA_WIDTH, 1, LIO$K_WORD)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   Make sure the width of the data path is in agreement with the data type of the buffer you declared or allocated in step 3 of this procedure.

   Setting the data path width is optional. However, many external devices are only 8 or 16 bits wide. Setting the data path width to be the same as the size of the external device is memory efficient. This saves DMA transfers because the data buffer does not contain unused bytes that require I/O if the data path is wider than necessary.

7. To perform continuous DMA with the DRB32 device, you must lock all I/O buffers in memory prior to beginning DMA transfers.

```
 status = LIO$SET_I (drb_id, LIO$K_LOCK_BUFFER, 2, buffer_address,
1            4096)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

Use the LIO$SET_I routine call with the LIO$K_LOCK_BUFFER parameter, specifying the buffer address and the size (in bytes) of each buffer to lock. You can lock a maximum of 16 buffers that contain up to 960K bytes each. If you need to lock additional buffers, you must unlock a buffer after it completes processing and lock a new one.

```
 status = LIO$SET_I (drb_id, LIO$K_UNLOCK_BUFFER, 2, buffer_address,
1            4096)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Start the data transfer. Use the LIO$ENQUEUE routine to pass buffers to the DRB32 device. The program remains active and can do other processing during the data transfers. However, you need some method to signal your main program at the end of the buffer transfer. The simplest way to do this is to specify an event flag to associate with each buffer when you enqueue it. The event flag is set when the buffer completes processing, for example:

```
status = LIB$GET_EF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
status = LIO$ENQUEUE (drb_id, buffer, 4096, , event_flag, ,)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The LIO$ENQUEUE routine queues a buffer to the DRB32 device and associates the value of the **event_flag** argument with the buffer.

9. Return the buffer to the main program. Use the LIO$DEQUEUE routine to return buffers to the main program. If you specified an event flag in the LIO$ENQUEUE routine call, you can use this flag to ensure that the data transfer on a buffer is complete before you dequeue the buffer. Specify the value of the LIO$DEQUEUE **wait** argument as a nonzero integer. The LIO$DEQUEUE routine waits for the buffer to become available on the device's user queue and then dequeues it.

```
status = LIO$DEQUEUE (drb_id, buffer_address, buffer_length, , 1, , ,)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

Typically, you enqueue all buffers to the device before the data transfer begins. Then, you dequeue and process each buffer, one at a time, as it completes. When you finish processing a dequeued buffer, you can enqueue it again to maintain continuous throughput. This process can continue until your application completes execution.

10. Detach the device.

```
status = LIO$DETACH (drb_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.3.2 DRB32W Support

The DRB32W is a DR11W-compatible port for the VAXBI bus. It is designed for fast data transfers with external devices. The DRB32W uses handshake lines to control the flow of data and uses DMA to store the data in memory.

The DRB32W device can be set for either input or output. The direction is controlled in two ways:

- By an external line that can be either tied high or tied low to set the direction
- By an external device to change direction dynamically

The device also contains three programmable control lines and three user-readable sense lines.

The LIO support provided for this device is identical to the device support provided for the DRV11-WA, with the exception that you must specify a three-letter device type for the DRB32W.

For more information about the DRB32W, see the *DRB32 Technical Manual*.

2.3.2.1 Attaching the DRB32W

Attaching the DRB32W means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the DRB32W device.

```
status = LIO$ATTACH (drbw_id, 'UQWA0', LIO$K_QIO)
        IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drbw_id** argument returns the LIO-assigned device ID for the device. The DRB32W is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification UQWA0 specifies a DRB32W (UQW) device with controller letter A and unit number 0. If you have only one DRB32W device configured in your system, specifying the three-letter device type UQW is sufficient.

NOTE

The DRB32W is the only device supported by VAXlab for which you must specify a three-letter device type. All other devices require that you specify a two-letter device type.

The LIO$K_QIO value specifies the I/O type. This is the only I/O type supported for use with the DRB32W device.

2.3.2.2 Setting Up the DRB32W

Before you can begin data transfer using the DRB32W, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRB32W device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–11: DRB32W LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DIRECTION	Sets the direction (input or output) of the device.

Table 2-11 (Cont.): DRB32W LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_SYNCH	Sets up the device to use synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.

2.3.2.3 Using the DRB32W for Synchronous I/O

To set up the DRB32W to use synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Declare or allocate a buffer up to 64K bytes in size.

```
INTEGER*2 buffer(1024)
```

4. Attach the DRB32W device as described in Section 2.3.2.1, Attaching the DRB32W.
5. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (drbw_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

6. Start the data transfer. You can use the **device_specific** argument of the LIO$READ and LIO$WRITE routines to set the state of the output function bits.

```
func_bits = 5  !decimal 5 = binary 101
status = LIO$READ (drbw_id, buffer, 2048, data_length, func_bits)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

7. Process the data.
8. Detach the device.

```
status = LIO$DETACH (drbw_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.3.2.4 Using the DRB32W for Asynchronous I/O

To set up the DRB32W to use asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Declare or allocate a buffer up to 64K bytes in size.

```
INTEGER*2 buffer(1024)
```

4. Attach the DRB32W device as described in Section 2.3.2.1, Attaching the DRB32W.
5. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (drbw_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

6. Set the device for input or output.

```
status = LIO$SET_I (drbw_id, LIO$K_DIRECTION, 1, LIO$K_INPUT)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

7. Get a VMS event flag for synchronizing with the I/O

```
status = LIB$GET_EF(event_flag)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

8. Start the data transfer. You can use the **device_specific** argument of the LIO$ENQUEUE routine to set the state of the output function bits.

```
func_bits = 5  !decimal 5 = binary 101
status = LIO$ENQUEUE (drbw_id, buffer, 2048, event_flag, , func_bits)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

9. Dequeue the buffer, specifying a nonzero **wait** argument.

```
wait = 1
status = LIO$DEQUEUE (drbw_id, buff_addr, buff_len, , wait, ,)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

10. Detach the device.

```
status = LIO$DETACH (drbw_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.3.3 DRQ3B Support

The DRQ3B is a high-speed, 16-bit parallel interface device that supports Q-bus block mode direct memory access (DMA) transfers. The device contains one input port and one output port. Each port contains a 512-word first-in/first-out (FIFO) buffer.

The DRQ3B device driver supports double-buffer DMA operations on the device. Double buffering is used whenever multiple I/O requests are queued with a buffer size large enough to allow the next buffer to be enqueued before the previous buffer completes.

The DRQ3B supports an end-of-process (EOP) bit on the input port which is strobed into the FIFO. When the EOP bit is read out of the FIFO, the DMA controller terminates the current buffer and starts the next buffer in the queue. The **data_length** argument of both the LIO$READ and LIO$DEQUEUE routines returns the number of bytes actually transferred.

See Chapter 1 for more information about FIFOs and double buffering.

See the *DRQ3B Parallel DMA I/O Module User's Guide* for more information about the DRQ3B.

2.3.3.1 Attaching the DRQ3B

Attaching a DRQ3B device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the DRQ3B device.

```
status = LIO$ATTACH (drq_id, 'HXA0', LIO$K_QIO) !Input
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drq_id** argument returns the LIO-assigned device ID for the device. The DRQ3B is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification HXA0 attaches a DRQ3B (HX) device for input. The controller letter and unit number (A0) specify the **input** port of the DRQ3B.

To specify the **output** port of the DRQ3B, attach the device using HXA1 as the value of the **devspec** argument.

```
status = LIO$ATTACH (drq_id, 'HXA1', LIO$K_QIO) !Output
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

When attaching the DRQ3B device, you must use the complete device specification to signal LIO which port on the device to attach.

The LIO$K_QIO attaches the device to use QIOs. This is the only supported I/O type for the DRQ3B device.

NOTE

You can use the DRQ3B device for both input and output at the same time. However, the LIO facility treats the input and output ports of the device as separate devices, so each must be attached and set up as a separate device.

2.3.3.2 Setting Up the DRQ3B

Before you can begin data transfers with the DRQ3B, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRQ3B device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-12: DRQ3B LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_BUFF_SIZE	Sets the maximum size, in bytes, of the asynchronous buffers to use.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.

Table 2–12 (Cont.): DRQ3B LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_FUNCTION_BITS	Sets the function bits associated with the DRQ3B device.
LIO$K_STOP	Stops the device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.3.3.3 Using the DRQ3B for Synchronous I/O

To use the DRQ3B device for synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the DRQ3B device as described in Section 2.3.3.1, Attaching the DRQ3B.
4. Set the DRQ3B to use the synchronous I/O interface.

```
status = LIO$SET_I (drq_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the buffer size (in bytes).

```
status = LIO$SET_I (drq_id, LIO$K_BUFF_SIZE, 1, 16384)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies an 8K-word (16384-byte) buffer. The maximum allowable value for this parameter is a 64K-byte (32768-word) buffer.

6. Use the LIO$READ routine or the LIO$WRITE routine to read or to write the data.
7. Detach the device.

```
status = LIO$DETACH (drq_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

NOTE

Using the DRQ3B in synchronous mode effectively disables double-buffering.

2.3.3.4 Using the DRQ3B for Asynchronous I/O

To use the DRQ3B for asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the DRQ3B device as described in Section 2.3.3.1, Attaching the DRQ3B.
4. Set the DRQ3B to use the asynchronous I/O interface.

```
status = LIO$SET_I (drq_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify the buffer size (in bytes).

```
status = LIO$SET_I (drq_id, LIO$K_BUFF_SIZE, 1, 16384)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine specifies an 8K-word (16384-byte) buffer. The maximum allowable value for this parameter is a 64K-byte (32768-word) buffer.

6. Enqueue one or more buffers to the DRQ3B device.

```
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , ,
1            LIO$M_HX_HOLD_DMA)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , ,
1            LIO$M_HX_HOLD_DMA)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , ,
1            LIO$M_HX_HOLD_DMA)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , ,
1            LIO$M_HX_HOLD_DMA)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , ,
1            LIO$M_HX_HOLD_DMA)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
 status = LIO$ENQUEUE (drq_id, buffer, 16384, , , , )
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine segment queues six buffers to the DRQ3B device. The LIO$M_HX_HOLD_DMA **device_specific** value inhibits the start of double-buffer DMA transfers until all buffers are enqueued to the device. The absence of the LIO$M_HX_HOLD_DMA value of the last LIO$ENQUEUE routine call starts the double-buffer DMA.

You can use the **device_specific** argument of the LIO$ENQUEUE routine to enqueue all data buffers before starting DMA data transfers (as shown). You can also use it to prevent the setting of an event flag or the delivering of an AST routine until the buffer on the output port has been output.

The value of the **device_specific** argument can be either of the following:

- LIO$M_HX_HOLD_DMA

  This argument value inhibits the DMA transfers from starting until the last buffer is enqueued to the device. Use this argument with LIO$ENQUEUE when you enqueue all but the last buffer. The absence of this argument on the last enqueue to the device actually signals the DMA transfers to start.

- LIO$M_HX_RUN_DOWN

  This argument value prevents the setting of an event flag or the delivering of an AST routine until the buffer on the output port has been completely transferred. This argument is valid only when used with asynchronous writes and effectively disables double buffering (for that buffer only).

7. Dequeue the buffers or use one of the other asynchronous I/O buffer-handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.

   The **device_specific** argument of the LIO$DEQUEUE routine and the AST routine return the contents of the status register in the high 16 bits, and the contents of the DMA status register in the low 16 bits. See the *DRQ3B Parallel DMA Input/Output Module User's Guide* for more information.

8. Detach the device.

```
status = LIO$DETACH (drq_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.3.4 DRV11-J Support

The DRV11-J is a parallel interface device with four separate 16-bit ports (A, B, C, and D).

You can set each port individually for either input or output. Each bit of port A can be assigned a separate AST routine or an event flag. Each AST routine is called or the event flag is set when the bit to which either the AST routine or the event flag is assigned is cleared or set, depending on the value of the LIO$K_POLARITY parameter.

The device can be jumpered for a two-wire handshake. If the device is set for a two-wire handshake, only the low 12 bits of port A are available for AST routines. The value of the LIO$K_HANDSHAKE parameter signals LIO whether or not the two-wire handshake is software-enabled or disabled.

The DRV11-J must be hardware jumpered for a two-wire handshake to transfer more than one data point per buffer. The setting of Jumper W11 on the DRV11-J device determines whether the hardware is jumpered for a two-wire handshake. See the *DRV11-J Parallel Line Interface User's Guide* for information about how to jumper the device.

2.3.4.1 Attaching the DRV11-J

Attaching the DRV11-J means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device. Use the LIO$ATTACH routine to attach the DRV11-J.

```
status = LIO$ATTACH (drj_id, 'DNA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drj_id** argument returns the LIO-assigned device ID for the device. The DRV11-J is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification DNA0 specifies a DRV11-J (DN) device with controller letter A and unit number 0. If you have only one DRV11-J device configured in your system, specifying the device type DN is sufficient.

The LIO$K_QIO values sets the device to use QIOs. If you do not specify the I/O type when you attach the DRV11-J device, by default it is attached to use QIOs.

The LIO facility also supports memory-mapped (LIO$K_MAP) I/O for the device. Only the synchronous I/O interface is supported for mapped I/O. When set to use synchronous I/O, the bits of port A cannot be assigned AST routines or event flags.

2.3.4.2 Setting Up the DRV11-J

Before you can begin data transfer using the DRV11-J, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRV11-J device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–13: DRV11-J LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DIRECTION	Sets the direction (input or output) of the four DRV11-J ports.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on DRV11-J port-A bit events.
LIO$K_EVENT_EF	Specifies the event flag to set on an external event or clock overflow.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_HANDSHAKE	Specifies whether or not the DRV11-J is jumpered to use a two-wire handshake for each port.

Table 2–13 (Cont.): DRV11-J LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_POLARITY	Sets the bits of port A to call their AST routines on either a negative-going or positive-going edge and the polarity of the handshake, if any.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.

To set up the DRV11-J, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the DRV11-J as described in Section 2.3.4.1, Attaching the DRV11-J.
4. Set the device to use either synchronous or asynchronous I/O.
5. Set the direction of the four ports.

```
 status = LIO$SET_I (drj_id, LIO$K_DIRECTION, 4, LIO$K_INPUT,
1            LIO$K_INPUT, LIO$K_OUTPUT, LIO$K_OUTPUT)
           IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine segment sets the direction of ports A and B for input, and sets the direction of ports C and D for output.

6. Enable or disable handshaking.

```
status = LIO$SET_I (drj_id, LIO$K_HANDSHAKE, 1, LIO$K_OFF)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine disables handshaking.

7. If the device is set to use synchronous I/O, initiate the data transfer by using the LIO$READ or LIO$WRITE routine. If the device is set to use asynchronous I/O, initiate the data transfer by using the LIO$ENQUEUE routine.

   The **device_specific** argument of the LIO$ENQUEUE, LIO$READ, and LIO$WRITE routines is an unsigned longword integer. The low word selects the port, and the high word is a mask that selects the bits of the port.

The value of the first word can be one of the following:

Value	DRV11-J Port
0	Port A
1	Port B
2	Port C
3	Port D

If any bits are set in the second word of the argument, then only those bits are written to on output, or read from on input. On output, bits not selected are not changed. On input, bits not selected are returned as zeros. If the second word is zero, all bits are written to on output, and read from on input.

If all bits are to be selected, then the **device_specific** argument can be treated as a normal integer containing the port number.

8. If the device is set up asynchronous I/O, dequeue the buffer or use one of the other asynchronous I/O buffer handling mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
9. Detach the device.

```
status = LIO$DETACH (drj_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.3.5 DRV11-WA Support

The DRV11-WA is a 16-bit parallel I/O DMA interface device. It is designed for fast data transfers with external devices. The DRV11-WA uses handshake lines to control the flow of data and uses DMA to store the data in memory.

The DRV11-WA device can be set for either input or output. The direction is controlled in two ways:

- By an external line that can be either tied high or tied low to set the direction
- By an external device to change direction dynamically

The device also contains three programmable control lines and three user-readable sense lines.

See the *DRV11-WA General Purpose DMA Interface User's Guide* for more information about the DRV11-WA.

2.3.5.1 Attaching the DRV11-WA

Attaching the DRV11-WA means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device. Use the LIO$ATTACH routine to attach the DRV11-WA device.

```
status = LIO$ATTACH (drw_id, 'XAA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **drw_id** argument returns the LIO-assigned device ID for the device. The DRV11-WA is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification XAA0 specifies a DRV11-WA (XA) device with controller letter A and unit number 0. If you have only one DRV11-WA device configured in your system, specifying the device type XA is sufficient.

The LIO$K_QIO value specifies the I/O type. This is the only I/O type supported for use with the DRV11-WA device.

2.3.5.2 Setting Up the DRV11-WA

Before you can begin data transfer using the DRV11-WA, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show DRV11-WA device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–14: DRV11-WA LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DIRECTION	Sets the direction (input or output) of the device.

Table 2–14 (Cont.): DRV11-WA LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.

2.3.5.3 Using the DRV11-WA for Synchronous I/O

To set up the DRV11-WA to use synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Declare or allocate a buffer up to 64K bytes in size.

```
INTEGER*2 buffer(1024)
```

4. Attach the DRV11-WA device as described in Section 2.3.5.1, Attaching the DRV11-WA.
5. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (drw_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

6. Start the data transfer. You can use the **device_specific** argument of the LIO$READ and LIO$WRITE routines to set the state of the output function bits.

```
func_bits = 5  !decimal 5 = binary 101
status = LIO$READ (drw_id, buffer, 2048, data_length, func_bits)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

7. Process the data.
8. Detach the device.

```
status = LIO$DETACH (drw_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.3.5.4 Using the DRV11-WA for Asynchronous I/O

To set up the DRV11-WA to use asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Declare or allocate a buffer up to 64K bytes in size.

```
INTEGER*2 buffer(1024)
```

4. Attach the DRV11-WA device as described in Section 2.3.5.1, Attaching the DRV11-WA.
5. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (drw_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

6. Set the device for input or output.

```
status = LIO$SET_I (drw_id, LIO$K_DIRECTION, 1, LIO$K_INPUT)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

7. Get a VMS event flag for synchronizing with the I/O

```
status = LIB$GET_EF(event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Start the data transfer. You can use the **device_specific** argument of the LIO$READ and LIO$WRITE routines to set the state of the output function bits.

```
func_bits = 5  !decimal 5 = binary 101
status = LIO$ENQUEUE (drw_id, buffer, 2048, event_flag, , func_bits)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

9. Dequeue the buffer, specifying a nonzero **wait** argument.

```
wait = 1
status = LIO$DEQUEUE (drw_id, buff_addr, buff_len, , wait, ,)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

10. Detach the device.

```
status = LIO$DETACH (drw_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4 Isolated Real-Time I/O Devices

The following isolated real-time I/O devices are supported by VSL:

- IAV11-A
- IAV11-AA
- IAV11-B
- IAV11-C
- IAV11-CA
- IDV11-A
- IDV11-B
- IDV11-C
- IDV11-D

These devices, also called IXV11 or IXV devices, are available only in Europe. For more information about these devices, see the *Industrial I/O Modules for Q-Bus*.

2.4.1 IAV11-A, IAV11-AA, IAV11-C, and IAV11-CA Support

The IAV11-A[1] and IAV11-AA[1] are 16-channel analog-to-digital converters.

The IAV11-A has 12 single-ended input channels and four differential flying capacitator input channels. The IAV11-AA has 16 single-ended input channels.

Both devices have the following:

- 12-bit resolution
- Unipolar and bipolar input ranges
- Programmable channel gain
- Optional external trigger input

[1] This device is available only in Europe.

The IAV11-C[1] and IAV11-CA[1] are multiplexer devices you can use to expand the IAV11-A and IAV11-AA A/D converters to a maximum of 128 channels.

NOTE

The IAV11-A, IAV11-AA, IAV11-C, and IAV11-CA input devices are all referred to as IAV11-A in subsequent sections in this chapter.

2.4.1.1 Attaching the IAV11-A

Attaching an IAV11-A means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the IAV11-A.

```
status = LIO$ATTACH (iava_id, 'IVA0', LIO$K_QIO)
       IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The **iava_id** argument returns the LIO-assigned device ID for the IAV11-A device. The IAV11-A device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IVA0 specifies an IAV11-A device with controller letter A and unit number 0. Specifying the unit number is optional. However, if you do specify a unit number, it must always be 0.

NOTE

Each IXV11 device, whether it is an analog input, an analog output, a digital input, a digital output, or a counter device, is represented as a unique controller. Unit numbers are not used. For example, if you have more than one IXV11 device configured in your system, then the first IXV11 device is IVA0, the second IXV11 device is IVB0, and so on.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the IAV11-A devices.

[1] This device is available only in Europe.

2.4.1.2 Setting Up the IAV11-A

Before you can begin data transfers using the IAV11-A, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IAV11-A device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–15: IAV11-A LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_N_AD_CHAN	Returns the number of analog-to-digital channels currently in use.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.4.1.3 Using the IAV11-A for Synchronous Input

To use the IAV11-A for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IAV11-A device as described in Section 2.4.1.1, Attaching the IAV11-A.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (iava_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Read data from the IAV11-A device using the LIO$READ routine. The **device_specific** argument is an array of longwords that you use to specify control information about the A/D channels. Each longword specifies information about one channel. Byte 1 specifies the channel number. Byte 2 specifies the channel gain. Byte 3 specifies the trigger source. Byte 4 specifies the timeout in seconds.

   The **buffer** argument is an array of longwords returning information about the data I/O transfer. Each longword returns information about one channel. Byte 1 returns the channel number. Byte 2 returns the channel gain. The high word returns the actual A/D value read from the channel.

   Specify the **buffer_length** argument as a multiple of four.

```
STRUCTURE /chan_val/              !Set up structure of channel list
        BYTE chan                 !Byte to hold channel number
        BYTE gain                 !Byte to hold channel gain
        BYTE trigger              !Byte to hold trigger source
        BYTE timeout              !Byte to hold timeout in second
END STRUCTURE                     !End structure
RECORD /chan_val/ chan_list(10)   !The device-specific argument

STRUCTURE /buff_val/              !Set up structure of one A/D value
        BYTE chan                 !Byte to hold channel number
        BYTE gain                 !Byte to hold channel gain
        INTEGER*2 value           !Word to hold actual A/D value
END STRUCTURE                     !End structure
RECORD /buff_val/ buffer(10)      !Buffer of 10 A/D values

DO i=1,10                         !Loop to set up 10 A/D channels
     chan_list(i).chan = i        !Channel number
     chan_list(i).gain = 1        !Channel gain
     chan_list(i).trigger = 1     !External trigger
     chan_list(i).timeout = 255 !255 second timeout
ENDDO

status = LIO$READ (iava_id, buffer, 10 * 4, idata_len, chan_list)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

   The LIO$READ routine returns when data is read from the specified channels.

6. Detach the device.

```
status = LIO$DETACH (iava_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.1.4 Using the IAV11-A for Asynchronous Input

To use the IAV11-A for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IAV11-A device as described in Section 2.4.1.1, Attaching the IAV11-A.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (iava_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Enqueue a buffer to the device to start A/D conversions. The **device_specific** argument is an array of longwords that you use to specify control information about the A/D channels. Each longword specifies information about one channel. Byte 1 specifies the channel number. Byte 2 specifies the channel gain. Byte 3 specifies the trigger source. Byte 4 specifies the timeout in seconds.

   The **buffer** argument is an array of longwords that returns information about the data I/O transfer. Each longword returns information about one channel. Byte 1 returns the channel number. Byte 2 returns the channel gain. The high word returns the actual A/D value read from the channel.

   Specify the **buffer_length** argument as a multiple of four.

```
STRUCTURE /chan_val/            !Set up structure of channel list
        BYTE chan               !Byte to hold channel number
        BYTE gain               !Byte to hold channel gain
        BYTE trigger            !Byte to hold trigger source
        BYTE timeout            !Byte to hold timeout in second
END STRUCTURE                   !End structure
RECORD /chan_val/ chan_list(10) !The device-specific argument

STRUCTURE /buff_val/            !Set up structure of one A/D value
        BYTE chan               !Byte to hold channel number
        BYTE gain               !Byte to hold channel gain
        INTEGER*2 value         !Word to hold actual A/D value
END STRUCTURE                   !End structure
RECORD /buff_val/ buffer(10)    !Buffer of 10 A/D values

DO i=1,10                       !Loop to set up 10 A/D channels
     chan_list(i).chan = i      !Channel number
     chan_list(i).gain = 1      !Channel gain
     chan_list(i).trigger = 1   !External trigger
     chan_list(i).timeout = 255 !255 second timeout
ENDDO

status = LIO$ENQUEUE (iava_id, buffer, 10 * 4, , , , chan_list)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The LIO$ENQUEUE routine returns as soon as it enqueues a buffer to the IAV11-A device.

6. Dequeue the buffer using the LIO$DEQUEUE routine, or use one of the other buffer synchronization mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
7. Detach the device.

```
status = LIO$DETACH (iava_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.2 IAV11-B Support

The IAV11-B[1] is a four-channel group-isolated digital-to-analog converter with 12-bit resolution.

[1] This device is available only in Europe.

2.4.2.1 Attaching the IAV11-B

Attaching an IAV11-B means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the IAV11-B.

```
status = LIO$ATTACH (iavb_id, 'IVA0', LIO$K_QIO)
       IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The **iavb_id** argument returns the LIO-assigned device ID for the IAV11-B device. The IAV11-B device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IVA0 specifies an IAV11-B device with controller letter A and unit number 0. Specifying the unit number is optional. However, if you do specify a unit number, it must always be 0.

NOTE

Each IXV11 device, whether it is an analog input, an analog output, a digital input, a digital output, or a counter device, is represented as a unique controller. Unit numbers are not used. For example, if you have more than one IXV11 device configured in your system, then the first IXV11 device is IVA0, the second IXV11 device is IVB0, and so on.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the IAV11-B counter device.

2.4.2.2 Setting Up the IAV11-B

Before you can begin data transfers using the IAV11-B, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IAV11-B device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–16: IAV11-B LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.4.2.3 Using the IAV11-B for Synchronous Output

To use the IAV11-B for synchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IAV11-B device as described in Section 2.4.2.1, Attaching the IAV11-B.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (iavb_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Write data to the IAV11-B device using the LIO$WRITE routine. The **buffer** argument of the LIO$WRITE routine is a four-element array of longword integers. The lower 12 bits of each longword are used to write data to an output channel. If a longword contains a −1, then the respective output channel is not changed.

   The LIO$WRITE routine returns after data is written to all the requested output channels.

6. Detach the device.

```
status = LIO$DETACH (iavb_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.2.4 Using the IAV11-B for Asynchronous Output

To use the IAV11-B for asynchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IAV11-B device as described in Section 2.4.2.1, Attaching the IAV11-B.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (iavb_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Enqueue a buffer to the device to start D/A conversions. The **buffer** argument of the LIO$ENQUEUE routine is a four-element array of longword integers. The lower 12 bits of each longword are used to write data to an output channel. If a longword contains a −1, then the respective output channel is not changed.
6. Dequeue the buffer using the LIO$DEQUEUE routine, or use one of the other buffer synchronization mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
7. Detach the device.

```
status = LIO$DETACH (iavb_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.3 IDV11-A Support

The IDV11-A[1] is a 16-channel optically isolated digital input device. The IDV11-A offers programmable interrupt capability on one channel, programmable contact bounce elimination, and programmable input voltage range selection.

[1] This device is available only in Europe.

2.4.3.1 Attaching the IDV11-A

Attaching the IDV11-A means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the IDV11-A.

```
status = LIO$ATTACH (idva_id, 'IVA0', LIO$K_QIO)
       IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The **idva_id** argument returns the LIO-assigned device ID for the IDV11-A device. The IDV11-A device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IVA0 specifies an IDV11-A device with controller letter A and unit number 0. Specifying the unit number is optional. However, if you do specify a unit number, it must always be 0.

NOTE

Each IXV11 device, whether it is an analog input, an analog output, a digital input, a digital output, or a counter device, is represented as a unique controller. Unit numbers are not used. For example, if you have more than one IXV11 device configured in your system, then the first IXV11 device is IVA0, the second IXV11 device is IVB0, and so on.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the IDV11-A devices.

2.4.3.2 Setting Up the IDV11-A

Before you can begin data transfers using the IDV11-A, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IDV11-A device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–17: IDV11-A LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_BOUNCE	Sets the contact bounce elimination response time delay for the IDV11-A device.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on IDV11-A channel 15 events.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_POLARITY	Defines the interrupt to occur on either a positive-going or negative-going edge.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_VOLTAGE	Specifies the input voltage range for the IDV11-A device.

2.4.3.3 Using the IDV11-A for Synchronous Input

To use the IDV11-A for synchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-A device as described in Section 2.4.3.1, Attaching the IDV11-A.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idva_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Read data from the IDV11-A device using the LIO$READ routine.
6. Detach the device.

```
status = LIO$DETACH (idva_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.3.4 Using the IDV11-A for Asynchronous Input

To use the IAV11-A for asynchronous input, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-A device as described in Section 2.4.3.1, Attaching the IDV11-A.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (idva_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Enqueue a buffer to the device to start A/D conversions.
6. Dequeue the buffer using the LIO$DEQUEUE routine, or use one of the other buffer synchronization mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.
7. Detach the device.

```
status = LIO$DETACH (idva_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.4 IDV11-B and IDV11-C Support

The IDV11-B[1] is an isolated digital output device that provides 16 single optically isolated DC outputs.

The IDV11-C[1] is a relay output device that provides 16 latched reed contact output channels.

2.4.4.1 Attaching the IDV11-B

Attaching an IDV11-B means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the IDV11-B.

```
status = LIO$ATTACH (idvb_id, 'IVAO', LIO$K_QIO)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

[1] This device is available only in Europe.

The **idvb_id** argument returns the LIO-assigned device ID for the IDV11-B device. The IDV11-B device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IVA0 specifies an IDV11-B device with controller letter A and unit number 0. Specifying the unit number is optional. However, if you do specify a unit number, it must always be 0.

NOTE

Each IXV11 device, whether it is an analog input, an analog output, a digital input, a digital output, or a counter device, is represented as a unique controller. Unit numbers are not used. For example, if you have more than one IXV11 device configured in your system, then the first IXV11 device is IVA0, the second IXV11 device is IVB0, and so on.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the IDV11-B devices.

2.4.4.2 Setting Up the IDV11-B

Before you can begin data transfers using the IDV11-B, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IDV11-B device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–18: IDV11-B LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.4.4.3 Using the IDV11-B for Synchronous Output

To use the IDV11-B for synchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-B device as described in Section 2.4.4.1, Attaching the IDV11-B.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idvb_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Write data to the IDV11-B device using the LIO$WRITE routine. The LIO$WRITE routine returns after the data is written to all the requested output channels.
6. Detach the device.

```
status = LIO$DETACH (idvb_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.4.4 Using the IDV11-B for Asynchronous Output

To use the IDV11-B for asynchronous output, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-B device as described in Section 2.4.4.1, Attaching the IDV11-B.
4. Set up the device to use the asynchronous I/O interface.

```
status = LIO$SET_I (idvb_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Enqueue a buffer to the device to start D/A conversions.
6. Dequeue the buffer using the LIO$DEQUEUE routine, or use one of the other buffer synchronization mechanisms described in Section 1.5, Asynchronous I/O Buffer-Handling Mechanisms.

7. Detach the device.

```
status = LIO$DETACH (idvb_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.5 The IDV11-D Real-Time Counter Device

The IDV11-D[1] is a counter device consisting of five 16-bit counters, which can be programmed to increment or decrement the count. The count inputs can be controlled either by external signals or internally. The counter outputs generate interrupts that can be processed by a user-defined AST routine.

2.4.5.1 Attaching the IDV11-D

Attaching the IDV11-D means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device. Use the LIO$ATTACH routine to attach the IDV11-D.

```
status = LIO$ATTACH (idvd_id, 'IVA0', LIO$K_QIO)
       IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The **idvd_id** argument returns the LIO-assigned device ID for the IDV11-D device. The IDV11-D is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IVA0 specifies an IXV11 device with controller letter A and unit number 0. Specifying the unit number is optional. However, if you do specify a unit number, it must always be 0.

NOTE

Each IXV11 device, whether it is an analog input, an analog output, a digital input, a digital output, or a counter device, is represented as a unique controller. Unit numbers are not used. For example, if you have more than one IXV11 device configured in your system, then the first IXV11 device is IVA0, the second IXV11 device is IVB0, and so on.

The LIO$K_QIO value sets up the device to use QIOs. This is the only I/O type supported for the IDV11-D counter device.

[1] This device is available only in Europe.

2.4.5.2 Setting Up the IDV11-D

Before you can begin data transfers using the IDV11-D, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IDV11-D device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–19: IDV11-D LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O.
LIO$K_CC_FOUT	Sets the Frequency Output (FOUT) reference signal for the IDV11-D device.
LIO$K_CC_SETUP	Sets up the operating characteristics of one channel on the IDV11-D real-time counter device.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_START	Starts one or more of the five IDV11-D counter channels.
LIO$K_STOP	Stops one or more of the five IDV11-D counter channels.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.4.5.3 Using the IDV11-D to Count External Events

This section describes how to write a program that uses the IDV11-D counter to call an AST routine after the IDV11-D counts a defined number of external pulses.

In this example, counter channel 1 generates an interrupt after 5 pulses are counted on source 1. Counter channel 1 is initialized to count down, counting on the positive edge of source 1 with no gating or active low pulse. The precount is set to 5. The counter starts immediately on the positive edge of source 1.

The counter counts down four pulses: 4, 3, 2, 1. With the next positive edge of source 1, instead of counting down to 0 (not a legal value when counting down), the counter reloads the value of the load register (which is 5 in this example). Then, the IDV11-D generates an interrupt and calls your AST routine. The counter continues to run until it is explicitly stopped using the LIO$K_STOP parameter.

To count external events using the IDV11-D counter, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-D device as described in Section 2.4.5.1, Attaching the IDV11-D.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idvd_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the operating characteristics of counter channel 1. The following table shows the values of a longword integer array of length nine called **cc_array**. The **cc_array** argument contains the operating characteristics for counter channel 1. Use **cc_array** as an argument to the LIO$K_CC_SETUP parameter.

Index	Value	Function
1	1	Selects counter channel 1
2	5	Sets precount of 5 pulses
3	0	Sets countdown switch
4	0	Sets count on positive-edge switch
5	1	Sets immediate start
6	user_ast	Enter your AST routine address
7	user_param	Enter your AST routine parameter
8	1	Selects source 1
9	0	Selects no gating

The following code segment sets up the operating characteristics:

```
INTEGER*4 cc_array(9)    !Declare integer array of length 9

cc_array(1)=1            !Channel 1
cc_array(2)=5            !Precount 5 pulses
cc_array(3)=0            !Countdown
cc_array(4)=0            !Positive edge
cc_array(5)=1            !Immediate start
cc_array(6)=user_ast     !Enter your AST routine name
cc_array(7)=user_param   !Enter your AST routine parameter
cc_array(8)=1            !Source 1
cc_array(9)=0            !No gating
   .
   .
   .
status = LIO$SET_I (idvd_id, LIO$K_CC_SETUP, 1, %LOC(cc_array))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The counter begins counting immediately. When the counter reaches the fifth pulse, your AST routine is called.

6. Stop the counter. The following table shows the values of a longword integer array of length five called **cc_stop**. The **cc_stop** array contains the IDV11-D counter channels to stop. Use **cc_stop** as an argument value to the LIO$K_STOP parameter to stop the counter channels.

Index	Value	Function
1	0	Counter channel 0 not used
2	1	Stop counter channel 1
3	0	Channel 2 is not used
4	0	Channel 3 is not used
5	0	Channel 4 is not used

The following code segment sets up the IDV11-D counter channels to stop and stops them.

```
INTEGER*4 cc_stop(5)     !Declare integer array of length 5

cc_stop(1)=0             !Channel 0 not used
cc_stop(2)=1             !Stop Channel 1
cc_stop(3)=0             !Channel 2 not used
cc_stop(4)=0             !Channel 3 not used
cc_stop(5)=0             !Channel 4 not used
   .
   .
   .
status = LIO$SET_I (idvd_id, LIO$K_STOP, 1, %LOC(cc_stop))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

7. Detach the device.

```
status = LIO$DETACH (idvd_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.5.4 Using the IDV11-D to Measure Pulse Duration

This section describes how to use the IDV11-D to measure the duration of an external pulse.

This example uses two of the IDV11-D counters. The first counter is set up to count the internal reference clock (the 5 MHz clock) and is gated by the active high state of the external pulse. The external pulse is connected to gate 2. The second counter is set up to interrupt, thus calling your AST routine, on the negative edge of the external pulse.

To measure pulse duration using the IDV11-D, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-D device as described in Section 2.4.5.1, Attaching the IDV11-D.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idvd_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the operating characteristics of counter channel 1. The following table shows the values of a longword integer array of length nine called **setup1_array**. The **setup1_array** argument contains the operating characteristics for counter channel 1. Use **setup1_array** as an argument value to the LIO$K_CC_SETUP parameter.

Index	Value	Function
1	1	Selects counter channel 1
2	0	Sets precount
3	1	Sets count up switch
4	0	Sets count on positive-edge switch
5	0	Sets explicit start
6	0	No AST routine
7	0	No AST routine parameter
8	11	Selects 5 MHz clock source
9	2	Selects gate 2

The following code segment sets up the operating characteristics:

```
INTEGER*4 setup1_array(9)    !Declare integer array of length 9

setup1_array(1)=1            !Channel 1
setup1_array(2)=0            !Initial precount
setup1_array(3)=1            !Count up
setup1_array(4)=0            !Positive edge
setup1_array(5)=0            !Explicit start
setup1_array(6)=0            !No AST routine
setup1_array(7)=0            !No AST routine parameter
setup1_array(8)=11           !5 MHz clock source
setup1_array(9)=2            !Gate 2
    .
    .
    .
status = LIO$SET_I (idvd_id, LIO$K_CC_SETUP, 1, %LOC(setup1_array))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

When the LIO$SET_I routine returns, the device is set up, but is not started.

6. Set up the operating characteristics of counter channel 2. The following table shows the values of a longword integer array of length nine called **setup2_array**. The **setup2_array** argument contains the operating characteristics for counter channel 2. Use **setup2_array** as an argument value to the LIO$K_CC_SETUP parameter.

Index	Value	Function
1	2	Selects counter channel 2
2	1	Sets precount
3	0	Sets countdown switch
4	1	Sets count on negative-edge switch
5	0	Explicit start
6	0	No AST routine
7	0	No AST routine parameter
8	7	Selects Gate 2 source
9	0	No gating

The following code segment sets up the operating characteristics:

```
INTEGER*4 setup2_array(9)    !Declare integer array of length 9

setup2_array(1)=2            !Channel 2
setup2_array(2)=1            !Initial precount
setup2_array(3)=0            !Countdown
setup2_array(4)=1            !Negative edge
setup2_array(5)=0            !Explicit start
setup2_array(6)=0            !No AST routine
setup2_array(7)=0            !No AST routine parameter
setup2_array(8)=7            !Source = Gate 2
setup2_array(9)=0            !No gating
    .
    .
    .
status = LIO$SET_I (idvd_id, LIO$K_CC_SETUP, 1, %LOC(setup2_array))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

When the LIO$SET_I routine returns, the device is set up, but is not started.

7. Explicitly start both counters. The following table shows the values for a longword integer array of length five called **cc_start**. The **cc_start** array contains the IDV11-D counter channels to start. Use **cc_start** as an argument value to the LIO$K_START parameter to start the counter channels.

Index	Value	Function
1	0	Channel 0 not used
2	1	Starts counter channel 1
3	1	Starts counter channel 2
4	0	Channel 3 is not used
5	0	Channel 4 is not used

The following code segment sets up the IDV11-D counter channels to start and starts them.

```
INTEGER*4 cc_start(5)     !Declare integer array of length 5

cc_start(1)=0             !Channel 0 not used
cc_start(2)=1             !Start channel 1
cc_start(3)=1             !Start channel 2
cc_start(4)=0             !Channel 3 not used
cc_start(5)=0             !Channel 4 not used
    .
    .
    .
status = LIO$SET_I (idvd_id, LIO$K_START, 1, %LOC(cc_start))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

8. After counter 2 generates an interrupt and calls your AST routine, read the contents of counter 1. Divide the value returned by 5 MHz to determine the duration of the external pulse. Use the **device_specific** argument to specify which counter to read.

```
BYTE           device_specific(4)
INTEGER*2      chan_number
BYTE           disarm, save

EQUIVALENCE (device_specific(1), chan_number)
EQUIVALENCE (device_specific(3), disarm)
EQUIVALENCE (device_specific(4), save)

chan_number = 1

disarm = 1                ! channel will be disarmed (stopped), with
save   = 1                ! LIO$READ content of the counter saved in
                          ! the hold register
```

```
 status = LIO$READ (idvd_id, buffer, buffer_length, data_length,
1            device_specific)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Detach the device.

```
status = LIO$DETACH (idvd_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.5.5 Using the IDV11-D to Generate Pulse Trains

This section describes how to use the IDV11-D to generate output pulses at specified time intervals.

This example uses counter channel 3 to generate an active low output pulse every 0.5 milliseconds. The pulse width output is defined by the period of the counting source. This example uses the 5-MHz internal reference frequency as the counting source.

To generate pulse trains using the IDV11-D, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the IDV11-D device as described in Section 2.4.5.1, Attaching the IDV11-D.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idvd_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the operating characteristics of counter channel 3. The following table shows the values of a longword integer array of length nine called **setup3_array**. The **setup3_array** argument contains the operating characteristics for counter channel 3. Use **setup3_array** as an argument value to the LIO$K_CC_SETUP parameter.

Index	Value	Function
1	3	Selects counter channel 3
2	2500	Sets precount for 2500-pulse interval
3	0	Sets countdown switch
4	0	Sets count on positive-edge switch
5	1	Sets immediate start

Index	Value	Function
6	0	No AST routine
7	0	No AST routine parameter
8	11	Selects 5 MHz clock source
9	0	Selects no gating

The following code segment sets up the operating characteristics:

```
INTEGER*4 setup3_array(9)    !Declare integer array of length 9

setup3_array(1)=3            !Channel 3
setup3_array(2)=2500         !Precount for 2500 pulse interval
setup3_array(3)=0            !Countdown
setup3_array(4)=0            !Positive edge
setup3_array(5)=1            !Immediate start
setup3_array(6)=0            !No AST routine
setup3_array(7)=0            !No AST routine parameter
setup3_array(8)=11           !5-MHz clock source
setup3_array(9)=0            !No gating
    .
    .
    .
status = LIO$SET_I (idvd_id, LIO$K_CC_SETUP, 1, %LOC(setup3_array))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

The counter begins to generate pulses immediately.

6. Detach the device.

```
status = LIO$DETACH (idvd_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.4.5.6 Using the IDV11-D to Generate Output Frequencies

This section describes how to use the IDV11-D to generate output frequencies.

This example sets up the frequency output signal to provide a quartz-controlled output of 5 MHz divided by 256 by using one of the internal reference frequencies.

To generate output frequencies using the IDV11-D, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.

3. Attach the IDV11-D device as described in Section 2.4.5.1, Attaching the IDV11-D.
4. Set up the device to use the synchronous I/O interface.

```
status = LIO$SET_I (idvd_id, LIO$K_SYNCH, 0)
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

5. Set up the parameters for the frequency output (FOUT) reference signal. The following table shows the values of a longword integer array of length three called **cc_fout**. The **cc_fout** argument contains the parameters for the frequency output reference signal. Use **cc_fout** as an argument value to the LIO$K_CC_FOUT parameter.

Index	Value	Function
1	0	Turn FOUT on switch
2	13	Source = 5 MHz/256
3	1	Divide by 1

The following code segment sets up the FOUT parameters:

```
INTEGER*4 cc_fout(3)    !Declare integer array of length 3

cc_fout(1)=0              !Turn FOUT on
cc_fout(2)=13             !Source = 5 MHz/256
cc_fout(3)=1              !Divide by 1
    .
    .
    .
status = LIO$SET_I (idvd_id, LIO$K_CC_FOUT, 1, %LOC(cc_fout))
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

6. Detach the device.

```
status = LIO$DETACH (idvd_id, )
         IF (.NOT. status) CALL LIB$SIGNAL (%VAL(status))
```

2.5 IEEE-488 Bus Devices

LIO provides support for an IEEE-488 device as:

- The system controller
- A controller
- An instrument

The role of the IEEE-488 device is defined when the device is attached.

LIO provides support for two Digital IEEE-488 devices—the IEQ11 and the IEZ11—and for the IOtech Micro488A.

2.5.1 IEQ11 and IEZ11

The IEQ11 contains two units, each of which can interface up to 14 instruments. Each unit can be attached as a separate LIO device.

The VMS and LIO device name for the IEQ11 is IX. The first unit on the first IEQ11 device is IXA0 and the second unit is IXA1. The first unit on the second IEQ11 device is IXB0, and so on.

The IEZ11 allows up to 14 instruments to be controlled from the external SCSI port of a MicroVAX 3100 series or VAXstation 3100 series system.

The VMS and LIO device name for the IEZ11 is EK. On the VAXstation 3100 Model 30, the device should be attached as EKA0 or EKA1. On the VAXstation 3100 Model 40 and the MicroVAX 3100, the device should be attached as EKB0 or EKB1, on the "B" (external) SCSI controller.

The IEQ11 and the IEZ11 are similar devices and have nearly identical functionality under LIO.

For more information about the IEQ11, see the *IEU11-A/IEQ11-A User's Guide*.

For more information about the IEZ11, see the *IEZ11 Software Installation Guide*, the *IEZ11 Hardware Installation Guide*, and the *DEC IEZ11 VMS Class Driver User's Guide*.

2.5.2 IOtech Micro488A

The IOtech Micro488A bus controller is an RS232 to IEEE-488 converter. The Micro488A supports synchronous I/O only.

The Micro488A can be used to add IEEE-488 functionality to any VSL-supported processor with a serial line interface.

The LIO device name for the Micro488A is IT. This name tells LIO the type of device to attach, but does not specify which physical port will be used.

The physical device name defaults to TTA2, because this is the serial port on the VAXstation 3100. To override this assignment, you must define the logical name IOTECH_PORT as the desired physical port. For example, the following logical name defines the physical port as a serial line on a DECserver.

```
$ DEFINE IOTECH_PORT LTA5:
```

The Micro488A functionality under LIO is a subset of the IEQ11 and IEZ11 functionality. The individual LIO$SET and LIO$SHOW parameters document the differences and limitations.

NOTE

You cannot use the IOtech Micro488A to transfer raw binary data.

For more information about the Micro488A, see the documentation from IOtech, Inc.

2.5.2.1 IOtech Micro488A DIP Switch Settings

You must set certain switches to use the Micro488A with LIO, as shown in the following list:

- Serial baud rate: SW1-1 through SW1-4. For LIO, the range is from 110 to 19200, to be determined by the user. The typical setting is 9600. You also need to "SET TERM/PERM/SPEED = baud rate" from the DCL prompt.
- Serial handshake selection: SW1-5 open (for XON/XOFF software control).
- Serial word length selection: SW1-6 closed (for 8 bits). You also need to "SET PORT/PERM/EIGHT" from the DCL prompt.
- Controller pass-thru selection: SW1-7. Controller pass-thru mode is not supported for LIO, so this setting does not matter.
- Serial stop bit selection: SW1-8 closed (for 1 bit).
- Serial echo selection: SW2-5 closed (for echo disabled).
- Serial parity selection: SW2-6 closed (for parity disabled).

- Mode selection: SW2-1 and SW2-2 closed for system controller mode; SW2-1 open and SW2-2 closed for peripheral mode. LIO supports system controller and peripheral modes only.
- IEEE address selection: SW3-1 through SW3-5. The address must be unique.

2.5.3 An IEEE-488 Device as the System Controller

An IEEE-488 device can be attached as the IEEE-488 bus system controller. An IEEE-488 device attached as the system controller is the device that is the controller-in-charge of the IEEE-488 bus when your application program first attaches the device.

NOTE

Only one device on the IEEE-488 bus can be the system controller. If a device is already functioning as the system controller, do not attach an IEEE-488 device as the system controller.

The system controller can pass control (controller-in-charge status) to any other device that is attached as a controller. The system controller can resume control of the IEEE-488 bus by having control passed back to it, or by sending the "interface clear" request.

Only the controller-in-charge of the IEEE-488 bus is able to perform certain functions:

- Respond to service requests from devices on the bus
- Send commands to devices on the bus
- Parallel poll the devices on the bus
- Serial poll the devices on the bus

2.5.4 An IEEE-488 Device as a Controller

An IEEE-488 device attached as a controller is initially not the controller-in-charge. This means that the device cannot perform the controller-in-charge functions described in Section 2.5.3, An IEEE-488 Device as the System Controller, until it becomes the controller-in-charge.

An IEEE-488 device becomes a controller-in-charge when the current controller-in-charge passes control to it. The controller remains the controller-in-charge until it passes control to another controller or until the system controller resumes control with the "interface clear" request.

2.5.5 An IEEE-488 Device as an Instrument

An instrument on the IEEE-488 bus is a slave to the controller-in-charge. An instrument sending a message is a talker. Only one instrument on the bus may talk at any one time. An instrument receiving a message is a listener. Any number of instruments can listen to a message being sent by the talker. In general, some IEEE-488 instruments can be talkers only, listeners only, or both talkers and listeners. An application program can use an IEEE-488 device as a talker only, as a listener only, or as both a talker and a listener.

An instrument talks when the controller-in-charge addresses it to talk, and it listens when the controller-in-charge addresses it to listen. The only independent action an instrument can take is to request service (SRQ). An IEEE-488 device on the IEEE-488 bus acts as an instrument unless it is the controller-in-charge.

2.5.6 IOtech Micro488A Device Modes

The IOtech Micro488A supports the system controller, controller, and instrument modes discussed above, but with one major distinction: You enable or disable the modes by setting a DIP switch in the device to either System Controller or Peripheral mode.

By default the System Controller mode is enabled, which means that the Micro488A can be attached as a controller or as the system controller. This is fine for most applications since controllers can behave as instruments.

In some applications, however, you may want to use the Micro488A as an instrument (as an add-on to existing IEEE-488 buses, for example). In this case you should set the DIP switch to Peripheral mode and attach the Micro488A as an instrument. See Section 2.5.2.1, IOtech Micro488A DIP Switch Settings, for more information.

2.5.7 Attaching an IEEE-488 Device

Attaching an IEEE-488 device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device. The role of the IEEE-488 device is defined when the device is attached.

Use the LIO$ATTACH routine to attach an IX (IEQ11) or EK (IEZ11) device.

```
status = LIO$ATTACH (ieee_id, 'IXA0', LIO$K_SYS_CTRL) !System controller
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ATTACH (ieee_id, 'IXA1', LIO$K_CTRL)     !Non-system Controller
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ATTACH (ieee_id, 'IXB0', LIO$K_INSTRUMENT) !Instrument
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$ATTACH (ieee_id, 'EKB0', LIO$K_SYS_CTRL) !System controller
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **ieee_id** argument returns the LIO-assigned device ID for the device. The device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification IXA0 specifies an IEQ11 (IX) device, with controller letter A and unit number 0. To attach another IX device, specify unit number 1.

The IEQ11 supports two units per device. Each device must have a unique device specification. The devices can be attached in any order.

The device specification EKB0 specifies an IEZ11 (EK) device, with controller letter B and unit number 0.

The IEZ11 supports one unit per device, attached as unit 0 or 1.

The I/O type LIO$K_SYS_CTRL sets up an IEEE-488 device as the system controller. This is the default value. The I/O type LIO$K_CTRL sets up an IEEE-488 device as a controller. The I/O type LIO$K_INSTRUMENT sets up an IEEE-488 device as an instrument.

The IT (IOtech Micro488A) device is attached the same way as the IX and EK devices, but the controller and unit designation have no meaning. To use a port other than TTA2 you must define the logical name "IOTECH_PORT" to be the physical device name. See

Section 2.5.2, IOtech Micro488A, for an example of defining the logical name.

The controller function for the IT device is selected by setting DIP switches in the device. See Section 2.5.2.1, IOtech Micro488A DIP Switch Settings, for more information.

See the description of the LIO$ATTACH routine in Chapter 3 for more information about supplying device specifications and I/O types to the LIO$ATTACH routine.

2.5.8 Setting Up the IEEE-488 Device

Before you can begin using an IEEE-488 device, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show IEEE-488 device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–20: IEEE-488 Device LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ASYNCH	Sets up a device to use the asynchronous I/O interface. (LIO$K_ASYNCH is not supported for the IOtech Micro488A.)
LIO$K_AUX_COMMAND	Sends an auxiliary command to an IEEE-488 device.
LIO$K_COMMAND	Sends the specified IEEE-488 commands on the bus.
LIO$K_CTRL_ACTIVE	Activates the IEEE-488 controller function.
LIO$K_CTRL_STANDBY	Deactivates the IEEE-488 controller function.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_EOI	Asserts or does not assert the end-or-identify (EOI) line when the last byte of data is output.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.

Table 2–20 (Cont.): IEEE-488 Device LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on IEEE-488 bus events. (LIO$K_EVENT_AST is not supported for the IOtech Micro488A.)
LIO$K_EVENT_ENA	Enables the recognition of specified IEEE-488 bus events.
LIO$K_EVENT_WAIT	Waits for enabled IEEE-488 bus events to occur.
LIO$K_IEEE_ADDR	Sets the IEEE-488 bus address of the device, and enables or disables the recognition of secondary addressing. (Note that the IEZ11 and the IOtech Micro488A cannot be addressed as a secondary listener or talker.)
LIO$K_LEAVE_IN_STATE	Specifies whether or not to leave an IEQ11 device in the state required to process the subsequent I/O request. (LIO$K_LEAVE_IN_STATE is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_PAR_POLL	Performs a parallel poll of IEEE-488 bus instruments.
LIO$K_PAR_POLL_CONFIG	Sets up the list of IEEE-488 instruments for parallel polling.
LIO$K_PAR_POLL_STATUS	Sets up an instrument's parallel poll status register.
LIO$K_PASS_CTRL	Passes control to another IEEE-488 bus device.
LIO$K_SER_POLL	Serial polls a predetermined list of IEEE-488 instruments.
LIO$K_SER_POLL_CONFIG	Sets up the list of IEEE-488 instruments to serial poll.
LIO$K_SRQ	Defines an IEEE-488 device's serial poll status byte and, optionally, sends a service request to the controller-in-charge.

Table 2–20 (Cont.): IEEE-488 Device LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TERM_CHAR	Defines a termination character to mark the end of a data transfer.
LIO$K_TERM_SRQ	Enables or disables termination of I/O transfers by a service request. (LIO$K_TERM_SRQ is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.

2.5.9 Assigning IEEE-488 Bus Addresses

The LIO facility recognizes the location of a device on the IEEE-488 bus by its address. **Immediately after you attach an IEEE-488 device, you must assign it a primary IEEE-488 bus address before you set up any other device characteristics.** A device does not actually exist on the IEEE-488 bus until it is assigned a primary address.

You also have the option of enabling a device to recognize secondary addresses.

NOTE

An IEZ11 or an IOtech Micro488A device cannot be addressed as a secondary listener or a secondary talker. However, an IEZ11 or a Micro488A can generate secondary addresses when it is controller-in-charge.

Use the LIO$K_IEEE_ADDR parameter to assign the device's IEEE-488 primary bus address, and to enable the recognition of secondary addressing, for example:

```
status = LIO$SET_I (ieee_id, LIO$K_IEEE_ADDR, 2, 0, LIO$K_ON)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The LIO$K_IEEE_ADDR constant is the LIO$SET_I parameter being set.

The 2 tells LIO that you are specifying two parameter values for the LIO$K_IEEE_ADDR parameter.

The 0 assigns IEEE-488 bus address zero as the primary address of this device.

The LIO$K_ON value enables the recognition of secondary addressing.

See the description of the LIO$SET_I routine in Chapter 3 for more information about specifying the appropriate arguments for this routine.

See the description of the LIO$K_IEEE_ADDR parameter in Chapter 4 for the acceptable values of this parameter.

NOTE

In addition to using LIO$K_IEEE_ADDR, you must also set DIP switches on the IOtech Micro488A device to assign the bus address. See Section 2.5.2.1, IOtech Micro488A DIP Switch Settings, for more information.

2.5.10 Enabling IEEE-488 Events

When a device is the controller-in-charge, a service request (SRQ) is an event.

When a device is not the controller-in-charge, IEEE-488 commands received from the controller-in-charge are treated as events by the device set up to respond to them.

To detect events on the IEEE-488 bus, the desired event types must be enabled with LIO$SET_I. Use the set parameter LIO$K_EVENT_ENA to enable specific events. The LIO$K_EVENT_ENA parameter values specify the bus events for as shown in the following table.

Value	Meaning
LIO$K_DEADDR_EVT	The device has been deaddressed. This event is detectable only when the device is not the controller-in-charge.
LIO$K_DEV_CLR_EVT	The controller-in-charge has sent the "device clear" command. The instrument should reset itself to its power-up state. Remember that user-written application programs are responsible for all instrument functions. The instrument should return to its initial state. This event is detectable only when the device is not the controller-in-charge.
LIO$K_DEV_TRIG_EVT	The controller-in-charge has sent the "device trigger" command. The instrument should trigger as specified in the user-written application program. This event is detectable only when the device is not the controller-in-charge.
LIO$K_EXT_LNR_EVT	The controller-in-charge is addressing the device as an extended (secondary) listener. This event is detectable only when the device is not the controller-in-charge. (LIO$K_EXT_LNR_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_EXT_TKR_EVT	The controller-in-charge is addressing the device as an extended (secondary) talker. This event is detectable only when the device is not the controller-in-charge. (LIO$K_EXT_TKR_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_IFC_EVT	The system controller is signalling the device to clear its bus interface. This does not generally affect the internal state of the instrument. (LIO$K_IFC_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_LNR_ADDR_EVT	The controller-in-charge is addressing the device as a listener. This event is detectable only when the device is not the controller-in-charge.

Value	Meaning
LIO$K_PAR_POLL_CONFIG_EVT	The controller-in-charge is signalling the device to configure itself for parallel polling. This event is detectable only when the device is not the controller-in-charge. (LIO$K_PAR_POLL_CONFIG_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_PAR_POLL_UNCONFIG_EVT	The controller-in-charge is signalling the device to unconfigure itself for parallel polling. This event is detectable only when the device is not the controller-in-charge. (LIO$K_PAR_POLL_UNCONFIG_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_REC_CTRL_EVT	The device has received control from the current controller-in-charge. This event is detectable only when the device is not the controller-in-charge, and it is attached as controller or system controller.
LIO$K_REM_LOCAL_EVT	The device state has changed from remote to local, or from local to remote. The current state of the device is returned by the LIO$K_EVENT_WAIT parameter, or by the AST routine set up by the LIO$K_EVENT_AST parameter. This event is detectable only when the device is not the controller-in-charge. (LIO$K_REM_LOCAL_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_SRQ_EVT	A device is requesting service. This event is detectable only when the device is the controller-in-charge.
LIO$K_TKR_ADDR_EVT	The controller-in-charge is addressing the device as a talker. For this event to be detectable, the device must be the controller-in-charge.

Multiple events can be specified in one call, for example:

```
 status = LIO$SET_I (ieee_id, LIO$K_EVENT_ENA, 2, LIO$K_PAR_POLL_CONFIG_EVT,
1        LIO$K_TKR_ADDR_EVT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This routine enables an IEEE-488 device to respond to a "parallel poll configure" event or an "addressed as talker" event. This means that the device detects that the controller-in-charge is performing a parallel poll configuration or is addressing this device as a talker.

See the description of the LIO$K_EVENT_ENA parameter in Chapter 4 for more information.

2.5.11 Detecting IEEE-488 Bus Events

The LIO facility supports the following two methods of detecting the occurrence of IEEE-488 bus events. Before an event can be detected, it must be enabled by the LIO$K_EVENT_ENA parameter.

- You can set up an IEEE-488 device with a user-written AST routine (LIO$K_EVENT_AST) that is called when an enabled event occurs, for example:

  ```
  status = LIO$SET_I (ieee_id, LIO$K_EVENT_AST, 1, event_ast)
           IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
  ```

  In this example, the LIO facility calls the user-written AST routine, **event_ast**, when an enabled event occurs. Your AST routine should take whatever action is appropriate for the device on this event. See Section 2.5.12, Supplying AST Routines, for information about supplying user-written AST routines.

- You can call the LIO$SHOW routine for an IEEE-488 device with the LIO$K_EVENT_WAIT parameter. This routine waits for an enabled event to occur, and then returns it, for example:

  ```
  INTEGER*4 event(2)
  INTEGER*4 length
      .
      .
      .
  LIO$SHOW (ieee_id, LIO$K_EVENT_WAIT, event, length)
  ```

This routine segment declares the variable **event** to be an integer array of length 2, and the variable **length** to be an integer. The IEEE-488 bus event and event-specific information are returned in the elements of the **event** array. The **length** variable returns the number of elements contained in the **event** array. The first element in the event array returns the IEEE-488 event code and the second element returns event specific information.

The nature of your application determines which method is more appropriate.

See the descriptions of the LIO$K_EVENT_AST and LIO$K_EVENT_WAIT parameters in Chapter 4 for more information about setting up a device to wait for IEEE-488 bus events.

2.5.12 Supplying AST Routines

You can supply the following two types of AST routines when using IEQ11 or IEZ11 devices:

- A buffer completion AST routine. A buffer completion AST routine is a normal user-written subroutine that the LIO facility calls to receive completed buffers from a device, usually for processing. When a device finishes a buffer transaction, it calls the AST routine and passes the buffer to it.

  Section 1.5.3, Asynchronous System Traps (ASTs), describes buffer completion AST routines in more detail. Also see Chapter 3, Program Development, in *Getting Started with VAXlab* for information about writing buffer completion AST routines that is specific to certain programming languages.

- An event AST routine. Here an event AST routine is a normal user-written subroutine that the LIO facility calls when it detects the occurrence of an IEEE-488 bus event. In order to detect an event, the detection of the event must be enabled through the LIO$K_EVENT_ENA parameter.

NOTE

You cannot use AST routines with the IOtech Micro488A device.

2.5.12.1 Example

The following VAX FORTRAN program segment shows how to set up an AST routine for an IEQ11 or IEZ11 device to detect the occurrence of IEEE-488 bus events.

```
    .
    .
    .
C Supply the address of the event AST routine
status = LIO$SET_I (ieee_id, LIO$K_EVENT_AST, 1, event_ast)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
C The device event AST routine takes four argument values.
C The first value, event_code, returns the code of the event
C that was detected. The second value, event_specific, returns
C information that is specific to the detected event. (For the
C IEZ11, this information is always 0.) The third value, unit,
C specifies the unit number of the device. The fourth value,
C controller, is the controller designation.

SUBROUTINE event_ast (event_code, event_specific, unit, controller)

         INTEGER*4 event_code     !IEEE-488 bus event code
         INTEGER*4 event_specific !Information specific to the event
         INTEGER*4 unit           !Unit number of the device that
                                  !detected the event
         INTEGER*4 controller     !Controller designation

C Body of your event AST routine goes here.  Write the routine to
C perform whatever tasks are appropriate for handling the events
C that might occur.
    .
    .
    .
          RETURN

          END
```

The **event_code** argument returns the event code of the detected event. See the description of the LIO$K_EVENT_ENA parameter for a list and descriptions of valid IEEE-488 bus event codes.

The **event_specific** argument returns information specific to the detected event according to the following general rules:

- For the IEZ11, the **event_specific** argument returns a 0.
- If the IEQ11 is addressed as extended listener or talker, the **event_specific** argument returns the device's secondary address. In this case, the value of the **event_code** argument is either LIO$K_LNR_ADDR_EVT or LIO$K_TKR_ADDR_EVT.
- If a remote/local change occurred, the **event_specific** argument returns a 0 if the new state is local mode. The **event_specific** argument returns a 1 if the new state is remote mode. In this case, the value of the **event_code** argument is LIO$K_REM_LOCAL_EVT.
- If a parallel poll configure occurred, the **event_specific** argument returns the PPE (parallel poll enable) byte or the PPD (parallel poll disable) byte. In this case, the value of the **event_code** argument is either LIO$K_PAR_POLL_CONFIG_EVT or LIO$K_PAR_POLL_UNCONFIG_EVT.
- For all other events, the **event_specific** argument returns a 0.

The **unit** argument is the number of the device that detected the event (for example, unit = 0 for IXA0 or unit = 1 for IXA1).

The **controller** argument is a single ASCII code containing the controller designation (''A'' for IXA0 or ''B'' for IXB0).

2.5.13 Requesting Service with an SRQ

To request service from the controller-in-charge, use the LIO$K_SRQ parameter, for example:

```
INTEGER*1 status_byte  !SRQ status byte
    .
    .
    .
status_byte = 64       !Bit 6 set, all other bits clear
status = LIO$SET_I (ieee_id, LIO$K_SRQ, 1, status_byte)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This routine segment declares variable **status_byte** as a one-byte integer and sets bit 6. When bit 6 is set in the SRQ status byte, a service request (SRQ) is sent to the controller-in-charge. The routine call waits until the controller-in-charge serially polls the device. If bit 6 is cleared in the SRQ status byte, the status is saved in the device, and no SRQ is sent to the controller-in-charge. The status is read by the controller-in-charge if it polls the device.

See the description of the LIO$K_SRQ parameter in Chapter 4 for more information.

2.5.14 Passing and Receiving Control

To pass control to another IEEE-488 device attached as a controller, use the LIO$K_PASS_CTRL parameter to specify the address of the instrument to which to pass control, for example:

```
status = LIO$SET_I (ieee_id, LIO$K_PASS_CTRL, 1, 6)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This routine passes control from the current controller-in-charge to the device at IEEE-488 bus address 6.

To receive control from the current controller-in-charge, an IEEE-488 device must be attached as a controller and must be set up with the receive control event (LIO$K_REC_CTRL) recognition enabled by the LIO$K_EVENT_ENA parameter, for example:

```
status = LIO$SET_I (ieee_id, LIO$K_EVENT_ENA, 1, LIO$K_REC_CTRL)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

Then, the device can either:

- Supply a user-written AST routine with the LIO$K_EVENT_AST parameter to act on the received control when the event occurs. (The advantage of this approach is that other processing can take place while waiting for control.)
- Call LIO$SHOW with the LIO$K_EVENT_WAIT parameter to wait until the controller-in-charge passes control to the device. (This is acceptable if the controller does not have any other functions it needs to perform while waiting.)

2.5.15 Responding to a Service Request

The controller-in-charge responds to a service request from an IEEE-488 instrument by performing a serial poll of all instruments that might be requesting service.

To set your program to respond to service requests, do the following:

1. Use the LIO$K_SER_POLL_CONFIG parameter to set up a list of instrument addresses to poll.

```
INTEGER addr1(4)    !Integer array of length 4 to hold primary addresses
INTEGER addr2(4)    !Integer array of length 4 to hold secondary addresses

addr1(1) = 2
addr1(2) = 4
addr1(3) = 6
addr1(4) = 8

addr2(1) = 142
addr2(2) = 144
addr2(3) = 146
addr2(4) = 148

status = LIO$SET_I (ieee_id, LIO$K_SER_POLL_CONFIG, 2, addr1, addr2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine sets up the devices at IEEE-488 bus addresses 2, 4, 6, and 8 for serial polling. If the list of devices you initially configure for serial polling does not need to change during the execution of your application, then you should use this setup parameter only once.

   If you need to serial poll other devices during the execution of your application, you must use this setup parameter again, specifying the IEEE-488 bus addresses of the other devices you need to poll.

2. Enable the service request (LIO$K_SRQ_EVT) event with the LIO$K_EVENT_ENA parameter.

```
status = LIO$SET_I (ieee_id, LIO$K_EVENT_ENA, 1, LIO$K_SRQ_EVT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This enables the controller-in-charge to recognize service requests from other devices.

3. Supply an event AST routine (LIO$K_EVENT_AST) to respond to the service request (LIO$K_SRQ) event.

```
status = LIO$SET_I (ieee_id, LIO$K_EVENT_AST 1, srq_ast)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

See the description of the LIO$K_EVENT_AST parameter in Section 2.5.12, Supplying AST Routines, and Chapter 4 for information about the device-specific information you must supply when you call the AST routine from your user program.

4. When the service request event occurs, your AST routine can call the LIO$SHOW routine with the LIO$K_SER_POLL parameter.

```
BYTE serial_poll_status(4)
INTEGER*4 length
    .
    .
    .
LIO$SHOW (ieee_id, LIO$K_SER_POLL, serial_poll_status, length)
```

This routine segment declares the variable **serial_poll_status** to be an integer array of length four, and the variable **length** to be an integer. The status bytes from the four devices polled (as set up with LIO$K_SER_POLL_CONFIG in step 1 of this procedure) are returned in the elements of the **serial_poll_status** array. For example, **serial_poll_status(1)** returns the status byte from the instrument at IEEE-488 bus address 2, **serial_poll_status(2)** returns the status byte from the instrument at IEEE-488 bus address 4, and so on. The **length** variable returns the number of status bytes contained in the **serial_poll_status** array.

For each instrument whose status byte has bit 6 set (requesting service), your AST routine should perform whatever action is appropriate at that time for the instrument requesting service.

NOTE

The actual response to a service request depends entirely on the instrument. The value of the other bits in the instrument's status byte can determine the appropriate action. See the description of the LIO$K_SRQ parameter in Chapter 4 for information about setting up IEEE-488 instruments to issue service requests. The user's manuals for IEEE-488 instruments usually contain information about the condition that causes these instruments to request service.

2.5.16 Sending Data and Receiving Data When the IEEE-488 Device Is Controller-In-Charge

When an IEEE-488 device is the controller-in-charge, it can send data to or receive data from an instrument.

To set up the device that is the controller-in-charge to send and receive data, do the following:

1. Call the appropriate LIO routine to make a buffer available to the controller-in-charge. (Call LIO$READ or LIO$WRITE if the device is set for synchronous I/O. Call LIO$ENQUEUE if the device is set for asynchronous I/O.)

   If the controller-in-charge will be sending data to an instrument, specify "LIO$M_LNR .OR. instrument_address" as the **device_specific** argument of the routine call, because the instrument will be a "listener".

   If the controller-in-charge will be receiving data from an instrument, specify "LIO$M_TKR .OR. instrument_address" as the **device_specific** argument of the routine call, because the instrument will be a "talker".

```
INTEGER*2 buffer(256)
INTEGER*4 event_flag
    .
    .
    .
LIB$GET_EF (event_flag)
LIO$SET_I (ctrl_id, LIO$K_DEVICE_EF, 1, event_flag)
    .
    .
    .
LIO$ENQUEUE (ctrl_id, buffer, 512, , event_flag, LIO$M_TKR .OR. 3)
    .
    .
    .
```

   This routine segment declares the variable buffer to be a word array of length 256, or 512 bytes. The **event_flag** variable is an integer and contains a system-assigned VMS event flag associated with the buffer. LIO$M_TKR .OR. 3 is a device-specific argument that signals the IEEE-488 instrument, at IEEE-488 bus address 3 (.OR. 3), to "talk" to the controller-in-charge.

When the buffer is enqueued (LIO$ENQUEUE), the IEEE-488 device tells the addressed instrument to "talk," and places the data sent by the addressed instrument in the enqueued buffer. The data transfer is complete when one of the following occurs:

- The buffer is filled.
- The instrument asserts the EOI line when the last byte of data is output. (See the description of the LIO$K_EOI parameter in Chapter 4.)
- The termination character is received. (See Section 2.5.18.1, Using Termination Characters to Terminate Read Requests, and the description of the LIO$K_TERM_CHAR parameter in Chapter 4.)

See the description of the LIO$ENQUEUE routine in Chapter 3 for more information about supplying arguments to this routine.

2. Call the LIO$DEQUEUE routine to return the buffer to the calling program, for example:

```
INTEGER*4 buffer_address
INTEGER*4 buffer_length
INTEGER*4 device_specific
   .
   .
   .
status = LIO$DEQUEUE (ctrl_id, buffer_address, buffer_length,
1            data_length, 1, , device_specific)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The LIO$DEQUEUE routine waits for the buffer to become available on the device's device queue and then returns the buffer to the controller-in-charge.

The **data_length** argument returns the number of bytes in the buffer. The value of **status** specifies the reason for terminating the buffer (for example, success, encountered termination character, or any error status code).

See the description of the LIO$DEQUEUE routine in Chapter 3 for more information about supplying arguments to this routine.

2.5.17 Sending Data to Multiple IEEE-488 Devices

The IEEE-488 bus allows more than one device to listen simultaneously. The LIO$K_COMMAND set parameter can send several listener addresses out on the bus.

To address multiple devices to listen simultaneously, do the following:

1. Attach the IEEE-488 device as a system controller. This ensures that it will be controller-in-charge. (It could also have been attached as a controller and waited to receive control.)

```
INTEGER*4 addr1, addr2, addr3
INTEGER*4 com1, com2, com3

status = LIO$ATTACH (ieee_id, 'IXAO', LIO$K_SYS_CTRL)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
```

2. Build listener addresses for IEEE-488 devices 2, 4, and 6 by ORing the LIO$M_LNR bit to each address.

```
addr1 = 2
com1 = LIO$M_LNR .OR. addr1

addr2 = 4
com2 = LIO$M_LNR .OR. addr2

addr3 = 6
com3 = LIO$M_LNR .OR. addr3
    .
    .
    .
```

3. Send the listener addresses out on the bus. This causes devices 2, 4, and 6 to listen.

```
status = LIO$SET_I (ieee_id, LIO$K_COMMAND, 3, com1, com2, com3)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
```

4. Write data out on the bus. Since devices 2, 4, and 6 are enabled to listen, they should receive this data.

```
status = LIO$WRITE (ieee_id, buffer, data_length)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
```

To send data asynchronously to multiple IEEE-488 devices, do the following:

1. Attach the IEEE-488 device as a system controller. This ensures that it will be controller-in-charge. (It could also have been attached as a controller and waited to receive control.)

```
INTEGER*4 addr1, addr2, addr3
INTEGER*4 com1, com2

status = LIO$ATTACH (ieee_id, 'IXA0', LIO$K_SYS_CTRL)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
```

2. Build listener addresses for IEEE-488 devices 2 and 4 by ORing the LIO$M_LNR bit to each address.

```
addr1 = 2
com1 = LIO$M_LNR .OR. addr1

addr2 = 4
com2 = LIO$M_LNR .OR. addr2
    .
    .
    .
```

3. Send the listener addresses out on the bus. This causes devices 2 and 4 to listen.

```
status = LIO$SET_I (ieee_id, LIO$K_COMMAND, 2, com1, com2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
```

4. Enqueue the send buffer. This example uses the listener address of device 6 as the device-specific argument. If no device-specific argument is given, device 0 will be addressed as the listener.

```
  status = LIO$ENQUEUE (ieee_id, buffer, buffer_length, , event_flag,
1                      , LIO$M_LNR .OR. addr3)
           IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Dequeue the send buffer. The "1" specifies that LIO$DEQUEUE is to wait for a buffer to become available.

```
status = LIO$DEQUEUE (ieee_id, buffer, buffer_length, , 1, , ,)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.5.18 Sending Data and Receiving Data When the IEEE-488 Device Is Attached as an Instrument

The LIO facility provides several ways in which instruments can send data to and receive data from the controller-in-charge.

To perform data transfers between an IEEE-488 instrument and the controller-in-charge, do the following:

1. The simplest way is to call LIO$READ or LIO$WRITE and wait until the controller-in-charge addresses the instrument to talk or listen.

```
status = LIO$READ (ieee_id, buffer, buffer_length, data_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$WRITE (ieee_id, buffer, data_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2. Often it is more useful to use the LIO$ENQUEUE and/or LIO$DEQUEUE routines, specifying input or output with the **device_specific** argument.

```
 status = LIO$ENQUEUE (instrument_id, buffer, buffer_length,
1          , , , LIO$M_TKR)
           IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

This enables the user program to continue while the instrument waits to be addressed as a talker by the controller-in-charge.

The **device-specific** argument LIO$M_TKR tells LIO that this is a read request (the instrument is a talker). If the **device-specific** argument were LIO$M_LNR, this would mean a write request (the instrument is a listener).

The IEEE-488 device processes buffers in the order in which they are enqueued. If an output buffer is enqueued before an input buffer, the device must be addressed first as a talker, and then as a listener.

3. If an instrument must be able to respond to both talk and listen requests, then it can enable the LIO$K_TKR_ADDR and LIO$K_LNR_ADDR events with the LIO$K_EVENT_ENA parameter.

```
 status = LIO$SET_I (instrument_id, LIO$K_EVENT_ENA, 2,
1            LIO$K_TKR_ADDR_EVT, LIO$K_LNR_ADDR_EVT)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
    .
    .
    .
 status = LIO$SHOW (instrument_id, LIO$K_EVENT_WAIT,
1            event, length)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **event** argument returns the IEEE-488 event that occurred. The **length** argument returns the number of events contained in the **event** argument.

4. The most flexible way to send and receive data is to supply user-written AST routines to be called when the enabled LIO$K_TKR_ADDR_EVT and LIO$K_LNR_ADDR_EVT events occur. The AST routines can then enqueue (LIO$ENQUEUE) buffers when the instrument is addressed. This way the instrument can handle requests to talk or listen in any order.

NOTE

The format in which data is transferred is entirely dependent on the instrument. However, using printable ASCII strings for commands, such as "T" for trigger, and ASCII-encoded decimal values for numbers, such as "5" "."'"3" for 5.3, are recommended. String values are usually terminated by a termination character. See Section 2.5.18.1, Using Termination Characters to Terminate Read Requests, for more information.

2.5.18.1 Using Termination Characters to Terminate Read Requests

An IEEE-488 device can be set to terminate input on receipt of a termination character.

Although only one termination character can be recognized at a time, the termination character can be repeated any number of times for an IEQ11 device. For example:

```
status = LIO$SET_I (instrument_id, LIO$K_TERM_CHAR, 2, 10, 2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

specifies that a line-feed (ASCII decimal 10) character, repeated twice in succession, signals the end of an input buffer.

The IEZ11 and the IOtech Micro488A devices do not support a repeat count. They will terminate on receiving the termination character once.

To disable termination character recognition, specify −1 as the value of the LIO$K_TERM_CHAR parameter, for example:

```
status = LIO$SET_I (instrument_id, LIO$K_TERM_CHAR, 1, -1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

Data transfers are faster when termination character recognition is disabled, because the driver does not have to check each character that is transferred to determine if it is a termination character.

2.5.18.2 Using EOI to Terminate Write Requests

An IEEE-488 device can signal the end of a write request by asserting an EOI line after the last byte of an output buffer is transferred. To cause the assertion of an EOI line, use the LIO$K_EOI parameter to enable EOI, for example:

```
status = LIO$SET_I (ieee_id, LIO$K_EOI, 1, LIO$K_ON)
```

This routine causes subsequent write requests to assert EOI after the last byte of a transfer.

2.6 Serial Line Devices

This section describes the serial line devices supported by VAXlab. If you are unfamiliar with serial line devices and the functions they perform, see the *VMS I/O User's Reference Manual* for more information.

See Table 3–2, Device Specifications and I/O Types, for a list of the serial line devices supported by VAXlab Software Library.

2.6.1 Attaching Serial Line Devices

Attaching a serial line device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the serial line device.

```
status = LIO$ATTACH (serial_id, 'TTA0', LIO$K_QIO)
         IF(.NOT. status) CALL LIB$SIGNAL(%VAL(status))
```

The **serial_id** argument returns the LIO-assigned device ID for the serial line device. The device is referenced by this device ID in subsequent routine calls to the device in a user program.

This example shows the device specification, TTA0, for the DZ11 serial line device. See Table 3–2 for the appropriate device types of the serial line devices supported by VSL.

The LIO$K_QIO value attaches the serial line device to use QIOs. LIO$K_QIO is the only I/O type supported for use with serial line devices.

2.6.2 Setting Up Serial Line Devices

Before you can begin data transfers using a serial line device, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show serial line device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-21: Serial Line LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_ACK_NAK_TERMINATOR	Establishes a termination character for the ACK/NAK string received from an external device.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up the device for asynchronous I/O.
LIO$K_BAUD_RATE	Sets the speed at which data is transmitted over a serial line.
LIO$K_BITS_PER_CHAR	Establishes the number of data bits per character.
LIO$K_BREAK	Generates a break condition on a terminal line for a specified period of time.
LIO$K_CTRL_AST	Specifies a user-written AST routine to be called on receipt of a specified control character.
LIO$K_CTRL_HANDLING	Sets up a flag that indicates what action to take on receipt of a control character specified using the LIO$K_CTRL_AST parameter.
LIO$K_DEVICE_ACK_NAK_BUFF	Supplies the buffer to be used when receiving an ACK or a NAK from a device.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DUPLEX	Specifies whether read/write requests are executed in half-duplex or full-duplex mode.
LIO$K_ECHO	Enables or disables the echoing of characters received on a serial line.
LIO$K_ERR_HANDLE	Specifies the way in which the serial line device handles errors.
LIO$K_ERROR_ENABLE	Enables or disables parity error handling for serial line devices.
LIO$K_FLOW_CONTROL	Establishes the method of flow control for a serial line device.
LIO$K_FLOW_MASTER	Establishes the XON/XOFF flow control scheme.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_HANGUP	Disconnects a terminal that is on a dial-up line.

Table 2–21 (Cont.): Serial Line LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_INPUT_TERMINATOR	Specifies a termination character or characters on the input side of a serial port.
LIO$K_MODEM	Specifies that the serial line is a modem.
LIO$K_MODEM_STATUS	Sets and returns modem status information.
LIO$K_OUTPUT_PREFIX	Specifies a prefix character string on the output side of a serial line.
LIO$K_OUTPUT_TERMINATOR	Specifies a termination character string on the output side of a serial line.
LIO$K_PARITY	Specifies the parity checking mode for a serial line.
LIO$K_PROTOCOL	Enables or disables the serial line user protocol feature.
LIO$K_PURGE	Purges all characters in the type-ahead buffer.
LIO$K_READ_PROMPT	Specifies a read-prompt to prefix each input data buffer.
LIO$K_STOP	Stops the device.
LIO$K_SYNCH	Sets up the device for synchronous I/O.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.
LIO$K_TIMEOUT_ENABLE	Enables a timeout for read requests.
LIO$K_TYPE_AHEAD	Enables or disables the typeahead buffer.
LIO$K_UNSOLICITED	Returns the number of characters in the type-ahead buffer.
LIO$K_USER_ACK_AST	Specifies the address of a user-supplied AST routine to transmit the ACK string on successful completion of a data transfer.
LIO$K_USER_ACK_STRING	Specifies the ACK string to be sent out by an AST routine on successful completion of a data transfer.
LIO$K_USER_NAK_AST	Specifies the address of a user-supplied AST routine to transmit the NAK string on unsuccessful completion of a data transfer.

Table 2–21 (Cont.): Serial Line LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_USER_NAK_STRING	Specifies the NAK string to be sent out by an AST routine on unsuccessful completion of a data transfer.
LIO$K_USER_READ_PROTOCAL_AST	Specifies the address of a user-supplied AST routine to be called on receipt of either a terminator or a full buffer of characters from a read request.
LIO$K_USER_WRITE_NAK_HANDLING	Specifies whether or not a sending device attempts to retransmit a buffer after receiving a NAK from the intended receiving device.
LIO$K_XON	Forces the sending of an XON character to reprime the serial line.

2.6.3 Using Serial Line Devices for Synchronous I/O

To make a serial line available to a nonprivileged user, use the following commands from a suitably privileged account:

```
$ SET PROCESS/PRIV=OPER
$ SET PROTECTION=W:RWLP/DEVICE dev_name:
```

To set up a serial line device for synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the serial line device as described in Section 2.6.1, Attaching Serial Line Devices.
4. Specify the I/O interface.

```
status = LIO$SET_I (serial_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set the baud rate. The following sample routine sets the baud rate at 1200.

```
status = LIO$SET_I (serial_id, LIO$K_BAUD_RATE, 1, 1200)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Set a device timeout. The following sample routine sets a 60 second timeout.

```
status = LIO$SET_I (serial_id, LIO$K_TIMEOUT, 1, 60)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify an input buffer terminator. The following sample routine specifies the letter "z" as the input buffer terminator.

```
status = LIO$SET_S (serial_id, LIO$K_INPUT_TERMINATOR, 'z')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Specify a read prompt. The following sample routine sets up the string "Enter data:" as the string thats prompts for data input.

```
status = LIO$SET_S (serial_id, LIO$K_READ_PROMPT, 'Enter data:')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Start the data transfer. The following sample routine reads and writes 10 bytes of data synchronously.

```
status = LIO$READ (serial_id, buffer, 10, buffer_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$WRITE (serial_id, buffer, 10, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Detach the device.

```
status = LIO$DETACH (serial_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.6.4 Using Serial Line Devices for Asynchronous I/O

To set up a serial line device for asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the serial line device as described in Section 2.6.1, Attaching Serial Line Devices.
4. Specify the I/O interface.

```
status = LIO$SET_I (serial_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Set the baud rate. The following sample routine sets the baud rate at 2400.

```
status = LIO$SET_I (serial_id, LIO$K_BAUD_RATE, 1, 2400)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Set a device timeout. The following sample routine sets a 30 second timeout.

```
status = LIO$SET_I (serial_id, LIO$K_TIMEOUT, 1, 30)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify an input buffer terminator. The following sample routine specifies the letter "q" as the input buffer terminator.

```
status = LIO$SET_S (serial_id, LIO$K_INPUT_TERMINATOR, 'q')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Enable the user-defined protocol feature.

```
status = LIO$SET_I (serial_id, LIO$K_PROTOCOL, 1, LIO$K_ON)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Specify the address of the AST routine.

```
 status = LIO$SET_I (serial_id, LIO$K_USER_READ_PROTOCOL_AST,
1            1, receive_buff)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Set up the acknowledge (ACK) and negative acknowledge (NAK) buffer. This buffer receives ACKs or NAKs from a device.

```
 status = LIO$SET_I (serial_id, LIO$K_DEVICE_ACK_NAK_BUFF, 4,
1            buffer, buffer_size, term_char, timeout)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Specify how a device handles negative acknowledges (NAKs) from a device.

```
 status = LIO$SET_I (serial_id, LIO$K_USER_WRITE_NAK_HANDLING, 1,
1            LIO$K_RESEND_LAST)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Get an event flag and clear it.

```
status = LIB$GET_EF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = SYS$CLREF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

13. Enqueue the buffer and wait for the event flag to be set by the AST routine. The AST routine will set the event flag when it receives an ACK from the device.

```
 status = LIO$ENQUEUE (serial_id, buffer, buffer_length, 0, 0,
1           LIO$K_OUTPUT)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

14. Wait for the event flag to be set by the AST routine. When the event flag is set, free the event flag and the memory allocated to the buffer.

```
status = SYS$WAITFR (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
status = LIB$FREE_EF (event_flag)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

15. Detach the device.

```
status = LIO$DETACH (serial_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_SERIAL.C in the LIO$EXAMPLES directory is a complete VAX C program that uses the user-defined protocol feature to transfer data. The AST routine, **receive_buff**, specified in step 9 is shown in LIO_SERIAL.C.

2.7 Software Pseudodevices

This section describes the software pseudodevices supported by VAXlab. The pseudodevices are:

- Disk file
- Memory queue
- Real-time plotting

2.7.1 Disk File Support

The disk file device moves data to and from disk files using QIOs. The QIOs move data buffers directly to and from disk using block I/O; each file is read or written in bytes. The size of the data buffers must be a multiple of 512 bytes, which is one VMS block.

2.7.1.1 Attaching a Disk File

Attaching a disk file means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the disk file device.

```
status = LIO$ATTACH (file_id, 'FLA0', LIO$K_QIO)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **file_id** argument returns the LIO-assigned device ID of the file device. The file is referenced by this device ID in subsequent routine calls to the file in a user program.

The device specification FLA0 specifies a file (FL) device with controller letter A and unit number 0. If you attach only one file device, specifying the device type FL is sufficient.

The LIO$K_QIO value attaches the file device to use QIOs. This is the only I/O type supported for use with the disk files.

2.7.1.2 Setting Up the Disk File Device

Before you can begin transferring data to the file device, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show disk file device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-22: Disk File LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DIRECTION	Sets the direction (input or output) of the file.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.

Table 2–22 (Cont.): Disk File LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_FILE_EXTENT	Extends an output file by the specified number of blocks.
LIO$K_FILE_POS	Repositions the current block pointer in an output file.
LIO$K_FILE_REMAIN	Returns the number of blocks remaining to be written in an output file.
LIO$K_FILE_SIZE	Sets the size (in blocks) of the output file.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_NAME	Specifies the file name.
LIO$K_OPEN_FILE	Opens a file.
LIO$K_SYNCH	Sets up the device for synchronous I/O.

2.7.1.3 Using Disk Files for Synchronous I/O

To set up the disk file device for synchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the file device as described in Section 2.7.1.1, Attaching a Disk File.
4. Specify the I/O interface.

```
status = LIO$SET_I (file_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify whether the file is an input or output device.

```
status = LIO$SET_I (file_id, LIO$K_DIRECTION, 1, LIO$K_OUTPUT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the file size.

```
status = LIO$SET_I (file_id, LIO$K_FILE_SIZE, 1, 10) !10-block file
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify the file name.

```
status = LIO$SET_S (file_id, LIO$K_NAME, 'AD_DATA')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Open the file.

```
status = LIO$SET_I (file_id, LIO$K_OPEN_FILE, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Write data to the file.

```
status = LIO$WRITE (file_id, buffer, data_length, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Detach the device.

```
status = LIO$DETACH (file_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

2.7.1.4 Using Disk Files for Asynchronous I/O

To set up the disk file device for asynchronous I/O, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the file device as described in Section 2.7.1.1, Attaching a Disk File.
4. Specify the I/O interface.

```
status = LIO$SET_I (file_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

5. Specify whether the file is an input or output device.

```
status = LIO$SET_I (file_id, LIO$K_DIRECTION, 1, LIO$K_OUTPUT)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

6. Specify the file size.

```
status = LIO$SET_I (file_id, LIO$K_FILE_SIZE, 1, 10) !10-block file
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify the file name.

```
status = LIO$SET_S (file_id, LIO$K_NAME, 'AD_DATA')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Open the file.

```
status = LIO$SET_I (file_id, LIO$K_OPEN_FILE, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Get a free VMS event flag to associate with the buffer and enqueue the output buffer to the device.

```
 status = LIB$GET_EF (e_flag)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

 status = LIO$ENQUEUE (file_id, buffer, buffer_length, data_length,
1            e_flag, , )
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Dequeue the buffer from the device.

```
status = LIO$DEQUEUE (file_id, buffer, buffer_length, , 1, , )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Detach the device.

```
status = LIO$DETACH (file_id, )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_BUF_FWD.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the synchronous I/O interface and single-buffer DMA with buffer forwarding to read A/D values from the ADV11-D device and forward the data buffers to a disk file.

2.7.2 Memory Queue Support

The LIO facility provides the memory queue pseudodevice as a convenient way to manage buffers. You can use the memory queue device to manage buffers in a global section shared by several processes. You can also use the memory queue device locally to manage buffers used by asynchronous parts of a program, and to allocate buffers dynamically from virtual memory.

2.7.2.1 Attaching the Memory Queue Device

Attaching a memory queue device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the memory queue device.

To attach a memory queue device to manage memory local to a user's process, use the following routine:

```
status = LIO$ATTACH (memory_id, 'APA0', LIO$K_LOCAL) !Local memory
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

To attach a memory queue device to manage an interprocess global section, use the following routine:

```
status = LIO$ATTACH (memory_id, 'APA0', LIO$K_INTER_PROC) !Global section
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **memory_id** argument in each routine returns the LIO-assigned device ID for the memory queue device. Each device is referenced by the device ID in subsequent routine calls to the device in a user program.

The device specification APA0 in each routine specifies a memory queue (AP) device with controller letter A and unit number 0.

The value of the *io_type* argument determines which function a memory queue device is attached to perform. The value LIO$K_LOCAL attaches the device to manage memory local to a user's process. The value LIO$K_INTER_PROC attaches the device to manage an interprocess global section.

2.7.2.2 Setting Up the Memory Queue Device

Before you can begin data transfers with the memory queue device, you must set up the device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show memory queue device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2–23: Memory Queue LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets the device for asynchronous I/O.
LIO$K_BUFF_SIZE	Sets the size, in bytes, of the asynchronous buffers to allocate for the device.
LIO$K_BUFF_SOURCE	Specifies the sources from which memory allocation is to occur.
LIO$K_DEVICE_EF	Establishes the event flag that is set when a buffer becomes available.
LIO$K_DISPLAY_ONLY	Sets an interprocess memory queue to display data buffers to a second process.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.
LIO$K_N_BUFFS	Sets the number of channels in the data buffer.
LIO$K_NAME	Specifies the name of a global section.
LIO$K_PAGE_ALIGN	Page-aligns buffers allocated for use with the memory queue.
LIO$K_READ_ONLY	Establishes read-only access to a global section.
LIO$K_SYNCH	Sets up the device for synchronous I/O when attached for QIO only.
LIO$K_TRANSFER	Sets an interprocess memory queue to transfer data buffers between processes.

Some of these parameters, such as LIO$K_BUFF_SOURCE, are appropriate for setting up the memory queue device for both managing local memory and managing a global section. Others, such as LIO$K_READ_ONLY and LIO$K_DISPLAY_ONLY, are appropriate only when a memory queue device is attached to manage memory in a global section.

2.7.2.3 Using a Memory Queue Device to Manage Local Memory

When a memory queue device is attached locally to a process, the device can be used in the following two ways:

- **To communicate between asynchronous parts of a program.** For example, the memory queue can be used to pass buffers between an AST routine and the main program. The AST routine performs time-critical processing on the buffers, such as verifying out-of-range conditions, and then queues the buffers to the memory device. The main program dequeues the buffers from the memory device, and performs further processing of the data.

  To set up the memory queue device to perform this task, do the following:

  1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
  2. Declare the data types and variables you are using in your program.
  3. Attach the memory queue device as described in Section 2.7.2.1, Attaching the Memory Queue Device.
  4. Specify the buffer source as a user-supplied buffer.

```
status = LIO$SET_I (memory_id, LIO$K_BUFF_SOURCE, 1, LIO$K_USER)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

NOTE

When specifying a user-supplied buffer as the buffer source for the memory queue device, it is not necessary to use LIOK_N_BUFFS, LIOK_BUFF_SIZE, or LIO$K_PAGE_ALIGN to set up buffer characteristics.

- **To allocate buffers dynamically from virtual memory.** This preallocates a specified number of buffers of a specified size.

  To set up the memory queue device to allocate buffers dynamically from virtual memory, do the following:

  1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
  2. Declare the data types and variables you are using in your program.

3. Attach the memory queue device as described in Section 2.7.2.1, Attaching the Memory Queue Device.
4. Specify the number of buffers to allocate. This sample routine allocates five buffers.

```
status = LIO$SET_I (memory_id, LIO$K_N_BUFFS, 1, 5)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
5. Specify the size (in bytes) of each buffer. This sample routine specifies 200-byte buffers.

```
status = LIO$SET_I (memory_id, LIO$K_BUFF_SIZE, 1, 200)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
6. The first buffer allocated is longword-aligned. You can use the LIO$K_PAGE_ALIGN parameter to page-align the buffers. Page-aligning the buffers ensures that the first buffer begins on a page boundary (the address is a multiple of 512 bytes).

```
status = LIO$SET_I (memory_id, LIO$K_PAGE_ALIGN, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
7. Specify the buffer source as virtual memory.

```
 status = LIO$SET_I (memory_id, LIO$K_BUFF_SOURCE, 1,
1            LIO$K_VIRTUAL_MEM)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This routine actually allocates the buffers from virtual memory.

The other LIO devices have two queues: the **device queue**, to which the main program enqueues buffers, and the **user queue**, from which a user-written AST routine or the main program dequeues buffers. The memory queue device provides a third queue, called the **free queue**. When virtual memory allocation occurs, the buffers are placed on the device's free queue.

Use the LIO$DEQUEUE routine, supplying LIO$K_FREE_Q as the value of the **device_specific** argument, to dequeue a buffer from the free queue. For example:

```
 status = LIO$DEQUEUE (memory_id, buffer_address, buffer_length,
1           , , , LIO$K_FREE_Q)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

Buffers are initially dequeued from the free queue in ascending order. If you supply the **buffer_index** argument in the LIO$DEQUEUE routine call, the variable returns the number of the buffer, beginning with buffer number one. If you subsequently enqueue the buffers back to the free queue out of order, they remain out of order.

When no buffers are left on the free queue, the LIO$DEQUEUE routine tries to get a buffer from the user queue. If there are no buffers on the user queue, the LIO$DEQUEUE routine returns the LIO$EMPTYQ condition value.

Buffers allocated from virtual memory can be enqueued to the memory queue device using the LIO$ENQUEUE routine. When the **device_specific** argument is defaulted, a buffer is enqueued to the memory queue. When the value of the **device_specific** argument is LIO$K_FREE_Q, a buffer is placed on the free queue. In this case, information about the buffer, such as the values of the **data_length** and **buffer_index** arguments, are initialized to default values.

2.7.2.4 Setting Up a Memory Queue Device for Interprocess Communications

When a memory queue is attached for interprocess communications, it can be used in the following two ways:

- **To transfer data.** The memory queue can be set up to act as a door between processes. The processes can then transfer data buffers back and forth through a shared global section of memory managed by the memory queue device.

  To set up the memory queue device to transfer data, do the following: **The memory queue device in each process must be set up identically.**

  1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.

2. Declare the data types and variables you are using in your program.
3. Attach the memory queue device as described in Section 2.7.2.1, Attaching the Memory Queue Device.
4. Specify the number of buffers to allocate. This sample routine allocates five buffers.

```
status = LIO$SET_I (memory_id, LIO$K_N_BUFFS, 1, 5)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
5. Specify the size (in bytes) of each buffer. This sample routine specifies 200-byte buffers.

```
status = LIO$SET_I (memory_id, LIO$K_BUFF_SIZE, 1, 200)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
6. Specify the global section name.

```
status = LIO$SET_S (memory_id, LIO$K_NAME, 'XFER_DATA')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
7. Set up the memory queues to transfer data between processes.

```
status = LIO$SET_I (memory_id, LIO$K_TRANSFER, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.
8. Specify the buffer source. To perform this task, the buffer source can be either virtual memory or an array. If the buffer source is an array, the array must be page-aligned. The array must also be large enough to contain the buffer overhead. See the description of the LIO$K_BUFF_SOURCE parameter in Chapter 4 for more information.

```
 status = LIO$SET_I (memory_id, LIO$K_BUFF_SOURCE, 1,
1            LIO$K_VIRTUAL_MEM)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

When the main program in each process sets up the buffer source (LIO$K_BUFF_SOURCE), the memory queue device maps to the global section. Then, the two processes can pass data back and forth through the memory queue.

To each process, the interprocess memory queue device looks the same as the virtual memory queue described in Section 2.7.2.3, Using a Memory Queue Device to Manage Local Memory. The buffers all begin on the free queue. You can dequeue buffers from the free queue by specifying LIO$K_FREE_Q as the value of the **device_specific** argument. You can enqueue buffers to the memory queue by defaulting the **device_specific** argument.

The online sample program LIO_MQ_XFER.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the interprocess memory queue's transfer function to transfer data buffers from one memory queue device to a second memory queue device running in a second process.

- **To display and copy data.** The memory queue device can also be set up to act as a window between processes. The first process uses the memory queue device for temporary storage of data, such as data on its way from an A/D to a disk file. The second process makes a copy of the data as it passes by the window.

  To set up the memory queue device to display and copy data, do the following:

  1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
  2. Declare the data types and variables you are using in your programs.
  3. Attach the memory queue device as described in Section 2.7.2.1, Attaching the Memory Queue Device.
  4. Specify the number of buffers to allocate. This sample routine allocates five buffers.

     ```
     status = LIO$SET_I (memory_id, LIO$K_N_BUFFS, 1, 5)
              IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
     ```

     You must set up this parameter before the memory allocation occurs.

5. Specify the size (in bytes) of each buffer. This sample routine specifies 200-byte buffers.

```
status = LIO$SET_I (memory_id, LIO$K_BUFF_SIZE, 1, 200)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   You must set up this parameter before the memory allocation occurs.

6. Specify the global section name.

```
status = LIO$SET_S (memory_id, LIO$K_NAME, 'GET_DATA')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. When you set up the memory queue device in the **first process**, perform steps a through c. Then, skip step 8 and perform step 9.

   a. Set up the device to **display** the data as it passes by the window.

```
status = LIO$SET_I (memory_id, LIO$K_DISPLAY_ONLY, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

      The first process is typically running at a high priority handling critical I/O functions, such as moving data from an I/O device to a disk file.

   b. Set up the device to use asynchronous I/O.

```
status = LIO$SET_I (memory_id, LIO$K_ASYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

      This process uses the LIO$ENQUEUE routine to enqueue buffers to the memory queue device.

   c. Set up the device with a device event flag. This event flag is supplied as the value of the **device_specific** argument to dequeue a buffer from the device's free queue.

```
status = LIO$SET_I (memory_id, LIO$K_DEVICE_EF, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

      The LIO$DEQUEUE routine gets a buffer from the device's free queue.

8. When you set up the memory queue device in the **second process**, perform steps a and b.

   a. Set up the device to **read** the data as it passes by the window.

```
status = LIO$SET_I (memory_id, LIO$K_READ_ONLY, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This second process is monitoring the data transfer, and perhaps plotting an occasional buffer to a graphics terminal. This process is running at a low priority and is not able to interrupt the flow of data buffers past the window. It misses buffers if the I/O process is using most of the CPU.

   b. Set up the device to use synchronous I/O.

```
status = LIO$SET_I (memory_id, LIO$K_SYNCH, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

   This second process must use LIO$READ to read from the queue. The buffer supplied to the LIO$READ routine must not be from the global section. The data read from the global section is copied into the buffer supplied in the LIO$READ routine.

9. Specify the buffer source. To perform this task, the buffer source can be either virtual memory or an array. If the buffer source is an array, the array must be page-aligned.

```
 status = LIO$SET_I (memory_id, LIO$K_BUFF_SOURCE, 1,
1            LIO$K_VIRTUAL_MEM)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The online sample program LIO_MQ_DISPLAY.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the interprocess memory queue's display-only function to display data buffers to a read-only memory queue device.

The online sample program LIO_MQ_READONLY.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that shows how to use the interprocess memory queue's read-only function to read data buffers acquired by a display-only memory queue device.

2.7.3 Real-Time Plotting

The LIO facility supports the real-time plotting pseudodevice for UIS-based (VWS-based) devices as a method of monitoring real-time I/O. For example, as data is collected by an A/D converter, it can be plotted on your terminal screen. The data is plotted in strip chart form that scrolls from left-to-right across the terminal screen, or in the form of oscilloscope-type output.

The real-time plotting device works only with floating-point data.

NOTE

The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS. Attempting to attach the real-time plotting device on MicroVAX-based systems returns the unknown device, LIO$_UNKDEV, condition value.

2.7.3.1 Real-Time Plotting Device Parameters

Before you can begin data transfers to the real-time plotting device, you must set up certain device characteristics. The following table lists the LIO$SET and LIO$SHOW parameters you can use to set up and show real-time plotting device characteristics. See Chapter 4 for reference descriptions of the parameters listed in this table.

Table 2-24: Real-Time Plotting LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_CURRENT_CHANNEL	Specifies which channel is affected by channel-specific set calls.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_MAX_CHANNELS	Sets the maximum number of channels that can be plotted using the real-time plotting device.

Table 2–24 (Cont.): Real-Time Plotting LIO$SET and LIO$SHOW Parameters

Parameter	Function
LIO$K_MULTIPLE_X_AXES	Specifies the x-axis representation for the plotting window.
LIO$K_N_BUFFS	Sets the number of channels in the data buffer.
LIO$K_PLOT_SIZE	Sets the size of the plotting window.
LIO$K_PLOT_TYPE	Specifies the type of plotting.
LIO$K_PO_CHAN	Specifies the channel numbers of the channels plotted.
LIO$K_POSITION	Establishes the position of the plotting window on the display screen.
LIO$K_SKIP_COUNT	Specifies how many points are skipped and not plotted.
LIO$K_START	Starts the device.
LIO$K_TITLE	Specifies the title for the graph of the current channel.
LIO$K_TITLE_n	Specifies the title for the graph of each channel plotted.
LIO$K_X_LABEL	Specifies the label on the x-axis of the current channel.
LIO$K_X_RANGE	Specifies the number of points to display, and the number of points to shift, along the x axis.
LIO$K_Y_LABEL	Specifies the label on the y-axis of the current channel.
LIO$K_Y_MAX	Sets the maximum y-axis value for each channel plotted.
LIO$K_Y_MIN	Sets the minimum y-axis value for each channel plotted.

2.7.3.2 Attaching the Real-Time Plotting Device

Attaching the real-time plotting device means assigning a VMS I/O channel to the device and initializing LIO data structures for, and pointers to, the device.

Use the LIO$ATTACH routine to attach the real-time plotting device.

```
status = LIO$ATTACH (graphics_id, 'POA0', )
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

The **graphics_id** argument returns the LIO-assigned device ID for the real-time plotting device. The real-time plotting device is referenced by this device ID in subsequent routine calls to the device in a user program.

The device specification POA0 specifies a real-time plotting (PO) device with controller letter A and unit number 0.

When attaching a real-time plotting device, you do not need to specify an **io_type** argument.

2.7.3.3 Setting Up and Using the Real-Time Plotting Device

To set up and to use the real-time plotting device, do the following:

1. Include the symbolic definition files required by the VAXlab facilities and programming language you are using.
2. Declare the data types and variables you are using in your program.
3. Attach the device as described in Section 2.7.3.2, Attaching the Real-Time Plotting Device.
4. Set up the device to use synchronous I/O.

   ```
   status = LIO$SET_I (graphics_id, LIO$K_SYNCH, 0)
            IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
   ```

   Performing this step is optional. It is included in this procedure for clarity. **The real-time plotting device supports only synchronous write-only operations.**
5. Specify the maximum number of channels that can be plotted. This example specifies that four channels can be plotted.

   ```
   status = LIO$SET_I (graphics_id, LIO$K_MAX_CHANNELS, 1, 4)
            IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
   ```

6. Specify the channel numbers of the channels to plot. This example sets up two channels: channel 0 and channel 1.

```
status = LIO$SET_I (graphics_id, LIO$K_PO_CHAN, 2, 0, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

7. Specify the type of plotting–either oscilloscope-type output or scrolling output. This example sets up scrolling output.

```
status = LIO$SET_I (graphics_id, LIO$K_PLOT_TYPE, 1, LIO$K_STRIPCHART)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

8. Specify the number of points to display along the x-axis, and the number of points to shift along the x-axis. This example sets up 360 to display, 45 to shift along the x-axis.

```
status = LIO$SET_R (graphics_id, LIO$K_X_RANGE, 2, 360.0, 45.0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

9. Specify how many points are to be skipped and not plotted. This example skips every nine points, which means that every tenth point is plotted.

```
status = LIO$SET_I (graphics_id, LIO$K_SKIP_COUNT, 1, 9)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

10. Specify the minimum y-axis value for both channels.

```
status = LIO$SET_R (graphics_id, LIO$K_Y_MIN, 2, -3.0, -3.0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

11. Specify the maximum y-axis value for both channels.

```
status = LIO$SET_R (graphics_id, LIO$K_Y_MAX, 2, 3.0, 3.0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

12. Specify the graph title of each channel plotted.

```
status = LIO$SET_I (graphics_id, LIO$K_CURRENT_CHANNEL, 1, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$SET_S (graphics_id, LIO$K_TITLE, 'Channel 0')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$SET_I (graphics_id, LIO$K_CURRENT_CHANNEL, 1, 1)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

status = LIO$SET_S (graphics_id, LIO$K_TITLE, 'Channel 1')
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

LIO$K_CURRENT_CHANNEL sets each channel in turn to the current channel, and the graph of each channel is then given a title.

13. Specify the number of channels in each data buffer.

```
status = LIO$SET_I (graphics_id, LIO$K_N_BUFFS, 1, 2)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

14. Start the plotting.

```
status = LIO$SET_I (graphics_id, LIO$K_START, 0)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

15. Write a buffer to the graphics device.

```
status = LIO$WRITE (graphics_id, graph_buffer, graph_size,)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))
```

16. Detach the device.

The online sample program LIO_AXV_RTPLOT.FOR in the LIO$EXAMPLES directory is a complete VAX FORTRAN program that uses the synchronous I/O interface and QIOs to read A/D values from the AXV11-C. This program then plots the values on the terminal screen using the real-time plotting device in a continuous real-time display.

Chapter 3

Laboratory I/O Routine Reference Descriptions

This chapter presents an overview and detailed reference descriptions of the Laboratory I/O (LIO) routines. You use these routines to set up, initiate, and control I/O to and from laboratory I/O devices.

The LIO routines are presented in alphabetical order, with a brief description of what the routine does. The routines are described using the following format:

- **Format** gives the routine entry point name and the argument list in the correct syntactical form.
- **Returns** lists the information returned by the routine.
- **Arguments** provides the following information: what each argument passes to or returns from the routine, and the data type, access, mechanism, and acceptable values of the argument.
- **Description** contains information about the specific actions taken by the routine, such as:
  - Interaction between routine arguments
  - Interactions and dependencies between the routine and other LIO routines
  - Restrictions for use
  - Actions specific to the routine when used with certain devices

The following table summarizes the LIO routines.

Table 3–1: Laboratory I/O Routine Summary

Routine Call	Function
LIO$ATTACH	Assigns a VMS I/O channel to the specified device, initializes LIO data structures for and pointers to the device, and returns an LIO-assigned device ID for the device.
LIO$DEQUEUE	Dequeues a buffer from a device set up to use the asynchronous I/O interface, and returns the buffer to the user program.
LIO$DETACH	Detaches the specified device, returns any associated storage to the system, and closes and deallocates associated VMS devices.
LIO$ENQUEUE	Queues a buffer to a device that is set up to use the asynchronous I/O interface.
LIO$READ	Reads a buffer from an input device that is set up to use the synchronous I/O interface.
LIO$SET_I	Sets up a device according to a parameter code and any number of integer values.
LIO$SET_R	Sets up a device according to a parameter code and any number of real values.
LIO$SET_S	Sets up a device according to a parameter code and a character-string value.
LIO$SHOW	Returns the current values of a specified parameter.
LIO$WRITE	Writes a buffer to an output device that is set up to use the synchronous I/O interface.

LIO$ATTACH

This routine does the following:

- Assigns a VMS I/O channel to the specified device
- Initializes LIO data structures for and pointers to the device
- Returns the LIO-assigned device ID for the device

The LIO$ATTACH routine must be called before any other routine call to a device.

Format LIO$ATTACH *(device_id, devspec, [io_type])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the LIO-assigned device ID. The **device_id** argument is the address of a signed longword integer into which the LIO facility writes the device ID. The device is then referenced by this device ID in subsequent routine calls to the device in a user program.

devspec
VMS Usage: **device_name**
type: **character string**
access: **read only**
mechanism: **by descriptor**
Contains a character string with a maximum of 131 characters in length specifying the device or a VMS logical name that translates to a device specification. The **devspec** argument is the address of a descriptor that points to this device specification. See Table 3–2 for a list of the argument values.

io_type
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the type of I/O to use. The **io_type** argument is the address of a signed longword integer containing the I/O type.

Description

Table 3–2 shows the I/O types supported for each device.

Table 3–2: Device Specifications and I/O Types

Device	Specification	I/O Type
AAF01[1]	UUcu	LIO$K_QIO
AAV11-D	AYcu	LIO$K_MAP LIO$K_QIO
ADF01[1]	UUcu	LIO$K_QIO
ADQ32	AWcu	LIO$K_QIO
ADV11-D	AZcu	LIO$K_MAP LIO$K_QIO
AXV11-C	AXcu	LIO$K_MAP LIO$K_CTI LIO$K_QIO

[1]This device is available only in Europe.

Table 3–2 (Cont.): Device Specifications and I/O Types

Device	Specification	I/O Type
DRB32	UQcu	LIO$K_QIO
DRB32W	UQWcu	LIO$K_QIO
DRQ11-C[1]	UUcu	LIO$K_QIO
DRQ3B	HXc0 (input port) HXc1 (output port)	LIO$K_QIO LIO$K_QIO
DRV11-J	DNcu	LIO$K_MAP LIO$K_QIO
DRV11-WA	XAcu	LIO$K_QIO
IAV11-A[1]	IVcu	LIO$K_QIO
IAV11-AA[1]	IVcu	LIO$K_QIO
IAV11-B[1]	IVcu	LIO$K_QIO
IAV11-C[1]	IVcu	LIO$K_QIO
IAV11-CA[1]	IVcu	LIO$K_QIO
IDV11-A[1]	IVcu	LIO$K_QIO
IDV11-B[1]	IVcu	LIO$K_QIO
IDV11-C[1]	IVcu	LIO$K_QIO
IDV11-D[1]	IVcu	LIO$K_QIO
IEQ11[2]	IXcu	LIO$K_CTRL LIO$K_INSTRUMENT LIO$K_SYS_CTRL
IEZ11[2]	EKcu	LIO$K_CTRL LIO$K_INSTRUMENT LIO$K_SYS_CTRL
IOtech Micro488A[2]	ITcu	LIO$K_CTRL LIO$K_INSTRUMENT LIO$K_SYS_CTRL
KWV11-C	KZcu	LIO$K_MAP LIO$K_QIO
Preston (DRB32W)	PGcu	LIO$K_QIO

[1]This device is available only in Europe.

[2]The **io_type** argument is used here to attach the device either as a controller, as an instrument, or as the system controller.

Table 3–2 (Cont.): Device Specifications and I/O Types

Device	Specification	I/O Type
Preston (DRQ3B)	PFcu	LIO$K_QIO
Preston (DRV11-WA)	PGcu	LIO$K_QIO
Simpact RTC01	KBcu	LIO$K_MAP LIO$K_QIO
Disk files	FLcu	Not applicable
Memory queue[3]	APcu	LIO$K_LOCAL LIO$K_INTER_PROC
Real-time plotting	POcu	Not applicable
Serial line devices:		
DH-	TXcu	LIO$K_QIO
DM-	TXcu	LIO$K_QIO
DZ-	TTcu	LIO$K_QIO
LAT	LTcu	LIO$K_QIO
MicroVAX2000	TTcu	LIO$K_QIO
MicroVAX3100	TTcu	LIO$K_QIO
VAXstation 2000	TTcu	LIO$K_QIO
VAXstation 3100	TTcu	LIO$K_QIO

[3]The **io_type** argument is used here to attach the device to manage memory local to a user's process, or to manage an interprocess global section.

Note that all devices configured in your system are prefixed with a two- or three-letter device type, such as AX or UQW, followed by a variable controller letter (A, B, C ...) and the unit number, which is usually 0.

The controller number specifies which device is addressed when more than one device is installed. The first device is assigned a controller letter A, the second B, and so forth.

Some devices provide multiple functionality on a single board. For example, a single serial line board might have multiple serial ports. Each port is assigned a unit number, starting with 0.

If you have only one of any of the devices configured in your system, specifying only the two- or three-letter device type is sufficient.

When using the DRQ3B device, the **devspec** HXA0 is used to specify the **input** port of the device. The **devspec** HXA1 is used to specify the **output** port of the device. For this device, you must supply the complete device specification.

When you attach a device to use QIOs, the following restrictions apply:

- If the device is capable of direct memory access (DMA), attaching it with LIO$K_QIO is the only way to use the DMA feature, which allows the fastest transfer rate. Because of the system overhead associated with each QIO call, QIO is best used when moving large amounts of data in large buffers.
- With devices set for buffer forwarding, the LIO$ENQUEUE and LIO$DEQUEUE routines are only available when each device is attached with LIO$K_QIO and set to use the asynchronous I/O interface.
- User and external event AST routines can only be used when devices are attached with LIO$K_QIO.
- When a device is attached with QIO, the maximum size of an individual data buffer is 65,534 bytes.

When you attach a device to use polled, or memory-mapped, I/O, the following restrictions apply:

- Only the synchronous I/O interface is available.
- The software cannot use the direct memory access (DMA) feature of the hardware, so the maximum transfer rate is limited.

When you attach a device to use connect-to-interrupt I/O, the following restrictions apply:

- Only the synchronous I/O interface is available.
- The software cannot use the direct memory access (DMA) feature of the hardware, so the maximum transfer rate is limited.

LIO$DEQUEUE

This routine does the following:

- Dequeues a buffer from a device set up to use the asynchronous I/O interface
- Returns the buffer to the main program

Format LIO$DEQUEUE *(device_id, buffer, buffer_length, [data_length], [wait], [buffer_index], [device_specific])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device from which the LIO$DEQUEUE routine dequeues the buffer. The **device_id** argument is the address of a signed longword integer that contains this device ID.

buffer
VMS Usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Returns the address of the oldest data buffer. The **buffer** argument is the address of an unsigned longword into which the LIO facility writes the starting virtual address of this data buffer.

buffer_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the length of the data buffer in bytes. The **buffer_length** argument is the address of a signed longword integer into which the LIO facility writes the length of **buffer** in bytes.

data_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the length of the data in the buffer in bytes. The **data_length** argument is the address of a signed longword integer into which the LIO facility writes the length of the data in **buffer** in bytes.

wait
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies whether or not the LIO$DEQUEUE routine is to wait for a buffer to become available. If the value of this argument is nonzero, the routine waits for a buffer to become available.

You must set up an event flag for the device or specify an event flag through the LIO$ENQUEUE call, or this argument is ignored. If the value of this argument is zero and no buffer is available, the LIO$_EMPTYQ condition value is returned. The **wait** argument is the address of a signed longword integer containing a zero or nonzero integer.

buffer_index
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the buffer index of the buffer previously assigned using the **buffer_index** argument of the LIO$ENQUEUE routine when the buffer was enqueued. The **buffer_index** argument is the address of a signed longword integer into which the LIO facility writes the buffer index number.

This value is not used by the LIO facility. It is included in the LIO$ENQUEUE routine call argument list as a way for a user program to identify buffers by assigning unique numbers to them. If a buffer index number is supplied, it is returned by the LIO$DEQUEUE routine when the buffer is dequeued.

device_specific
VMS Usage: **longword_unsigned**
type: **longword integer (unsigned)**
access: **device-dependent**
mechanism: **by reference**
Specifies or returns device-specific information about the buffer. When used with certain devices, the **device_specific** argument is the address of an unsigned longword integer containing information required by the device to perform the data transfer. When used with other devices, the **device_specific** argument is the address of an unsigned longword integer to which the LIO facility writes information about the data transfer.

The following table lists the devices that support the use of this argument with the LIO$DEQUEUE routine, the LIO-supplied values of this argument, if any, and the information that is being supplied or returned about the data transfer.

Table 3–3: LIO$DEQUEUE Device-Specific Argument Values

Argument Values	Description
	AAF01,[1] ADF01,[1] DRQ11-C[1]
User-supplied I/O status block	Here you use the **device_specific** argument to specify the address of a user-supplied I/O status block. The user-supplied I/O status block returns additional information about the completed I/O transfer.
	AXV11-C
User-supplied output variable	Here you use the **device_specific** argument to return the source of the buffer. This means whether it is an input buffer from the ADC or an output buffer from the DACs.
	DRQ3B
User-supplied output variable	If the **device_specific** argument is supplied, it returns the contents of the status register in the high 16 bits, and the contents of the DMA status register in the low 16 bits.
	DRV11-WA
User-supplied output variable	If the **device_specific** argument is supplied, it returns the state of the three control lines in bits 0, 1, and 2, and the status of the three sense lines in bits 9, 10, and 11.

[1]This device is available only in Europe.

Table 3–3 (Cont.): LIO$DEQUEUE Device-Specific Argument Values

Argument Values	Description
	Memory queue
LIO$K_FREE_Q LIO$K_USER_Q	Here you use the **device_specific** argument to signal whether a buffer is to be dequeued from the device's user queue or from the free queue. Specifying a LIO$K_FREE_Q signals the device to dequeue a buffer from the free queue. If the free queue is empty, the routine attempts to dequeue a buffer from the user queue. If the routine obtains a buffer from the user queue, this argument returns LIO$K_USER_Q. Specifying LIO$K_USER_Q signals the device to dequeue a buffer from the user queue. Check the value of this argument if the source of the buffer is important.
	Serial line
LIO$K_INPUT LIO$K_OUTPUT	Here you use the **device_specific** argument to specify whether the serial line device is to dequeue an input request or an output request.

Description

Use the LIO$DEQUEUE routine to return previously enqueued (LIO$ENQUEUE) buffers to a user program. If no buffers are available and the **wait** argument is nonzero, the routine call does not return until an enqueued buffer becomes available. You can use this routine to return an enqueued buffer or to wait for a forwarding loop to complete.

You use this routine call with the memory queue device to obtain buffers allocated from virtual memory from the device's free queue. You must dequeue a buffer from the free queue before you can begin using the buffer for data transfers.

LIO$DETACH

This routine does the following:

- Detaches the specified device
- Returns any associated storage to the system
- Closes and deallocates associated VMS devices

Format LIO$DETACH *(device_id, [rundown])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device the LIO$DETACH routine is to detach. The **device_id** argument is the address of a signed longword integer containing this device ID.

rundown
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies when to detach the device. If the value of this argument is nonzero, the device is not detached until all outstanding I/O requests are complete. If the value of this argument is zero, the device is detached immediately. All outstanding I/O requests are cancelled.

This argument is only valid when the device is set to use the asynchronous I/O interface. The **rundown** argument is the address of a signed longword integer that contains the detach condition.

Description

See the **Arguments** for details.

LIO$ENQUEUE

This routine queues a buffer to a device set up to use the asynchronous I/O interface.

Format

LIO$ENQUEUE *(device_id, buffer, buffer_length, [data_length], [event_flag], [buffer_index], [device_specific])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device to which the LIO$ENQUEUE routine enqueues the buffer. The **device_id** argument is the address of a signed longword integer containing this device ID.

buffer
VMS Usage: **array**
type: **array**
access: **read or write or read/write**
mechanism: **by reference**
Specifies the address of the data buffer. The **buffer** argument is the address of the array which is the data buffer. The user defines the type of array, but it should be a type acceptable to the device.

For the IAV11-A[1] devices, this argument returns information about the data transfer. This argument is an array of longwords. Each longword returns information about one A/D channel. Byte 1 returns the channel number. Byte 2 returns the channel gain. The high word (bytes 3 and 4) returns the actual A/D value.

buffer_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the length of the data buffer in bytes. The **buffer_length** argument is the address of a signed longword integer containing the length of the data buffer.

For IAV11-A[1] devices, specify **buffer_length** as a multiple of four.

data_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
For input devices, specifies the maximum number of bytes to read into the buffer. For output devices, specifies the amount of data in the buffer, in bytes. The **data_length** argument is the address of a signed longword integer that contains this information. If the **data_length** argument is omitted, it defaults to the value of the **buffer_length** argument.

event_flag
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies a VMS event flag to be associated with the buffer. The **event_flag** argument is the address of a signed longword integer that contains this event flag. **Specify a unique VMS event flag for each buffer.**

[1] This device is available only in Europe.

You can use the VMS Run-Time Library Routine LIB$GET_EF to obtain a free VMS event flag. This routine allocates one local event flag from a process-wide pool and returns the number of the allocated flag to the calling program.

You can also specify a value 1 through 23, or 32 through 127 that is unique to the process. Event flags 24 through 31 are reserved for use by DIGITAL. Event flags 1 through 23 and 32 through 63 are local to the process. Event flags 64 through 127 are in global event flag clusters that may or may not currently be associated with the process. The default value is zero.

This argument is required if the device is attached with LIO$K_QIO and the LIO$DEQUEUE routine is to wait for the buffer transaction to complete.

buffer_index
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies an index number for the buffer. The **buffer_index** argument is the address of a signed longword integer that contains the buffer index number. The buffer index number is returned by the LIO$DEQUEUE call, and can be used by a user program to identify the buffer.

This value is not used by the LIO facility. It is included in the LIO$ENQUEUE routine call argument list as a way for a user program to identify buffers by assigning unique numbers to them. If a buffer index number is supplied, it is returned by the LIO$DEQUEUE routine when the buffer is dequeued.

device_specific
VMS Usage: **longword_unsigned**
type: **longword integer (unsigned)**
access: **device-dependent**
mechanism: **by reference**
Specifies or returns device-specific information about the buffer. When used with certain devices, the **device_specific** argument is the address of an unsigned longword integer that contains information required by the device to perform the data transfer. When used with other devices, the **device_specific** argument is the address of an unsigned longword

integer to which the LIO facility writes information about the data transfer.

The following table lists the devices that support the use of this argument with the LIO$ENQUEUE routine, the LIO-supplied values of this argument, if any, and the information that is being supplied or returned about the data transfer.

Table 3–4: LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	AAF01,[1] ADF01,[1] DRQ11-C[1]
User-supplied parameter block	Here you use the **device_specific** argument to specify the address of a user-supplied parameter block. The user-supplied parameter block is an integer array of length six supplying information required for the data transfer, such as buffer address and buffer length. See the AAF01, ADF01, or DRQ11-C device-specific section in Chapter 2 for information about using the **device_specific** argument to supply the parameter block.
	AAV11-D
User-supplied input variable	Here you use the **device_specific** argument to write the four digital control lines. Before outputting a buffer, the control lines are set with the complement of the value in the low four bits of the argument. When the buffer transaction is complete, the bits are cleared.

[1]This device is available only in Europe.

Table 3–4 (Cont.): LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	ADQ32
LIO$M_HOLD_DMA LIO$M_DONE_DBL_BUF	When using the ADQ32 device to perform double-buffer DMA transfers, the LIO$M_HOLD_DMA value is included in all LIO$ENQUEUE routine calls to the device to inhibit the start of DMA transfers until all buffers are enqueued. Omitting the LIO$M_HOLD_DMA value on the last LIO$ENQUEUE routine call to the device causes the DMA transfers to begin as soon as the last buffer is enqueued. The LIO$M_DONE_DBL_BUF value signals LIO that you are done double buffering. Include this **device-specific** value when you enqueue the last buffer in a double-buffering sequence. See Section 1.6.3.4, Double-Buffer DMA, for more information.
	AXV11-C
LIO$K_INPUT LIO$K_OUTPUT	Here you use the **device_specific** argument to signal whether a program is to read (LIO$K_INPUT) from the ADC or to write (LIO$K_OUTPUT) to the DACs.
	DRB32
User-supplied input variable	Specifying the **device_specific** argument may be necessary to place the external device in the proper mode for the data transfer. Here you specify the address of a longword whose low-order byte contains the bit pattern you want to write to the output control port. This byte is written to the output control port before the data transfer starts.

Table 3–4 (Cont.): LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	DRQ3B
LIO$M_HOLD_DMA LIO$M_RUN_DOWN	When using the DRQ3B device to perform double-buffer DMA transfers, the LIO$M_HOLD_DMA value is included in all LIO$ENQUEUE routine calls to the device to inhibit the start of DMA transfers until all buffers are enqueued. Omitting the LIO$M_HOLD_DMA value on the last LIO$ENQUEUE routine call to the device causes the DMA transfers to begin as soon as the last buffer is enqueued. The LIO$M_RUN_DOWN value prevents the setting of an event flag or the delivering of an AST routine until the buffer on the output port has been output. This value effectively disables double buffering.
	DRV11-J
User-supplied input variable	An unsigned longword integer. The low word selects the port, where 0 specifies port A, 1 specifies port B, 2 specifies port C, and 3 specifies port D. The high word is a mask that selects the bits of the port. If any bits are set in the high word of the argument, then only those bits are written to output or read from on input. On output, bits not selected are not changed. On input, bits not selected are returned as zeros. If the second word is zero, all bits are written to on output and read from on input. If all bits are to be selected, then this argument can be treated as a normal integer specifying only the port number.

Table 3–4 (Cont.): LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	DRV11-WA
User-supplied output variable	The low three bits of the integer are written to the control lines when the buffer is input (if the board is jumpered for input) or output (if the board is jumpered for output). The actual direction of the device is set by an external control line that can be tied either high or low, or can be controlled by an external instrument control.
	IAV11-A[1], IAV11-AA[1]
User-supplied input variable	The **device_specific** argument is an array of longwords you use to specify information about the A/D channels. See Section 2.4.1.4, Using the IAV11-A for Asynchronous Input, for information about using the **device_specific** argument to read A/D values from the IAV11-A devices.
	IDV11-D[1]
User-supplied input variable	The **device_specific** argument is a structure with three arguments. The first argument is a word that contains the channel number. The second argument is a byte containing the ''disarm'' parameter. The third argument is a byte containing the ''save'' parameter.

[1]This device is available only in Europe.

Table 3–4 (Cont.): LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	IEQ11, IEZ11, IOtech Micro488A

LIO$M_LNR
LIO$M_TKR
	Use the **device_specific** argument to specify the direction of an I/O transfer. If the IEEE-488 device is attached as a controller-in-charge, the **device_specific** argument can also specify the IEEE-488 address of the destination or source device. **This is a required argument.**
	
	When an IEEE-488 device is attached as an instrument, the **device_specific** argument only specifies the direction of the data transfer. LIO$M_TKR causes LIO to perform an asynchronous input. LIO$M_LNR causes LIO to perform an asynchronous output.
	
	When an IEEE-488 device is the controller-in-charge, the **device_specific** argument contains the source or destination address as well as the direction of the data transfer. LIO$M_TKR ORed with a device address causes LIO to read data from the specified device (the device becomes a talker). LIO$M_LNR ORed with a device address causes LIO to write data to the specified device (the device becomes a listener). For example:
	
	`dev_addr = 3`
	`dev_spec = LIO$M_LNR .OR. dev_addr`

Table 3–4 (Cont.): LIO$ENQUEUE Device-Specific Argument Values

Argument Values	Description
	Serial line
LIO$K_INPUT LIO$K_OUTPUT LIO$K_BREAKOUT	Here you use the **device_specific** argument to signal the serial line device to enqueue an input request, to enqueue an output request, or to force a write request over the current read request. During normal operation, an active read request means that an input buffer has been sent to the serial line device driver and is set up to receive any incoming bytes of data from the serial line. Using the **device_specific** argument forces the write request to execute ahead of an active read request. The read request resumes execution on completion of the forced write request.

Description

You can use this routine only when the specified device is set for asynchronous I/O.

If an event flag is supplied, it is set when the buffer transaction completes. The event flag is required by the memory queue device. It defaults to zero for all other devices using QIOs.

To ensure continuous data transfers, several LIO$ENQUEUE calls can be made to queue several buffers ahead.

When a device is attached with QIO, the maximum size of an individual data buffer is 65,534 bytes.

LIO$READ

This routine reads a buffer from an input device set up to use the synchronous I/O interface.

Format

LIO$READ *(device_id, buffer, buffer_length, data_length, [device_specific])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device from which the LIO$READ routine reads the buffer. The **device_id** argument is the address of a signed longword integer that contains this device ID.

buffer
VMS Usage: **array**
type: **array**
access: **write only**
mechanism: **by reference**
Specifies the address of the data buffer. The **buffer** argument is the address of the array which is the data buffer. The user defines the type of array, but it should be a type acceptable to the device.

For the IAV11-A[1] devices, this argument returns information about the data transfer. This argument is an array of longwords. Each longword returns information about one A/D channel. Byte 1 returns the channel number. Byte 2 returns the channel gain. The high word (bytes 3 and 4) returns the actual A/D value.

buffer_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the length of the data buffer in bytes. The **buffer_length** argument is the address of a signed longword integer that contains the length of the data buffer.

For IAV11-A[1] devices, specify **buffer_length** as a multiple of four.

data_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the length of the data buffer in bytes. The **data_length** argument is the address of a signed longword integer into which the LIO facility writes the length of the data buffer.

device_specific
VMS Usage: **longword_unsigned**
type: **longword integer (unsigned)**
access: **device-dependent**
mechanism: **by reference**
Specifies or returns device-specific information about the buffer. When used with certain devices, the **device_specific** argument is the address of an unsigned longword integer containing information required by the device to perform the data transfer. When used with other devices, the **device_specific** argument is the address of an unsigned longword integer to which the LIO facility writes information about the data transfer.

[1] This device is available only in Europe.

The following table lists the devices that support the use of this argument with the LIO$READ routine, the LIO-supplied values of this argument, if any, and the information that is being supplied or returned about the data transfer.

Table 3–5: LIO$READ Device-Specific Argument Values

Argument Values	Description
	AAF01,[1] ADF01,[1] DRQ11-C[1]
User-supplied parameter block	Here you use the **device_specific** argument to specify the address of a user-supplied parameter block. The user-supplied parameter block is an integer array of length six supplying information required for the data transfer, such as buffer address and buffer length. See the AAF01, ADF01, or DRQ11-C device-specific section in Chapter 2 for information about using the **device_specific** argument to supply the parameter block.
	DRB32
User-supplied input variable	Specifying the **device_specific** argument may be necessary to place the external device in the proper mode for the data transfer. Here you specify the address of a longword whose low-order byte contains the bit pattern you want to write to the output control port. This byte is written to the output control port before the data transfer starts.

[1]This device is available only in Europe.

Table 3–5 (Cont.): LIO$READ Device-Specific Argument Values

Argument Values	Description
	DRV11-J
User-supplied input variable	An unsigned longword integer. The low word selects the port, where 0 specifies port A, 1 specifies port B, 2 specifies port C, and 3 specifies port D. The high word is a mask that selects the bits of the port. If any bits are set in the high word of the argument, then only those bits are written to output or read from on input. On output, bits not selected are not changed. On input, bits not selected are returned as zeros. If the second word is zero, all bits are written to on output and read from on input. If all bits are to be selected, then this argument can be treated as a normal integer specifying only the port number.
	DRV11-WA
User-supplied output variable	The low three bits of the integer are written to the control lines when the buffer is transferred. The argument returns the state of the three control lines in bits 0, 1, 2, and the state of the three sense lines in bits 9, 10, and 11.
	IAV11-A,[1] IAV11-AA[1]
User-supplied input variable	The **device_specific** argument is an array of longwords you use to specify information about the A/D channels. See Section 2.4.1.3, Using the IAV11-A for Synchronous Input, for complete information about using the **device_specific** argument to read A/D values from the IAV11-A devices.

[1]This device is available only in Europe.

Table 3–5 (Cont.): LIO$READ Device-Specific Argument Values

Argument Values	Description
	IDV11-D[1]
User-supplied input variable	The **device_specific** argument is a structure with three arguments. The first argument is a word that contains the channel number. The second argument is a byte containing the "disarm" parameter. The third argument is a byte containing the "save" parameter.
	IEQ11, IEZ11, IOtech Micro488A
Talker address	If an IEEE-488 device is the controller-in-charge, you can use this argument to specify a valid IEEE-488 talker address. This causes LIO to address the specified device as a talker before starting the transfer.

[1]This device is available only in Europe.

Description

You can use this routine only when the specified device is set for synchronous I/O. This routine call does not return until the buffer is complete.

When a device is attached with QIO, the maximum size of an individual data buffer is 65,534 bytes.

LIO$SET_I

This routine sets up a device according to a parameter code and any number of integer values.

Format

LIO$SET_I *(device_id, param_code, n_values, [param_values])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device the LIO$SET_I routine is to set up. The **device_id** argument is the address of a signed longword integer that contains this device ID.

param_code
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the parameter code being set. The **param_code** argument is the address of a signed longword integer containing the parameter code.

n_values
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Contains the number of parameter code values being set. The **n_values** argument is the address of a signed longword integer containing this number.

param_values
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Contains zero or more values that are the parameter code values to set. The values are separated by commas. The **param_values** argument is the address of a signed longword integer that contains these values.

Description

See Chapter 2 for listings of the valid parameters for each device.

See Chapter 4 for information about valid parameter codes and parameter-code values.

LIO$SET_R

This routine sets up a device according to a parameter code and any number of real values.

Format

LIO$SET_R *(device_id, param_code, n_values, [param_values])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device the LIO$SET_R routine is to set up. The **device_id** argument is the address of a signed longword integer that contains this device ID.

param_code
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the parameter code being set. The **param_code** argument is the address of a signed longword integer that contains the parameter code.

n_values
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the number of parameter code values being set. The **n_values** argument is the address of a signed longword integer that contains this number.

param_values
VMS Usage: **floating point**
type: **F_floating**
access: **read only**
mechanism: **by reference**
Contains zero or more values that are the parameter code values to set. The values are separated by commas. The **param_values** argument is the address of a signed longword integer that contains these values.

Description

See Chapter 2 for listings of the valid parameters for each device.

See Chapter 4 for information about valid parameter codes and parameter-code values.

LIO$SET_S

This routine sets up a device according to a parameter code and a character-string value.

Format LIO$SET_S *(device_id, param_code, string)*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device the LIO$SET_S routine is to set up. The **device_id** argument is the address of a signed longword integer that contains this device ID.

param_code
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the parameter code being set. The **param_code** argument is the address of a signed longword integer containing the parameter code.

string
VMS Usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**
The character string value being set. The **char_string** argument is the address of a string descriptor pointing to the character string.

Description

See Chapter 2 for listings of the valid parameters for each device.

See Chapter 4 for information about valid parameter codes and parameter-code values.

LIO$SHOW

This routine returns the current values of a specified parameter.

Format

LIO$SHOW *(device_id, param_code, value_list, list_length)*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device about which the LIO$SHOW routine is to show information. The **device_id** argument is the address of a signed longword integer that contains this device ID.

param_code
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the parameter code whose values you want the LIO$SHOW routine to return. The **param_code** argument is the address of a signed longword integer that contains the parameter code.

value_list
VMS Usage: **varying_arg**
type: **unspecified**
access: **write only**
mechanism: **by reference**
Returns the values of the parameter you want shown. The **value_list** argument is the address of a floating-point variable, a string, or a longword in which the LIO facility writes the current values of the parameter. The data type of **value_list** must be appropriate for the expected data. For example, specify an integer if an integer is expected; specify a string if a string is expected.

list_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **write only**
mechanism: **by reference**
Returns the number of elements in **value_list**. The **list_length** argument is the address of a signed longword integer into which the LIO facility writes the number. This can be the number of longwords, the number of single-precision, floating-point values, the length of the string, or (for a serial poll in IEEE-488 devices) the number of bytes.

Description

For any device, any parameter value that can be set can also be shown.

In addition, some devices have parameter values that cannot be set but can be shown, such as the state of an external sense line. Use the LIO$SHOW routine to return any parameter value appropriate for use with a device.

LIO$WRITE

This routine writes a buffer to an output device set up to use the synchronous I/O interface.

Format LIO$WRITE *(device_id, buffer, data_length, [device_specific])*

Returns

VMS Usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

Arguments

device_id
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the device ID of the device to which the LIO$WRITE routine is to write the buffer. The **device_id** argument is the address of a signed longword integer that contains this device ID.

buffer
VMS Usage: **array**
type: **array**
access: **read only**
mechanism: **by reference**
Specifies the address of the data buffer. The **buffer** argument is the address of the array which is the data buffer. The user defines the type of array, but it should be a type acceptable to the device.

data_length
VMS Usage: **longword_signed**
type: **longword integer (signed)**
access: **read only**
mechanism: **by reference**
Specifies the length of the data buffer in bytes. The **data_length** argument is the address of a signed longword integer that contains the length of the data buffer.

device_specific
VMS Usage: **longword_unsigned**
type: **longword integer (unsigned)**
access: **device-dependent**
mechanism: **by reference**
Specifies or returns device-specific information about the buffer.

When used with certain devices, the **device_specific** argument is the address of an unsigned longword integer that contains information required by the device to perform the data transfer.

When used with other devices, the **device_specific** argument is the address of an unsigned longword integer to which the LIO facility writes information about the data transfer.

The following table lists the devices that support the use of this argument with the LIO$WRITE routine, the LIO-supplied values of this argument, if any, and the information that is being supplied or returned about the data transfer.

Table 3–6: LIO$WRITE Device-Specific Argument Values

Argument Values	Description
	AAF01,[1] ADF01,[1] DRQ11-1[1]
User-supplied parameter block	Here you use the **device_specific** argument to specify the address of a user-supplied parameter block. The user-supplied parameter block is an integer array of length six supplying information required for the data transfer, such as buffer address and buffer length. See the AAF01, ADF01, or DRQ11-C device-specific section in Chapter 2 for information about using the **device_specific** argument to supply the parameter block.
	AAV11-D
User-supplied input variable	Here you use the **device_specific** argument to write the four digital control lines. Before outputting a buffer, the control lines are set with the value in the low four bits of the argument. When the buffer transaction is complete, the bits are cleared.
	DRB32
User-supplied input variable	Here you specify the address of a longword whose low order byte contains the bit pattern you want to write to the output control port. This byte is written to the output control port before the data transfer starts. Specifying the **device_specific** argument may be necessary to place the external device in the proper mode for the data transfer.

[1]This device is available only in Europe.

Table 3–6 (Cont.): LIO$WRITE Device-Specific Argument Values

Argument Values	Description
	DRV11-J
User-supplied input variable	An unsigned longword integer. The low word selects the port, and the high word is a mask that selects the bits of the port. If any bits are set in the high word of the argument, then only those bits are written to on output. The bits not selected are not changed. If the second word is zero, all bits are written to on output. If all bits are to be selected, then this argument can be treated as a normal integer containing only the port number.
	DRV11-WA
User-supplied output variable	The low three bits of the integer are written to the control lines when the buffer is transferred. The argument returns the state of the three control lines in bits 0, 1, 2, and the state of the three sense lines in bits 9, 10, and 11.
	IEQ11, IEZ11, IOtech Micro488A
Listener address	If an IEEE-488 device is the controller-in-charge, you can use this argument to specify a valid IEEE-488 address. This causes LIO to address the specified device as a listener before starting the transfer.

Description

You can use this routine only when the specified device is set for synchronous I/O. This routine call does not return until the buffer is empty.

When a device is attached with QIO, the maximum size of an individual data buffer is 65,534 bytes.

Chapter 4

LIO$SET and LIO$SHOW Parameter Reference Descriptions

This chapter presents an overview and detailed reference descriptions of the LIO$SET and LIO$SHOW parameters you use to set up and display I/O device parameters.

The LIO parameters are presented in alphabetical order and described using the following format:

- **Supported Devices** lists the LIO devices with which the parameter can be used.
- **Parameter Values** describes the data type and number of values the parameter accepts.
- **Description** contains information about the specific actions taken by the parameter. This information includes interactions or dependencies between the parameter and other LIO parameters, and actions specific to the parameter when used with certain devices.
- **Restrictions** lists those things you should consider when you set up a device using the parameter. Restrictions include the limitations imposed on the device by setting it up using the parameter, and other parameters that must be used in conjunction with this parameter to set up the device properly.
- **Example** contains one or more programming examples to illustrate the use of the parameter. An explanation follows each example.

NOTE

All programming examples in this chapter are presented in VAX FORTRAN (except where noted).

The following table summarizes the LIO$SET and LIO$SHOW parameters.

Table 4-1: LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_ACK_NAK_TERMINATOR	Establishes a termination character for the ACK/NAK string received from an external device.
LIO$K_AD_CHAN	Specifies the A/D channels to use for input.
LIO$K_AD_DIFFERENTIAL	Specifies whether the A/D channels set up with the LIO$K_AD_CHAN parameter use single-ended or differential input.
LIO$K_AD_GAIN	Specifies the amount of amplification or gain applied to each A/D channel specified for use.
LIO$K_ADD_AD_CHAN	Specifies that an additional A/D channel be added to the Preston device's current A/D channel list.
LIO$K_ANA_OUT	Outputs a voltage value to one of the D/A channels on the AAF01[1] device.
LIO$K_AST_RTN	Specifies a user-written AST routine to receive buffers when a device finishes processing them.
LIO$K_ASYNCH	Sets up a device for asynchronous I/O
LIO$K_AUX_COMMAND	Sends an auxiliary command to an IEEE-488 device.
LIO$K_BAUD_RATE	Sets the speed at which data is transmitted over a serial line.
LIO$K_BIN_DDR	Moves a complementary offset binary-coded output voltage into the DAC Data Register (DDR) of the ADF01[1] device.
LIO$K_BITS_PER_CHAR	Establishes the number of data bits per character for serial line devices.
LIO$K_BOUNCE	Sets the contact bounce elimination response time delay for the IDV11-A[1] device.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_BREAK	Generates a break (spacing) condition on a terminal line for a specified amount of time.
LIO$K_BUFF_SIZE	For the ADQ32, DRQ3B, and Preston, sets the maximum allowable buffer size. For the memory queue device, sets the size of the buffers to allocate.
LIO$K_BUFF_SOURCE	Specifies the source from which memory allocation is to occur.
LIO$K_BURST_DIV	Establishes the divisor for the internal burst rate clock of the Preston device.
LIO$K_BURST_RATE	Specifies the rate of the internal burst rate clock of the Preston device.
LIO$K_CANCEL	Cancels all pending I/O requests on the specified channel and stops continuous DMA for the AAF01,[1] ADF01,[1] and DRQ11-C[1] devices.
LIO$K_CC_FOUT	Sets the Frequency Output (FOUT) reference signal for the IDV11-D[1] counter.
LIO$K_CC_SETUP	Sets up the operating characteristics of a channel on the IDV11-D[1] counter.
LIO$K_CHANNEL	For the AAF01,[1] specifies the channel to be used for output. For the ADF01,[1] specifies the channel to be use for input.
LIO$K_CLK_BASE	Sets up the base crystal frequency of the Preston internal clock.
LIO$K_CLK_DIV	Specifies the sampling rate for the Preston internal clock.
LIO$K_CLK_RATE	Takes an ideal frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_CLK_SRC	Specifies the source frequency and divider for clock ticks.
LIO$K_CLR_LBO	Clears the large buffer overflow condition on the AAF01,[1] ADF01,[1] and DRQ11-C[1] devices.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_COB	Reads or writes the Command Output (COUT) bit in the Command and Status Register (CSR) of the AAF01[1] and ADF01[1] devices.
LIO$K_COMMAND	Sends the specified IEEE-488 commands on the bus.
LIO$K_CONT	Sets up the device to perform continuous direct memory access data transfers.
LIO$K_COUNTER	Reads the count register of the Simpact RTC01. of the KWV11-C does not allow reading of the count register while the clock is operating.
LIO$K_CTA	Reads or writes the Control Table Address (CTA) register of the AAF01[1] and ADF01[1] devices.
LIO$K_CTI_BUF	Specifies the buffer and event flag for an AXV11-C device that is attached to use connect-to-interrupt I/O.
LIO$K_CTI_OVERHD	Returns the size in bytes of the connect-to-interrupt overhead.
LIO$K_CTRL_ACTIVE	Activates the IEEE-488 device controller function.
LIO$K_CTRL_AST	Assigns a user-written AST routine to be called when an external device writes data to the input control port of the DRB32, or on receipt of a specified control character for serial line devices.
LIO$K_CTRL_HANDLING	Sets up a flag that indicates what action to take on receipt of a control character specified by the LIO$K_CTRL_AST parameter for serial line devices.
LIO$K_CTRL_STANDBY	Deactivates the IEEE-488 bus controller function.
LIO$K_CURRENT_CHANNEL	Specifies which channel is to be affected by channel-specific set calls.
LIO$K_CWT	Reads the Control Word Registers from, or writes the Control Word Registers to, the AAF01[1] and ADF01[1] devices.
LIO$K_DA_CHAN	Specifies the D/A channels to use for output.
LIO$K_DATA	Performs a data transfer to the parallel data path of the DRB32 without using direct memory access.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_DATA_PATH	Selects the data path and channel number for the AAF01,[1] ADF01,[1] and DRQ11-C[1] devices.
LIO$K_DATA_WIDTH	Sets the width of the data path for DRB32 devices.
LIO$K_DBL_BUF	Enables double-buffer DMA data transfers for the ADQ32 device.
LIO$K_DEVICE_ACK_NAK_BUFF	Supplies the buffer to be used when receiving an ACK or a NAK from a device.
LIO$K_DEVICE_EF	Sets the event flag for output buffer available.
LIO$K_DIAG_CHAN	Enables or disables diagnostic inputs to ADQ32 channels 0, 1, and 2.
LIO$K_DIRECTION	Sets the direction (input or output) of the four DRV11-J ports, signals to the LIO facility the direction (input or output) in which the DRV11-WA hardware is set, sets the direction of the DRB32 data path, or sets the direction of disk file devices.
LIO$K_DISPLAY_ONLY	Sets an interprocess memory queue to display data buffers to a second process.
LIO$K_DRX_AST_RTN	Specifies a user-written AST routine to receive buffers when an AAF01,[1] ADF01,[1] or DRQ11-C[1] finishes processing them.
LIO$K_DRX_STAT	Returns the status of the DRQ11-C[1] device.
LIO$K_DUPLEX	Specifies whether serial line read/write requests are executed in half-duplex or full-duplex mode.
LIO$K_ECHO	Enables or disables the echoing of characters received on a serial line.
LIO$K_ED_CTT	For the AAF01,[1] enables or disables the Memory Transfer (MET) bit in the Command and Status Register (CSR). For the ADF01,[1] enables or disables the Control Table Transfer (CTT) bit in the Command and Status Register (CSR).
LIO$K_ED_ECE	Enables or disables the External Clock Enable (ECE) bit in the Command and Status Register (CSR) of the AAF01[1] and ADF01[1] devices.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_ED_SBE	Enables or disables the Sequence Break Enable (SBE) bit in the Command and Status Register (CSR) of the AAF01[1] and ADF01[1] devices.
LIO$K_EOI	Enables or disables the assertion of the end-or-identify (EOI) line after the last byte of data is output.
LIO$K_ERR_HANDLE	Specifies the way in which a device returns error conditions.
LIO$K_ERROR_ENABLE	Enables or disables parity error handling for serial line devices.
LIO$K_EVENT_AST	Assigns a user-written AST routine to be called on AAF01,[1] ADF01,[1] and DRQ11-C[1] unsolicited interrupts, DRV11-J port A bit events, IDV11-A[1] channel 15 events, IEEE-488 bus events, and KWV11-C or Simpact RTC01 clock overflows or ST2 events.
LIO$K_EVENT_EF	Specifies the event flag to set on an external event or clock overflow.
LIO$K_EVENT_ENA	Enables the recognition of IEEE-488 bus events.
LIO$K_EVENT_WAIT	Waits for an IEEE-488 bus event to occur.
LIO$K_FILE_EXTENT	Sets the size, in blocks, by which an output file can be extended if necessary.
LIO$K_FILE_POS	Repositions the current block pointer in a file.
LIO$K_FILE_REMAIN	Returns the number of blocks remaining to be written in an output file.
LIO$K_FILE_SIZE	Establishes the size in blocks of the output file.
LIO$K_FLOW_CONTROL	Establishes the method of flow control for a serial line device.
LIO$K_FLOW_MASTER	Specifies how LIO uses XON/XOFF flow control.
LIO$K_FORWARD	Specifies the device to which completed buffers are forwarded.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_FUNCTION	Specifies the function the KWV11-C or Simpact RTC01 clock device is to perform.
LIO$K_FUNCTION_BITS	Sets the function bits associated with the AAF01,[1] ADF01,[1] DRQ11-C,[1] DRB32, and DRQ3B devices.
LIO$K_GATE	Specifies the type of external gating used with the ADQ32 device.
LIO$K_HANDSHAKE	Specifies whether or not the DRV11-J is jumpered to use a two-wire handshake for each port.
LIO$K_HANGUP	Disconnects a terminal that is on a dial-up line.
LIO$K_IEEE_ADDR	Sets the IEEE-488 bus address of an IEEE-488 device.
LIO$K_INIT_AD_CHAN	Initializes or clears the existing Preston A/D channel list.
LIO$K_INPUT_TERMINATOR	Specifies a termination character or characters on the input side of a serial port.
LIO$K_INTERRUPT_LEVEL	Sets the level at which interrupts occur for the Simpact RTC01.
LIO$K_LEAVE_IN_STATE	Specifies whether or not to leave an IEQ11 device in the state required to process the subsequent I/O request.
LIO$K_LOCK_BUFFER	Locks buffers before beginning direct memory access transfers with the DRB32 device.
LIO$K_LOOP_BACK	Enables or disables loopback mode.
LIO$K_MAX_CHANNELS	Specifies the maximum number of channels that can be plotted using the real-time plotting device.
LIO$K_MODEM	Specifies that the serial line is a modem.
LIO$K_MODEM_STATUS	Sets and returns modem status.
LIO$K_MULTIPLE_X_AXES	Specifies the x-axis representation for the plotting window.
LIO$K_N_AD_CHAN	Returns the number of A/D channels currently in use.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_N_BUFFS	Sets the number of buffers to allocate for the memory queue device, and specifies the number of channels in the data buffer for the real-time plotting device.
LIO$K_N_DA_CHAN	Returns the number of D/A channels currently in use.
LIO$K_NAME	Specifies the name of a file or a global section.
LIO$K_OPEN_FILE	Opens a file.
LIO$K_OUTPUT_PREFIX	Specifies a prefix character string on the output side of a serial line.
LIO$K_OUTPUT_TERMINATOR	Specifies a suffix character string on the output side of a serial line.
LIO$K_PAGE_ALIGN	Page-aligns buffers allocated for use with the memory queue device.
LIO$K_PAR_POLL	Performs a parallel poll of IEEE-488 bus instruments.
LIO$K_PAR_POLL_CONFIG	Sets up the list of IEEE-488 instruments for parallel polling.
LIO$K_PAR_POLL_STATUS	Sets up a parallel poll status register for an IEEE-488 instrument.
LIO$K_PARITY	Specifies the parity checking type for a serial line.
LIO$K_PASS_CTRL	Passes control to another IEEE-488 bus device.
LIO$K_PCR	Specifies the number of steps in the Programmable Clock Register (PCR) of the AAF01[1] and ADF01[1] devices.
LIO$K_PLOT_SIZE	Establishes the size of the plotting window.
LIO$K_PLOT_TYPE	Specifies the style of plotting.
LIO$K_PO_CHAN	Specifies the channel numbers to be plotted.
LIO$K_POLARITY	Sets the bits of port A to call their AST routines on either a negative-going or positive-going edge and the polarity of the handshake, if any.
LIO$K_POSITION	Establishes the position of the plotting window on the display surface.

[1] This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_PROTOCOL	Enables or disables the serial line user-defined protocol feature.
LIO$K_PURGE	Purges all characters in the type-ahead buffer.
LIO$K_READ_ONLY	Establishes read-only access to a global section.
LIO$K_READ_PROMPT	Specifies a read prompt to prefix each input data byte.
LIO$K_READ_STAT	Returns the status of the read-only bits in the Command and Status Register (CSR) of the AAF01[1] and ADF01[1] devices.
LIO$K_RESET_AXF	Resets the AAF01[1] and ADF01[1] devices.
LIO$K_RESET_DRX	Resets the DRQ11-C[1] direct memory access (DMA) interface.
LIO$K_SCHMITT_TRIGGER	Sets the mode of operation for the two Schmitt triggers on the Simpact RTC01.
LIO$K_SER_POLL	Serial polls a predetermined list of IEEE-488 instruments.
LIO$K_SER_POLL_CONFIG	Sets up the list of IEEE-488 instruments to serial poll.
LIO$K_SGL_BUF	Sets the device to stop direct memory access between buffers. Transfer is not continuous.
LIO$K_SKIP_COUNT	Specifies how many points are to be skipped and not plotted.
LIO$K_SRQ	Defines a serial poll status byte for an IEEE-488 device and, optionally, sends a service request to the controller-in-charge.
LIO$K_ST0_1	Writes to the 23-bit counter contained in the Sequence Timer Registers ST0 and ST1 of the AMF01[1] device.
LIO$K_START	Starts the device.
LIO$K_STAT_BITS	Returns status information about the DRQ11-C[1] device.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_STE	Clears the Sequence Timer Enable (STE) in the AMF01[1] Sequence Timer Register (ST1).
LIO$K_STOP	Stops the device.
LIO$K_SWEEP_RATE	For the ADQ32 sweep rate clock, takes an ideal frequency and produces the best internal crystal rate and divider to approximate that frequency.
LIO$K_SYNCH	Sets up a device for synchronous I/O.
LIO$K_TERM_CHAR	Defines a termination character to mark the end of data received.
LIO$K_TERM_SRQ	Enables or disables terminations of I/O transfers by a service request.
LIO$K_TIMEOUT	Sets the length of time (in seconds) before an I/O request is aborted.
LIO$K_TIMEOUT_ENABLE	Enables or disables the timeout for read requests.
LIO$K_TITLE	Specifies the graph title for the current channel.
LIO$K_TITLE_n	Specifies the title for the graph of each channel plotted.
LIO$K_TRANSFER	Sets an interprocess memory queue to transfer data buffers between processes.
LIO$K_TRIG	Sets the device trigger mode or source.
LIO$K_TYPE_AHEAD	Enables or disable a serial line type-ahead buffer.
LIO$K_UNLOCK_BUFFER	Unlocks buffers previously locked with the LIO$K_LOCK_BUFFER parameter.
LIO$K_UNSOLICITED	Returns the number of characters in the type-ahead buffer.
LIO$K_UPDATE	Updates the Preston device to the current set-up specifications.
LIO$K_USER_ACK_AST	Specifies the address of a user-supplied AST routine that transmits an ACK message on successful completion of a data transfer.
LIO$K_USER_ACK_STRING	Supplies the ACK string to be sent out by an AST routine on successful completion of a data transfer.

[1]This device is available only in Europe.

Table 4–1 (Cont.): LIO$SET and LIO$SHOW Parameter Summary

Parameter	Function
LIO$K_USER_NAK_AST	Specifies the address of a user-supplied AST routine to transmit the NAK string on unsuccessful completion of a data transfer.
LIO$K_USER_NAK_STRING	Supplies the NAK string to be sent out by an AST routine on unsuccessful completion of a data transfer.
LIO$K_USER_READ_PROTOCOL_AST	Specifies the address of a user-supplied AST routine to be called on receipt of either a terminator or a full buffer of characters from a read request.
LIO$K_USER_WRITE_NAK_HANDLING	Specifies whether or not a sending device attempts to retransmit a buffer after receiving a NAK from the intended receiving device.
LIO$K_VOLTAGE	Specifies the input voltage range for the IDV11-A[1] device.
LIO$K_VLT_DDR	Converts a voltage into its corresponding complementary binary-coded value and moves it to the DAC Data Register (DDR) of the ADF01[1] device.
LIO$K_X_LABEL	Specifies the label to be placed on the x-axis of the channel specified by LIO$K_CURRENT_CHANNEL.
LIO$K_X_RANGE	Specifies the number of points to display along the x-axis, and the increment at which points are plotted.
LIO$K_XON	Forces the sending of an XON character to reprime the serial line.
LIO$K_Y_LABEL	Specifies the label to be placed on the y-axis of the channel specified by LIO$K_CURRENT_CHANNEL.
LIO$K_Y_MAX	Sets the maximum y-axis value for each channel to be plotted.
LIO$K_Y_MIN	Sets the minimum y-axis value for each channel to be plotted.

[1]This device is available only in Europe.

LIO$K_ACK_NAK_TERMINATOR

This parameter establishes a termination character for the ACK/NAK string received from an external device. This parameter is used only for the user-defined protocol for serial line devices.

Supported Devices

Serial line (when used with user-defined protocol)

Parameter Values

A character string specifying the valid termination character.

This value is passed by descriptor.

Description

This parameter defines the termination character for an ACK/NAK string received from an external device after a write operation.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

You can specify only one termination character for use at any given time. If you change the termination character, the new termination character supercedes any previously specified termination character. This parameter is only used with user-defined protocols for serial line devices.

Example

```
status = LIO$SET_S (serial_id, LIO$K_ACK_NAK_TERMINATOR, 'A')
```

This routine specifies the letter A as the valid termination character for ACK/NAK strings.

LIO$K_AD_CHAN

This parameter specifies the A/D channels to use for input.

Supported Devices

ADQ32
ADV11-D
AXV11-C A/D
Preston

Parameter Values

One or more longword integers specifying the A/D channels to use.

Description

When the ADQ32 is set for differential input (using the LIO$K_AD_DIFFERENTIAL parameter), only channels 0 through 15 are available. Channel numbers can be specified in any order. The channels are sampled in the order specified. See Appendix A for more information.

When the ADV11-D is attached for mapped I/O, you can specify up to 16 channels in any order. The channels can repeat. When the ADV11-D is attached for QIOs, you must specify either one channel or all channels in ascending order. When the device is jumpered for single-ended input, channels 0 to 15 are available for use. When the device is jumpered for differential input, channels 0 through 7 are available for use.

When the AXV11-C is attached for mapped I/O, you can specify up to 16 channels in any order. The channels can repeat. When the AXV11-C is attached for QIOs, you can specify up to 16 channels in ascending order. When the device is jumpered for single-ended input, channels 0 to 15 are available for use. When the device is jumpered for differential input, channels 0 through 7 are available for use.

Restrictions

The following restrictions apply only to the ADV11-D and the AXV11-C:

- When the device is attached for QIOs, you must specify the channels in ascending order (see the **Description**).
- No buffers can be currently enqueued to the device when you set up the device using this parameter.

The following restriction applies only to Preston devices:

- Use this parameter to set up Preston A/D channels only when you do not intend to use either or both the LIO$K_INIT_AD_CHAN and LIO$K_ADD_AD_CHAN parameters. The use of the LIO$K_AD_CHAN parameter with these Preston-specific parameters is mutually exclusive.

Example

```
status = LIO$SET_I (device_id, LIO$K_AD_CHAN, 5, 0, 1, 2, 3, 4)
```

This routine specifies five A/D channels, channels 0 through 4 on the device, for use.

LIO$K_AD_DIFFERENTIAL

This parameter specifies whether the A/D channels set up with the LIO$K_AD_CHAN parameter use single-ended or differential input.

Supported Devices

ADQ32

Parameter Values

One or more longword integer constants that enable single-ended or differential input for each A/D channel set up with the LIO$K_AD_CHAN parameter.

The values can be one of the following:

Constant Value	Meaning
LIO$K_OFF[1]	The channel is single-ended.
LIO$K_ON	The channel is differential.

[1]The default value.

Description

If all of your analog inputs are connected in single-ended mode, you do not need to use this parameter.

Each ADQ32 channel can be individually set for single-ended or differential input. Only channels 0 through 15 can be set for differential input.

When a channel is set for single-ended input, the difference in voltage between the signal line and analog ground is measured as data. Connect the analog ground for the device on that channel to the analog ground input.

LIO$K_AD_DIFFERENTIAL

When a channel is set for differential input, the signal is returned as the difference between two lines. The first line is connected to the channel input (only channels 0 through 15 are legal for differential input). The second line is connected to a channel number equal to the first channel plus 16. For example, if channel 3 is set for differential input, then the first line is connected to channel 3; the second line is connected to channel 19 (3 + 16).

Restrictions

- Only channels 0 through 15 can be set for differential input.
- When a channel is set for differential input, that channel number plus 16 must be used as the other channel of the differential pair.

Examples

1.
```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 5, 0, 1, 2, 20, 21)
```

   This routine specifies five A/D channels, channels 0 through 2 and channels 20 and 21 for input.

2.
```
status = LIO$SET_I (adq_id, LIO$K_AD_DIFFERENTIAL, 5, LIO$K_ON,
1                LIO$K_ON, LIO$K_ON, LIO$K_OFF, LIO$K_OFF)
```

   This routine sets channels 0, 1, and 2 for differential input, and channels 20 and 21 for single-ended input. Channels 0, 1, and 2 use channels 16, 17, 18, respectively, as the second input channel of the differential pair. Channels 20 and 21 use analog ground as the signal return.

LIO$K_AD_GAIN

This parameter specifies the amount of amplification or gain applied to each A/D channel specified for use.

Supported Devices

ADQ32
ADV11-D
AXV11-C A/D

Parameter Values

One or more longword integers specifying the A/D channel gains to use. Each gain can be 1, 2, 4, or 8, representing the amount of amplification applied to the input signal before conversion.

The default value is 1.

Description

See the **Parameter Values**.

Restrictions

The following restrictions apply to the ADQ32:

- You must specify the channels to use through the LIO$K_AD_CHAN parameter before you specify the channel gains to use.
- You must specify one gain per channel.
- No buffers can be queued to the device when you specify the channel gain.

LIO$K_AD_GAIN

The following restrictions apply to both the ADV11-D and the AXV11-C A/D:

- You must specify the channels to use through the LIO$K_AD_CHAN parameter before you specify the channel gains to use.
- You must specify one gain per channel.
- When the device is attached with QIO, only the first gain is used.

Example

```
status = LIO$SET_I (device_id, LIO$K_AD_GAIN, 5, 1, 1, 1, 1, 1)
```

This routine specifies a gain of one for each A/D channel specified for use in the **Example** presented in the reference description of the LIO$K_AD_CHAN parameter.

LIO$K_ADD_AD_CHAN

This parameter specifies one additional A/D channel to be added to the current A/D channel list of the Preston device.

Supported Devices

Preston

Parameter Values

A longword integer specifying the A/D channel to add to the channel list.

Description

This parameter enables you to add A/D channels to the Preston device channel list one at a time. This capability is desirable when you are creating an interactive application where the entire channel list is unknown at the beginning of your program. You can also use this parameter to add channels to the end of a channel list you previously specified through the LIO$K_AD_CHAN parameter.

Restrictions

- Use this parameter only after you have either:
  - Initialized a channel list using the LIO$K_INIT_AD_CHAN parameter
  - Created a channel list using the LIO$K_AD_CHAN parameter
- You can add only one channel per LIO$SET_I routine call.

Example

```
status = LIO$SET_I (device_id, LIO$K_ADD_AD_CHAN, 1, 1)
```

This routine adds an A/D channel, channel 1 on the device, to an initialized or existing channel list for a Preston device.

LIO$K_ANA_OUT

This parameter outputs a voltage value to a specified D/A channel.

Supported Devices

AAF01[1]

Parameter Values

The first value is a single-precision floating-point real value specifying the volts to be output.

The second value is a longword integer specifying the D/A channel to which you want the volts output. This value can be between 0 and 15, inclusive.

Description

Use this parameter to write a voltage value to a single channel.

Restrictions

None.

Example

```
REAL*4 real_values(2)

 real_values(1) = volt_value
 real_values(2) = channel_number

 status = LIO$SET_R (aaf_id, LIO$K_ANA_OUT, 2, real_values(1),
1            real_values(2))
```

This routine sends the voltage value specified by **volt_value** to the AAF01 channel specified by **channel_number**.

[1] This device is available only in Europe.

LIO$K_AST_RTN

This parameter specifies a user-written AST routine to receive buffers when a device finishes processing them. The completed buffers are passed to the AST routine instead of being placed on the device's user queue.

Supported Devices

AAV11-D
ADQ32
ADV11-D
AXV11-C A/D
DRB32
DRB32W
DRQ3B
DRV11-J
DRV11-WA
IAV11[1] devices
IDV11[1] devices
IEQ11
IEZ11
KWV11-C
Preston
Simpact RTC01
Disk file
Memory queue
Serial line

Parameter Values

A longword integer specifying the address of the AST routine.

[1] These devices are available only in Europe.

Description

See Chapter 3 in *Getting Started with VAXlab* and Section 1.5.3, Asynchronous System Traps (ASTs), in this guide for more information about using AST routines.

Restrictions

- The device must be set for asynchronous I/O.
- When using FORTRAN, the subroutine must be declared EXTERNAL. See the **Example**.

Examples

1. `EXTERNAL user_ast`

   This line declares the user-written AST routine to be external. See the **Restrictions**.

2. `status = LIO$SET_I (device_id, LIO$K_ASYNCH, 0)`

   This routine sets up the device to use asynchronous I/O.

3. `status = LIO$SET_I (device_id, LIO$K_AST_RTN, 1, user_ast)`

   This routine specifies a user-written AST routine, **user_ast**, to receive completed buffers.

LIO$K_ASYNCH

This parameter sets up a device to use the asynchronous I/O interface.

Supported Devices

AAF01[1]
AAV11-D
ADF01[1]
ADQ32
ADV11-D
AXV11-C
DRB32
DRB32W
DRQ11-C[1]
DRQ3B
DRV11-J
DRV11-WA
IAV11[2] devices
IDV11-A[2] devices
IEQ11
IEZ11
KWV11-C
Preston
Simpact RTC01
Disk file
Memory queue
Serial line

Parameter Values

None.

[1] This device is available only in Europe.

[2] These devices are available only in Europe.

Description

Use this parameter when you have already attached a device with the synchronous I/O interface, and you want to use the asynchronous I/O interface.

By default, all devices are set to use the asynchronous I/O interface when you attach them with the LIO$ATTACH routine specifying LIO$K_QIO as the value of the **io_type** argument. If the device is already set to use the asynchronous I/O interface, using this LIO$SET_I parameter has no effect.

When you attach both a memory queue device and a disk file, you use the **io_type** argument to specify the function these devices are to perform within the context of the program. By default, these devices use the asynchronous I/O interface. So, while this parameter is valid for use with these devices, neither device needs to be explicitly set up using this parameter.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_ASYNCH, 0)
```

This routine sets up the device to use asynchronous I/O.

LIO$K_AUX_COMMAND

This parameter sends a specified auxiliary command to an IEEE-488 device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword containing a valid IEEE-488 auxiliary command.

For the **IEQ11** and the **IEZ11**, the auxiliary command can be one of the following:

Table 4–2: IEQ11 and IEZ11 IEEE-488 Auxiliary Commands

Auxiliary Command	Function
LIO$K_DACR_ON	Release ACDS holdoff
LIO$K_DACR_OFF	Disable release ACDS holdoff
LIO$K_DAI_ON	Disable all interrupts
LIO$K_DAI_OFF	Enable all interrupts
LIO$K_FEOI	Send EOI with next byte
LIO$K_FGET_ON	Force group execute trigger
LIO$K_FGET_OFF	Disable force group execute trigger
LIO$K_GTS	Go to standby
LIO$K_HDFA_ON	Holdoff on all data
LIO$K_HDFA_OFF	Disable holdoff on all data
LIO$K_HDFE_ON	Holdoff on EOI only
LIO$K_HDFE_OFF	Disable holdoff on EOI only
LIO$K_LON_ON	Listen only
LIO$K_LON_OFF	Disable listen only
LIO$K_NBAF	Set new byte available false
LIO$K_PTS	Pass through next secondary
LIO$K_RHDF	Release RFD holdoff
LIO$K_RLC	Release control

Table 4–2 (Cont.): IEQ11 and IEZ11 IEEE-488 Auxiliary Commands

Auxiliary Command	Function
LIO$K_RPP_ON	Request parallel poll
LIO$K_RPP_OFF	Disable request parallel poll
LIO$K_RQC	Request control
LIO$K_RSV2_ON	Request service bit 2
LIO$K_RSV2_OFF	Disable request service bit 2
LIO$K_RTL_ON	Return to local
LIO$K_RTL_OFF	Disable return to local
LIO$K_SHDW_ON	Shadow handshake
LIO$K_SHDW_OFF	Disable shadow handshake
LIO$K_SIC_ON	Send interface clear
LIO$K_SIC_OFF	Disable send interface clear
LIO$K_SRE_ON	Send remote enable
LIO$K_SRE_OFF	Disable send remote enable
LIO$K_STDW_ON	Short T1 settling time
LIO$K_STDW_OFF	Disable short T1 settling time
LIO$K_SWSRST_ON	Chip reset
LIO$K_SWSRST_OFF	Disable chip reset
LIO$K_TCA	Take control asynchronously
LIO$K_TCS	Take control synchronously
LIO$K_TON_ON	Talk only
LIO$K_TON_OFF	Disable talk only
LIO$K_VSTD1_ON	Very short T1 delay
LIO$K_VSTD1_OFF	Disable very short T1 delay

For the **IOtech Micro488A**, the auxiliary command can be one of the following:

Table 4–3: IOtech Micro488A IEEE-488 Auxiliary Commands

Auxiliary Command	Function
LIO$K_FGET_ON	Force group execute trigger
LIO$K_SIC_ON	Assert interface clear for 500 microseconds
LIO$K_SRE_ON	Assert remote enable
LIO$K_SRE_OFF	Return to local
LIO$K_SWSRST_ON	Hardware reset

LIO$K_AUX_COMMAND

Description

Use this parameter to send one auxiliary command to an IEEE-488 device. See the *IEU11-A/IEQ11-A User's Guide* for information about IEEE-488 auxiliary commands.

Note that with the IOtech Micro488A, the functionality of the auxiliary commands is slightly different from the IEQ11 and IEZ11 functionality.

Restrictions

- The device sending the auxiliary command must be the controller-in-charge.
- You can send only one auxiliary command per routine call.

Examples

1. `status = LIO$SET_I (ieee_id, LIO$K_AUX_COMMAND, 1, LIO$K_SRE_ON)`

   This routine asserts remote enable on an IEEE-488 bus.

2. `status = LIO$SET_I (ieee_id, LIO$K_AUX_COMMAND, 1, LIO$K_SRE_OFF)`

   This routine deasserts remote enable on an IEEE-488 bus.

LIO$K_BAUD_RATE

This parameter sets the speed at which data is transmitted over a serial line.

Supported Devices

Serial line

Parameter Values

One or two longword integers specifying the baud rate for a serial line device.

If you specify one value, this value is used for both the transmit and receive speeds.

If you specify two values, the first value is used as the receive speed and the second value is used as the transmit speed. The values can be 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, or 19200 depending on the type of serial line device you are using.

The default value is 9600.

Not all serial line device drivers support all the baud rates listed. See the **Description** for the legal values for the type of serial line device you are using.

Description

The term **baud rate** usually refers to the number of data bits per second (bps) that are transmitted over a serial line. However, when transmitting a continuous stream of data, you should consider the overhead associated with the start and stop bits that prefix and suffix each character transmitted.[1]

[1] The number of stop bits is set by the device driver, and is based on the baud rate of the line. There is no way for the user to control this setting.

LIO$K_BAUD_RATE

To determine the actual number of characters per second (cps) that can be transmitted, use the following formula:

```
rate (cps) = baud rate / (start bits + bits/character + stop bits)
rate (bps) = rate (cps) * bits/character
```

For example, assume that the baud rate is set at 9600 (using the LIO$K_BAUD_RATE parameter), the number of bits per character is set at eight (using the LIO$K_BITS_PER_CHAR parameter), with one start bit and one stop bit. The number of characters per second (cps) and the number of bits per second (bps) that can be transmitted are determined as follows:

```
rate (cps) = 9600 / (1 + 8 + 1)
rate (cps) = 9600/10
rate (cps) = 960
rate (bps) = 960 * 8
rate (bps) = 7680
```

The following table lists the serial line devices and the maximum speed, or baud rate, for each device supported by VSL.

Device	Maximum Speed
DH11	9600
DHV11	9600
DMF32	19200
DS100	19200
DS200	19200
DZ11	9600
DZQ11	9600
DZV11	9600
μVAX2000	9600

You can specify any baud rate listed in the **Parameter Values** up to and including the maximum speed listed in this table.

Restrictions

The specified baud rates must be listed in the **Parameter Values**, but cannot exceed the maximum baud rate for the device.

Example

```
status = LIO$SET_I (device_id, LIO$K_BAUD_RATE, 1, 2400)
```

This routine sets the send and receive baud rates at 2400.

LIO$K_BIN_DDR

This parameter moves a complementary offset binary coded output voltage into the DAC Data Register (DDR).

Supported Devices

ADF01[1]

Parameter Values

A longword integer containing the binary code corresponding to the output voltage in complementary offset binary coding.

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_I (adf_id, LIO$K_BIN_DDR, 1, binary_code)
```

This routine moves the corresponding value of **binary_code** into the DAC Data Register (DDR). The **binary_code** argument contains the binary code of the desired output voltage.

[1] This device is available only in Europe.

LIO$K_BITS_PER_CHAR

This parameter establishes the number of data bits per character.

Supported Devices

Serial line

Parameter Values

A longword integer specifying the number of bits per character. The value can be 5, 6, 7, or 8.

The default value is 8.

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_BITS_PER_CHAR, 1, 7)
```

This routine specifies 7 bits per character.

LIO$K_BOUNCE

This parameter sets the contact bounce elimination response time delay.

Supported Devices

IDV11-A[1]

Parameter Values

A longword integer specifying the input response time for the contact bounce eliminator circuits for all 16 input channels.

The value can be one of the following:

Value	Response Time
1	0.5 ms
2[1]	5.0 ms
3	10.0 ms

[1]The default value.

Description

The contact bounce eliminator circuit takes an input signal from a bouncing contact and generates a clean digital signal four clock periods after the input stabilizes. The **Parameter Values** lists the bounce response time delays for which you can program the device.

[1] This device is available only in Europe.

Restrictions

None.

Example

```
status = LIO$SET_I (idva_id, LIO$K_BOUNCE, 1, 2)
```

This routine sets the bounce elimination to 5 milliseconds.

LIO$K_BREAK

This parameter generates a break (spacing) condition on a terminal line for the specified amount of time.

Supported Devices

Serial line

Parameter Values

A longword integer constant specifying the length of the break condition.

The value can be one of the following:

Constant Value	Break Length
LIO$K_SHORT_BREAK	0.4 seconds
LIO$K_LONG_BREAK	1.5 seconds

Description

A spacing condition (a series of null characters) is sent to the serial line for the specified length of time. A receiver can detect a break by checking for a framing error.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_BREAK, 1, LIO$K_LONG_BREAK)
```

This routine generates a long (1.5 second) break in a data transfer.

LIO$K_BUFF_SIZE

This parameter sets the following:

- Maximum buffer size, in bytes, for the ADQ32, DRQ3B, and Preston devices
- Size, in bytes, of the asynchronous buffers to allocate for the memory queue device

Supported Devices

ADQ32
DRQ3B
Memory queue
Preston

Parameter Values

A longword integer specifying the buffer size in bytes.

Description

The maximum allowable buffer size for the ADQ32, the DRQ3B, and the Preston is 65,534 bytes.

Using this parameter forces a reset of the DMA on the ADQ32 and the DRQ3B.

The default value for the Preston is an 8K-word (16,384-byte) buffer.

The default value for the DRQ3B is a 0-length buffer, so you must use this parameter.

Continuous data transfer using double buffering may be affected by the size of the buffer you choose. See Section 1.6.3.4, Double-Buffer DMA, for more information. For the DRQ3B, buffer sizes of 8K words or more usually ensure continuous data transfer.

LIO$K_BUFF_SIZE

When you set up the memory device with the value of the LIO$K_BUFF_SOURCE parameter as LIO$K_ARRAY or LIO$K_VIRTUAL_MEM, you must set up the device using the LIO$K_BUFF_SIZE parameter before you specify the buffer source (LIO$K_BUFF_SOURCE). The LIO facility does not supply a default value if you omit this parameter. If you set up the memory device with the value of the LIO$K_BUFF_SOURCE parameter as LIO$K_USER, then LIO ignores this parameter.

Restrictions

With the memory queue device, you must use this parameter before you set up the buffer source (LIO$K_BUFF_SOURCE).

Example

```
status = LIO$SET_I (device_id, LIO$K_BUFF_SIZE, 1, 200)
```

This routine specifies the buffer size as 200 bytes or 100 words.

LIO$K_BUFF_SOURCE

This parameter specifies the source from which to allocate buffers.

Supported Devices

Memory queue

Parameter Values

A longword integer constant specifying the buffer source.

The value can be one of the following:

Constant Value	Meaning
LIO$K_ARRAY	Preallocates buffers from a user array
LIO$K_VIRTUAL_MEM	Preallocates buffers from VMS virtual memory
LIO$K_USER	Uses buffers supplied by the user

Description

When you set up the memory queue device, you must use the LIO$K_BUFF_SOURCE parameter to specify where the LIO facility is to obtain the buffer.

If the value of this parameter is LIO$K_ARRAY, the user buffer must be large enough to hold the buffers allocated through the LIO$K_BUFF_SIZE and LIO$K_N_BUFFS parameters and allow for buffer overhead. You can use the following formula to determine the number of bytes of overhead to add to the array.

F + (B * N_BUFFS)

where:

F	is the fixed overhead
B	is the overhead per buffer
N_BUFFS	is the number of buffers supplied to LIO$K_N_BUFFS

LIO$K_BUFF_SOURCE

If the memory queue is attached local to the process (LIO$K_LOCAL as the value of the **io_type** argument), use 8 as the value of F and 16 as the value of B. If the memory queue is attached for interprocess I/O (LIO$K_INTER_PROC as the value of the **io_type** argument), use 72 as the value of F and 24 as the value of B.

Restrictions

- You must specify all other parameters associated with setting up the memory queue device before you use this parameter to allocate memory. Once you use this parameter, the memory is allocated immediately.
- If **io_type** is LIO$K_INTER_PROC and the value of LIO$K_BUFF_SOURCE is LIO$K_ARRAY, then **the array must be page-aligned**. See Section 1.6.3.2, Continuous DMA, for more information about page-aligning buffers.

Example

```
status = LIO$SET_I (device_id, LIO$K_BUFF_SOURCE, 1, LIO$K_VIRTUAL_MEM)
```

This routine specifies that the buffer is to be allocated from virtual memory.

LIO$K_BURST_DIV

This parameter establishes the sampling rate of the internal burst rate clock of the Preston device.

Supported Devices

Preston

Parameter Values

A longword integer specifying the desired frequency divisor. This value is used as the divisor for the internal clock 5 MHz crystal frequency of the Preston device.

The default value is 10.

Description

This parameter specifies the timing between channels in all channel sweep trigger modes. For example, if the sampling rate is set to 100 kHz (using the LIO$K_CLK_RATE parameter), the channel burst rate divisor is set to 5, and the A/D is set for multiple channels, then the A/D begins one channel sweep every 10 microseconds.

Use this parameter as an alternative to using a burst clock rate (LIO$K_BURST_RATE).

See the reference description of the LIO$K_TRIG parameter in this chapter for more information about trigger modes.

Restrictions

- This parameter is only useful when using the Preston internal clock or external clock input.
- The valid range for parameter values is 3 to 65,534.

Examples

This example shows how to set up multiple A/D channels for sampling, and how to set up the A/D sampling rate and the channel burst divisor. When you set up the Preston using the parameters and parameter values specified, the Preston A/D begins one channel sweep every 10 microseconds.

1. `status = LIO$SET_I (device_id, LIO$K_AD_CHAN, 5, 0, 1, 2, 3, 4)`

   This routine specifies 5 A/D channels for sampling.

2. `status = LIO$SET_R (device_id, LIO$K_CLK_RATE, 1, 100000.0)`

   This routine sets the A/D sampling rate to 100 kHz.

3. `status = LIO$SET_I (device_id, LIO$K_BURST_DIV, 1, 5)`

   This routine sets the Preston burst rate divisor to 5.

LIO$K_BURST_RATE

This parameter specifies the rate of the internal burst rate clock of the Preston device.

Supported Devices

Preston

Parameter Values

A single-precision, floating-point real value specifying the desired frequency.

The default value is 1 MHz.

Description

This parameter specifies the timing between channels in all channel sweep trigger modes. For example, if the sampling rate is set to 100 kHz (using LIO$K_CLK_RATE), the channel burst rate is set to 1 MHz, the A/D is set for multiple channels, and the channels within the sweep are sampled each microsecond, then the A/D begins one channel sweep each 10 microseconds.

Use this parameter as an alternative to selecting a burst clock divisor (LIO$K_BURST_DIV). The Preston device internal clock has a fixed base crystal frequency of 5 MHz, thus the granularity of the clock rate is .2 microseconds.

See the reference description of the LIO$K_TRIG parameter in this chapter for more information about trigger modes.

LIO$K_BURST_RATE

Restrictions

- This parameter is useful only when using the Preston internal clock.
- The valid range of parameter values is limited to 106 Hz to 1 MHz.

Example

```
status = LIO$SET_R (device_id, LIO$K_BURST_RATE, 1, 500000.0)
```

This routine specifies 500 kHz as the rate between channels in a channel sweep.

LIO$K_CANCEL

This parameter cancels outstanding I/O on the specified channel. It is used to stop continuous DMA transfers.

Supported Devices

AAF01[1]
ADF01[1]
DRQ11-C[1]

Parameter Values

None.

Description

Once a continuous DMA transfer has been started for the AAF01, ADF01, or DRQ11-C, it can be stopped only by issuing a CANCEL request.

Restrictions

None.

Example

```
status = LIO$SET_I (adf_id, LIO$K_CANCEL)
```

This routine causes all outstanding I/O on the channel assigned to **adf_id** to be stopped.

[1] This device is available only in Europe.

LIO$K_CC_FOUT

This parameter sets the Frequency Output (FOUT) reference signal.

Supported Devices

IDV11-D[1]

Parameter Values

A longword integer array of length three.

The array index values can be the following:

Index	Values	Function
1	0...1	Turns FOUT ON (0) or OFF (1)
2	0...15	Selects the FOUT source
3	0...15	Selects the FOUT divider

Description

Use this parameter to control the FOUT reference signal which is provided to the user cable for external timing control.

Restrictions

None.

[1] This device is available only in Europe.

Example

```
INTEGER*4 i_fout(3) !Declare integer array of length 3
i_fout(1) = 1            !Frequency output reference signal off switch
i_fout(2) = 0            !No source
i_fout(3) = 0            !No divider

status = LIO$SET_I (idvd_id, LIO$K_CC_FOUT, 1, %LOC(i_fout))
```

This routine turns off the frequency output reference signal.

LIO$K_CC_SETUP

This parameter sets up the operating characteristics of a channel.

Supported Devices

IDV11-D[1]

Parameter Values

A longword integer array of length nine.

The array index values can be the following:

Index	Values	Function
1	0...4	Selects counter channel
2	0...65534	Specifies precount
3	0 1	Counts down Counts up
4	0 1	Counts on positive edge Counts on negative edge
5	0 1	Explicit start Immediate start
6	User supplied	AST address
7	0...65535	AST parameter
8	0...15	Selects source signal
9	0...5	Selects gating signal

Description

Use this parameter to program the five-channel IDV11-D counter. You program each of the five counters with separate LIO$SET_I routine calls.

[1] This device is available only in Europe.

Restrictions

- You must use this parameter with the LIO$SET_I routine call to program each channel of the IDV11-D counter individually.
- If you use this parameter with the LIO$SHOW routine, the first entry in the **value_list** argument must specify the IDV11-D counter for which you want the operating characteristics returned.

Example

```
INTEGER*4 cc_array(9)    !Declare integer array of length 9
cc_array(1) = 4          !Counter channel 3
cc_array(2) = 2500       !Pulse internal precount
cc_array(3) = 0          !Countdown switch
cc_array(4) = 0          !Count on positive-edge
cc_array(5) = 1          !Immediate start
cc_array(6) = 0          !AST routine not used
cc_array(7) = 0          !AST routine parameter not used
cc_array(8) = 1          !Source = 5 MHz clock
cc_array(9) = 0          !No gating

status = LIO$SET_I (idvd_id, LIO$K_CC_SETUP, 1, %LOC(cc_array))
```

This routine sets up the IDV11-D counter channel 3 to generate output pulses every 500 microseconds.

LIO$K_CHANNEL

This parameter sets the A/D or D/A channel to be used in subsequent I/O routine calls.

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer containing the device channel number.

Description

See the **Parameter Values**.

Restrictions

The channel selected must have been initialized with the LIO$K_DATA_PATH parameter.

Example

```
status = LIO$SET_I (adf_id, LIO$K_CHANNEL, 1, 1)
```

This routine sets channel number 1 as the channel for use by subsequent VSL routine calls.

[1] This device is available only in Europe.

LIO$K_CLK_BASE

This parameter specifies the base frequency of the Preston internal clock.

Supported Devices

Preston

Parameter Values

A single-precision, floating-point real value specifying the base frequency.

The acceptable values are 4,000,000.0 or 5,000,000.0.

The default value is 5,000,000.0.

Description

This parameter is useful because older Preston units contain a 5 MHz clock, while newer models contain a 4 MHz clock.

Using this parameter with the LIO$SHOW routine returns the base crystal frequency.

This is an optional parameter.

Restrictions

- You can use this parameter only with the Preston internal clock.
- The only acceptable values are 4 MHz or 5 MHz.
- If you use this parameter with LIO$K_CLK_RATE or LIO$K_BURST_RATE, you must set up this parameter first.

Example

```
status = LIO$SET_R (device_id, LIO$K_CLK_BASE, 1, 4000000.0)
```

This routine tells LIO that your Preston device contains a 4 MHz clock.

LIO$K_CLK_DIV

This parameter specifies the divider to be used for selecting the clock frequency you want.

Supported Devices

Preston

Parameter Values

A longword integer specifying the clock divider. This value is used as the divider for the base frequency of the Preston internal clock.

The default value is 10, which provides a clock rate of 500 kHz for Prestons with 5 MHz base frequency, and a clock rate of 400 kHz for Prestons with 4 MHz base frequency.

Description

This parameter specifies the divider to be used for selecting a clock rate for the Preston (with base frequency either 4 MHz or 5 MHz, depending on the model). The divider specifies the number of base frequency clock ticks to be counted before a sampling pulse is issued. The clock sampling rate is determined by the formula

$$rate = \frac{base\ frequency}{divider}$$

In the point sampling modes, the divider specifies the rate at which the A/D converts successive samples. In the sweep trigger modes, the divider specifies the rate at which channel sweeps are initiated.

Use this parameter as an alternative to selecting the clock rate using LIO$K_CLK_RATE.

LIO$K_CLK_DIV

With the LIO$K_CLK_BASE and LIO$K_CLK_DIV parameters, the rate actually used is determined by the sampling rate formula. With the LIO$K_CLK_RATE parameter, the rate actually used may be different from what you specify. For more information, see the description of the LIO$K_CLK_RATE parameter.

For the Preston internal clock with a 5 MHz base frequency, the granularity of the clock rate is .2 microseconds.

Restrictions

- This parameter is only useful when using the Preston internal clock or external clock input.
- The valid range for parameter values is 3 to 65,535.

Example

```
status = LIO$SET_I (device_id, LIO$K_CLK_DIV, 1, 100)
```

This routine specifies a divider of 100, which provides a clock rate of 50 kHz (if your Preston has a 5 MHz clock) or 40 kHz (if your Preston has a 4 MHz clock).

LIO$K_CLK_RATE

This parameter takes an ideal frequency and selects the internal base frequency and divider to best approximate that frequency.

Supported Devices

ADQ32
KWV11-C
Preston
Simpact RTC01

Parameter Values

A single-precision floating-point real value specifying the desired frequency. Using this parameter with LIO$SHOW returns the actual frequency.

When you set up the ADQ32, the KWV11-C, or the Simpact RTC01 without using this parameter, LIO uses a default value of 100.0 Hz.

When you set up the Preston without using this parameter, LIO uses a default value of 500.0 kHz.

Description

When used with the KWV11-C or the Simpact RTC01, this parameter is an alternative to selecting a source frequency and divider with LIO$K_CLK_SRC.

The KWV11-C can produce rates from 1 MHz to .001526 Hz—from one tick every millionth of a second to approximately one tick every 11 minutes.

The Simpact RTC01 can produce rates from 10 MHz to 2.3×10^{-8} Hz—from one tick every 10 millionths of a second to one tick every 497 days.

Any rates you specify outside of the range of the clock are trimmed to the maximum and minimum rates of which the clock is capable.

LIO uses the following algorithm to approximate some clock rates: The "ideal" divider is first rounded and then converted to an integer value. (A remainder greater than or equal to .5 is rounded up to 1; less than .5 is rounded down.)

Suppose you want a clock rate of 1500 Hz. To select a clock rate you use the following formula:

$$rate = \frac{base\ frequency}{divider}$$

If your base frequency is 5 MHz, according to the formula you would use a divider of 3333.33. However, a divider must be an integer, so you cannot obtain an exact rate of 1500 Hz.

In this case, LIO uses a divider of 3333. The actual clock rate produced is 1500.15 Hz, which is the closest you can get to 1500 Hz.

This parameter returns no warning if the clock is not set to the exact rate you request. You must use this parameter with LIO$SHOW to see the actual rate.

When used with a Preston device, this parameter specifies the sampling rate for the A/D. In the point sampling modes, it specifies the rate at which the A/D converts successive channels. In the sweep trigger modes, it describes the rate at which channel sweeps are initiated.

When used with the ADQ32 device, this parameter sets the rate for the primary clock on the device. See Appendix A for more information.

Restrictions

The following restrictions apply to a Preston device:

- This parameter is useful only when using the Preston internal clock.
- The valid range of parameter values is 106 Hz to 1 MHz.

The following restriction applies to the KWV11-C and the Simpact RTC01:

- This parameter is useful only for single-count and repeat-count functions because the event-timing functions use only the base crystal rate, not the divider.

Example

```
status = LIO$SET_R (device_id, LIO$K_CLK_RATE, 1, 2.0)
```

This routine sets the clock at 2 Hz (two ticks per second).

LIO$K_CLK_SRC

This parameter sets the source frequency and the divider for clock ticks. The source frequency and divider together specify the clock rate. This parameter also sets the source frequency for event timing.

LIO$K_CLK_SRC is an alternative to LIO$K_CLK_RATE.

Supported Devices

KWV11-C
Simpact RTC01

Parameter Values

A longword integer value or array of length two.

The first value, which is required, specifies the clock source frequency.

The clock source can be one of the following:

Value	Clock Source
1	Internal 1 MHz clock
2	Internal 100 kHz clock
3	Internal 10 kHz clock
4	Internal 1 kHz clock
5	Internal 100 Hz clock
6	Schmitt trigger 1
7	For the KWV11-C, line frequency—50 Hz or 60 Hz, depending on country[1]
7	For the Simpact RTC01, 10 MHz

[1]Line frequency is obtained by accessing the BEVNT line of the Q-bus. On many systems, including the MicroVAX II and the MicroVAX 3000 series, the BEVNT line is not enabled. Therefore, line frequency is not available on these systems.

The second value, which is optional, specifies the divider. The divider can be any value between 1 and 65,536 inclusive.

The default divider value is 1.

Description

The KWV11-C and the Simpact RTC01 can be used for single-count or repeat-count functions or for event-timing functions. You select the function with the LIO$K_FUNCTION parameter. The LIO$K_CLK_SRC parameter is used differently for each function.

For single-count or repeat-count functions, you can use the LIO$K_CLK_SRC parameter to select a clock rate by specifying the source frequency and the divider.

For event-timing functions, only the clock source frequency is used. The divider is not used. Event timing can be done in absolute or relative time. For more information, see the description of LIO$K_FUNCTION.

The clock rate is specified using the following formula:

$$rate = \frac{source\ frequency}{divider}$$

The choices for the source frequency are the following:

- Internal KWV11-C or Simpact RTC01 clock crystal (one of five frequencies)
- Power line frequency[1]
- Schmitt trigger 1 (ST1)

When ST1 is selected as the clock source, the clock's counters count each external event on the ST1. (An external event is an input voltage; the threshold level is selected by potentiometers and the positive or negative slope by switches on the UDIP.)

When the number of external events equals the number specified by the divider, the clock generates a pulse. Note that if the ST1 input is a regularly timed frequency, the input can be used as a source frequency for more specific clock rates.

[1] Line frequency is obtained by accessing the BEVNT line of the Q-bus. On many systems, including the MicroVAX II and the MicroVAX 3000 series, the BEVNT line is not enabled. Therefore, line frequency is not available on these systems.

Restrictions

For single-count or repeat-count functions, you must specify a divider (or use the default value of 1). For event-timing functions, no divider is used.

Examples

1. `status = LIO$SET_I (clock_id, LIO$K_CLK_SRC, 2, 6, 1)`

   This routine specifies ST1 as the clock source with a divider of 1. The KWV11-C and the Simpact RTC01 issue a clock pulse every time the Schmitt trigger fires.

2. `status = LIO$SET_I (clock_id, LIO$K_CLK_SRC, 2, 1, 150)`

   This routine selects a clock rate of 6666.67 Hz by specifying a source frequency of 1 MHz and a divider of 150.

3. `status = LIO$SET_I (clock_id, LIO$K_CLK_SRC, 1, 5)`

   This routine selects the 100 Hz source frequency to be used for event timing.

LIO$K_CLR_LBO

This parameter clears the large buffer overflow (LBO) condition.

Supported Devices

AAF01[1]
ADF01[1]
DRQ11-C[1]

Parameter Values

None.

Description

When you want to transfer a large buffer more than once, use this parameter to clear the large buffer overflow condition. This parameter allows a large buffer transfer to wrap to the beginning of the large buffer. If the large buffer is to be transferred more than once, a user program must perform the following steps:

1. Issue an LIO$K_CLR_LBO immediately after starting the large buffer transfer.
2. Clear the device event flag each time one sub-buffer is processed.
3. Count the sub-buffers.
4. Issue an LIO$K_CLR_LBO each time **n** sub-buffers are transferred, where **n** sub-buffers is one large buffer.

Restrictions

None.

[1] This device is available only in Europe.

Example

```
status = LIO$SET_I (aaf_id, LIO$K_CLR_LBO, 0)
```

This routine clears the large buffer overflow condition.

LIO$K_COB

This parameter sets the Command Output (COUT) bit in the Command and Status Register (CSR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

None.

Description

Use this parameter to set the Command Output (COUT) bit in the Command and Status Register (CSR).

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_COB, 0)
```

This routine sets the Command Output (COUT) bit in the Command and Status Register (CSR).

[1] This device is available only in Europe.

LIO$K_COMMAND

This parameter sends one or more IEEE-488 commands on the bus.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

One or more longword integer values specifying IEEE-488 commands. Valid command values are in the following octal ranges:

Octal Value	Command Type
0..17	Addressed commands
20..37	Universal commands
40..77	Listener commands
100..137	Talker commands
140..177	Secondary commands

The following addressed commands are effective only in devices that have been addressed as a listener or a talker.

Table 4–4: Address Command Group

Octal	Function	Command	Description
001	Listener	Go To Local	Causes addressed listeners to go from remote mode to local mode. When local is true, a device is controlled by its front or back panel controls.
004	Listeners	Selected Device Clear	Causes the addressed listeners to be reset (initialization).

Table 4–4 (Cont.): Address Command Group

Octal	Function	Command	Description
005	Listener	Parallel Poll Configuration	Causes the addressed listeners to enter the parallel poll configuration mode so that the addressed listeners can participate in a parallel poll. The next command must be Parallel Poll Enable from the Secondary Command Group to allow the listener to respond to ATN or EDI becoming true.
010	Listener	Group Execute Trigger	Causes the addressed listeners to start basic operation of the device of which the listener is a part.
011	Talker	Take Control	Causes a device that has been addressed as the talker to enable its controller to become the controller-in-charge after the current controller-in-charge unasserts ATN.

The following universal commands affect all devices that are able to respond without having to be previously addressed.

Table 4–5: Universal Command Group

Octal	Command	Description
021	Local Lockout	Causes all devices to ignore their local message RTL (Return To Local).
024	Device Clear	Causes all devices to be reset (initialization).
025	Parallel Poll Unconfigure	Causes all parallel poll configurations to become unconfigured.

Table 4–5 (Cont.): Universal Command Group

Octal	Command	Description
030	Serial Poll Enable	Causes all talkers to enter the serial poll mode to allow a talker to send a status byte after being addressed by the controller-in-charge.
031	Serial Poll Disable	Causes all talkers to exit the serial poll mode and return to the normal data mode.

The following listener commands are used to address one or more listeners or to address all listeners at once. Addressed listeners become active when the controller-in-charge unasserts ATN. These commands can be followed by a secondary address command (see Table 4–8) to address an extended listener.

Table 4–6: Listener Address Group

Octal	Address	Description
040	My Listen Address 0	Any listener that recognizes its own address becomes an addressed listener and is able to receive data bytes from a talker as soon as the controller-in-charge unasserts ATN.
041	My Listen Address 1	"
.	.	.
.	.	.
.	.	.
076	My Listen Address 30	"
077	Unlisten	Causes all listeners to become unaddressed.

The following talker commands are used to address or unaddress one talker. An addressed talker becomes active when the controller-in-charge unasserts ATN. These commands can be followed by a secondary address command (see Table 4–8) to address an extended talker.

Table 4–7: Talker Address Group

Octal	Address	Description
100	My Talk Address 0	A talker that recognizes its own address becomes an addressed talker, whereas all other talkers become unaddressed. Only one talker is able to send data on the IEEE-488 bus.
101	My Talk Address 1	"
.	.	.
.	.	.
.	.	.
136	My Talk Address 30	"
137	Untalk	Causes the addressed talker to become unaddressed.

The meaning of the following secondary commands is defined by the preceding primary commands of the primary command group (addressed, universal, listener, or talker).

Table 4–8: Secondary Command Group

Octal	Command	Description
140	Parallel Poll Enable 1	Each Parallel Poll Enable command must follow a Parallel Poll Configuration command, which forces currently addressed listeners into their parallel poll configuration state. Each Parallel Poll Enable command tells a device how to respond to a parallel poll request from the controller-in-charge. A device responds by sending one status bit on one of the eight data I/O lines.

Table 4-8 (Cont.): Secondary Command Group

Octal	Command	Description
141	Parallel Poll Enable 2	"
146	Parallel Poll Enable 7	"
147	Parallel Poll Enable 8	"
150	Parallel Poll Enable 11	"
151	Parallel Poll Enable 12	"
156	Parallel Poll Enable 17	"
157	Parallel Poll Enable 18	"
160	Parallel Poll Disable	This command must follow its associated Parallel Poll Configuration command and inhibits devices from responding to the parallel poll request.
140	My Secondary Address 0	This command must follow a talker or listener address. Devices that use extended addressing do not become addressed as long as the associated secondary address follows the primary address.
141	My Secondary Address 1	"
175	My Secondary Address 29	"
176	My Secondary Address 30	"

Description

Use this parameter to send one or more IEEE-488 bus commands. See the *IEU11-A/IEQ11-A User's Guide* for more information about IEEE-488 commands.

Restrictions

The IEEE-488 device must be controller-in-charge of the bus.

Example

```
instr_addr = 1
command = inst_addr .OR. LIO$M_LNR
status = LIO$SET_I (ieee_id, LIO$K_COMMAND, 1, command)
```

This routine sets the device at IEEE-488 bus address 1 to be a listener. This is an alternative to specifying its address in the **device_specific** argument of LIO$WRITE.

LIO$K_CONT

This parameter sets the device to continuous direct memory access (DMA) mode.

Supported Devices

AAV11-D
ADV11-D
Preston devices (DRQ3B interface only)

Parameter Values

None.

Description

The AAV11-D, ADV11-D, and Preston devices perform single-buffer DMA, by default, when attached to use QIOs (LIO$K_QIO).

Use the LIO$K_CONT parameter to signal LIO that you want a device to use its DMA mode for subsequent data transfers. Continuous DMA mode provides the greatest speed from the hardware by sending buffers of data continuously with no stops between buffers.

See Section 1.6.3.2, Continuous DMA, for more information about using continuous DMA for high-speed data transfers.

Restrictions

The following restrictions apply to both the AAV11-D and the ADV11-D:

- The device must be set for asynchronous I/O.
- No buffer can currently be enqueued to the device.

Example

```
status = LIO$SET_I (device_id, LIO$K_CONT, 0)
```

This routine sets the device for continuous DMA mode.

LIO$K_COUNTER

This parameter reads the count register of the Simpact RTC01 clock device.

Supported Devices

Simpact RTC01

Parameter Values

A single longword integer.

Description

LIO$K_COUNTER used with LIO$SHOW reads and returns the current contents of the Simpact RTC01 counter.

For example, if the clock is requested to generate a 100 KHz pulse train, the clock would be set to operate at 10 MHz and the count register would be loaded with the two's complement of the value 100. Each time the count register counts down from −100 to zero, a pulse is generated. Executing LIO$SHOW with the LIO$K_COUNTER argument allows the count register to be read while the clock is operating.

Restrictions

- You can use this parameter only with the Simpact RTC01 clock device. (The design of the KWV11-C does not allow reading of the count register while the clock is operating.)
- This is an LIO$SHOW parameter only.

Example

```
status = LIO$SHOW (device_id, LIO$K_COUNTER, value_list, list_len)
```

This routine reads the count register of the Simpact RTC01. The **value_list** argument returns the Simpact RTC01 count register contents. The **list_len** argument returns 1.

LIO$K_CTA

This parameter loads the Control Table Address (CTA) register.

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer specifying the position in the 1K-word Control Table.

The value can be between 0 and 1023, inclusive.

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_I (axf_id, LIO$K_CTA, 1, control_table_pos)
```

This routine loads the position within the Control Table specified by **control_table_pos** into the Control Table Address (CTA) register.

[1] This device is available only in Europe.

LIO$K_CTI_BUF

This parameter sets the buffer and event flag for devices set for connect-to-interrupt (CTI) I/O.

Supported Devices

AXV11-C A/D

Parameter Values

A maximum of three longword integers specifying the buffer address (optional), the buffer size in bytes, and the device event flag.

If you do not supply the buffer address, the LIO facility allocates one from dynamic virtual memory.

The buffer size should be the size of the desired user buffer **not including** the overhead. Note that the actual array must be declared to be approximately 300 bytes larger than the buffer size you specify through the LIO$K_CTI_BUF parameter value. The CTI handler uses the extra space at the high end of the buffer for the interrupt service routine.

The LIO$K_CTI_OVERHD parameter of the LIO$SET routine returns the exact amount of CTI handler overhead (in bytes).

Description

The AXV11-C is the only device that currently supports CTI I/O. Use this parameter to supply the AXV11-C with a buffer for the CTI.

The connect-to-interrupt driver requires less overhead for each I/O call than the QIO driver, but more overhead for each I/O call than memory-mapped I/O.

Because the interrupt service routine performs the I/O, a user program does not need to run at high priority. Also, the I/O does not preempt all other processes running on the system.

Restrictions

- The AXV11-C must be attached to use CTI I/O using LIO$K_CTI.
- The data buffer must be large enough to contain the connect-to-interrupt overhead in addition to the data itself.
- The data buffer size cannot be more than 65,536 minus the CTI overhead (in bytes).

Examples

This example shows how to attach the AXV11-C device to use CTI I/0, and how to set up the connect-to-interrupt buffer and an event flag. Variable declarations are included to make this example easier to understand.

1. ```
   INTEGER*2 buffer(20 + 150)
   ```

   This line allocates a 20-word buffer, plus a 150-word (300-byte) buffer overhead area.

2. ```
   INTEGER buffer_length
   INTEGER event_flag
   ```

   These two lines declare the variables **buffer_length** and **event_flag**, and the data type.

3. ```
   status = LIO$ATTACH (device_id, 'AXA0', LIO$K_CTI)
   ```

   This routine attaches the AXV11-C to use CTI I/O. The device ID of the AXV11-C is returned in the **device_id** parameter and is used in the subsequent routine lines to identify this AXV11-C device.

4. ```
   status = LIB$GET_EF(event_flag)
   ```

   This routine uses the VMS Run-Time Library Routine, LIB$GET_EF, to allocate one local event flag from a process-wide pool. The allocated event flag is returned as the value of **event_flag**.

5. `buffer_length = 40`

   This line specifies 40 bytes as the length of the data portion of the buffer argument.

6. ```
   status = LIO$SET_I (device_id, LIO$K_CTI_BUF, 3, buffer, buffer_length,
   1                  event_flag)
   ```

   This routine specifies the CTI buffer address, the length of the CTI buffer (in bytes), and the device event flag.
   ```

LIO$K_CTI_OVERHD

This parameter returns the size of the connect-to-interrupt (CTI) handler overhead in bytes.

Supported Devices

AXV11-C A/D

Parameter Values

This parameter returns one longword integer.

Description

Use this LIO$SHOW parameter to determine the size of the CTI handler overhead you must allocate at the end of the buffer passed to LIO$K_CTI_BUF.

Restrictions

The AXV11-C device must be attached to use CTI I/O using LIO$K_CTI.

Example

```
status = LIO$SHOW (device_id, LIO$K_CTI_OVERHD, buffer, buffer_size)
```

The routine line returns a longword integer containing the size of the CTI handler overhead in bytes. The **buffer** argument returns the size of the CTI handler, the **buffer_size** argument returns the number of integers returned in the **buffer** argument.

LIO$K_CTRL_ACTIVE

This parameter signals an IEEE-488 device to activate its controller function.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

None.

Description

Using this parameter reactivates the controller-in-charge function of an IEEE-488 device. This is the default for the device.

You need to set this parameter only if you have deactivated the controller-in-charge function by setting the LIO$K_CTRL_STANDBY parameter. Using this parameter in conjunction with LIO$K_CTRL_STANDBY improves the performance of successive data transfers.

The controller function must be active for the IEEE-488 device to perform any of the functions of the controller-in-charge. When the controller is active and the IEEE-488 device prepares to send or receive data, the device must first put the controller function on standby (LIO$K_CTRL_STANDBY). The device then transfers the data and reactivates the controller function.

Deactivating and reactivating the controller function adds overhead to each send or receive operation. If an application requires a series of data transfers, you can improve the rate of the transfers by setting the controller function on standby for the duration of the data exchange. Then, your application should reactivate the controller function when all data transfers complete.

LIO$K_CTRL_ACTIVE

Restrictions

The IEEE-488 device must be the controller-in-charge.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_CTRL_ACTIVE, 0)
```

This routine reactivates the controller function of an IEEE-488 device.

LIO$K_CTRL_AST

This parameter supplies a user-written AST routine to be called in the following situations:

- When an external device writes data to the input control port of the DRB32
- On receipt of a specified control character by serial line devices

Supported Devices

DRB32
Serial line

Parameter Values

A longword integer specifying the address of the AST routine. Setting this parameter without supplying a parameter value cancels the AST routine.

Description

This can be a mechanism for synchronizing with state changes in the external device.

Restrictions

When using FORTRAN, the subroutine must be declared EXTERNAL.

Examples

1. `status = LIO$SET_I (device_id, LIO$K_CTRL_AST, 1, user_ast)`

   This routine specifies a user-supplied AST routine called **user_ast**.

2. `status = LIO$SET_I (device_id, LIO$K_CTRL_AST, 0)`

   This routine cancels the delivery of a previously specified AST routine.

LIO$K_CTRL_HANDLING

This parameter specifies a flag that indicates what action to take on receipt of a control character specified by LIO$K_CTRL_AST for serial line devices.

Supported Devices

Serial line

Parameter Values

A longword integer constant specifying the action to take on receipt of the control character.

The value can be one of the following:

Constant Value	Action Taken
LIO$K_CTRL_ABORT	Aborts the current operation on receipt of the control character. The status value SS$_CONTROLC is returned.
LIO$K_CTRL_INCLUDE	Includes the control character in the buffer returned to the user program.
LIO$K_CTRL_INC_ABORT	Aborts the current operation and includes the control character in the buffer returned to the user program.

Description

Use this parameter in conjunction with the LIO$K_CTRL_AST parameter to specify what action to take on receipt of the control character specified by the LIO$K_CTRL_AST parameter. The **Parameter Values** lists the valid action you can specify.

LIO$K_CTRL_HANDLING

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_CTRL_HANDLING, 1,
1                  LIO$K_CTRL_INCLUDE)
```

This routine specifies that the LIO facility include the control character in the buffer returned to the user program.

LIO$K_CTRL_STANDBY

This parameter deactivates the controller function of an IEEE-488 device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

None.

Description

You must place the controller-in-charge in controller standby mode when two other devices on the IEEE-488 bus are going to transfer data between each other. Use the LIO$K_CTRL_STANDBY parameter to place the controller-in-charge in standby mode only when the controller-in-charge is not participating in the data transfer.

This parameter can also be used in conjunction with the LIO$K_CTRL_ACTIVE parameter to improve the performance of successive data transfers if the controller-in-charge is going to issue several LIO$READ routines or LIO$ENQUEUE routines to read a large buffer of data.

Restrictions

The IEEE-488 device must be the controller-in-charge.

Example

```
status = LIO$SET_I (device_id, LIO$K_CTRL_STANDBY, 0)
```

This routine deactivates the controller function of an IEEE-488 device.

LIO$K_CURRENT_CHANNEL

This parameter specifies which channel is to be affected by channel-specific set calls (LIOK_TITLE, LIOK_X_LABEL, and LIOK_Y_LABEL).

Supported Devices

Real-time plotting

Parameter Values

A longword integer specifying the channel to be changed by future set calls.

The default channel is 0.

Description

You can select any channel up to the maximum number of channels as the current channel. All future channel-specific set calls will modify the selected channel.

Restrictions

- The current channel must be within the maximum number of channels.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_CURRENT_CHANNEL, 1, 3)
```

This routine selects channel 3 as the current channel to be changed by future set calls.

LIO$K_CWT

This parameter reads the Control Word Registers from, or writes the Control Word Registers to, the Control Table Memory of the AAF01 and ADF01 devices.

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

Five longword integer values.

The first value specifies the direction of the operation (read or write).

This value can be one of the following:

Constant Value	Meaning
LIO$K_INPUT	Reads from the Control Table
LIO$K_OUTPUT	Writes to the Control Table

The second value specifies the address of an integer array from which to read the Control Table, or to which to write the Control Table. The data type of the buffer itself must be a word (2-byte) array.

The third value specifies the position in the Control Table at which you want the transfer to begin.

The fourth value specifies the position in the Control Table at which you want the transfer to end.

For LIO$K_INPUT (read), the fifth value specifies the length of the integer array in bytes. For LIO$K_OUTPUT (write), the fifth value specifies the first position in the Control Table at which loading begins.

[1] This device is available only in Europe.

LIO$K_CWT

Description

See the **Examples**.

Restrictions

None.

Examples

1.
```
status = LIO$SET_I (axf_id, LIO$K_CWT, 5, LIO$K_INPUT,
1                  %LOC(control_table), start_pos, end_pos, table_length)
```

When reading from the Control Word Table, the contents of the table, beginning at **start_pos** and ending at **end_pos**, are loaded into the **control_table** buffer. The **table_length** argument specifies the length of the **control_table** buffer. When the routine call returns, the **start_pos** argument contains the position within the Control Table at which the transfer ended.

2.
```
status = LIO$SET_I (axf_id, LIO$K_CWT, 5, LIO$K_OUTPUT,
1                  %LOC(control_table), start_pos, end_pos, table_pos)
```

When writing to the Control Word Table, the contents of the **control_table** buffer, beginning at **start_pos** and ending at **end_pos**, are loaded into the Control Table beginning at **table_pos**. When the routine call returns, the **end_pos** argument contains the position + 1 within the Control Table at which the transfer ended.

LIO$K_DA_CHAN

This parameter specifies the D/A channels to use for output.

Supported Devices

AAV11-D
AXV11-C D/A

Parameter Values

One or two longword integers specifying the D/A channels to use.

The values can be one or both of the following:

Value	Meaning
0	Specifies channel X
1	Specifies channel Y

When you set up either of these two devices without using the LIO$K_DA_CHAN parameter to specify the D/A channels, LIO defaults to 0.

Description

If you specify both channels, the first output value in the buffer is always sent to channel X; the second output value is sent to channel Y; the third output value is sent to channel X; and so on, until all of the values in the buffer are output.

Restrictions

- For the AAV11-D, no buffers can be currently enqueued to the device when you set up the AAV11-D using this parameter.
- For the AAV11-D and AXV11-C D/A devices, you can specify each channel only once.

Example

```
status = LIO$SET_I (device_id, LIO$K_DA_CHAN, 2, 0, 1)
```

This routine specifies two D/A channels, channel X and channel Y, for output.

LIO$K_DATA

This parameter enables you to perform data transfers to and from the DRB32 parallel data path without using DMA.

Supported Devices

DRB32

Parameter Values

Two longword integer values.

The first value specifies the number of bytes to transfer. A maximum of four bytes is allowed for each transfer.

The second value specifies the address of the buffer to transfer.

Description

To perform an output operation, use the LIO$SET_I routine. To perform an input operation, use the LIO$SHOW routine.

Restrictions

The data transfer is limited to four bytes in length in either direction.

Examples

1. `status = LIO$SET_I (drb_id, LIO$K_DATA, 2, 4, buffer_address)`

   This routine outputs a four-byte buffer.

2. `status = LIO$SHOW (drb_id, LIO$K_DATA, buffer_address, buffer_length)`

   This routine inputs the array at starting virtual address **buffer_address** of length **buffer_length**.

LIO$K_DATA_PATH

This parameter selects the data path and channel number.

Supported Devices

AAF01[1]
ADF01[1]
DRQ11-C[1]

Parameter Values

Two longword integer values.

The first value specifies either a direct or a buffered data path.

This value can be one of the following:

Constant Value	Meaning
LIO$K_BUFPATH	Buffered data path
LIO$K_DIRPATH	Direct data path

The second value, which is optional, specifies a device channel number. This value can be between 1 and 8, inclusive.

Description

If you are programming the AAF01 Digital/Analog Conversion Subsystem or the ADF01 Data Acquisition Subsystem and you want to access multiple devices, you must specify a device channel number. Use the LIO$K_SET_CHAN parameter to specify the current channel number for use in subsequent routine calls to the device.

[1] This device is available only in Europe.

Restrictions

The value LIO$K_BUFPATH is provided for compatibility reasons only. You must specify LIO$K_DIRPATH for use with all VAXlab devices.

Example

```
status = LIO$SET_I (adf_id, LIO$K_DATA_PATH, 1, LIO$K_DIRPATH)
```

This routine specifies a direct data path.

LIO$K_DATA_WIDTH

This parameter specifies the width of the DRB32 parallel data path.

Supported Devices

DRB32

Parameter Values

A longword integer constant specifying the width of the parallel data path.

The value can be one of the following:

Constant Value	Meaning
LIO$K_BYTE	The data path width is a byte
LIO$K_WORD	The data path width is a word
LIO$K_LONG[1]	The data path width is a longword

[1]The default value.

Description

See the **Parameter Values**.

Restrictions

Specifying any value other than those listed in the **Parameter Values** generates an error.

Example

```
status = LIO$SET_I (device_id, LIO$K_DATA_WIDTH, 1, LIO$K_BYTE)
```

This routine sets the parallel path width to a byte.

LIO$K_DBL_BUF

This parameter enables double-buffer DMA data transfers for the ADQ32 device.

Supported Devices

ADQ32

Parameter Values

None.

Description

Use this parameter to enable double-buffer DMA data transfers for the ADQ32 device. Buffers must be larger than 8K words and queued continuously to be double-buffered.

See Section 1.6.3.4, Double-Buffer DMA, for more information. Also see Appendix A for details on how external triggers function differently in single-buffer and double-buffer mode.

Restrictions

None.

Example

```
status = LIO$SET_I (adq_id, LIO$K_DBL_BUF, 0)
```

This routine enables double-buffer DMA data transfers for the ADQ32 device.

LIO$K_DEVICE_ACK_NAK_BUFF

This parameter supplies the buffer to be used when receiving an ACK or a NAK from a serial device.

Supported Devices

Serial line

Parameter Values

Three longword integer values.

The first value specifies the address of the buffer.

The second value specifies the length of the buffer, in bytes.

The third value specifies the timeout period, in seconds.

Description

Use the LIO$K_DEVICE_ACK_NAK_BUFF parameter to supply the buffer into which LIO returns ACKs and NAKs from a serial device.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_DEVICE_ACK_NAK_BUFF, 4,
1                  ack_nak_buff, 40, 60)
```

This routine supplies a buffer called **ack_nak_buff**, 40 bytes in length, with a 60 second timeout period.

LIO$K_DEVICE_EF

This parameter supplies the device with an event flag to set when it completes a buffer. This parameter is valid only for devices set for buffer forwarding.

Supported Devices

AAV11-D
ADQ32
ADV11-D
AXV11-C A/D
DRB32
DRB32W
DRQ3B
DRV11-J
DRV11-WA
IAV11[1] devices
IDV11[1] devices
IEQ11
IEZ11
IOtech Micro488A
KWV11-C
Preston
Simpact RTC01
Disk file
Memory queue
Serial line

[1] These devices are available only in Europe.

LIO$K_DEVICE_EF

Parameter Values

A longword integer specifying an event flag.

The integer value can be any legal event flag—1 through 23, or 32 through 127—that is unique to the process. Event flags 24 through 31 are reserved for use by DIGITAL. Event flags 1 through 23 and 32 through 63 are local to the process. Event flags 64 through 127 are in global event flag clusters that may or may not currently be associated with the process.

You can also use the VMS Run-Time Library routine, LIB$GET_EF, to get a free VMS event flag to use as the value of this parameter. See the **Example**.

Description

You can use this parameter in addition to any optional event flag associated with each buffer.

Restrictions

- The device must be set for asynchronous I/O.
- The device must be set up using the LIO$K_FORWARD parameter to forward completed buffers to another device.
- The event flag should be unique to the process.
- Do not specify event flag zero. This is the VMS default event flag when no other event flag is specified.

Examples

1. `status = LIB$GET_EF(event_flag)`

   This routine gets a free VMS event flag number.

2. `status = LIO$SET_I (device_id, LIO$K_DEVICE_EF, 1, event_flag)`

   This routine sets up the device with the VMS event flag number obtained through the LIB$GET_EF routine.

LIO$K_DIAG_CHAN

This parameter enables or disables diagnostic inputs to ADQ32 channels 0, 1, and 2.

Supported Devices

ADQ32

Parameter Values

A longword integer constant enabling or disabling the diagnostic channels.

The value can be one of the following:

Constant Value	Function
LIO$K_ON	Enables the diagnostic channels
LIO$K_OFF[1]	Disables the diagnostic channels

[1]The default value.

Description

When the diagnostic channels are turned on, a precision voltage chip is connected to channels 0, 1, 2, and 3. The voltages used are shown below.

Channel	Voltage	Voltage in Bipolar Setting	Voltage in Unipolar Setting
0	Analog ground	0.0000	0.0000
1	Negative full scale plus 1/2 LSB	-9.9975	+0.0012

Channel	Voltage	Voltage in Bipolar Setting	Voltage in Unipolar Setting
2	Positive full scale minus 1.5 LSB	+9.9925	+9.9963
3	+5	+5	+5

Enabling the ADQ32 diagnostic channels is useful when you want to ensure that your software is working properly without having to connect the A/D inputs to known values.

Restrictions

None.

Example

```
status = LIO$SET_I (adq_id, LIO$K_DIAG_CHAN, 1, LIO$K_ON)
```

This routine enables the ADQ32 diagnostic channels.

LIO$K_DIRECTION

This parameter sets the direction (input or output) of a device.

Supported Devices

DRB32
DRB32W
DRV11-J
DRV11-WA
Disk file

Parameter Values

For the **DRV11-J**, four longword integer constants specifying the direction (input or output) of ports A, B, C, and D.

For the **DRB32, DRB32W, DRV11-WA**, and **disk files**, one longword integer constant specifying the direction (input or output) of the device.

The value can be one of the following:

Constant Value	Meaning
LIO$K_INPUT[1]	Sets the device or port to input or read
LIO$K_OUTPUT	Sets the device or port to output or write

[1]The default value.

Description

For the **DRB32** and **DRB32W**, this parameter sets the direction of data transfer when the device is set to use asynchronous I/O. If the device is set to use synchronous I/O, this parameter need not be set, and is ignored if it is set.

For the **DRV11-J**, this parameter sets the direction of all four ports.

LIO$K_DIRECTION

For the **DRV11-WA**, this parameter signals the LIO facility in which direction the device hardware is set.

For **disk files**, this parameter specifies whether a file is an input (read) or an output (write) file.

Restrictions

- Each of the supported devices or ports operates in one direction at a time and cannot be set to perform simultaneous input and output transfers.
- You should explicitly set up the device or port direction before you use it.

Examples

1.
```
status = LIO$SET_I (drj_id, LIO$K_DIRECTION, 4, LIO$K_INPUT,
1                   LIO$K_INPUT, LIO$K_OUTPUT, LIO$K_OUTPUT)
```

   This routine specifies the direction of the four ports of the DRV11-J device. Ports A and B are set for input; ports C and D are set for output.

2.
```
status = LIO$SET_I (drw_id, LIO$K_DIRECTION, 1, LIO$K_INPUT)
```

   This routine signals the LIO facility that the DRV11-WA device is jumpered for input.

LIO$K_DISPLAY_ONLY

This parameter sets up an interprocess memory queue to display data buffers from a global section to the second process.

Supported Devices

Memory queue

Parameter Values

None.

Description

This parameter sets the memory queue device to display data from a global section to a second process. See Chapter 2 for complete information about interprocess memory I/O.

Restrictions

- The memory queue must be set up to use the asynchronous (LIO$K_ASYNCH) I/O interface.
- The memory queue must be attached for interprocess I/O (LIO$K_INTER_PROC as the value of the **io_type** argument).
- The process that reads the data must set up its memory queue to be read-only using the LIO$K_READ_ONLY parameter.

Example

```
status = LIO$SET_I (device_id, LIO$K_DISPLAY_ONLY, 0)
```

This routine sets up the memory queue device in the first process in the main program to display data to the second process as it passes by the window.

LIO$K_DRX_AST_RTN

This parameter specifies a user-written AST routine to receive buffers when a device finishes processing them. The completed buffers are passed to the AST routine instead of being placed on the device's user queue.

Supported Devices

AAF01[1]
ADF01[1]
DRQ11-C[1]

Parameter Values

A longword integer specifying the address of the AST routine.

Description

See Chapter 3 in *Getting Started with VAXlab* and Section 1.5.3, Asynchronous System Traps (ASTs), in this guide for more information about using AST routines.

Restrictions

- The device must be set for asynchronous I/O.
- When using FORTRAN, the subroutine must be declared EXTERNAL. See the **Examples**.

[1] This device is available only in Europe.

Examples

1. `EXTERNAL user_ast`

   This line declares the user-written AST routine to be external. See the **Restrictions**.

2. `status = LIO$SET_I (device_id, LIO$K_ASYNCH, 0)`

   This routine sets up the device to use asynchronous I/O.

3. `status = LIO$SET_I (device_id, LIO$K_DRX_AST_RTN, 1, user_ast)`

   This routine specifies a user-written AST routine, **user_ast**, to receive completed buffers.

LIO$K_DRX_STAT

This parameter returns the contents of the hardware registers on the DRQ11-C device.

Supported Devices

DRQ11-C[1]

Parameter Values

A word array of length 10 returning the contents of the hardware registers on the DRQ11-C device. Each array index returns the contents of a different hardware register.

Index	Returns the contents of:
1	Command and Status Register (SCR)
2	Control Register (COR)
3	Bus Address Register 1 (BAR1)
4	Word Count Register 1 (WCR1)
5	Bus Address Register 2 (BAR2)
6	Word Count Register 2 (WCR2)
7	Current Word Counter (WCO)
8	Current Address Counter (ACO)
9	Extended Address Register (XA22)

Description

See the **Parameter Values**.

Restrictions

This is an LIO$SHOW parameter only.

[1] This device is available only in Europe.

Example

```
INTEGER*2 drx_value_list(10)
 status = LIO$SET_I (drq_id, LIO$K_DRX_STAT, drx_value_list,
1            list_length)
          IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

 SCR  = drx_value_list(1)  !Status and Command Register
 COR  = drx_value_list(2)  !Control Register
 BAR1 = drx_value_list(3)  !Bus Address Register 1
 WCR1 = drx_value_list(4)  !Word Count Register 1
 BAR2 = drx_value_list(5)  !Bus Address Register 2
 WCR2 = drx_value_list(6)  !Word Count Register 2
 WC0  = drx_value_list(7)  !Current Word Counter
 AC0  = drx_value_list(8)  !Current Address Counter
 XA22 = drx_value_list(9)  !Extended Address Register
```

This routine returns the contents of the DRQ11-C hardware registers.

LIO$K_DUPLEX

This parameter specifies whether serial line read/write requests are executed in half-duplex or full-duplex mode.

Supported Devices

Serial line

Parameter Values

A longword integer constant specifying the duplex mode.

The value can be one of the following:

Constant Value	Duplex Mode
LIO$K_HALF[1]	Half-duplex mode
LIO$K_FULL	Full-duplex mode

[1]The default value.

Description

In half-duplex mode, a single queue of read/write requests is maintained. Requests are executed sequentially in the order in which they are issued. In full-duplex mode, two queues are maintained, and requests are not retired sequentially. Write requests normally have priority. Once a read request becomes active, all write requests are queued until the read completes.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_DUPLEX, 1, LIO$K_FULL)
```

This routine sets a serial line device to operate in full-duplex mode.

LIO$K_ECHO

This parameter enables or disables the echoing of characters received on the serial line.

Supported Devices

Serial line

Parameter Values

A longword integer constant enabling or disabling echoing. The value can be one of the following:

Constant Value	Meaning
LIO$K_OFF[1]	Disables echoing
LIO$K_ON	Enables echoing

[1]The default value.

Description

This parameter enables or disables the echoing of characters received on the serial line. This parameter is typically of little use in a serial line data acquisition environment, but may be helpful while debugging application programs using serial line devices.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_ECHO, 1, LIO$K_ON)
```

This routine enables the echoing of characters received on the serial line.

LIO$K_ED_CTT

For the AAF01, this parameter enables or disables the Memory Transfer (MET) bit in the Command and Status Register (CSR).

For the ADF01, this parameter enables or disables the Control Table Transfer (CTT) bit in the Command and Status Register (CSR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer constant specifying whether to enable or disable the bit.

The value can be one of the following:

Constant Value	Function
LIO$K_ENABLE	Enables the bit
LIO$K_DISABLE	Disables the bit

Description

The DMA cycle contains the 12-bit contents of the control word that belong to the following conversion. This affects the conversion data rate by requiring another DMA cycle.

[1] This device is available only in Europe.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_ED_CTT, 1, LIO$K_ENABLE)
```

This routine either enables the MET bit in the CSR of the AAF01 device or enables the CTT bit in the CSR of the ADF01 device.

LIO$K_ED_ECE

This parameter enables or disables the External Clock Enable (ECE) bit in the Command and Status Register (CSR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer constant specifying whether to enable or disable the bit.

The value can be one of the following:

Constant Value	Function
LIO$K_ENABLE	Enables the bit
LIO$K_DISABLE	Disables the bit

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_ED_ECE, 1, LIO$K_ENABLE)
```

This routine enables the ECE bit in the CSR of the AAF01 and ADF01 devices.

[1] This device is available only in Europe.

LIO$K_ED_SBE

This parameter enables or disables the Sequence Break Enable (SBE) bit in the Command and Status Register (CSR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer constant specifying whether to enable or disable the bit.

The value can be one of the following:

Constant Value	Function
LIO$K_ENABLE	Enables the bit
LIO$K_DISABLE	Disables the bit

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_ED_SBE, 1, LIO$K_ENABLE)
```

This routine enables the SBE bit in the CSR of the AAF01 and ADF01 device.

[1] This device is available only in Europe.

LIO$K_EOI

This parameter enables or disables the assertion of the end-or-identify (EOI) line after the last byte of data is output.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer constant enabling or disabling the assertion of the EOI line.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Disables the assertion of the EOI line
LIO$K_ON	Enables the assertion of the EOI line

[1]The default value.

Description

This parameter performs the same function as the LIO$_TERM_CHAR parameter. If you enable LIO$K_EOI, LIO asserts the EOI line after the last byte of data is output.

Restrictions

None.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_EOI, 1, LIO$K_ON)
```

This routine enables the assertion of the EOI line.

LIO$K_ERR_HANDLE

This parameter specifies the way in which a device returns error conditions.

Supported Devices

All

Parameter Values

A longword integer constant specifying the error handling method.

The value can be one of the following:

Constant Value	Meaning
LIO$K_FATAL	Prints the error message to both SYS$OUTPUT and SYS$ERROR. On fatal errors, the program stops execution.
LIO$K_MESSAGE	Prints the error message to both SYS$OUTPUT and SYS$ERROR, and returns the symbolic status as the value of the routine call.
LIO$K_STATUS[1]	Returns the symbolic status as the value of the routine call.

[1]The default value.

Description

LIO$K_FATAL simplifies program execution because the program need not check the status after each LIO routine call.

LIO$K_MESSAGE simplifies the debugging of a program that handles error conditions.

LIO$K_STATUS enables a program to handle error recovery. Typically, this is useful during an interactive program execution because the program can prompt the user to correct error conditions.

Restrictions

If an LIO$_INTERR internal software error is generated, it is always fatal. This condition value indicates the existence of some error from which LIO cannot recover.

Example

```
status = LIO$SET_I (device_id, LIO$K_ERR_HANDLE, 1, LIO$K_FATAL)
```

This routine sets up a device to terminate program execution on fatal errors.

LIO$K_ERROR_ENABLE

This parameter enables or disables parity error handling for serial line devices.

Supported Devices

Serial line

Parameter Values

A longword integer constant enabling the type of parity error checking selected using the LIO$K_PARITY parameter.

Description

Use this parameter to enable the parity error checking for a serial device.

Restrictions

You must select the parity checking type using the LIO$K_PARITY parameter.

Examples

1. `status = LIO$SET_I (serial_id, LIO$K_PARITY, 1  LIO$K_EVEN)`

   This routine selects even parity checking for a serial line device.

2. `status = LIO$SET_I (serial_id, LIO$K_ERROR_ENABLE, 1, LIO$K_PARITY)`

   This routine enables the even parity checking selected by the LIO$K_PARITY parameter in the previous routine line.

LIO$K_EVENT_AST

This parameter assigns a user-written AST routine to be called on the following conditions:

- AAF01,[1] ADF01,[1] and DRQ11-C[1] unsolicited interrupts
- DRV11-J port A bit events
- IDV11-A[1] channel 15 events
- IEEE-488 bus events
- KWV11-C or Simpact RTC01 clock overflows or Schmitt trigger 2 events

Supported Devices

AAF01[1]
ADF01[1]
DRQ11-C[1]
DRV11-J
IDV11-A[1]
IEQ11
IEZ11
KWV11-C
Simpact RTC01

Parameter Values

For the **AAF01, ADF01**, and **DRQ11-C**, three longword integer values.

The first value specifies the address of the user-supplied AST routine. If this value is zero, a disconnect from unsolicited interrupt is performed. This disables the delivery of further attention AST's.

The second value specifies the address of a 16-bit parameter passed to the AST routine. On entry to the AST routine, this parameter contains the Status and Command Register (SCR) of the DRQ11-C at the time of the interrupt.

[1] This device is available only in Europe.

The third value, which is optional, may contain the integer constant LIO$K_CANCEL. This value clears any previously set interrupt condition.

ASTs are immediately disabled after the delivery of one AST to the user program. The user must specify another attention AST to reenable notification.

For the **DRV11-J**, two longword integer values. The first value specifies the address of the user-supplied AST routine. The second value specifies the number of the bit to which you are assigning the AST routine (0 is the lowest bit, 15 is the highest bit).

For the **IDV11-A**, two longword integer values. The first value specifies the address of the user-supplied AST routine. The second value specifies a 16-bit parameter to be passed to the AST routine.

For the **IEQ11**, the **IEZ11**, the **KWV11-C**, and the **Simpact RTC01**, a longword integer value specifying the address of the user-supplied AST routine.

Description

For the **DRV11-J**, an event is a transition on one of the bits of port A. Whether an event is a positive-going transition or a negative-going transition is determined by the value of the LIO$K_POLARITY parameter.

For the **IDV11-A**, an event is a transition on channel 15 of the device. Whether an event is a positive-going transition or a negative-going transition is determined by the value of the LIO$K_POLARITY parameter.

For the **IEQ11** and the **IEZ11**, an event is an IEEE-488 bus event enabled by the LIO$K_EVENT_ENA parameter.

For the **KWV11-C** and the **Simpact RTC01**, an event is either a clock overflow or a pulse on Schmitt trigger 2, depending on the value of the LIO$K_FUNCTION parameter.

Restrictions

The following restrictions apply to the DRV11-J:

- The DRV11-J must be attached through the LIO$ATTACH routine with LIO$K_QIO as the value of the **io_type** argument.
- The AST routine is called when the specified bit is cleared or set, depending on the value of the LIO$K_POLARITY parameter.
- The AST routine must have one parameter which is the device ID of the DRV11-J that interrupted.
- Bits 12 through 15 are not available if the device is jumpered for handshaking.

The following restriction applies to the IDV11-A:

- You must set up the IDV11-A using the LIO$K_POLARITY parameter before you supply an event AST routine using the LIO$K_EVENT_AST parameter.

The following restriction applies to the IEQ11 and the IEZ11:

- If you declare an event AST using this parameter, you cannot use the LIO$K_EVENT_WAIT parameter.

The following restriction applies to the KWV11-C and the Simpact RTC01:

- The KWV11-C and the Simpact RTC01 must be attached through the LIO$ATTACH routine with LIO$K_QIO as the value of the **io_type** argument.

Examples

An AST routine is a normal user-written subroutine that can require one or more arguments, depending on the LIO device with which it is used. The first argument of every AST routine specifies the device ID of the device that detects the event.

1. `EXTERNAL drv_ast(drj_id)`

   This line associates an AST routine, **drv_ast**, with the DRV11-J device referred to by the **drj_id** argument. When used with the DRV11-J devices, you must also specify **drj_id**.

2. `status = LIO$SET_I (drj_id, LIO$K_EVENT_AST, 2, drv_ast, 0)`

   This routine sets up the DRV11-J device to call the AST routine, **drv_ast**, on an event when bit 0 is either set or cleared.

3. `EXTERNAL ieee_ast(event_code, event_specific, unit, controller)`

   This line associates an AST routine, **ieee_ast**, with the IEQ11 or the IEZ11 device referred to by the **ieee_id** argument. You also need to include the **event_code** and **event_specific** arguments to return the event that occurred, and event-specific information associated with the event, if any. (For the IEZ11, the event-specific information is always 0.)

4. `status = LIO$SET_I (ieee_id, LIO$K_EVENT_AST, 1, ieee_ast)`

   This routine sets up the IEEE-488 device to call the AST routine, **ieee_ast**, when an event (enabled by LIO$K_EVENT_ENA) occurs.

5. `EXTERNAL clock_ast(clock_id)`

   This line associates an AST routine, clock_ast, with the KWV11-C or the Simpact RTC01 device referred to by the **clock_id** argument. When used with the KWV11-C and the Simpact RTC01 devices, you only need to specify **clock_id**.

6. `status = LIO$SET_I (clock_id, LIO$K_EVENT_AST, 1, clock_ast)`

   This routine sets up the KWV11-C or the Simpact RTC01 device to call the AST routine, **clock_ast**, on an event.

LIO$K_EVENT_EF

This parameter specifies an event flag to be set on an external event or clock overflow.

Supported Devices

DRV11-J
KWV11-C
Simpact RTC01

Parameter Values

For the **DRV11-J**, two longword integer values specifying the event flag number and the number of the port A bit to which you are assigning the event flag (0 is the lowest bit, 15 is the highest bit).

For the **KWV11-C** and the **Simpact RTC01**, a longword integer specifying the event flag number.

For all three devices, the event flag number can be any legal event flag—1 through 23, or 32 through 127—that is unique to the process. Event flags 24 through 31 are reserved for use by DIGITAL.

Event flags 1 through 23 and 32 through 63 are local to the process. Event flags 64 through 127 are in global event flag clusters that may or may not currently be associated with the process.

You can also use the VMS Run-Time Library routine, LIB$GET_EF, to get a free VMS event flag to use as the value of this parameter. See the **Example**.

Description

See the **Parameter Values**.

Restrictions

The following restrictions apply to the DRV11-J:

- The DRV11-J must be attached through the LIO$ATTACH routine with the LIO$K_QIO parameter.
- The event flag is set when the specified bit is cleared or set, depending on the value of the LIO$K_POLARITY parameter.
- Bits 12 through 15 are not available if the device is jumpered for handshaking.
- The bit interrupt capability in the DRV11-J works only on port A.

The following restrictions apply to the KWV11-C and the Simpact RTC01:

- The KWV11-C and the Simpact RTC01 must be attached through the LIO$ATTACH routine with LIO$K_QIO as the value of the **io_type** argument.
- The clock rate must be slow enough to allow the event flag to be set between clock overflows.

Examples

1. `status = LIB$GET_EF (event_flag)`

   This routine gets a free VMS event flag number.

2. `status = LIO$SET_I (device_id, LIO$K_EVENT_EF, 1, event_flag)`

   This routine sets up the device with the VMS event flag number obtained through the LIB$GET_EF routine.

LIO$K_EVENT_ENA

This parameter enables recognition of specified IEEE-488 bus events. When the specified event occurs, the device (the IEQ11, IEZ11, or IOtech Micro488A) can then recognize the event and respond if appropriate.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

One or more longword integer constants specifying the bus events for the IEEE-488 bus to recognize.

The values can be any of the following:

Constant Value	Event Meaning
LIO$K_DEADDR_EVT	The device has been deaddressed. This event is detectable only when the device is not the controller-in-charge.
LIO$K_DEV_CLR_EVT	The controller-in-charge has sent the "device clear" command. The instrument should reset itself to its power-up state. Remember that user-written application programs are responsible for all instrument functions. The instrument should return to its initial state. This event is detectable only when the device is not the controller-in-charge.
LIO$K_DEV_TRIG_EVT	The controller-in-charge has sent the "device trigger" command. The instrument should trigger as specified in the user-written application program. This event is detectable only when the device is not the controller-in-charge.

LIO$K_EVENT_ENA

Constant Value	Event Meaning
LIO$K_EXT_LNR_EVT	The controller-in-charge is addressing the device as an extended (secondary) listener. This event is detectable only when the device is not the controller-in-charge. (LIO$K_EXT_LNR_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_EXT_TKR_EVT	The controller-in-charge is addressing the device as an extended (secondary) talker. This event is detectable only when the device is not the controller-in-charge. (LIO$K_EXT_TKR_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_IFC_EVT	The system controller is signalling the device to clear its bus interface. This does not generally affect the internal state of the instrument. (LIO$K_IFC_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_LNR_ADDR_EVT	The controller-in-charge is addressing the device as a listener. This event is detectable only when the device is not the controller-in-charge.
LIO$K_PAR_POLL_CONFIG_EVT	The controller-in-charge is signalling the device to configure itself for parallel polling. This event is detectable only when the device is not the controller-in-charge. (LIO$K_PAR_POLL_CONFIG_EVT is not supported for the IOtech Micro488A.)
LIO$K_PAR_POLL_UNCONFIG_EVT	The controller-in-charge is signalling the device to unconfigure itself for parallel polling. This event is detectable only when the device is not the controller-in-charge. (LIO$K_PAR_POLL_UNCONFIG_EVT is not supported for the IOtech Micro488A.)

Constant Value	Event Meaning
LIO$K_REC_CTRL_EVT	The device has received control from the current controller-in-charge. This event is detectable only when the device is not the controller-in-charge, and it is attached as controller or system controller.
LIO$K_REM_LOCAL_EVT	The device state has changed from remote to local, or from local to remote. The current state of the device is returned by the LIO$K_EVENT_WAIT parameter, or by the AST routine set up by the LIO$K_EVENT_AST parameter. This event is detectable only when the device is not the controller-in-charge. (LIO$K_REM_LOCAL_EVT is not supported for the IEZ11 or the IOtech Micro488A.)
LIO$K_SRQ_EVT	A device is requesting service. This event is detectable only when the device is the controller-in-charge.
LIO$K_TKR_ADDR_EVT	The controller-in-charge is addressing the device as a talker. For this event to be detectable, the device must be the controller-in-charge.

Description

The parameter enables the detection of the events listed in the **Parameter Values**. All events must be enabled using this parameter before they can be detected with either the LIO$K_EVENT_WAIT or the LIO$K_EVENT_AST parameters.

Restrictions

- Some events are only detectable when the device is in a specific state, such as controller-in-charge; however, they can be selected in any state.
- If either the LIO$K_DEV_CLEAR_EVT or LIO$K_DEV_TRIG_EVT events are enabled in conjunction with either the LIO$K_LNR_ADDR_EVT or the LIO$K_TKR_ADDR_EVT events, then a false listener or a false talker AST is delivered before the "device clear" or the "device trigger" AST.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_EVENT_ENA, 1, LIO$K_TKR_ADDR_EVT)
```

This routine enables an IEEE-488 device to recognize and respond when the controller-in-charge addresses the device as a talker.

LIO$K_EVENT_WAIT

This parameter waits for any enabled IEEE-488 bus event to occur, and then returns it.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer array of length two.

The first value returns the event that has occurred. (See the reference description of the LIO$K_EVENT_ENA parameter for information about the event values returned.)

The second value returns event-specific information associated with the event, if any.

The following list describes which events return event-specific information, and what the information means.

- The events LIO$K_LNR_ADDR_EVT and LIO$K_TKR_ADDR_EVT return the secondary address, if secondary addressing is enabled by the LIO$K_IEEE_ADDR parameter.
- The event LIO$K_REM_LOCAL_EVT returns the current state of the device: zero for local, one for remote.
- The LIO$K_PAR_POLL_CONFIG_EVT event returns either parallel poll disable byte or parallel poll enable byte: zero for disabled, one for enabled.
- All other events return a zero.

LIO$K_EVENT_WAIT

Description

This parameter waits for an enabled event to occur, and then returns the event and any event-specific information associated with it.

Restrictions

- This is an LIO$SHOW parameter only.
- You must first enable the recognition of one or more events using the LIO$K_EVENT_ENA parameter.
- You cannot use this parameter if an event AST is declared by the LIO$K_EVENT_AST parameter.

Examples

1. `INTEGER*4 event(2)`

   This line declares the variable **event** to be an integer array of length two. The event and any event-specific information are returned in **event(1)** and **event(2)**, respectively.

2. `INTEGER*4 length`

   This line declares the variable **length** to be an integer. The **length** argument returns the number of bytes in the **event** argument.

3. `status = LIO$SHOW (ieee_id, LIO$K_EVENT_WAIT, event, length)`

   This routine waits for an IEEE-488 bus event to occur and then returns the event and event-specific information in **event**. The **length** argument returns the number of bytes in the **event** argument.

4. `IF (event(1) .EQ. LIO$K_SQR_EVT)`

   ```
   .
   .
   .
   ```

   This line checks for a service request and then process flow continues.

LIO$K_FILE_EXTENT

This parameter enables the extension of the output file beyond the size (in blocks) set through the LIO$K_FILE_SIZE parameter.

Supported Devices

Disk file

Parameter Values

A longword integer specifying the number of blocks by which to extend the file.

Description

Normally, outputting a buffer to a file too small to contain the buffer generates an error. Through this parameter, you can specify a file extension size. If you output a buffer to a file that has too few remaining blocks to contain the buffer, the software extends the file by the number of blocks you specified through LIO$K_FILE_EXTENT, and continues writing the buffer. **LIO extends the output file only once per write operation.** However, if successive writes overrun the extent area, the file is extended again.

Extents are allocated with best try at contiguous disk space.

Allocating a file extension takes time. When using time-critical applications, try to allocate enough file space through the LIO$K_FILE_SIZE parameter.

Restrictions

- This parameter is valid only for output files.
- You must use this parameter in your program before you open the output file.
- The file extension fails if the size of the buffer being written to the file overruns the file after it has been extended. The file can be extended only once for each buffer or write operation.

Example

```
status = LIO$SET_I (device_id, LIO$K_FILE_EXTENT, 1, 1)
```

This routine specifies the file extension size as one VMS block.

LIO$K_FILE_POS

This parameter repositions the current block pointer in a file.

Supported Devices

Disk file

Parameter Values

A longword integer specifying the block number where the pointer should be positioned.

Description

You can use this parameter to move to a different part of a disk file. The pointer moves to the beginning of the block you specify.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_FILE_POS, 1, 20)
```

This routine sends the file pointer to the beginning of block 20 of the file.

LIO$K_FILE_REMAIN

This parameter returns the number of remaining blocks available in an output file.

Supported Devices

Disk file

Parameter Values

A longword integer returning the number of remaining blocks available in the output file.

Description

You can use this parameter with disk files to determine the number of blocks remaining in an output file.

Restrictions

- You must open the file using the LIO$K_OPEN_FILE parameter before you can determine the number of remaining blocks.
- This is an LIO$SHOW parameter only.

Examples

1. `INTEGER*4 number`

   This line declares the variable **number** to be an integer. The number of blocks remaining in an output file is returned in this variable.

2. `INTEGER*4 length`

   This line declares the variable **length** to be an integer. The **length** argument returns the number of bytes in the **number** argument.

3. `status = LIO$SHOW (device_id, LIO$K_FILE_REMAIN, number, length)`

   This routine returns the number of blocks remaining to be written in an output file. The **number** argument returns the number of remaining blocks. The **length** argument returns the number of bytes in the **number** argument.

LIO$K_FILE_SIZE

This parameter specifies the size of an output file in blocks.

Supported Devices

Disk file

Parameter Values

A longword integer specifying the file size in number of blocks. A VMS block is 512 bytes or 256 words (INTEGER*2) in size.

Description

Use this parameter to specify the size of an output file in blocks.

This parameter allocates the file as a single, logically contiguous piece of disk space. If sufficient logically contiguous disk space is not available, the file creation will fail. To correct this problem, allocate a small file and then extend it, or backup and restore your disk to reduce fragmentation. Subsequent extents are allocated with best try at contiguous disk space.

Restrictions

- This parameter is valid only for output files.
- You must specify the output file size before opening the file using the LIO$K_OPEN_FILE parameter.

Example

```
status = LIO$SET_I (device_id, LIO$K_FILE_SIZE, 1, 10)
```

This routine allocates an output file 10 blocks in size.

LIO$K_FLOW_CONTROL

This parameter establishes the method of flow control for serial line devices.

Supported Devices

Serial line

Parameter Values

A longword integer constant that establishes the flow control.

The value can be one of the following:

Constant Value	Flow Control
LIO$K_XOFF_XON[1]	XOFF/XON
LIO$K_DSR_DTR[2]	DSR/DTR
LIO$K_NO_FLOW	No flow control
LIO$K_RTS_CTS[2]	RTS/CTS

[1]The default value.

[2]Supported on DECserver 200-based lines only.

Description

Flow control is a method of temporarily suspending the data flow to or from a device when there are not enough buffers available to meet the demands of the device.

Restrictions

None.

LIO$K_FLOW_CONTROL

Example

```
status = LIO$SET_I (serial_id, LIO$K_FLOW_CONTROL, 1, LIO$K_XOFF_XON)
```

This routine establishes XOFF/XON flow control for a serial line device.

LIO$K_FLOW_MASTER

This parameter specifies how XOFF/XON flow control is used for serial line devices.

Supported Devices

Serial line

Parameter Values

A longword integer constant.

The value can be one of the following:

Constant Value	Function
LIO$K_DEVICE	The device sends XOFF/XON to the host to signal when to start or stop a data transfer.
LIO$K_HOST[1]	The hosts sends XOFF/XON to the device to signal when to start or stop a data transfer.
LIO$K_BOTH	Enables both LIO$K_DEVICE and LIO$K_HOST.
LIO$K_READ	The host explicitly solicits all read operations with XON and terminates each operation with XOFF.

[1]The default value.

Description

See the **Parameter Values**.

LIO$K_FLOW_MASTER

Restrictions

You must use the LIO$K_FLOW_CONTROL parameter to enable the flow control type as LIO$K_XOFF_XON before you use this parameter to specify how the flow control is used.

Example

```
status = LIO$SET_I (serial_id, LIO$K_FLOW_MASTER, 1, LIO$K_DEVICE)
```

This routine sets up the serial device to send XOFF/XON to the host to signal when to start or stop a data transfer.

LIO$K_FORWARD

This parameter specifies the device to forward buffers to when this device completes a buffer.

Supported Devices

AAV11-D
ADQ32
ADV11-D
AXV11-C A/D
DRB32
DRB32W
DRQ3B
DRV11-J
DRV11-WA
IAV11[1] devices
IDV11-A[1] devices
IEQ11
IEZ11
KWV11-C
Preston
Simpact RTC01
Disk file
Memory queue
Serial line

Parameter Values

A longword integer specifying the device ID of the device to which you want the current device to forward buffers.

[1] These devices are available only in Europe.

LIO$K_FORWARD

Description

Use this parameter to designate another device to receive buffers from the current device when the buffers are complete. The buffers are automatically enqueued to the specified receiving device instead of being placed on the user queue of the current device.

Restrictions

- Both the forwarding and receiving devices must be set to use the asynchronous I/O interface.
- The forwarding device must be set up with a device event flag (LIO$K_DEVICE_EF).

Example

```
status = LIO$SET_I (fwd_device_id, LIO$K_FORWARD, 1, rec_device_id)
```

This routine sets up the device referred to in this example as **fwd_device_id** to forward buffers to the device referred to in this example as **rec_device_id**.

LIO$K_FUNCTION

This parameter sets up the function the clock is to perform.

Supported Devices

KWV11-C
Simpact RTC01

Parameter Values

A longword integer constant specifying the function the clock is to perform. The function can be one of the following:

Constant Value	Clock Function
LIO$K_SGL_COUNT	Single count. The clock counts one clock tick (clock source rate divided by the divider), then produces a pulse and stops. The rate of the clock can be specified using LIO$K_CLK_SRC or LIO$K_CLK_RATE. If the device is set for QIO and you specified an AST routine through the LIO$K_EVENT_AST parameter, then the AST routine is called. If the LIO$K_EVENT_EF parameter was also used to set up the device, then the event flag is set. Using the Schmitt trigger 1 (ST1) as the clock source provides a way to produce a pulse after a fixed number of external events (pulses on the ST1 input).
LIO$K_REP_COUNT	Repeat count. The clock runs continuously, producing pulses at the base rate divided by the divider. The rate of the clock can be specified using LIO$K_CLK_SRC or LIO$K_CLK_RATE.

Constant Value	Clock Function
	If the clock output of the KWV11-C or the Simpact RTC01 is wired to another device, such as the AAV11-D, ADV11-D, or AXV11-C, it can then serve as the clock source for that device. If the clock is set for QIO and an AST routine is specified through the LIO$K_EVENT_AST parameter, then the AST routine is called every time the clock ticks. If Schmitt trigger 1 is specified as the source, the divider represents the number of ST1 pulses that must occur before the KWV11-C or the Simpact RTC01 produces a clock tick.
LIO$K_EVENT_ABS	Event timer. In this mode, the clock begins counting either immediately (when the LIO$READ or LIO$ENQUEUE statement executes) or at the first ST1 trigger. (See LIO$K_TRIG for more information.) On each ST1 trigger (including the first if LIO$K_TRIG selects ST1 start), the current value of the clock's counter is written to the buffer. The value is in ticks of the clock source specified in the LIO$K_CLK_SRC parameter. Time is determined by multiplying the number of ticks by the frequency of the clock source. The clock continues to increment after each ST2 event. If the clock is set for QIO and an AST routine is specified through the LIO$K_EVENT_AST parameter, then the AST routine is called when each Schmitt trigger 2 pulse is received. For the KWV11-C, the time is 16 bits wide. When it reaches 65,535 ($2^{16}-1$) and increments, it wraps to zero. For the Simpact RTC01, the time is 32 bits wide. When it reaches $2^{32}-1$ and increments, it wraps to zero. If the counter overflows between ST2 pulses, the input buffer is terminated.

Constant Value	Clock Function
	If the clock source (set by LIO$K_CLK_SRC) is the internal crystal frequency or the line frequency, then the time interval can be calculated from the base rate. This is useful for timing Schmitt trigger 2 pulses.
	If the clock source is the Schmitt trigger 1, then the interval is in ticks of the external source. You can use this to count the number of ST1 pulses that occur in a known interval (when Schmitt trigger 2 is connected to a known frequency source).
LIO$K_EVENT_REL	Event timer, relative time.
	This parameter value operates exactly the same way as the event timer with absolute time, except the counter is reset to zero on each Schmitt trigger 2 pulse.

Description

See the **Parameter Values**.

Restrictions

- For the LIO$K_EVENT_ABS and LIO$K_EVENT_REL functions, if the counter overflows twice between ST2 pulses, the input buffer is terminated.
- Timeout values only apply to the LIO$K_EVENT_ABS and LIO$K_EVENT_REL functions.

Example

```
status = LIO$SET_I (clock_id, LIO$K_FUNCTION, 1, LIO$K_REP_COUNT)
```

This routine specifies the repeat count function. The clock runs continuously, producing pulses at the rate specified by the LIO$K_CLK_RATE parameter.

LIO$K_FUNCTION_BITS

This parameter sets the function bits associated with the device.

Supported Devices

AAF01[1]
ADF01[1]
DRB32
DRQ11-C[1]
DRQ3B

Parameter Values

For the **AAF01, ADF01**, and **DRQ11-C**, a word integer value specifying the function bits to be placed in the SCR register.

The following table lists the valid parameter values and the effect each value has on the state of the function bits.

	Function Bits			
Value	**4**	**3**	**2**	**1**
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	1	0	0
12	1	1	0	1
13	1	1	1	1

[1] This device is available only in Europe.

For the **DRB32**, a longword integer value whose least significant byte contains a value to write to the output control port. When used with the LIO$SHOW routine, this parameter returns a value whose least significant byte contains the value of the input status port.

For the **DRQ3B**, a longword integer value that sets the state of the 3 function bits on either the input port or the output port, depending on whether you attach the device as HXA0 (input port) or HXA1 (output port).

The following table lists the valid parameter values and the effect each value has on the state of the function bits.

Value	Function Bits 3	2	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

You do not specify a parameter value when you use this parameter with the LIO$SHOW routine.

Description

The DRQ3B has six input and six output general purpose bits. Three input pins and three output pins are located on each port's connector. The number of each pin corresponds to the bit in the function registers.

These pins are completely independent of the operation of the port, and are available for customized use.

Examples of such use might be remote power-on of external devices, or additional information transfer, such as monitoring the status of an external device.

The types of uses possible with the output function bits depend on the signals present on the external device. Data written to an output function register is latched into the output function bits, that is, the output function bits continually assert the signals written to the output function register until new data is written to the register.

An input function register allows your program to read data present on the input function bits. This register is not latched and continually reflects the signals present on the input functions bits.

When used with the LIO$SET_I routine, this parameter sets function bits 1-3 on the input or output port, depending on which port you attach using the LIO$ATTACH routine.

When used with the LIO$SHOW routine, the high word of the longword is unused. The low word contains the state of all the function bits on the DRQ3B (both input and output ports).

Figure 4-1 shows how to read what is returned by the LIO$SHOW routine.

Figure 4-1: State of the Function Bits on the DRQ3B

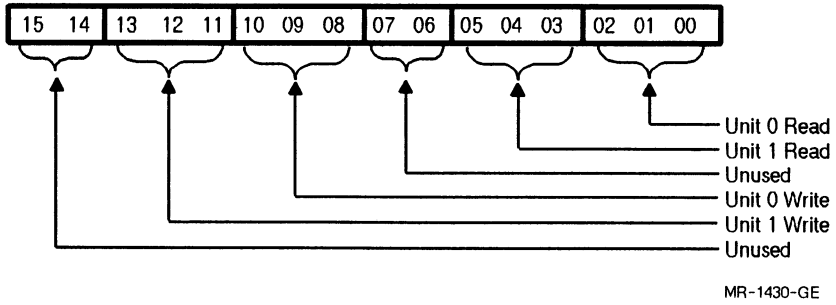


Table 4–9 shows the correspondence between this word and the actual pins on the DRQ3B connectors.

Table 4–9: Pin Numbers on the DRQ3B

Unit	Bit Number	Function Bit Pin Number
Unit 0 Read	00	FUNCT IN 0
	01	FUNCT IN 1
	02	FUNCT IN 2
Unit 1 Read	03	FUNCT IN 3
	04	FUNCT IN 4
	05	FUNCT IN 5
Unused	06	
Unused	07	
Unit 0 Write	08	FUNCT OUT 0
	09	FUNCT OUT 1
	10	FUNCT OUT 2
Unit 1 Write	11	FUNCT OUT 3
	12	FUNCT OUT 4
	13	FUNCT OUT 5
Unused	14	
Unused	15	

The DRB32 has eight output function bits called the output control port, and eight input function bits called the input status port. The LIO$SET_I routine passes a parameter whose least significant byte contains the value to write to the output control port. The LIO$SHOW routine returns a longword value whose least significant byte contains the value read from the input status port.

LIO$K_FUNCTION_BITS

Restrictions

None.

Example

```
status = LIO$SET_I (drq_id, LIO$K_FUNCTION_BITS, 1, 7)
```

This routine sets the three DRQ3B function bits on the input port or the output port, depending on which port you attached.

LIO$K_GATE

This parameter sets up the external gating used with the ADQ32 device.

Supported Devices

ADQ32

Parameter Values

A longword integer constant specifying how external gating is used with the ADQ32 device.

The value can be one of the following:

Constant Value	Meaning
LIO$K_EDGE	The A/D converter begins taking data when the external gate input goes low, and stops taking data when the external gate input goes low again.
LIO$K_EDGE_DELAY	The same as LIO$K_EDGE except that the gate is delayed one tick of the sweep clock before it takes effect. The delay time is 1/(sweep rate), where the sweep rate is set using the LIO$K_SWEEP_RATE parameter.
LIO$K_LEVEL	The A/D converter begins taking data when the external gate input is high, and stops taking data when the external gate goes low.
LIO$K_OFF[1]	No gating.

[1]The default value.

LIO$K_GATE

Description

Use the LIO$K_GATE parameter to control when the A/D takes data. When the external gate opens, the A/D takes data until the external gate closes. Starting and stopping the data acquisition continues as long as there are buffers to fill.

When level gating or edge gating the ADQ32 with the sweep trigger source different from the point trigger source, the gate controls the sweep trigger source, but not the point trigger source. For example, when using level gating, the gate may go low after the sweep starts but before the sweep completes. The current sweep still completes, but no further sweeps can start until the gate goes high again.

See Appendix A for more information.

Restrictions

- Gating is not possible when the external gate/trigger input is used to trigger sweeps or to trigger the buffer.
- Level gating is not available in burst point mode (LIO$K_TRIG values LIO$K_BURST, LIOK_SAME, LIOK_SAME), or external trigger mode (LIO$K_TRIG values LIO$K_EXTERNAL, LIOK_SAME, LIOK_SAME).
- Delayed gating is not available in sweep modes.

Examples

This example shows how to set up the ADQ32 trigger mode, the primary clock rate, the sweep clock rate, and external gating with a delay.

1.
```
status = LIO$SET_I (device_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                  LIO$K_SAME, LIO$K_SAME)
```

   This routine specifies the primary clock of the ADQ32 as the trigger source for the device, and defaults the sweep trigger and the buffer trigger values.

2.
```
status = LIO$SET_R (device_id, LIO$K_CLK_RATE, 1, 1000.0)
```

   This routine specifies the rate for the primary clock as 1 kHz.

3.
```
status = LIO$SET_R (device_id, LIO$K_SWEEP_RATE, 1, 10.0)
```

   This routine specifies the rate for the sweep clock as 10 Hz. This sets a delay time of 1/10 of a second.

4.
```
status = LIO$SET_I (device_id, LIO$K_GATE, 1, LIO$K_EDGE_DELAY)
```

   This routine specifies external gating on an edge with a delay. The length of the delay (1/10 of a second) is specified by the value of the LIO$K_SWEEP_RATE parameter.

LIO$K_HANDSHAKE

This parameter specifies whether or not the DRV11-J is jumpered to use a two-wire handshake for each port. This affects all four ports.

Supported Devices

DRV11-J

Parameter Values

A longword integer constant specifying whether or not the DRV11-J is jumpered to use a two-wire handshake.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Disables handshake
LIO$K_ON	Enables handshake

[1]The default value.

Description

If the DRV11-J is used to transfer data to or from an external device, using a two-wire handshake controls the flow and prevents data from being lost. If the DRV11-J is used to sense or set individual control lines, you should not use handshaking.

The DRV11-J hardware must be physically jumpered to perform two-wire handshaking by inserting jumper W11. The LIO$K_HANDSHAKE parameter is used to enable or disable handshaking in the software context. If jumper W11 is inserted to enable the hardware for two-wire handshaking, but the value of LIO$K_HANDSHAKE is set to LIO$K_OFF, then handshaking is not used.

Restrictions

- The DRV11-J must be jumpered for a two-wire handshake to transfer more than one data point for each buffer.
- If handshaking is enabled, only the low 12 bits of port A can be set to call AST routines.
- The value of the LIO$K_POLARITY parameter also affects the polarity of the handshake.

Example

```
status = LIO$SET_I (device_id, LIO$K_HANDSHAKE, 1, LIO$K_ON)
```

This routine enables the two-wire handshaking feature of the DRV11-J when jumper W11 is inserted.

LIO$K_HANGUP

This parameter disconnects or "hangs up" a modem line.

Supported Devices

Serial line

Parameter Values

None.

Description

Use this parameter to disconnect a serial line device connected as a modem.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_HANGUP, 0)
```

This routine disconnects a serial line device connected as a modem.

LIO$K_IEEE_ADDR

This parameter sets up the IEEE-488 bus address of the device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

Two longword values.

The first value specifies the primary address of the device on the IEEE-488 bus. This value can be between 0 and 30, inclusive.

The second value, which is optional, enables or disables the recognition of the secondary addressing.

This value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Disables secondary addressing
LIO$K_ON	Enables secondary addressing

[1]The default value.

Description

This parameter must be set immediately after the device is attached using the LIO$ATTACH routine. Until a primary bus address is assigned to the device, it does not exist on the IEEE-488 bus. No further LIO routine or LIO$SET calls can be made to the device until it exists on the IEEE-488 bus.

An IEEE extended address is a primary address followed by a secondary address.

LIO$K_IEEE_ADDR

When extended addressing is enabled, the device can detect being addressed as a secondary listener or as a secondary talker. The LIO$K_EVENT_ENABLE set parameter enables detection of these events.

The secondary address is passed as the event specific argument if an event AST is used. If the show parameter LIO$K_EVENT_WAIT is used, the secondary address is returned in the second element of the LIO$SHOW array. The action taken based on a secondary address depends on your application.

NOTE

In addition to using this parameter to set up the IEEE-bus address for the IOtech Micro488A, you must also set DIP switches on the device. See Section 2.5.2.1, IOtech Micro488A DIP Switch Settings, for more information.

Restrictions

- You must use this parameter to set the primary address of the device on the IEEE-488 bus before you set up any other device parameters.
- An IEZ11 or an IOtech Micro488A device cannot be addressed as a secondary listener or a secondary talker. However, an IEZ11 or a Micro488A can generate secondary addresses when it is controller-in-charge.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_IEEE_ADDR, 2, 3, LIO$K_ON)
```

This routine sets the primary address of the device on the IEEE-488 bus and enables secondary addressing. Once the primary address is set, the device exists on the IEEE-488 bus and device setup can continue.

LIO$K_INIT_AD_CHAN

This parameter initializes a Preston A/D channel list and clears any existing channel list.

Supported Devices

Preston

Parameter Values

None.

Description

This parameter initializes the channel list for subsequent use with the LIO$K_ADD_AD_CHAN parameter. This parameter also clears any existing channels and sets the channel count to zero.

Restrictions

Any previously set channel list is lost.

Example

```
status = LIO$SET_I (device_id, LIO$K_INIT_AD_CHAN, 0)
```

This routine initializes a Preston A/D channel list.

LIO$K_INPUT_TERMINATOR

This parameter establishes a termination character on the input side of a serial line.

Supported Devices

Serial line

Parameter Values

A character string specifying the valid termination character.

This value is passed by descriptor.

To disable a previously set-up input termination character, specify 0.

Description

Data is input into a user buffer until the user-specified termination character is encountered in the input stream. When the termination character is encountered, the user buffer is completed. A termination character is always the last character in a buffer.

Restrictions

You can specify only one termination character for use at any given time. If you change the termination character, the new termination character supersedes any previously specified termination character.

Example

```
status = LIO$SET_S (serial_id, LIO$K_INPUT_TERMINATOR, 'A')
```

This routine specifies the letter A as the valid input buffer termination characters. Data is input into a buffer until an A is encountered.

LIO$K_INTERRUPT_LEVEL

This parameter sets the level at which interrupts occur for the Simpact RTC01 clock device.

Supported Devices

Simpact RTC01

Parameter Values

A longword integer constant specifying the interrupt level.

The value can be 4, 5, 6, or 7. (These values are equivalent to the VAX interrupt priority levels 20, 21, 22, and 23.)

The default value is 4.

Description

Use this parameter to set the interrupt level for requesting interrupts to the host when internal or external events occur.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_INTERRUPT_LEVEL, 1, 4)
```

This routine sets the interrupt level to 4.

LIO$K_LEAVE_IN_STATE

This parameter specifies whether or not to leave the IEEE-488 device in the controller-standby state after data I/O.

Supported Devices

IEQ11

Parameter Values

A longword integer constant specifying whether or not to leave the IEEE-488 device in the operating state required to process the subsequent I/O request.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Changes the operating state
LIO$K_ON	Leaves the device in the current operating state

[1]The default value.

Description

When the IEQ11 device is controller-in-charge, it automatically changes its operating state to process an I/O request.

For increased efficiency, specify the parameter value as LIO$K_ON to prevent the device from automatically changing state. The device changes to the appropriate state to perform the next I/O request, and then remains in that state for all subsequent requests until the feature is disabled.

When the I/O transfers complete, you disable this feature by setting this parameter specifying LIO$K_OFF.

Restrictions

- The IEEE-488 device must be the controller-in-charge.
- This parameter is valid only for LIO$READ, LIO$WRITE, and LIO$ENQUEUE calls.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_LEAVE_IN_STATE, 1, LIO$K_ON)
```

This routine causes the IEEE-488 device to remain in its current operating state during subsequent I/O transfers.

LIO$K_LOCK_BUFFER

This parameter locks buffers before beginning DMA data transfers with the DRB32 device.

Supported Devices

DRB32

Parameter Values

Two longword integer values.

The first value specifies the buffer size in bytes.

The second value specifies the buffer to lock.

Description

This parameter signals the device driver that a buffer is locked and that paging it into memory is not necessary. Using LIO$K_LOCK_BUFFER saves the overhead of locking buffers while the I/O is in progress.

Using this parameter with the LIO$SHOW routine returns the address and size of the buffer in bytes.

You can unlock buffers using the LIO$K_UNLOCK_BUFFER parameter.

Restrictions

- You must lock buffers before performing DMA data transfers.
- You can lock a maximum of 16 buffers, each with a maximum size of 960K bytes.

Example

```
status = LIO$SET_I (device_id, LIO$K_LOCK_BUFFER, 2, buffer_address, 4096)
```

This routine locks one 4096-byte buffer at address **buffer_address**.

LIO$K_LOOP_BACK

This parameter enables and disables the loopback mode of the DRB32.

Supported Devices

DRB32

Parameter Values

A longword integer enabling or disabling the loopback mode. Specifying a nonzero value enables loopback. Specifying zero disables loopback.

Description

This parameter places the DRB32 into internal loopback mode for testing.

Restrictions

The DRB32 device must be idle, that is, no data transfers in progress or pending when this parameter is set.

Examples

1. `status = LIO$SET_I (device_id, LIO$K_LOOP_BACK, 1, 1)`

   This routine enables loopback mode.

2. `status = LIO$SET_I (device_id, LIO$K_LOOP_BACK, 1, 0)`

   This routine disables loopback mode.

LIO$K_MAX_CHANNELS

This parameter specifies the maximum number of channels that can be plotted.

Supported Devices

Real-time plotting

Parameter Values

A longword integer specifying the maximum number of channels that can be plotted. The default value is 8.

Description

Use this parameter to specify the maximum number of channels to be plotted.

Restrictions

- The number of channels is limited by the amount of available memory and the size of the workstation screen.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_MAX_CHANNELS, 1, 5)
```

This routine specifies that five channels can be plotted.

LIO$K_MODEM

This parameter enables a serial line device as a modem.

Supported Devices

Serial line

Parameter Values

A longword integer constant that enables or disables a serial line as a modem.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Disables a serial line as a modem
LIO$K_ON	Enables a serial line as a modem

[1]The default value.

Description

If a serial line device is not enabled as a modem (LIO$K_OFF), then a user program can manipulate the modem control signals. If a serial line device is enabled as a modem (LIO$K_ON), then VMS handles the modem control signals.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_MODEM, 1, LIO$K_OFF)
```

This routine sets up a serial line device so that a user program can manipulate the modem control signals.

LIO$K_MODEM_STATUS

This parameter sets and returns the status of a serial line device set up as a modem.

Supported Devices

Serial line

Parameter Values

A longword integer constant setting or returning modem line status.

The value can be one of the following:

Constant Value	Meaning
LIO$M_DTR[1]	Data terminal ready
LIO$M_RTS	Request to send
LIO$M_CD[2]	Carrier detect
LIO$M_CTS[2]	Clear to send
LIO$M_DSR[2]	Data set ready
LIO$M_RI[2]	Ring indicator

[1]The default value.

[2]An LIO$SHOW parameter value only.

Description

You can use this parameter to set up the data terminal ready (DTR) and request to send (RTS) signals.

You can also use this parameter with the LIO$SHOW routine to display the status of all the signals listed in the table in the **Parameter Values**.

Calling LIO$SHOW with LIO$K_MODEM_STATUS returns an array of six values, shown in the table below. If the signal is detected, the symbolic value is returned in the corresponding position in the array. If the signal is not detected, a 0 is placed in the array.

Array Position	Signal	Symbolic Value
1	DTR	LIO$M_DTR
2	RTS	LIO$M_RTS
3	CARRIER	LIO$M_CD
4	DSR	LIO$M_DSR
5	RING	LIO$M_RI
6	CTS	LIO$M_CTS

Restrictions

- You must enable the serial line as a modem line (using the LIO$K_MODEM parameter) before you set up the LIO$K_MODEM_STATUS parameter.
- Only the DTR and RTS signals can be set up by the user.

Example

```
status = LIO$SET_I (serial_id, LIO$K_MODEM_STATUS, 1, LIO$M_DTR)
```

This routine sets the status of a serial modem to data terminal ready (DTR).

LIO$K_MULTIPLE_X_AXES

This parameter specifies the x-axis format for the plotting window.

Supported Devices

Real-time plotting

Parameter Values

A longword integer constant.

The value can be one of the following:

Constant Value	Meaning
LIO$K_OFF	Specifies a single x,y axis for the entire window
LIO$K_ON[1]	Specifies an x,y axis for each channel

[1]The default value.

Description

Use this parameter to specify whether you want a single x-axis for a window or a separate x-axis for each channel.

If you specify a single x-axis for the entire window (LIO$K_OFF), all channels are moved simultaneously and then updated one at a time.

If you specify an x-axis for each channel (LIO$K_ON), one channel gets moved and updated, followed by the next channel, and so on. The x-axes are positioned one on top of the other. The y-axis is the same axis for each channel, but each channel has a separate y-axis plot line.

Restrictions

The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_MULTIPLE_X_AXES, 1, LIO$K_ON)
```

This routine specifies that each channel has a separate x,y axis.

LIO$K_N_AD_CHAN

This parameter returns the number of A/D channels currently in use.

Supported Devices

ADQ32
ADV11-D
AXV11-C A/D
IAV11-A[1]
IAV11-AA[1]
IAV11-C[1]
IAV11-CA[1]
Preston

Parameter Values

A longword integer array returning the number of A/D channels the device is using for input.

Description

See the **Parameter Values**.

Restrictions

This is an LIO$SHOW parameter only.

Example

```
status = LIO$SHOW (device_id, LIO$K_N_AD_CHAN, ad_array, length)
```

This routine call returns the number of A/D channels currently in use in the **ad_array** argument. The **length** argument returns the number of bytes in the **ad_array** argument.

[1] This device is available only in Europe.

LIO$K_N_BUFFS

This parameter specifies either of the following:

- Number of buffers to allocate for the memory queue device
- Number of channels in the data buffer for the real-time plotting device

Supported Devices

Memory queue
Real-time plotting

Parameter Values

A longword integer specifying either the number of buffers to allocate for the memory queue device or the number of channels in the data buffer for the real-time plotting device.

Description

When you set up the memory device with the value of the LIO$K_BUFF_SOURCE parameter as LIO$K_ARRAY or LIO$K_VIRTUAL_MEM, you must also set up the device using the LIO$K_N_BUFFS parameter. The LIO facility does not supply a default value if you omit this parameter.

If you set up the memory device with the value of the LIO$K_BUFF_SOURCE parameter as LIO$K_USER, then LIO ignores this parameter.

With the real-time plotting device, use this parameter to specify the number of A/D channels contained in each data buffer output to the real-time plotting device.

Restrictions

- For the memory queue device, you must specify the number of buffers to be allocated before the memory allocation occurs through the LIO$K_BUFF_SOURCE parameter.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Examples

1. `status = LIO$SET_I (memory_id, LIO$K_N_BUFF, 1, 5)`

   This routine specifies the allocation of five buffers for the memory queue device to use.

2. `status = LIO$SET_I (graphics_id, LIO$K_N_BUFF, 1, 2)`

   This routine specifies that there is data from two A/D channels in each buffer output to the real-time plotting device.

LIO$K_N_DA_CHAN

This parameter returns the number of D/A channels in use.

Supported Devices

AAV11-D
AXV11-C D/A

Parameter Values

A longword variable returning the number of D/A channels a device is currently using for output.

Description

See the **Parameter Values**.

Restrictions

This is an LIO$SHOW parameter only.

Example

```
status = LIO$SHOW (device_id, LIO$K_N_DA_CHAN, da_array, length)
```

This routine returns the number of D/A channels currently in use in the **da_array** argument. The **length** argument returns the number of bytes in the **da_array** argument.

LIO$K_NAME

This parameter sets the name of a file or a global section.

Supported Devices

Disk file
Memory queue

Parameter Values

A character string specifying a legal VMS file name or global section name, or a VMS logical name that translates to a legal VMS file name or global section name.

This value is passed by descriptor.

Description

Use this parameter to specify input and output file names and global section names.

When you set up disk file and memory queue devices, you must use the LIO$K_NAME parameter to supply a file name or global section name. The LIO facility does not supply a default name if you omit this parameter.

Restrictions

- For disk files, you must specify the file name before opening the file. The LIO facility does not supply a default file extension if you omit a file extension from the file name specification.
- For memory queue devices, if the global section already exists and if two memory queue devices are already mapped to it, then read-only access is available.

Examples

1. `status = LIO$SET_S (file_id, LIO$K_NAME, 'AD_FILE.DAT')`

   This routine specifies a disk file name as AD_FILE.DAT.

2. `status = LIO$SET_S (memory_id, LIO$K_NAME, 'XFER_DAT')`

   This routine specifies a global section name as XFER_DAT.

LIO$K_OPEN_FILE

This parameter opens input and output files.

Supported Devices

Disk file

Parameter Values

None.

Description

File space is allocated with best try at contiguous disk space.

Restrictions

- For all disk files, you must specify the direction of the file (input or output) through the LIO$K_DIRECTION parameter before you open the file.
- For all disk files, you must set the file name through the LIO$K_NAME parameter before you open the file.
- For output files, you must set the size through the LIO$K_FILE_SIZE parameter before you open the file.

Example

```
status = LIO$SET_I (device_id, LIO$K_OPEN_FILE, 0)
```

This routine opens a disk file device.

LIO$K_OUTPUT_PREFIX

This parameter establishes a prefix character string on the output side of a serial line.

Supported Devices

Serial line

Parameter Values

A character string specifying the string to be sent out the serial line before the buffer is output.

This value is passed by descriptor.

To disable a previously set-up output prefix, specify 0.

Description

The output prefix string set up by this parameter is output on the serial line before each output buffer.

Restrictions

None.

Examples

1. `status = LIO$SET_S (serial_id, LIO$K_OUTPUT_PREFIX, 'Transfer beginning')`

   This routine specifies an output buffer prefix. This prefix signals the start of an output buffer transfer.

2. `status = LIO$SET_S (serial_id, LIO$K_OUTPUT_PREFIX, 0)`

   This routine disables the output prefix.

LIO$K_OUTPUT_TERMINATOR

This parameter establishes a suffix character string on the output side of a serial line.

Supported Devices

Serial line

Parameter Values

A character string specifying the string to be sent out the serial line after each buffer has been output.

This value is passed by descriptor.

To disable a previously set-up output suffix, specify 0.

Description

This string is appended to each output buffer.

Restrictions

None.

Examples

1. `status = LIO$SET_S (serial_id, LIO$K_OUTPUT_TERMINATOR, 'Transfer ended')`

   This routine sets up a serial line device to output the phrase "Transfer ended" after each buffer has been output.

2. `status = LIO$SET_S (serial_id, LIO$K_OUTPUT_TERMINATOR, 0)`

   This routine disables the output suffix.

LIO$K_PAGE_ALIGN

This parameter page-aligns the first buffer allocated from virtual memory.

Supported Devices

Memory queue

Parameter Values

None.

Description

Page-aligning the buffer ensures that the first buffer begins on a page boundary. The address is a multiple of 512 bytes.

Restrictions

- The memory queue device must be attached to manage memory local to a process.
- You must set up the LIO$K_PAGE_ALIGN parameter before the memory allocation occurs.
- The memory allocation must be from virtual memory.

Example

```
status = LIO$SET_I (memory_id, LIO$K_PAGE_ALIGN, 0)
```

This routine page-aligns the buffers.

LIO$K_PAR_POLL

This parameter enables the controller-in-charge to parallel poll a maximum of eight instruments on the IEEE-488 bus and return a status bit from each device that is able to respond.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer returning the status byte read from the IEEE-488 bus. The interpretation of this information is up to the application program.

Description

A parallel poll is significantly faster than a serial poll because all devices able to respond do so simultaneously.

The usefulness of a parallel poll is limited by two factors:

- A single bit is supplied for each instrument
- Many devices are unable to respond to a parallel poll

See the reference descriptions of the LIO$K_PAR_POLL_CONFIG and the LIO$K_PAR_POLL_STATUS parameters for information about how to configure the devices to be polled and how to specify the bit mask of each device's status register.

Restrictions

- This is an LIO$SHOW parameter only.
- The device must be the controller-in-charge.
- The responding devices must be configured by the LIO$K_PAR_POLL_CONFIG parameter before they can respond to a parallel poll.

Example

```
status = LIO$SHOW (ieee_id, LIO$K_PAR_POLL, stat_array, length)
```

This routine parallel polls the devices on the IEEE-488 bus and returns the status bit from each device in the **stat_array** argument. The **length** argument returns the number of bytes in **stat_array**.

LIO$K_PAR_POLL_CONFIG

This parameter configures a maximum of eight IEEE-488 bus instruments to respond to a parallel poll by the controller-in-charge.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

Four values or arrays of values.

The first value is a longword integer specifying the number of IEEE-488 instruments to configure. This value can be 0 through 8, inclusive.

The second value is a longword integer array of up to eight values specifying the IEEE-488 bus addresses of the devices to configure.

The third value is a longword integer array of up to eight values specifying the number of the bit each device should use.

The fourth value is a longword integer array of up to eight values specifying the values to which each associated device should set its assigned status bit. Specify a 1 if the device should set its bit to indicate a true condition. Specify a 0 if the device should clear its bit to indicate a true condition.

Description

Before you set up this parameter, you must use the LIO$K_PAR_POLL parameter to enable IEEE-488 bus instruments to recognize and respond to a parallel poll configuration event from the controller-in-charge. Note that some IEEE-488 devices do not support parallel polling.

Restrictions

The device must be the controller-in-charge to issue a parallel poll configuration event.

Examples

1.
```
INTEGER number
number = 2
```

   This routine segment declares the variable **number** as an integer and assigns it a value of 2. This variable contains the number of instruments to configure.

2.
```
INTEGER*4 address(2)
address(1) = 3
address(2) = 4
```

   These lines declare **address** as an integer array of length two and assign bus addresses to each element in the array. The **address** array contains the IEEE-488 bus addresses of the instrument to configure.

3.
```
INTEGER*4 bits(2)
bits(1) = 6
bits(2) = 7
```

   These lines declare **bits** as an integer array of length two and assign individual status bits to each element of the array. The **bits** array contains the number of the bit each device should use.

4.
```
INTEGER*4 condition(2)
condition(1) = 1
condition(2) = 1
```

   These lines declare **condition** as an integer array of length two and assign 1 as the bit value on a true condition. The **condition** array specifies the value each device sets its associated **bits** to on a true condition.

5.
```
status = LIO$SET_I (ieee_id, LIO$K_PAR_POLL_CONFIG, 4, number,
1                   address, bits, condition)
```

This routine configures two devices, at IEEE-488 bus addresses 3 and 4, respectively, to set bits 6 and 7, respectively, to a value of 1 on a true condition.

LIO$K_PAR_POLL_STATUS

This parameter sets up the parallel poll status register of an IX device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer between 1 and 255 specifying the bit mask of the status to return to the controller-in-charge when the device is parallel polled. Only bits 0 through 7 are used, and only one bit is set.

Which bit is set is determined by the LIO$K_PAR_POLL_CONFIG parameter when the current controller-in-charge issues a parallel poll configuration event. See the reference description of the LIO$K_PAR_POLL_CONFIG parameter for information about specifying the status bit.

Description

This parameter sets up parallel poll status register of an IX device. This value is read by the current controller-in-charge when the device is parallel polled.

The current controller-in-charge, using the LIO$K_PAR_POLL_CONFIG parameter, signals an IX device which bit in the status register it is to use to return status information to the controller-in-charge. The LIO$K_PAR_POLL_STATUS parameter sets up the bit mask for this status bit.

Restrictions

An IEEE-488 device can be parallel polled only when it is not the controller-in-charge.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_PAR_POLL_STATUS, 1, 255)
```

This routine sets up bit 7 (bit mask 255) as the status bit of an IEEE-488 device. On a TRUE condition, the device sets bit 7 to 1. On a FALSE condition (error), the device clears bit 7.

Note that a user's program must determine when a device condition changes from TRUE to FALSE, or from FALSE to TRUE. When a user's program detects a change, it must then call LIO$K_PAR_POLL_STATUS to change the device's parallel poll status byte.

LIO$K_PARITY

This parameter enables or disables the DRB32 to accept parity from an external device. This parameter also establishes the type of parity checking used by serial line devices.

Supported Devices

DRB32
Serial line

Parameter Values

For the **DRB32**, a longword integer enabling or disabling whether the DRB32 interprets the parity bit of an external device. Specifying a nonzero value causes the DRB32 to interpret the parity bit of an external device. Specifying zero effectively disables parity checking.

For **serial line** devices, a longword integer constant specifying the type of parity checking for the serial line.

The value can be one of the following:

Constant Value	Meaning
LIO$K_EVEN	The number of ''1'' bits in a character sum to an even number.
LIO$K_ODD	The number of ''1'' bits in a character sum to an odd number.
LIO$K_NONE[1]	Disables parity checking.

[1]The default value.

Description

See the **Parameter Values**.

Restrictions

You cannot use this parameter to set the parity for LAT devices. To change parity for LAT devices, set the port's permanent characteristics.

Examples

1. `status = LIO$SET_I (drb_id, LIO$K_PARITY, 1, 1)`

   This routine enables the DRB32 to interpret the parity bit of an external device.

2. `status = LIO$SET_I (serial_id, LIO$K_PARITY, 1, LIO$K_NONE)`

   This routine disables checking parity for a serial line device.

LIO$K_PASS_CTRL

This parameter signals the current controller-in-charge to pass control to another device specified by its IEEE-488 bus address.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer specifying the IEEE-488 bus address of the device to which to pass control.

Description

This parameter passes control to the specified device. Once control is passed, this device acts as an instrument (not a controller) until control is passed back to this device.

Restrictions

- The device passing control must be the controller-in-charge.
- If the device receiving control is an IEQ11, an IEZ11, or an IOtech Micro488A device, then it must be attached as a controller and must be set up with receive control event (LIO$K_REC_CTRL) recognition enabled through the LIO$K_EVENT_ENA parameter.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_PASS_CTRL, 1, 3)
```

This routine passes control to the device at IEEE-488 bus address 3.

LIO$K_PCR

This parameter specifies the number of steps in the Programmable Clock Register (PCR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer specifying the number of steps in the PCR in multiples of 100 nanoseconds.

The value can be between 25 and 4095, inclusive.

Description

The contents of the PCR are received in a 12-bit counter that is clocked by a quartz generator of 10 MHz.

The counter and the clock are only enabled when the conversion starts and if the current Control Word Mode is not mode 3. The clock is also available as an output signal when enabled by the LIO$K_ED_ECE parameter.

Restrictions

None.

[1] This device is available only in Europe.

Example

```
status = LIO$SET_I (axf_id, LIO$K_PCR, 1, steps)
```

This routine writes the number of steps, specified as a multiple of 100 nanosecond steps, to the PCR.

LIO$K_PLOT_SIZE

This parameter specifies the size of the plotting window.

Supported Devices

Real-time plotting

Parameter Values

Two single-precision, floating-point real values specifying the size of the plotting window in centimeters.

The first value specifies the vertical size of the window. The default value is 10.0 centimeters.

The second value specifies the horizontal size of the window. The default value is 20.0 centimeters.

Description

See the **Parameter Values**.

Restrictions

- The values are limited to the size of the display screen.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_R (device_id, LIO$K_PLOT_SIZE, 2, 15.0, 30.0)
```

This routine specifies the size of the plotting window as 15.0 centimeters vertically and 30.0 centimeters horizontally.

LIO$K_PLOT_TYPE

This parameter specifies the style of plotting used by the real-time plotting device.

Supported Devices

Real-time plotting

Parameter Values

A longword integer constant specifying the plotting style.

The value can be one of the following:

Constant Value	Style
LIO$K_SCOPE	Oscilloscope-type output of data
LIO$K_STRIPCHART[1]	Scrolling output of data

[1]The default value.

Description

This parameter allows you to specify either scrolling or oscilloscope-type output of data. The oscilloscope-type output is similar to the output of the LGP$PLOTC routine in LGP, which updates a plot dynamically by erasing old data and displaying new data.

Restrictions

The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_PLOT_TYPE, 1, LIO$K_SCOPE)
```

This routine specifies that the output will be oscilloscope-type output.

LIO$K_PO_CHAN

This parameter specifies the channel or channels to be plotted on the display screen using the real-time plotting device.

Supported Devices

Real-time plotting

Parameter Values

One or more longword integer values specifying the channel or channels in the data buffer to be plotted.

The default value is channel 0.

Description

You can specify up to the number of channels specified by LIO$K_MAX_CHANNELS (a maximum of eight channels) to be plotted in any order. The channels can repeat.

Restrictions

- You can specify a maximum of eight channels.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_PO_CHAN, 2, 4, 2)
```

This routine specifies that data from channels 4 and 2 be plotted.

LIO$K_POLARITY

This parameter determines whether a negative-going or positive-going edge on the input bits causes an AST routine to execute or an event flag to be set.

This parameter also sets the polarity of the DRV11-J handshake signals.

Supported Devices

DRV11-J
IDV11-A[1]

Parameter Values

For the **DRV11-J**, a longword integer constant specifying whether the bits of port A call event AST routines or set event flags on a negative-going or positive-going edge.

The default value for the DRV11-J is a negative-going edge.

For the **IDV11-A**, a longword integer constant specifying whether the device calls an event AST routine on a negative-going or positive-going edge received on the interrupt line (line 15).

There is no default value for the IDV11-A. (Both negative-going and positive-going edge are disabled unless you use this parameter.)

The value can be one of the following:

Constant Value	Function
LIO$K_NEGATIVE	Bits call AST routines, set event flags, or handshake on a negative-going edge.
LIO$K_POSITIVE	Bits call AST routines, set event flags, or handshake on a positive-going edge.

[1] This device is available only in Europe.

Description

For the **DRV11-J**, the value of this parameter affects all the bits of port A. If the constant value is LIO$K_POSITIVE, all the bits in port A wait for a positive-going edge.

For example, when the voltage on the pin assigned to a particular bit goes from low (0 volts) to high (5 volts), this represents a positive-going edge. If an AST routine is assigned to the bit, that AST routine is called. If an event flag is assigned to the bit, that event flag is set.

For the **IDV11-A**, the value of this parameter affects input line 15 (counting from 0). If an AST routine is assigned, then it is called on the respective edge event. Note that line 15 can be set for both positive-going and negative-going edge.

Restrictions

- The DRV11-J must be attached to use QIOs.
- If the DRV11-J hardware is jumpered to use a two-wire handshake, and if handshaking is software-enabled by the LIO$K_HANDSHAKE parameter, changing the value of this parameter changes the polarity of the handshake.
- If the bits of port A are to call AST routines, you must set up an AST routine with the LIO$K_EVENT_AST parameter.
- If the bits of port A are to set event flags, you must specify event flags with the LIO$K_EVENT_EF parameter.

Example

```
status = LIO$SET_I (device_id, LIO$K_POLARITY, 1, LIO$K_POSITIVE)
```

This routine sets up the bits to respond to a positive-going edge.

LIO$K_POSITION

This parameter specifies the position of the real-time plotting window on the display screen.

Supported Devices

Real-time plotting

Parameter Values

Two single-precision, floating-point real values specifying the position of the real-time plotting window on the display screen in centimeters.

The first value specifies the x-axis offset from the lower lefthand corner of the display screen. The default value is 0.0 centimeters.

The second value specifies the y-axis offset from the lower lefthand corner of the display screen. The default value is 0.0 centimeters.

Description

Adjusting the x-axis offset moves the plotting window horizontally on the display screen.

Adjusting the y-axis offset moves the plotting window vertically on the display screen.

If the default values are used, the plotting window is displayed at the lower lefthand corner of the display screen.

Restrictions

- The offset values are limited to the size of the display screen.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_R (graphics_id, LIO$K_POSITION, 2, 1.0, 1.0)
```

This routine positions the real-time plotting window 1.0 centimeter, both horizontally and vertically, from the lower lefthand corner of the display screen.

LIO$K_PROTOCOL

This parameter enables or disables the use of a user-defined protocol for serial line data transfers.

Supported Devices

Serial line

Parameter Values

A longword integer constant enabling or disabling the user protocol feature.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF[1]	Disables user-defined protocol
LIO$K_ON	Enables user-defined protocol

[1]The default value.

Description

You define a protocol to interface to laboratory equipment that requires some sort of software handshaking to occur over the serial line. Specify the protocol to conform to the specific requirements of certain laboratory instruments.

Restrictions

None.

Example

```
param_code  = LIO$K_PROTOCOL;
param_value = LIO$K_ON;
param_n_val = 1;
sys_stat = LIO$SET_I(&dev_id,&param_code,&param_n_val,&param_value);
if (!(sys_stat & STS$M_SUCCESS)) lib$signal (sys_stat);

param_code  = LIO$K_USER_READ_PROTOCOL_AST;
param_value = receive_buff;
param_n_val = 1;
sys_stat    = LIO$SET_I(&dev_id,&param_code,&param_n_val,param_value);
if (!(sys_stat & STS$M_SUCCESS)) lib$signal (sys_stat);

param_code  = LIO$K_DEVICE_ACK_NAK_BUFF;
param_value = sizeof(ack_buff);
param_n_val = 3;
sys_stat    = LIO$SET_I(&dev_id,&param_code,&param_n_val,ack_buff,
                        &param_value,&timeout);
if (!(sys_stat & STS$M_SUCCESS)) lib$signal (sys_stat);

param_code  = LIO$K_ACK_NAK_TERMINATOR;
sys_stat    = LIO$SET_S(&dev_id,&param_code,&term_char);
if (!(sys_stat & STS$M_SUCCESS)) lib$signal (sys_stat);

param_code  = LIO$K_USER_WRITE_NAK_HANDLING;
param_value = LIO$K_RESEND_LAST;
param_n_val = 1;
sys_stat    = LIO$SET_I(&dev_id,&param_code,&param_n_val,&param_value);
if (!(sys_stat & STS$M_SUCCESS)) lib$signal (sys_stat);
```

This VAX C program section is an example of using a user-defined protocol to output to a serial device.

This program does the following:

- Enables a user-defined protocol
- Defines an AST routine to handle ACK/NAK reception
- Sets a buffer to receive the ACK/NAK string
- Defines a terminator to the ACK/NAK string
- Sets up the action to be taken on receipt of a NAK string

The ACK string sent by an external device on successful receipt of a buffer must be known.

The AST routine, called receive_buff, is shown in the program segment below taken from the example program LIO_SERIAL.C.

```
/*-------------------------------------------------------------------------*/


int receive_buff(status_ptr,device_id_ptr,buffer,buffer_length_ptr,
            data_length_ptr,buffer_index_ptr,device_specific_ptr)

int    *status_ptr;             /* returns the status of the i/o operation */
int    *device_id_ptr;          /* specifies the LIO assigned device id    */
char   buffer[];                /* the buffer of received data             */
int    *buffer_length_ptr;      /* the length of the buffer                */
int    *data_length_ptr;        /* the length of the data in the buffer    */
int    *buffer_index_ptr;       /* buffer index, NOT USED                  */
int    *device_specific_ptr;    /* device specific, NOT USED               */

{

if (buffer[0] == 'A')       /* The device ACKED, return DEVICE ACKED to LIO */
                {
                 return(LIO$K_DEVICE_ACKED);
                }

       else                 /* The device NAKED; return DEVICE NAKED to LIO  */
                            /* Packet will be retransmitted to serial device */

                 return(LIO$K_DEVICE_NAKED);

}
```

The AST routine compares the string actually received and placed in the buffer (called buffer in this example) against the known ACK string (the character A in this example).

If the received string matches the ACK string, the AST routine returns the constant LIO$K_DEVICE_ACKED. If it does not match, the AST routine returns the constant LIO$K_DEVICE_NAKED. It is evoked when the external device sends a string in response to your write operation.

LIO$K_PURGE

This parameter purges the contents of the type-ahead buffer.

Supported Devices

Serial line

Parameter Values

None.

Description

Use this parameter to ensure that the type-ahead buffer is empty for starting a data transfer.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_PURGE, 0)
```

This routine purges the contents of the type-ahead buffer.

LIO$K_READ_ONLY

This parameter sets a memory queue device or global section to be read-only.

Supported Devices

Memory queue

Parameter Values

None.

Description

This parameter sets the memory queue device to copy data from the global section into the buffer supplied in the LIO$READ routine call. See Section 2.7.2.4, Setting Up a Memory Queue Device for Interprocess Communications, for information about interprocess memory I/O.

Restrictions

- The memory queue must be attached for interprocess I/O.
- The memory queue must be set to use the synchronous I/O interface.
- The memory queue device running in the other process must be set up to display data buffers.

Example

```
status = LIO$SET_I (memory_id, LIO$K_READ_ONLY, 0)
```

This routine sets the memory queue device to read data buffers as they are passed by the window by the display-only memory queue.

LIO$K_READ_PROMPT

This parameter establishes a read prompt to prefix each input data buffer.

Supported Devices

Serial line

Parameter Values

A character string specifying the string to be sent out the serial line before a read request.

This value is passed by descriptor.

To disable a previously set-up read prompt, specify 0.

Description

You can use the read prompt string to prompt a device for data.

Restrictions

None.

Examples

1. `status = LIO$SET_S (serial_id, LIO$K_READ_PROMPT, 'Enter string: ')`

   This routine sets up "Enter string:" as the read prompt.

2. `status = LIO$SET_S (serial_id, LIO$K_READ_PROMPT, 0)`

   This routine disables a read prompt that was previously set up.

LIO$K_READ_STAT

This parameter returns the status of the read-only bits in the Command and Status Register (CSR).

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

A longword integer array of length five returning the status of the read-only bits in the CSR. The following list explains the return order of each bit within the array and what the contents of each bit indicates.

- The first array index returns the contents of the Unipolar Operation bit. A zero indicates a bipolar operation (0V to +10V). A one indicates a unipolar operation (-10V to +10V).
- The second array index returns:
  - For the AAF01, the contents of the Comparator Input bit. A one indicates that the COMP IN line is more negative than the output of channel 0.
  - For the ADF01, the contents of the Single-Ended Input bit. A zero indicates differential input. A one indicates single-ended input.
- The third array index returns:
  - For the AAF01, the contents of the End of Conversion Sequence bit. A zero indicates a normal end of conversion. A one indicates an abnormal end of conversion sequence.
  - For the ADF01, the contents of the Attention bit . A zero indicates that the bit is not set. A one indicates that the bit is set.

[1] This device is available only in Europe.

- The fourth array index returns:
  - For the AAF01, the contents of the Data Buffer Empty bit. A zero indicates that the data buffer was not empty during the conversion sequence. A one indicates that the data buffer was empty during the conversion sequence.
  - For the ADF01, the contents of the Data Buffer Full bit. A zero indicates that the data buffer is not full. A one indicates that the data buffer is full.
- The fifth array index returns the Sequence Break bit. A zero means that the sequence break is not set. A one means that the sequence break is set.

Description

See the **Parameter Values**.

Restrictions

This is an LIO$SHOW parameter only.

Example

```
INTEGER*4 axf_value_list(5)
status = LIO$SET_I (axf_id, LIO$K_READ_STAT, axf_value_list, list_length)
         IF (.NOT. status)  CALL LIB$SIGNAL (%VAL(status))

UNI  = axf_value_list(1)    !Unipolar operation bit (both)
SING = axf_value_list(2)    !Single-ended input bit (ADF01)
CMP  = axf_value_list(2)    !Comparator bit (AAF01)
ATT  = axf_value_list(3)    !Attention bit (ADF01)
EOC  = axf_value_list(3)    !End of conversion sequence bit (AAF01)
DBF  = axf_value_list(4)    !Data buffer full bit (ADF01)
DBE  = axf_value_list(4)    !Data buffer empty bit (AAF01)
SBE  = axf_value_list(5)    !Sequence break enable bit (both)
```

This routine returns the read-only bits in the Command and Status Register of the AAF01 and ADF01 devies.

LIO$K_RESET_AXF

This parameter resets a device.

Supported Devices

AAF01[1]
ADF01[1]

Parameter Values

None.

Description

Use this parameter to reset the AAF01 and ADF01 devices to their default operating characteristics.

For the AAF01, this parameter resets the ACS (AAF01 Command and Status Register), PCR (Programmable Clock Register), DBF (Data Buffer Register), and all the analog output channels.

For the ADF01, this parameter resets the ACS (ADF01 Command and Status Register), PCR (Programmable Clock Register), DBS (Data Buffer Silo), and DDC (DAC Data Register).

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_RESET_AXF, 0)
```

This routine resets the AAF01 or ADF01 subsystem.

[1] This device is available only in Europe.

LIO$K_RESET_DRX

This parameter resets the direct memory access (DMA) interface.

Supported Devices

DRQ11-C[1]

Parameter Values

Two longword integer values.

The first value specifies whether or not to set the FNCT0 bit.

This value can be one of the following:

Constant Value	Function
LIO$K_FNCT0	Sets the FNCT0 bit
LIO$K_NO_FNCT0	Clears the FNCT0 bit

The second value is a mask to be placed into the Data Buffer Register (DBR). A mask value of 0 does not affect the DBR.

Description

This parameter resets the DMA interface. The LIO$K_FNCT0 value resets the interface with the FNCT0 bit. The LIO$K_NO_FNCT0 value resets the interface without the FNCT0 bit. The specified mask value is written to the Data Buffer Register (DBR). A mask value of zero does not affect the DBR.

[1] This device is available only in Europe.

LIO$K_RESET_DRX

Restrictions

None.

Example

```
status = LIO$SET_I (drq_id, LIO$K_RESET_DRX, 2, LIO$K_NO_FNCTO, 0)
```

This routine resets the DMA interface without the FNCT0 bit. The 0 mask value does not affect the DBR register.

LIO$K_SCHMITT_TRIGGER

This parameter sets the mode of operation for the two Schmitt triggers on the Simpact RTC01 clock device.

Supported Devices

Simpact RTC01

Parameter Values

Two longword integer constants.

The first value specifies the Schmitt trigger number.

Constant Value	Meaning
1	Schmitt trigger 1
2	Schmitt trigger 2

The second value specifies the Schmitt trigger mode.

Constant Value	Meaning
LIO$K_ANALOG	Sets the trigger mode to analog
LIO$K_TTL	Sets the trigger mode to TTL

Description

The Schmitt trigger circuitry for the Simpact RTC01 accepts either TTL-compatible input or analog input. This parameter lets you set the Schmitt triggers to receive either kind of input.

For both TTL and analog mode, the polarity of the trigger is set with switches on the UDIP. If you do not have a UDIP, set pins 5 and 6 on the D25 connector.

When set for analog mode, the threshold level for triggering is set using the potentiometers on the UDIP. If you do not have a UDIP, set pins 3 and 4 on the D25 connector.

Restrictions

None.

Examples

1. `status = LIO$SET_I (device_id, LIO$K_SCHMITT_TRIGGER, 2, 1, LIO$K_TTL)`

   This routine sets the Schmitt trigger 1 to TTL mode.

2. `status = LIO$SET_I (device_id, LIO$K_SCHMITT_TRIGGER, 2, 2, LIO$K_ANALOG)`

   This routine sets the Schmitt trigger 2 to analog mode.

LIO$K_SER_POLL

This parameter performs a serial poll of the instruments on the IEEE-488 bus and returns the status from each device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A byte array and a longword integer value.

The array returns the status values from each instrument on the IEEE-488 bus configured for serial polling.

The integer value represents the length of the array, which depends on the number of primary addresses set up by the LIO$K_SER_POLL_CONFIG parameter.

Description

This parameter enables the controller-in-charge to obtain status information from each device on the IEEE-488 bus configured for serial polling.

If the device is requesting service, then the SRQ bit (bit 6) is set in the status byte of the device. Polling a device clears its SRQ bit and stops it from requesting service.

You can perform a serial poll only after a service request event has occurred.

See the description of the LIO$K_SER_POLL_CONFIG parameter for information about how to configure the devices to be polled.

Restrictions

- This is an LIO$SHOW parameter only.
- The IEEE-488 device must be the controller-in-charge.
- The responding devices must be configured for serial polling by the LIO$K_SER_POLL_CONFIG parameter.

Example

```
status = LIO$SHOW (ieee_id, LIO$K_SER_POLL, stat_array, length)
```

This routine serially polls the devices on the IEEE-488 bus. The array **stat_array** returns the status of the devices. The **length** argument returns the number of bytes (addresses) in **stat_array**.

LIO$K_SER_POLL_CONFIG

This parameter configures an IEEE-488 device that is controller-in-charge to perform a serial poll.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

A longword integer value and a longword integer array.

The integer value specifies the length of the array.

The array specifies the addresses of the instruments to be serially polled.

Description

Use this parameter to signal IEEE-488 devices to configure themselves for serial polling by the controller-in-charge.

You can mix primary and secondary addresses in the list of addresses.

If you want to specify the secondary address of a device, you must enable recognition of secondary addressing when you assign the primary address of the device using the LIO$K_IEEE_ADDR parameter.

Restrictions

- This is an LIO$SET_I parameter only.
- The device must be the controller-in-charge to issue a serial poll configuration event.
- You must enable IEEE-488 bus instruments to recognize and respond to a serial poll configuration event from the controller-in-charge before you set up this parameter.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_SER_POLL_CONFIG, 2, list_size,
1                            serial_list)
```

This routine sets up the list of instruments to be polled.

LIO$K_SGL_BUF

This parameter sets the device for single-buffer DMA. This means that the device stops between DMA buffers so the input or output is not continuous.

Supported Devices

AAV11-D
ADQ32
ADV11-D
Preston (DRQ3B interface only)

Parameter Values

None.

Description

Use single-buffer DMA when you want to examine the data before continuing data acquisition.

Single-buffer DMA is less restrictive on buffer sizes and alignments than continuous DMA.

Use this parameter to stop continuous DMA for the AAV11-D, the ADV11-D, and the Preston (DRQ3B interface).

Since single buffering is the default for the ADQ32, you need to use this parameter for the ADQ32 only if you want to stop double buffering.

Restrictions

The following restrictions apply to both the AAV11-D and the ADV11-D:

- The buffers must be word-aligned.
- There must be 512 bytes remaining at the end of the buffer to serve as an overrun area.
- See Section 1.6.3.1, Single-Buffer DMA, for more information.

Example

```
status = LIO$SET_I (device_id, LIO$K_SGL_BUF, 0)
```

This routine enables single-buffer DMA data transfers for the device.

LIO$K_SKIP_COUNT

This parameter specifies how many points are to be skipped and not plotted.

Supported Devices

Real-time plotting

Parameter Values

A longword integer specifying the number of points to skip.

Description

This parameter allows every $n+1$th point to be plotted, resulting in faster plotting speeds.

The first point is plotted, and then n points are skipped.

Restrictions

- The skip rate must be greater than or equal to 0.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_SKIP_COUNT, 1, 4)
```

This routine specifies that four points are to be skipped, which means that every fifth point will be plotted.

LIO$K_SRQ

This parameter defines the serial poll status byte of an IEEE-488 device and, optionally, sends a service request to the controller-in-charge.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

Two longword integer values.

The first value specifies the serial poll status byte.

The second value specifies the secondary address of the device. If you specify the secondary address of the device, you must enable the recognition of secondary addressing when you assign the primary address of the device using the LIO$K_IEEE_ADDR parameter. Specifying the secondary address is optional.

Description

Use this parameter to specify the serial poll status byte of an IEQ11, an IEZ11, or an IOtech Micro488A device.

If bit 6 in the status byte of the device is set (requesting service), then the SRQ is sent to the controller-in-charge. The LIO$SET_I call hangs until the device is serially polled.

If bit 6 in the status byte of the device is cleared (not requesting service), then the status is saved in the device and is read by the controller-in-charge if it serially polls the device.

The meaning of the other bits in the status byte of the device is defined by the user program (instrument).

For example, a user program can be written so that the SRQ status byte also displays the current state of the instrument. The current state of an instrument might be idle (status byte is 0), running (status byte is 1), or data available (status byte is 2) and bit 6 set to signal an SRQ.

Restrictions

- The device responding to the serial poll event cannot be the controller-in-charge.
- If the secondary address of the device is specified, then the device only responds to a serial poll of that secondary address.

Example

```
status = LIO$SET_I (ieee_id, LIO$K_SRQ, 2, status_byte, 140)
```

This routine sets up **status_byte** as the status byte for the IEEE-488 device with secondary address 140.

LIO$K_ST0_1

This parameter loads the specified number of steps into the 23-bit counter contained in the Sequence Timer Registers ST0 and ST1.

Supported Devices

ADF01/AMF01[1]

Parameter Values

Two longword integer values.

The first value specifies the number of steps in multiples of 1 microsecond. This value can be between 2 and 8,388,607, inclusive.

The second value enables or disables the sequence timer.

This value can be one of the following:

Constant Value	Function
LIO$K_ENABLE	Enables sequence timer
LIO$K_DISABLE	Disables sequence timer

[1] This device is available only in Europe.

Description

The counter contained in the Sequence Timer Registers ST0 and ST1 is clocked by a quartz generator of 1 MHz.

You can enable the sequence timer in two ways:

- By using the Sequence Timer Enable (STE) bit
- By an external sequence timer enable input signal connected to the (ST EN EXT) pin of any one of the three AMF01 user connectors J3, J4, or J5

See the *AMF01 48 channel Analogue Input Multiplexer for the ADF01 Data Acquisition Subsystem* for more information.

Restrictions

None.

Example

```
status = LIO$SET_I (adf_id, LIO$K_ST0_1, 2, steps, LIO$K_ENABLE)
```

This routine loads the number of steps specified as a multiple of 1 microsecond into ST0 and ST1, and enables the Sequence Timer Enable bit.

LIO$K_START

This parameter starts the following:

- Continuous DMA I/O
- IDV11-D counter channels
- KWV11-C or the Simpact RTC01 clock
- Continuous real-time plotting

Supported Devices

AAV11-D
ADV11-D
IDV11-D[1]
KWV11-C
Preston (DRQ3B interface only)
Real-time plotting
Simpact RTC01

Parameter Values

For the **IDV11-D**, a longword integer array of length five specifying which of the five IDV11-D channels to start. For each array index containing a one, the respective counter is started.

For the other devices, there is no parameter value.

Description

See the **Restrictions**.

[1] This device is available only in Europe.

Restrictions

The following restrictions apply to both the AAV11-D and the ADV11-D:

- This parameter is required to start continuous DMA I/O for the AAV11-D when it is set for continuous DMA output, and for the ADV11-D when it is set for continuous DMA input.
- The devices must already be set for continuous DMA I/O through the LIO$K_CONT parameter.
- 64K bytes of buffers must be enqueued to the device.

The following restriction applies to the KWV11-C and the Simpact RTC01:

- The clock must not already be running.
- LIO$K_START is necessary only when the clocks are set for asynchronous transfer and the trigger mode is LIO$K_IMMEDIATE.

The following restriction applies to the real-time plotting device:

- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Examples

1.
```
status = LIO$SET_I (device_id, LIO$K_START, 0)
```

   This routine does one of the following:

   - Starts continuous DMA I/O on the AAV11-D, ADV11-D, or Preston devices
   - Starts the KWV11-C or the Simpact RTC01 clock
   - Starts continuous real-time plotting

   Which function the parameter performs depends on the device for which you are using it.

2.
```
INTEGER*4 counter_start(5)
counter_start(1) = 0    !Channel 0 not used
counter_start(2) = 1    !Start channel 1
counter_start(3) = 1    !Start channel 2
counter_start(4) = 0    !Channel 3 not used
counter_start(5) = 0    !Channel 4 not used

status = LIO$SET_I (idvd_id, LIO$K_START, 1, %LOC(counter_start))
```

   This routine starts counter channels 1 and 2 of the IDV11-D device.

LIO$K_STAT_BITS

This parameter returns status information about the device.

Supported Devices

DRQ11-C[1]

Parameter Values

A longword integer returning the second word of the I/O Status Block (IOSB), which contains the status bits STAT0 through STAT3.

Description

See the **Parameter Values**.

Restrictions

This is an LIO$SHOW parameter only.

Example

```
status = LIO$SHOW (drc_id, LIO$K_STAT_BITS, status_bits, length)
```

This routine returns the contents of STAT0 through STAT3 in bits 0 through 3 of the **status_bits** argument.

[1] This device is available only in Europe.

LIO$K_STE

This parameter clears the Sequence Timer Enable (STE) bit in the Sequence Timer Register (ST1).

Supported Devices

ADF01/AMF01[1]

Parameter Values

None.

Description

You can use this parameter with an ADF01 Data Acquisition Subsystem that has the AMF01 multiplexer expansion option.

Restrictions

None.

Example

```
status = LIO$SET_I (adf_id, LIO$K_STE, 0)
```

This routine clears the Sequence Timer Enable (STE) bit in the AMF01 Sequential Timer Register (ST1).

[1] This device is available only in Europe.

LIO$K_STOP

This parameter stops continuous DMA I/O, stops the IDV11-D[1] counter, and stops the KWV11-C and the Simpact RTC01 clock.

Supported Devices

AAV11-D
ADV11-D
DRQ3B
IDV11-D[1]
KWV11-C
Preston (DRQ3B interface only)
Simpact RTC01
Serial line

Parameter Values

For the **IDV11-D**, a longword integer array of length five specifying which of the five IDV11-D channels to stop. For each array index containing a one, the respective counter is stopped.

For the other devices, there is no parameter value.

Description

Although the preferred way to stop continuous DMA I/O is not to enqueue any new buffers, you can also stop continuous DMA by using this parameter.

When used with the DRQ3B, this parameter cancels any outstanding DMA I/O requests.

If you stop continuous DMA in midstream, the filled buffers continue to dequeue. The data length that LIO$DEQUEUE returns on the buffer that was being filled when continuous DMA was stopped is not valid.

[1] This device is available only in Europe.

Restrictions

- For the AAV11-D and ADV11-D, continuous DMA I/O must already have been started using the LIO$K_START parameter before you stop it.
- For the IDV11-D, the counter channels must already have been started using the LIO$K_START parameter before you stop them.

Examples

1.
```
status = LIO$SET_I (device_id, LIO$K_STOP, 0)
```

   This routine stops continuous DMA I/O, or stops the KWV11-C or the Simpact RTC01 clock. Which function the parameter performs depends on the device for which you are using it.

2.
```
INTEGER*4 counter_stop(5)
counter_stop(1) = 0     !Channel 0 not used
counter_stop(2) = 1     !Stop channel 1
counter_stop(3) = 1     !Stop channel 2
counter_stop(4) = 0     !Channel 3 not used
counter_stop(5) = 0     !Channel 4 not used

status = LIO$SET_I (idvd_id, LIO$K_STOP, 1, %LOC(counter_stop))
```

   This routine segment stops counter channels 1 and 2 of the IDV11-D device.

LIO$K_SWEEP_RATE

This parameter sets the rate for the sweep clock of the ADQ32 device. The sweep clock controls sweep and, for delayed edge gate modes, specifies the delay time.

Supported Devices

ADQ32

Parameter Values

A single-precision, floating-point real value specifying the desired rate. The maximum value is 200 kHz.

The default value is 1 kHz.

Description

LIO supplies the closest approximation of the specified rate it can obtain with the sweep clock. A sweep of the channels specified in LIO$K_AD_CHAN is then made at every tick of the sweep clock.

When used in a sweep mode with an external gate signal, the gate controls the sweep clock rate. When the external gate closes, the sweep clock is turned off. If the gate closes in the middle of a sweep, all of the remaining channels in the sweep are sampled, and then the primary clock is turned off also.

For delayed edge gate modes, this parameter specifies the delay time. The delay is one tick of the clock rate you specify.

This parameter returns no warning if the clock is not set to the exact rate you request.

Using this parameter with LIO$SHOW returns the actual rate.

See Appendix A for more information.

Restrictions

The sweep rate must be fast enough to trigger all the channels (selected using LIO$K_AD_CHAN) before the next sweep starts. If sweeps are started by the A/D clock (LIO$K_AD_CLOCK), then the rate set with LIO$K_SWEEP_RATE must be at least the number of channels times the rate set using the LIO$K_CLK_RATE parameter.

Example

```
status = LIO$SET_R (device_id, LIO$K_SWEEP_RATE, 1, 1000.0)
```

This routine sets the sweep clock rate at 1 kHz.

LIO$K_SYNCH

This parameter sets up a device to use the synchronous I/O interface.

Supported Devices

AAF01[1]
AAV11-D
ADF01[1]
ADQ32
ADV11-D
AXV11-C
DRB32
DRB32W
DRQ11-C[1]
DRQ3B
DRV11-J
DRV11-WA
IAV11[2] devices
IDV11-A[1]
IEQ11
IEZ11
KWV11-C
Preston
Simpact RTC01
Disk file
Memory queue
Serial line

Parameter Values

None.

[1] This device is available only in Europe.

[2] These devices are available only in Europe.

LIO$K_SYNCH

Description

See the **Restrictions**.

Restrictions

No buffers can currently be on the user queue or device queue of the device.

Example

```
status = LIO$SET_I (device_id, LIO$K_SYNCH, 0)
```

This routine sets up the device to use the synchronous I/O interface.

LIO$K_TERM_CHAR

This parameter defines a termination character to signal the end of an input data transfer for an IEEE-488 device.

Supported Devices

IEQ11
IEZ11
IOtech Micro488A

Parameter Values

Two longword integer values.

The first value specifies the ASCII value of the termination character. Specifying -1 disables the termination character.

The second value, which can be used with the IEQ11 only, specifies the number of times the character must be repeated to be treated as a terminator. The default value is 1. Specifying this value is optional.

Description

This parameter defines a character that signals the termination of input data.

The termination character can be repeated any number of times for an IEQ11 device. The IEZ11 and the IOtech Micro488A, however, do not support a repeat count, and will terminate input on receiving the termination character once.

For the IOtech Micro488A, you can use LIO$K_EOI in place of the termination character to enable termination on assertion of the EOI line.

LIO$K_TERM_CHAR

Restrictions

None.

Examples

1. `status = LIO$SET_I (ieee_id, LIO$K_TERM_CHAR, 2, 10, 2)`

   This routine specifies that two line-feed (ASCII decimal 10) characters in succession signal the end of an input buffer for the IEQ11 device.

2. `status = LIO$SET_I (ieee_id, LIO$K_TERM_CHAR, 1, 10)`

   This routine specifies that one line-feed (ASCII decimal 10) character signals the end of an input buffer for the IEZ11 or the IOtech Micro488A device.

3. `status = LIO$SET_I (ieee_id, LIO$K_TERM_CHAR, 1, LIO$K_EOI)`

   This routine specifies that input data transfer will terminate on assertion of the EOI line for the IOtech Micro488A device.

LIO$K_TERM_SRQ

This parameter enables or disables terminations of I/O transfers by a service request.

Supported Devices

IEQ11

Parameter Values

A longword integer constant enabling or disabling terminations of I/O transfers by a service request.

The value can be one of the following:

Constant Value	Meaning
LIO$K_OFF[1]	I/O transfers are not terminated by a service request
LIO$K_ON	I/O transfers are terminated by a service request

[1]The default value.

Description

When enabled (LIO$K_ON), a service request results in the termination of the I/O transfer in progress.

Restrictions

None.

LIO$K_TERM_SRQ

Example

```
status = LIO$SET_I (ieee_id, LIO$K_TERM_SRQ, 1, LIO$K_ON)
```

This routine enables the termination of I/O transfers by a service request.

LIO$K_TIMEOUT

This parameter sets the I/O operation timeout for any device attached with QIO and for the disk file device.

Supported Devices

AAF01[1]
AAV11-D
ADF01[1]
ADV11-D
AXV11-C
DRB32
DRB32W
DRQ11-C[1]
DRV11-J
DRV11-WA
IEQ11
IEZ11
IOtech Micro488A
KWV11-C
Preston (DRB32W and DRV11-WA interfaces only)
Simpact RTC01
Serial line

Parameter Values

A longword integer indicating the number of seconds before timeout.

The value can be between 2 and 65,636, inclusive.

For all devices except the IEQ11 and the IEZ11, the default value is 10 seconds.

For the IEQ11, the default value is 20 seconds.

For the IEZ11, the default value is 30 seconds.

[1] This device is available only in Europe.

LIO$K_TIMEOUT

Description

Use this parameter to specify a data transfer timeout.

If a data transfer does not finish within the specified number of seconds, the buffer is terminated. The status %SYSTEM-F-TIMEOUT is returned by the LIO$READ, LIO$WRITE, and LIO$DEQUEUE routines. The status %SYSTEM-F-TIMEOUT is also returned to the user's AST routine.

Restrictions

- If you use this parameter with a serial line device, you must first enable the serial device for timeouts using the LIO$K_TIMEOUT_ENABLE parameter.
- The KWV11-C and the Simpact RTC01 clocks use the timeout value only when the event timing functions LIO$K_EVENT_ABS and LIO$K_EVENT_REL are set. The functions LIO$K_SGL_COUNT and LIO$K_REP_COUNT always use a timeout of 65,535 seconds.

Example

```
status = LIO$SET_I (device_id, LIO$K_TIMEOUT, 1, 15)
```

This routine specifies a 15-second timeout period for a data transfer.

LIO$K_TIMEOUT_ENABLE

This parameter enables or disables timeouts for serial line read requests.

Supported Devices

Serial line

Parameter Values

A longword integer constant.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF	Disables timeouts
LIO$K_ON[1]	Enables timeouts

[1]The default value.

Description

This parameter affects read requests only.

Restrictions

If you use this parameter to enable timeouts, you must specify the length of the timeout in seconds using the LIO$K_TIMEOUT parameter.

Example

```
status = LIO$SET_I (serial_id, LIO$K_TIMEOUT_ENABLE, 1, LIO$K_ON)
```

This routine enables a serial line device for timeouts.

LIO$K_TITLE

This parameter specifies the title placed on the graph of the channel specified by LIO$K_CURRENT_CHANNEL.

Supported Devices

Real-time plotting

Parameter Values

A character string less than or equal to 72 characters in length specifying the title.

This value is passed by descriptor.

Description

The title is placed over the graph of the current channel.

If a single x-axis is being used for the plotting window, use LIO$K_CURRENT_CHANNEL to set channel 0 as the current channel so that the title will be placed over the entire screen.

Use LIO$K_X_LABEL and LIO$K_Y_LABEL to place labels on the x-axis and y-axis.

Restrictions

- The title is limited to 72 characters.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_S (graphics_id, LIO$K_TITLE, 'Channel 2')
```

This routine specifies "Channel 2" as the title of the current channel.

LIO$K_TITLE_n

This parameter specifies the graph title and the position of the channel to be plotted in the LIO$K_PO_CHAN list for the real-time plotting device.

NOTE

The preferred method of specifying titles of graphs is to use LIO$K_CURRENT_CHANNEL followed by LIO$K_TITLE. Using LIO$K_TITLE_n is the non-preferred method.

Supported Devices

Real-time plotting

Parameter Values

A character string less than or equal to 72 characters in length specifying the title of the graph of the data from a channel.

This value is passed by descriptor.

Description

Use this parameter to specify titles for the graphs depicting the data plotted from the channels specified in the LIO$K_PO_CHAN list. The "n" in LIO$K_TITLE_n is a placeholder used to specify the position of the channel in the LIO$K_PO_CHAN list.

LIO$K_TITLE_0 is used to specify the title of the graph of the first channel specified in the LIO$K_PO_CHAN list. LIO$K_TITLE_1 is used to specify the title of the graph of the second channel specified in the LIO$K_PO_CHAN list, and so forth.

For example, if channels 4 and 2 are specified (in that order) in the LIO$K_PO_CHAN list, then you use LIO$K_TITLE_0 to specify the title for the graph of channel 4 and LIO$K_TITLE_1 to specify the title for the graph of channel 2.

A maximum of eight graphs, one graph for each channel listed in the LIO$K_PO_CHAN list, can be plotted in one window.

Restrictions

- Each character string (graph title) can be a maximum of 72 characters in length.
- You can specify only one graph title at a time.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Examples

1. `status = LIO$SET_I (graphics_id, LIO$K_PO_CHAN, 2, 4, 2,)`

   This routine specifies that channels 4 and 2 be plotted, in that order, by the real-time plotting device.

2. `status = LIO$SET_S (graphics_id, LIO$K_TITLE_0, 'Channel 4')`

   This routine specifies the title of the graph of channel 4, the first channel specified in the LIO$K_PO_CHAN list, as "Channel 4".

3. `status = LIO$SET_S (graphics_id, LIO$K_TITLE_1, 'Channel 2')`

   This routine specifies the title of the graph of channel 2, the second channel specified in the LIO$K_PO_CHAN list, as "Channel 2".

LIO$K_TRANSFER

This parameter sets up an interprocess memory queue device to transfer data buffers between processes.

Supported Devices

Memory queue

Parameter Values

None.

Description

This parameter sets up a memory queue device to pass buffers to and receive buffers from another process.

Restrictions

The memory queue device in the other process must be attached and set up identically to the memory queue device in this process.

Example

```
status = LIO$SET_I (device_id, LIO$K_TRANSFER, 0)
```

This routine sets up the interprocess memory queue to transfer data buffers between processes.

LIO$K_TRIG

This parameter sets the device trigger mode or source.

Supported Devices

AAV11-D
ADQ32
ADV11-D
AXV11-C
KWV11-C
Preston
Simpact RTC01

Parameter Values

For the **AAV11-D**, one or two longword integer constants.

The first value specifies the trigger mode value.

This value can be one of the following:

Table 4–10: AAV11-D Trigger Modes

Constant Value	Device Trigger Mode
LIO$K_IMM_BURST	Immediate start burst mode. This starts immediately on the LIO$WRITE or the LIO$ENQUEUE routine call, or on the LIO$SET_I call with the LIO$K_START parameter if the AAV11-D is set for continuous DMA mode. In continuous DMA mode, the D/A continues to send data from the successive buffers until stopped, or until the software stops enqueuing buffers. Because of Q-bus latency, a consistent rate cannot be guaranteed.

Table 4–10 (Cont.): AAV11-D Trigger Modes

Constant Value	Device Trigger Mode
LIO$K_EXT_BURST LIO$K_CLK_BURST	Triggered burst mode. On each (clock or external) trigger, this mode sends buffers as fast as the DMA can move buffers. When set for continuous DMA, the trigger starts the output. Then, the hardware moves the data as fast as possible. To use this trigger option with the AAV11-D, you must specify both D/A channels for use through the LIO$K_DA_CHAN parameter. Because of Q-bus latency, a consistent rate cannot be guaranteed. See the **Description** for additional information about supplying the clock device ID.
LIO$K_EXT_SWEEP LIO$K_CLK_SWEEP	Triggered sweep mode. This mode sweeps all selected channels on each trigger. See the **Description** for additional information about supplying the clock device ID.
LIO$K_EXT_POINT LIO$K_CLK_POINT	Triggered nonburst mode. This mode outputs to one channel on each external trigger or clock tick. If the AAV11-D is attached to use QIOS, using the LIO$K_CLK_POINT and LIO$K_EXT_POINT trigger modes are legal only when you select to output to one D/A channel. If you select to output to both D/A channels, the LIO facility returns an error. See the **Description** for additional information supplying the clock device ID.

The second value, which is optional, specifies the device ID of the clock. Specifying the clock device ID is valid for the external trigger modes only when the device is set to use the synchronous I/O interface.

For the ADQ32, a maximum of three longword integer constants.

The first value specifies the point trigger source.

This value can be one of the following:

Table 4–11: ADQ32 Point Trigger Sources

Constant Value	Trigger Source
LIO$K_EXTERNAL[1]	A data point is taken each time the external clock input goes low. If you specify this value, the sweep trigger and buffer trigger values must be LIO$K_SAME.
LIO$K_BURST	Data points are taken at the top speed of the A/D.
LIO$K_AD_CLOCK	This value specifies the primary A/D clock as the trigger source for the ADQ32 device. The rate for the clock is set through the LIO$K_CLK_RATE parameter.

[1]The ADQ32 has two external inputs: the **external gate/trigger** input and the **external frequency** input. The LIO$K_EXTERNAL value generally refers to the external gate/trigger input. However, when you are specifying all points triggered by the same source (LIO$K_EXTERNAL, LIO$K_SAME, LIO$_SAME), and you are using the external gate/trigger input to gate the trigger (specified by the LIO$K_GATE parameter), then the LIO$K_EXTERNAL value refers to the external frequency input.

The second value specifies the sweep trigger source. This is the trigger to take the first point in a channel sweep.

This value can be one of the following:

Table 4–12: ADQ32 Sweep Trigger Sources

Constant Value	Trigger Source
LIO$K_EXTERNAL[1]	Sweeps through the specified channels are controlled by negative transitions on the external gate/trigger input. If you specify this value, the buffer trigger value must be LIO$K_SAME.

[1]This trigger source cannot be used in conjunction with a gating mode specified using the LIO$K_GATE parameter because both require the use of the external gate/trigger input.

Table 4–12 (Cont.): ADQ32 Sweep Trigger Sources

Constant Value	Trigger Source
LIO$K_SAME	The A/D does not differentiate between the first point in the channel sweep and the rest of the points in the channel sweep.
LIO$K_SWEEP_CLOCK	The LIO$K_SWEEP_RATE parameter sets the rate of the sweep A/D clock. If the point trigger source is specified as LIO$K_AD_CLOCK, then the point clock rate (set by the LIO$K_CLK_RATE parameter) must be high enough for data from all channels in the channel list to be acquired before another sweep clock tick.

The third value specifies the buffer trigger source. This is the trigger to take the first point in the buffer.

This value can be one of the following:

Table 4–13: ADQ32 Buffer Trigger Sources

Constant Value	Trigger Source
LIO$K_EXTERNAL[1]	The A/D waits for the external clock input to go low before it starts to take data.
LIO$K_SAME	The A/D does not differentiate between the first point in the buffer and the first point of any other sweep.

[1]This trigger source cannot be used in conjunction with a gating mode specified using the LIO$K_GATE parameter because both require the use of the external gate/trigger input.

See Appendix A for more information on ADQ32 trigger modes.

For the **ADV11-D**, one or two longword integer constants. The first value specifies the trigger mode value.

This value can be one of the following:

Table 4–14: ADV11-D Trigger Modes

Constant Value	Trigger Mode
LIO$K_EXT_SWEEP LIO$K_CLK_SWEEP	Triggered sweep mode. This mode sweeps all selected channels on each trigger. This trigger mode is valid only when the ADV11-D is attached with mapped I/O. See the **Description** for additional information about supplying the clock device ID.
LIO$K_IMM_BURST	Immediate start burst mode. This starts immediately on the LIO$READ or the LIO$ENQUEUE routine call, or on the LIO$SET_I call with the LIO$K_START parameter specifying continuous DMA mode. In continuous DMA mode, the A/D continues to accept data into the successive buffers until stopped, or until the software stops enqueuing buffers. When the ADV11-D is attached to use QIOs, be sure that the channel list you set up using the LIO$K_AD_CHAN parameter specifies one A/D channel or all A/D channels. If you use all A/D channels, the buffer size must be greater than 32 bytes (one sweep of all 16 channels). The number of samples read from the device is the largest multiple of 16 (2-byte) samples that fit in the buffer. Because of Q-bus latency, a consistent rate cannot be guaranteed.
LIO$K_EXT_BURST	Triggered burst mode.
LIO$K_EXT_POINT LIO$K_CLK_POINT	Triggered nonburst mode. This mode inputs to one channel on each clock tick. When the ADV11-D is attached with LIO$K_QIO, the parameter value of the LIO$K_AD_CHAN parameter must specify either one channel only or all channels in ascending order. See the **Description** for information about supplying the clock device ID.

The burst modes of triggering run as fast as the hardware can. For QIOs, this is 50 kHz. For mapped I/O, this is as fast as the polling loop can read from the A/D.

The second value, which is optional, specifies the device ID of the clock. Specifying the clock device ID is valid for external trigger modes only when the device is set for the synchronous I/O interface.

For the **AXV11-C**, one or two longword integer constants.

The first value specifies the trigger mode. The trigger mode applies only to the AXV11-C A/D converter. The D/A converter always outputs data immediately and does not use trigger modes.

This value can be one of the following:

Table 4–15: AXV11-C Trigger Modes

Constant Value	Trigger Mode
LIO$K_IMM_BURST	Immediate start burst mode. This starts immediately on the LIO$READ or LIO$ENQUEUE routine call.
LIO$K_EXT_BURST LIO$K_CLK_BURST	Externally triggered burst mode. On each external trigger, this mode sends a buffer as fast as the A/D can move data. When the AXV11-C is attached to use QIOs, be sure that the channel list you set up using the LIO$K_AD_CHAN parameter specifies one A/D channel or all A/D channels. If you use all A/D channels, the buffer size must be greater than 32 bytes (one sweep of all 16 channels). The number of samples read from the device is the largest multiple of 16 (2-byte) samples that fit in the buffer. See the **Description** for additional information.

Table 4–15 (Cont.): AXV11-C Trigger Modes

Constant Value	Trigger Mode
LIO$K_EXT_SWEEP LIO$K_CLK_SWEEP	Externally triggered sweep mode. This mode sweeps all selected channels on each trigger. The buffer size must be greater than a single sweep of the selected channels. The number of samples read from the device is the largest multiple of the number of channels that fit in the buffer. See the **Description** for additional information.
LIO$K_EXT_POINT LIO$K_CLK_POINT	Externally triggered point mode. This mode sets the A/D converter to sample one channel on each external trigger. If you specify several channels for use, the next channel in the list is sampled on each successive external trigger. When the A/D converter reaches the end of the channel list, it begins sampling again at the first channel in the list.

The second value, which is optional, specifies the clock device ID. Specifying the clock device ID is valid for the triggered burst and triggered point modes.

The device can be set to use the asynchronous I/O or the synchronous I/O interface.

For the **KWV11-C** or the **Simpact RTC01**, a longword integer constant specifying the trigger mode.

The value can be one of the following:

Table 4–16: KWV11-C/Simpact RTC01 Trigger Modes

Constant Value	Trigger Mode
LIO$K_IMMEDIATE	Software start. Start the clock after the appropriate conditions are met. See the **Restrictions** for these conditions.
LIO$K_EXTERNAL	External trigger ST2. Start the clock on the first Schmitt trigger 2 after one of the appropriate conditions is met. See the **Restrictions** for these conditions.

See the **Restrictions** for appropriate KWV11-C and Simpact RTC01 start conditions.

For the **Preston**, a longword integer constant specifying the trigger mode.

The value can be one of the following:

Table 4–17: Preston Trigger Modes

Constant Value	Trigger Mode
LIO$K_IMM_START_CLK_POINT	Immediate start one point per clock pulse. This mode starts the A/D immediately on the LIO$READ or LIO$ENQUEUE routine call and reads one data point for each Preston internal clock pulse.
LIO$K_IMM_START_CLK_SWEEP	Immediate start one channel sweep for each clock pulse. This mode starts the A/D immediately after the LIO$READ or LIO$ENQUEUE routine and reads one channel sweep for each Preston internal clock pulse.
LIO$K_IMM_START_EXT_POINT	Immediate start one point per external clock pulse. This mode starts the A/D immediately after the LIO$READ or LIO$ENQUEUE routine, and reads one data point for each external clock pulse.

Table 4–17 (Cont.): Preston Trigger Modes

Constant Value	Trigger Mode
LIO$K_EXT_START_CLK_POINT	External start one point per internal clock pulse. This mode starts the A/D on the first external trigger pulse after the LIO$READ or LIO$ENQUEUE routine, and reads one data point for each internal clock pulse.
LIO$K_EXT_START_CLK_SWEEP	External start one channel sweep per internal clock pulse. This mode starts the A/D on the first external pulse after the LIO$READ or LIO$ENQUEUE routine, and reads one channel sweep for each internal clock pulse.
LIO$K_EXT_START_EXT_POINT	External start one point per external clock pulse. This mode starts the A/D on the first external trigger pulse after the LIO$READ or LIO$ENQUEUE routine, and reads one data point for each external clock pulse.
LIO$K_EXT_START_EXT_SWEEP	External start one channel sweep per external clock pulse. This mode starts the A/D on the first external trigger pulse after the LIO$READ or LIO$ENQUEUE routine, and reads one channel sweep for each external clock pulse.

Description

The clock input is (usually) wired to the clock device. If you specify the device ID of the clock as the second value of this parameter, then the input calls start and stop the clock. This prevents hardware errors resulting from clock pulses occurring before the A/D converter is ready to accept them.

Restrictions

- The restrictions for the AAV11-D, ADV11-D, and AXV11-C device-specific trigger modes are described in the **Parameter Values**.
- The clock must not already be running.

- The KWV11-C and Simpact RTC01 begin operating under different conditions depending on which trigger mode is selected and whether the transfer is synchronous or asynchronous, as explained below:
  - If the clock is set for synchronous reads, and the trigger mode is LIO$K_IMMEDIATE, the clock begins when the LIO$READ statement executes.
  - If the clock is set for synchronous reads, and the trigger mode is LIO$K_EXTERNAL, the clock begins when the external ST1 event occurs.
  - If the clock is set for asynchronous enqueues, and the trigger mode is LIO$K_IMMEDIATE, the clock begins when the LIO$K_START statement executes. (See LIO$K_START.)
  - If the clock is set for asynchronous enqueues, and the trigger mode is LIO$K_EXTERNAL, the clock begins when the external ST1 event occurs.
- The Simpact RTC01 does not use the trigger mode until the clock is started using LIO$SET_I with the LIO$K_START parameter.

Examples

1.
```
status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST, LIO$K_SAME,
1                          LIO$K_SAME)
```

This routine sets up the ADQ32 to take data at the burst rate of the A/D.

2.
```
status = LIO$SET_I (adv_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST)
```

This routine sets up the ADV11-D to begin taking data as soon as the program executes an LIO$READ or LIO$ENQUEUE routine. The ADV11-D fills the data buffer as fast as possible.

LIO$K_TYPE_AHEAD

This parameter enables or disables a serial line type-ahead buffer.

Supported Devices

Serial line

Parameter Values

A longword integer constant.

The value can be one of the following:

Constant Value	Function
LIO$K_OFF	Disables the type-ahead buffer. (Data is lost if there is no read request pending.)
LIO$K_ON[1]	Enables the type-ahead buffer. (Data is maintained in the typeahead buffer if no read request is pending and unsolicited data is received.)

[1]The default value.

Description

Enabling the type-ahead buffer allows the buffering of unsolicited data. The buffered data is returned during a read request. Use the LIO$K_UNSOLICITED parameter to return the number of characters in the type-ahead buffer, in bytes.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_TYPE_AHEAD, 1, LIO$K_ON)
```

This routine enables the type-ahead buffer.

LIO$K_UNLOCK_BUFFER

This parameter unlocks a buffer previously locked with the LIO$K_LOCK_BUFFER parameter.

Supported Devices

DRB32

Parameter Values

Two longword integer values.

The first value specifies the buffer to unlock.

The second value specifies the size of the buffer in bytes.

Description

Unlocking buffers is necessary when you have locked the maximum number of buffers (16) and you need to lock another buffer not previously locked.

Restrictions

This parameter only unlocks buffers that have been previously locked with the LIO$K_LOCK_BUFFER parameter. This parameter cannot unlock buffers that are locked by standard VMS mechanisms.

Example

```
status = LIO$SET_I (device_id, LIO$K_UNLOCK_BUFFER, 2, buffer_address,
1                  4096)
```

This routine unlocks one 4096-byte buffer at address **buffer_address**.

LIO$K_UNSOLICITED

This parameter returns the number of characters in the type-ahead buffer.

Supported Devices

Serial line

Parameter Values

A longword integer.

Description

The number of characters in the type-ahead buffer is useful to help determine if your instrument is outputting data (unsolicited input) before you issue a read request. Any data output by the device between the time you attach the device and the time you actually issue a read request is returned in the integer you supply.

Restrictions

This is an LIO$SHOW parameter only.

Example

```
INTEGER*4 number_char
status = LIO$SET_I (serial_id, LIO$K_UNSOLICITED, 1, number_char)
```

This routine returns the number of characters in the type-ahead buffer in the argument **number_char**.

LIO$K_UPDATE

This parameter causes the set-up information to be sent to the Preston device through the parallel port.

Supported Devices

Preston

Parameter Values

None.

Description

This parameter physically updates the Preston device to the currently specified set-up parameters.

Restrictions

- This must be the last parameter to be set, with the exception of the LIO$K_START and LIO$K_STOP parameters when using continuous DMA, before the first LIO$READ and LIO$ENQUEUE to the device.
- This is not a valid LIO$SHOW parameter.
- This is a required parameter.

Example

```
status = LIO$SET_I (device_id, LIO$K_UPDATE, 0)
```

This routine updates, or sets up, the Preston device with the set-up characteristics specified in previous LIO$SET routine calls in the user program.

LIO$K_USER_ACK_AST

This parameter specifies the address of a user-supplied AST routine that transmits an ACK message on successful completion of a data transfer.

Supported Devices

Serial line

Parameter Values

A longword integer.

Description

Use an AST routine to transmit ACK messages that consist of more than a fixed ACK string.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_USER_ACK_AST, 1, ack_ast)
```

This routine supplies the address of an AST routine called **ack_ast**.

LIO$K_USER_ACK_STRING

This parameter supplies an ACK string to be sent out on successful completion of a data transfer.

Supported Devices

Serial line

Parameter Values

A character string specifying what to transmit on successful completion of a data transfer.

This value is passed by descriptor.

Description

Use an ACK string to signal an acknowledge of successful completion to a serial device.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_S (serial_id, LIO$K_USER_ACK_STRING, 'Confirmed')
```

This routine specifies that the string "Confirmed" be transmitted to a serial device to signal an acknowledge.

LIO$K_USER_NAK_AST

This parameter specifies the address of a user-supplied AST routine to transmit the NAK message on unsuccessful completion of a data transfer.

Supported Devices

Serial line

Parameter Values

A longword integer.

Description

Use an AST routine to transmit NAK messages that consist of more than a fixed NAK string.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_USER_NAK_AST, 1, nak_ast)
```

This routine supplies the address of an AST routine called **nak_ast**.

LIO$K_USER_NAK_STRING

This parameter supplies the NAK string to be sent out on unsuccessful completion of a data transfer.

Supported Devices

Serial line

Parameter Values

A character string specifying what to transmit on unsuccessful completion of a data transfer.

This value is passed by descriptor.

Description

Use a NAK string to signal a negative acknowledge to a serial device.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_S (serial_id, LIO$K_USER_NAK_STRING, 'Not received')
```

This routine specifies that the string "Not received" be transmitted to a serial device to signal a negative acknowledge.

LIO$K_USER_READ_PROTOCOL_AST

This parameter sets up the protocol AST routine to specify what action to take on receipt of a terminator or full buffer of characters from a read request.

Supported Devices

Serial line

Parameter Values

A longword integer specifying the address of an AST routine to be called on receipt of either an input terminator or a full buffer of characters from a read request.

Description

This parameter enables users to define a protocol for the serial line. A user program can examine a buffer and determine if an acknowledge or a negative acknowledge is to be sent out on the serial line.

The status returned by the buffer determines what action is appropriate. The buffer can return one of the following status values:

Status Value	Action
LIO$K_ABORT	Abort.
LIO$K_ACK_STRING	Send the string defined by the LIO$K_USER_ACK_STRING parameter as the acknowledge to the device.
LIO$K_NAK_STRING	Send the string defined by the LIO$K_USER_NAK_STRING parameter as the negative acknowledge to the device.
LIO$K_ACK_ROUTINE	Call the AST routine defined by LIO$K_USER_ACK_AST to enable the user to build an acknowledge string to send to the device.

Status Value	Action
LIO$K_NAK_ROUTINE	Call the AST routine defined by LIO$K_USER_NAK_AST to enable the user to build a negative acknowledge string to send to the device.
LIO$K_DEVICE_ACKED	Call the AST routine defined by LIO$K_DEVICE_ACK_AST to take appropriate action on a successful write.
LIO$K_DEVICE_NAKED	Call the AST routine defined by LIO$K_DEVICE_NAK_AST to take appropriate action on an unsuccessful write.
LIO$K_NO_ACTION	Take no action.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

None.

Example

```
status = LIO$SET_I (serial_id, LIO$K_USER_READ_PROTOCOL_AST, 1,
1                  LIO$K_ACK_STRING)
```

This routine signals the sending of the string defined by the LIO$K_USER_ACK_STRING parameter as the acknowledge to the device.

LIO$K_USER_WRITE_NAK_HANDLING

This parameter specifies whether or not a sending device attempts to retransmit a buffer after receiving a NAK from the intended receiving device.

Supported Devices

Serial line

Parameter Values

A longword integer constant.

The value can be one of the following:

Constant Value	Function
LIO$K_RESEND_LAST	Retransmits the buffer to the device.
LIO$K_NO_RESEND[1]	Does not retransmit the buffer.

[1]The default value.

Description

This parameter determines the action that is taken on receiving a NAK from a serial line device.

See the description of LIO$K_PROTOCOL for more information.

Restrictions

This parameter is used with user-defined protocols for serial line devices.

LIO$K_USER_WRITE_NAK_HANDLING

Example

```
status = LIO$SET_I (serial_id, LIO$K_USER_WRITE_NAK_HANDLING, 1,
1                       LIO$K_RESEND_LAST)
```

This routine retransmits the buffer after receiving a NAK from the intended receiving device.

LIO$K_VLT_DDR

This parameter converts a voltage into its corresponding complementary binary-coded value and moves it to the DAC Data Register (DDR).

Supported Devices

ADF01[1]

Parameter Values

A single-precision, floating-point real value specifying the DAC voltage in the range −10.0000 to +9.9951.

Description

See the **Parameter Values**.

Restrictions

None.

Example

```
status = LIO$SET_R (adf_id, LIO$K_VLT_DDR, 1, real_value)
```

This routine moves the corresponding value of **real_value** in complementary offset binary code into the DDR. The **real_value** argument contains the DAC voltage value in the range −10.0000 to +9.9951.

[1] This device is available only in Europe.

LIO$K_VOLTAGE

This parameter specifies the input voltage range.

Supported Devices

IDV11-A[1]

Parameter Values

A longword integer constant.

The value can be one of the following:

Value	Meaning
0[1]	Standard input range
1	Low-level input range

[1]The default value.

Description

The standard input range for all 16-digital inputs is 24 to 48V DC.

The low-level range allows low voltage and low power signal sources, such as transistor-to-transistor logic (TTL) at 0 to 5V, and metal oxide semiconductor (MOS) at 0 to 3V.

Restrictions

None.

[1] This device is available only in Europe.

Example

```
status = LIO$SET_I (idva_id, LIO$K_VOLTAGE, 1, 1)
```

This routine sets the IDV11-A for low-level voltage input.

LIO$K_X_LABEL

This parameter specifies the label placed on the x-axis of the channel specified by LIO$K_CURRENT_CHANNEL.

Supported Devices

Real-time plotting

Parameter Values

A character string less than or equal to 72 characters in length specifying the label.

This value is passed by descriptor.

Description

The label is placed under the x-axis of the current channel.

If a single x-axis is being used for the plotting window, use LIO$K_CURRENT_CHANNEL to set channel 0 as the current channel so that the x-axis label is placed for the entire screen.

Use LIO$K_Y_LABEL to set the y-axis label.

Restrictions

- The label is limited to 72 characters.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_S (graphics_id, LIO$K_X_LABEL, 'Degrees')
```

This routine specifies "Degrees" as the x-axis label for the current channel.

LIO$K_X_RANGE

This parameter specifies the number of points to plot along the x axis, and the increment at which points are plotted.

Supported Devices

Real-time plotting

Parameter Values

Two longword integer values.

The first value specifies the number of data points to be plotted along the x-axis. The default value is 100.

The second value specifies the increment at which points are to be plotted. The default value is 10.

Description

The number of points plotted on the x axis determines how many data points are visible at one time. The increment at which points are plotted determines how many data points are plotted from a single channel at one time.

Restrictions

- The increment must be no more than one half of the number of points displayed along the x axis.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_I (graphics_id, LIO$K_X_RANGE, 2, 200, 20)
```

This routine sets up the real-time plotting device to plot 200 data points, with an increment of 20.

LIO$K_XON

This parameter forces the sending of an XON character to reprime the serial line.

Supported Devices

Serial line

Parameter Values

None.

Description

XON/XOFF signals control the data flow along the serial line. Sending an XON signal tells the device to start sending data.

Restrictions

None.

Example

```
status = LIO$SET_I (device_id, LIO$K_XON, 0)
```

This routine sends an XON signal along the serial line.

LIO$K_Y_LABEL

This parameter specifies the label placed on the y-axis of the channel specified by LIO$K_CURRENT_CHANNEL.

Supported Devices

Real-time plotting

Parameter Values

A character string less than or equal to 72 characters in length specifying the label.

This value is passed by descriptor.

Description

The label is placed parallel to the y-axis of the current channel.

Restrictions

- The label is limited to 72 characters.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_S (graphics_id, LIO$K_Y_LABEL, 'Amplitude')
```

This routine specifies "Amplitude" as the y-axis label for the current channel.

LIO$K_Y_MAX

This parameter specifies the maximum y value for each channel to be plotted.

Supported Devices

Real-time plotting

Parameter Values

A single-precision, floating-point real value or values specifying the maximum y value for each channel to be plotted.

The default value is 10.0.

You can specify up to eight maximum y values, one for each channel specified in the LIO$K_PO_CHAN list.

Description

See the **Parameter Values**.

Restrictions

- You must specify one maximum y value for each channel specified in the LIO$K_PO_CHAN list.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_R (graphics_id, LIO$K_Y_MAX, 2, 4.0, 4.0)
```

This routine specifies a maximum y value of 4.0 for each of two channels specified in the LIO$K_PO_CHAN list.

LIO$K_Y_MIN

This parameter specifies the minimum y value for each channel to be plotted.

Supported Devices

Real-time plotting

Parameter Values

A single-precision, floating-point real value or values specifying the minimum y value for each channel to be plotted.

The default value is −10.0.

You can specify up to eight minimum y values, one for each channel specified in the LIO$K_PO_CHAN list.

Description

See the **Parameter Values**.

Restrictions

- You must specify one minimum y value for each channel specified in the LIO$K_PO_CHAN list.
- The real-time plotting device is supported only on VAXstation-based VAXlab systems running VWS.

Example

```
status = LIO$SET_R (graphics_id, LIO$K_Y_MIN, 2, -2.0, -2.0)
```

This routine specifies a minimum y value of −2.0 for each of two channels specified in the LIO$K_PO_CHAN list.

Chapter 5

Laboratory I/O Error Handling

This chapter describes Laboratory I/O (LIO) error handling, explains the error messages, and provides recovery procedures.

5.1 Overview

When you execute an image that results in an LIO error, the system locates the error message associated with the error and directs it to the devices or files defined as SYS$ERROR and SYS$OUTPUT.

The system also generates messages when routine calls in an application program execute successfully. The LIO routines use the same standards as the VMS Run-Time Library and System Services for returning status information about routine calls.

The VMS Run-Time Library and System Services return a status value which is passed back to the user program through a longword variable when the routine is called as a function.

A successful operation returns an LIO success status value with bit zero set (true). An unsuccessful operation returns one of the LIO symbolic status values with bit zero clear (false). The symbols for the status values are defined in definition files for each of the programming languages listed in the following table:

Table 5–1: Error Handling Symbolic Status Definition Files

Language	Symbolic Status Definition File
VAX Ada	SYS$LIBRARY:LIOERRS.ADA
VAX BASIC	SYS$LIBRARY:LIOERRS.BAS
VAX C	SYS$LIBRARY:LIOERRS.H
VAX FORTRAN	SYS$LIBRARY:LIOERRS.FOR
VAX MACRO	SYS$LIBRARY:LIOERRS.MAR
VAX Pascal	SYS$LIBRARY:LIOERRS.PAS

See Section 5.4, Symbolic Status Values and Descriptions, for a description of the symbolic status values.

5.2 Checking Routine Call Status

A user program can check the status of routine calls in the following two ways:

- By testing status for success after each operation and by signaling the condition value to the device or file defined as SYS$ERROR and SYS$OUTPUT if the operation is not successful.

```
 status = LIO$DEQUEUE (device_id, buff_ptr, buff_len,
1            data_len, 0, ,)
          IF (.NOT. status) CALL LIB$SIGNAL(%VAL(STATUS))
```

- By testing status after each operation for a specific condition value.

```
  INCLUDE 'SYS$LIBRARY:LIOERRS.FOR' !Symbolic status definitions
      .
      . Declare variables, attach device, set up device
      .

C Poll for a completed buffer.  If a buffer is not available,
C print out the message.  If a buffer is available, process it.

5 CONTINUE

C Poll for a completed buffer:
  status = LIO$DEQUEUE (device_id, buff_ptr, buff_len,
 1            data_len, 0, ,)
        IF (status .EQ. LIO$_EMPTYQ) THEN  !Queue is empty

           TYPE *, 'No buffers on user queue'
           GOTO 5

       ELSE IF (.NOT. status) !Check status
           CALL LIB$SIGNAL (%VAL(STATUS)) !Signal error, if any

       ELSE
         .
         .   Buffer processing may occur here
         .
       ENDIF
```

This program segment tests if status equals the LIO$_EMPTYQ condition value. If status is equal to LIO$_EMPTYQ, then the program types out the "No buffers on user queue" message and attempts to dequeue the buffer again. The program continues to call LIO$DEQUEUE until it successfully dequeues the buffer. When this occurs, status is no longer equal to LIO$K_EMPTYQ. The new value of status is directed to the device or file defined as SYS$ERROR, and the program continues execution.

NOTE

If your program is coded to check for specific condition values after one or more operations, you must include the symbolic value definition file appropriate for your programming language. The symbolic definition files supplied by VAXlab are listed in Table 5-1, Error Handling Symbolic Status Definition Files. If you do not include the symbolic definition file, LIO does not recognize these values.

5.3 Setting Up Devices for Error Handling

You can set up an I/O device with the LIO$K_ERR_HANDLE parameter to specify the way the device returns errors. The LIO facility supports the following error-handling methods for all devices:

- The device returns the symbolic status as the value of the routine call.

```
status = LIO$SET_I (device_id, LIO$K_ERR_HANDLE, 1, LIO$K_STATUS)
         IF(.NOT. status) CALL LIB$SIGNAL(%VAL(STATUS))
```

  This error-handling method enables the user program to handle error recovery. **This is the default for all devices.** If you do not explicitly set up a device using the LIO$K_ERR_HANDLE parameter, the device's error-handling method defaults to LIO$K_STATUS.

- The device directs the status value to both SYS$OUTPUT and SYS$ERROR, and returns the symbolic status as the value of the routine call. This method does not terminate program execution on fatal errors. Code the program to handle error recovery.

```
status = LIO$SET_I (device_id, LIO$K_ERR_HANDLE, 1, LIO$K_MESSAGE)
         IF(.NOT. status) CALL LIB$SIGNAL(%VAL(STATUS))
```

  This error-handling method simplifies the debugging of a program that handles error conditions.

- The device directs all status values to both SYS$OUTPUT and SYS$ERROR. On fatal errors, the program stops execution.

```
C Check status of LIO$SET_I call because LIO$K_FATAL is not
C set up yet.

  status = LIO$SET_I (device_id, LIO$K_ERR_HANDLE, 1, LIO$K_FATAL)
           IF(.NOT. status) CALL LIB$SIGNAL(%VAL(STATUS))

C LIO$K_FATAL error handling method is set up now
    .
    .
    .
C No need to check status of routine call

  CALL LIO$DEQUEUE (device_id, buff_ptr, buff_len,
 1        data_len, 1,,)
```

NOTE

In this case, your program can call the LIO routines as subroutines instead of functions because the program does not have to check the returned status. However, you still need to check the status of any routine calls you make before you set up the device with LIO$K_FATAL. The device uses LIO$K_STATUS (the default) until you specify another method.

If you set up an IX device with LIO$K_FATAL, you still need to check the status of LIO$DEQUEUE calls to determine the reason for buffer completion. In this case, you are testing status for a specific condition value. Thus, you also need to include the symbolic status definition file appropriate for your programming language.

```
  INCLUDE 'SYS$LIBRARY:LIOERRS.FOR' !Get symbolic status definitions
      .
      . Declare variables, attach device, set up device, etc.
      .
C Check status of LIO$SET_I call because LIO$K_FATAL is not
C set up yet.
  status = LIO$SET_I (ix_id, LIO$K_ERR_HANDLE, 1, LIO$K_FATAL)
           IF (.NOT. status) CALL LIB$SIGNAL(%VAL(STATUS)
C LIO$K_FATAL error handling method is set up now
    .
    .
    .
  C No need to check status of routine call
  CALL LIO$ENQUEUE (ix_id, buff_ptr, buff_len, , , , )
    .
    .
    .
5 CONTINUE
C Check status of routine call
  status = LIO$DEQUEUE (ix_id, buff_ptr, buff_len,
 1            data_len, 1, ,)
            IF (status .EQ. LIO$_BUFFER_FULL) THEN
               GOTO 5  !There is more data to read
            ENDIF
```

This program segment checks the status of the LIO$DEQUEUE routine to determine if the buffer completes because it is full. If the buffer is full, then there is still more data to dequeue, so the program dequeues another buffer. This operation continues until an end condition other than LIO$K_BUFFER_FULL occurs.

5.4 Symbolic Status Values and Descriptions

This section presents the LIO symbolic status values and error messages alphabetically, with an explanation of each value and the appropriate action you should take to recover from each error condition.

LIO$_ACCVIO, Cannot access CTRL_AST or buffer invalid

Explanation: An address specified is not a valid virtual memory location.

User Action: Edit your program to ensure that you specify a valid address.

LIO$_ADDR_NOT_SET, IEEE-488 address not set for this unit

Explanation: An IEEE-488 device address has not been set up prior to the beginning of data transfers.

User Action: Use the LIO$K_IEEE_ADDR parameter to set up an IEEE-488 bus address for the device.

LIO$_ALREADY_ATTACHED, Device already attached

Explanation: The device is already attached, either by this program or by another program.

User Action: Your program may have tried to attach the device more than once. If this is the case, edit your program to remove the redundant LIO$ATTACH call. If the device is currently attached by another program, wait until the other program detaches the device.

LIO$_ARGREQ, Cannot default required argument

Explanation: You have omitted a required argument in the routine call argument list. The LIO facility does not supply a default value for the missing argument.

User Action: Check the reference description of the routine call for which this status value is returned to determine which argument was omitted. Edit your program to include the required argument in the argument list. If the argument requires a variable declaration, be sure to declare the variable and its appropriate data type.

LIO$_ATTACH_FAILED, Cannot attach device

Explanation: The LIO facility is unable to attach the device.

User Action: None.

LIO$_BUFF_OVERLAP, Specified buffer overlaps prelock buffer

Explanation: Your program attempted to perform an operation on a buffer that partially overlaps a prelocked buffer.

User Action: Ensure that all buffers, prelocked or otherwise, do not overlap each other.

LIO$_BUFFSIZE, Wrong buffer size for operation

Explanation: The size of the buffer declared is inappropriate for the operation. If this status value is returned after an LIO$ENQUEUE, LIO$READ, or LIO$WRITE routine call to a device, this indicates that the buffer is too small to contain a single data value.

If this status value is returned after an LIO$ENQUEUE routine call to a memory queue device, the buffer is larger than the buffers preallocated for the device. This applies to memory queues attached for local memory management and interprocess memory management.

This status value can also be returned if the buffer is too large for operation. If the device is the AXV11-C, buffer sizes must be 64K bytes or smaller for input operations.

User Action: For all devices, you must specify the **buffer_length** or **data_length** in numbers of bytes. For example, if you have a buffer that contains one data value that is a word (two bytes), then the **buffer_length** or the **data_length** must be specified as 2, which indicates one data value, two bytes in size.

For memory queue devices, preallocate larger buffers.

The AXV11-C can only input 64K bytes. Check to see that you did not exceed this number.

LIO$_BUFORDER, Continuous DMA buffers enqueued out of order

Explanation: Continuous DMA buffers are enqueued out of order. When a program enqueues continuous DMA buffers, they must be enqueued in ascending order beginning with the first buffer and incrementing the buffer array index with each successive buffer enqueued.

User Action: Check your program to be sure that the index value is included in your **buffer** argument in the LIO$ENQUEUE routine call, and that your program enqueues the buffers in ascending order, beginning with index value one.

See Section 1.6.3.2, Continuous DMA, for information about performing continuous DMA with VAXlab I/O devices.

LIO$_BUS_ERR, A bus error has occurred

Explanation: A VAXBI bus error was generated during a transaction.

User Action: If the condition persists, call your local Field Service office.

LIO$_CIC, Unit is controller-in-charge

Explanation: The requested operation cannot be performed while the IEEE-488 device is currently the controller-in-charge.

User Action: Use the LIO$K_PASS_CTRL parameter to specify another IEEE-488 device as the controller-in-charge. Or, use the LIO$K_CTRL_STANDBY parameter to place the current device's controller function on standby while the operation is performed.

LIO$_CLKOVERUN, Clock overrun

Explanation: The ADQ32 device is being triggered faster than it can take data.

User Action: Be sure that the device clock rates are set at values that are appropriate for the physical limitations of the device.

LIO$_CTGCDMA, Continuous DMA buffers are not contiguous

Explanation: The continuous DMA buffers do not form a contiguous 64K-byte block of memory.

User Action: To ensure that continuous DMA buffers form a contiguous 64K-byte block of memory, allocate the buffers as sections of a 64K-byte array. Also, check your program to ensure that you have used the correct array index and that you have supplied the correct number of bytes to the **buffer_length** argument.

LIO$_DETACH_FAILED, Cannot detach device

Explanation: The LIO facility is unable to detach the device.

User Action: None.

LIO$_DEVACTIVE, Device must be idle when changing parameters

Explanation: You attempted to change a DRB32 operating parameter while a data transfer is in progress.

User Action: Wait for all I/O requests to complete before issuing an LIO$SET_X call to change an operating parameter.

LIO$_DEV_ERR, I/O hardware detected an error

Explanation: The I/O hardware set its error bit. When using ADCs or DACs, the clock rate may be set too fast, or a condition known as "trigger slivering" has occurred.

User Action: Check your program to ensure that you have set up the clock rate and trigger sources appropriately for the device. Also, see Section 2.1.6, Using the KWV11-C to Avoid Trigger Slivering, to ensure that you have set up your device to avoid trigger slivering. This applies to AAV11-D, ADV11-D, and AXV11-C devices.

LIO$_DEVSPREQ, Device-specific argument required

Explanation: You have omitted the **device_specific** argument from a routine call argument list. This can occur in the following instances:

- On an LIO$ENQUEUE routine call to the AXV11-C, the **device_specific** argument is used to specify whether the buffer is sent to the ADC or a DAC on the device.

- On an LIO$ENQUEUE, LIO$READ, or LIOWRITE routine call to the DRV11-J, the **device_specific** argument is used to specify which of the four ports of the device is to be read from or written to by the routine call.

User Action: See the device support sections for the AXV11-C and DRV11-J devices in Chapter 2 for more information about specifying the **device_specific** argument and the acceptable values for this argument. Check your program to ensure that you have specified a **device_specific** argument appropriate for your device.

LIO$_EMPTYQ, No buffers on queue

Explanation: The LIO$DEQUEUE routine has attempted to dequeue a buffer from the device's user and found no buffers available.

User Action: Specify a nonzero value for the **wait** argument of the LIO$DEQUEUE routine to signal the routine to wait for a buffer to become available. Or, continue to attempt to dequeue the buffer until the operation is successful. The nature of your application determines which method is appropriate.

LIO$_FIL_OPEN, File is open

Explanation: Your program attempted to set up a file device that your program has already opened.

User Action: Check your program to ensure that you have set up all characteristics of the file device before you open the file. The LIO$K_OPEN_FILE parameter must be the last parameter you set. This parameter actually opens the file device with the characteristics you set up previously in the same program.

LIO$_FLAGREQD, An event flag is required

Explanation: When enqueueing buffers to the AAV11-D and the ADV11-D for continuous DMA, you must supply an event flag in the LIO$ENQUEUE routine call argument list. An event flag is required to signal the completion of the operation and/or the start of a subsequent operation. See Section 1.6.3.2, Continuous DMA, for more information.

User Action: Check your program to ensure that you have included a unique event flag for each buffer in every LIO$ENQUEUE routine argument.

LIO$_GBLACCESS, Access mode conflicts with global section

Explanation: The memory queue devices in both processes are set up with conflicting parameters. This can occur when the memory queue device in one process is set up for transfer (LIO$K_TRANSFER), but the memory queue device in a second process is not set up for transfer.

User Action: Check your programs to ensure that either the memory queue device in each process is set to transfer data, or that one memory queue device is set up to display data and the second memory queue device is set up to copy data. See Section 2.7.2.4, Setting Up a Memory Queue Device for Interprocess Communications, for more information.

LIO$_ILLBUFF, Illegal buffer address

Explanation: One of the following occurred:

- When using the AXV11-C device attached with LIO$K_CTI, you supplied a buffer not previously set up with the LIO$K_CTI_BUF parameter.
- When using the DRQ3B or the Preston/DRQ3B devices, you supplied a buffer larger than the LIO$K_BUFF_SIZE parameter.
- When using memory queue devices, you supplied a buffer not previously set up (preallocated) with the LIO$K_BUFF_SOURCE parameter.

User Action: Check your program to ensure that you have set up the device appropriately for the operations you intend to perform.

LIO$_ILLCHAN, A/D or D/A channel number out of range

Explanation: The LIO$K_AD_CHAN or LIO$K_DA_CHAN parameter values are inappropriate or out of range for the device.

User Action: Check your program to ensure that the values of the LIO$K_AD_CHAN or LIO$K_DA_CHAN parameter are appropriate for the device. See the reference descriptions of these parameters in Chapter 4 for the acceptable values of these parameters for each device.

LIO$_ILLDEVSPEC, Invalid device-specific argument

Explanation: The value contained in the **device_specific** argument is inappropriate for use with the operation.

User Action: See the device support description in Chapter 2 to verify the appropriate use and acceptable values of the **device_specific** argument.

LIO$_ILLFUNC, Device set up conflicts with I/O operation

Explanation: This status value is returned on the first LIO$ENQUEUE or LIO$READ routine call when the KWV11-C or Simpact RTC01 clock module is not set up for event timing.

User Action: Edit your program to set up the KWV11-C or Simpact RTC01 clock module for event timing. Use the LIO$K_FUNCTION parameter, specifying either LIO$K_EVENT_ABS or LIO$K_EVENT_REL as the parameter value.

LIO$_ILLGAIN, Gain number out of range

Explanation: A channel gain value you specified with the LIO$K_AD_GAIN parameter is out of range.

User Action: See the reference description of the LIO$K_AD_GAIN parameter for acceptable values of this parameter and the restrictions that apply.

LIO$_ILLID, Illegal device ID

Explanation: The **device_id** argument was not returned by the LIO$ATTACH routine.

User Action: Check your program to ensure that you attach a device using the LIO$ATTACH routine before you attempt to set up device characteristics. You must have an LIO-assigned device ID for a device before you can use the device.

If the LIO$ATTACH routine is the first routine call you make to the device, then check for spelling errors in the routine line. If you are programming in VAX C, ensure that all routine arguments are passed by reference.

LIO$_ILLSETUP, Device not set properly for I/O operation

Explanation: You used an illegal combination of set parameters.

User Action: See the reference description of the set routines and the set parameters to verify the acceptable values.

LIO$_ILLTRIG, Unknown trigger value

Explanation: A trigger value you specified with the LIO$K_TRIG parameter is unknown or not supported for the device.

User Action: See the reference description of the LIO$K_TRIG parameter to verify the acceptable values of this parameter for the device. Check your program to be sure you have included the LIOSET.ext file that defines the set parameter symbols for your programming language. Check your program to be sure the trigger value is entered correctly.

LIO$_ILLVAL, Illegal parameter values

Explanation: A parameter value you specified is unknown or not supported for the device.

User Action: See the reference description of the parameter to verify the acceptable values. Check your program to be sure you have included the LIOSET.ext file that defines the set parameter symbols for your programming language. Check your program to ensure that the parameter values are entered correctly.

LIO$_INSBUFHDR, Internal error, ran out of buffer header

Explanation: The LIO software detected an unrecoverable, inconsistent condition.

User Action: Submit a Software Performance Report (SPR) that describes the conditions leading to the error.

LIO$_INSFWS, Insufficient working set to lock user buffer

Explanation: The working set is too small to lock the specified buffer.

User Action: You should increase your working set by the size of the buffer. This may require assistance from the System Manager.

LIO$_INTERR, Internal software error

Explanation: The LIO software detected an unrecoverable, inconsistent condition.

User Action: Submit a Software Performance Report (SPR) that describes the conditions leading to the error.

LIO$_INV_ADDR, Invalid IEEE-488 address

Explanation: An IEEE-488 bus address either is out of range or does not match the address of any device currently on the IEEE-488 bus.

User Action: Check the IEEE-488 addresses of all devices currently on the IEEE-488 bus. Some of these can only be set manually.

LIO$_IOERROR, Error occurred during I/O

Explanation: The I/O hardware set its error bit. When using ADCs or DACs, the clock rate may be set too fast, or a condition known as "trigger slivering" has occurred.

User Action: Check your program to ensure that you have set up the clock rate and trigger sources appropriately for the device. Also, see Section 2.1.6, Using the KWV11-C to Avoid Trigger Slivering, to ensure that you have set up your device to avoid trigger slivering. This applies to AAV11-D, ADV11-D, and AXV11-C devices.

LIO$_MALFAIL, Unable to allocate memory

Explanation: A user account is not allowed to allocate as much memory as the LIO facility requires for internal data structures.

User Action: See your System Manager and ask about increasing your working set size.

LIO$_NAMTOOLONG, Device name too long

Explanation: A file or global section name is more than 131 or 43 characters in length, respectively.

User Action: Edit your program to shorten the file or global section name. If you are programming in FORTRAN, the CHARACTER*n declaration must have "n" no greater than 131 for disk files, and no greater than 43 for global sections.

LIO$_NIMP, Feature not implemented

Explanation: A feature, such as CTI support, was originally planned but is not yet implemented for the device.

User Action: See the device support section in Chapter 2 for information about the VAXlab-specific features supported for the device.

LIO$_NOASYNCH, Device not set for asynchronous I/O

Explanation: The current device is set for synchronous I/O, but your program is calling asynchronous (LIO$ENQUEUE and LIO$DEQUEUE) I/O routines.

User Action: Use the LIO$K_ASYNCH parameter to set the device for asynchronous I/O.

LIO$_NOCTI, Device not attached with CTI I/O

Explanation: Your program attempted to set up the CTI buffer without having been attached to use CTI I/O.

User Action: Edit the LIO$ATTACH routine call in your program to attach the device with CTI I/O.

LIO$_NODP, Data path has not been selected

Explanation: The data path for block mode transfers using the AAF01,[1] ADF01,[1] or DRQ11-C[1] device is not set up.

User Action: Use the LIO$K_DATA_PATH parameter to specify a direct data path using parameter value LIO$K_DIRPATH. A direct data path is the only valid data path for a MicroVAX II.

LIO$_NODRIVER, Driver not loaded

Explanation: The driver for a device is not connected to the device. This absence of the driver indicates that the device hardware is not installed at all, is not installed properly, or is not functioning properly.

[1] This device is available only in Europe.

User Action: If you are attempting to attach an ADQ32, a DRQ11-C,[1] an IAV11,[1] an IDV11,[1] or an IEQ11, be sure that the appropriate load command file has been executed. If you have purchased a service contract with your VAXlab system, reread it to determine your next course of action. If you have a Field Service Contract, call your local Field Service Office for assistance.

LIO$_NOENTRY, Buffer does not match a prelocked buffer

Explanation: A program attempted to unlock a buffer (LIO$K_UNLOCK_BUFFER) that was not previously locked (LIO$K_LOCK_BUFFER).

User Action: Check your program to ensure that there are no syntax or spelling errors. Be sure that the buffer you attempt to unlock was previously locked using the LIO$K_LOCK_BUFFER parameter.

LIO$_NOEVENT, No events enabled on this unit

Explanation: You called an LIO$K_EVENT_WAIT or LIO$K_EVENT_AST without first enabling specific events through the LIO$K_EVENT_ENA parameter.

User Action: Use the LIO$K_EVENT_ENA parameter to enable the recognition of specific events by the IEEE-488 device before attempting to wait for an event to occur (LIO$SHOW with the LIO$K_EVENT_WAIT parameter), or before specifying an AST routine to be called when an event occurs (LIO$SET_I with the LIO$K_EVENT_AST parameter).

LIO$_NOINPUT, Device not set for input

Explanation: A program attempted to read from an output device. If the device is a disk file or one of the parallel I/O devices, it is currently set for output.

User Action: If the device is a disk file or a parallel I/O device, use the LIO$K_DIRECTION parameter to set the device direction to input (LIO$K_INPUT).

[1] This device is available only in Europe.

If the device is an output-only device, such as a D/A converter, use the LIO$WRITE routine or the LIO$ENQUEUE routine to write to the device. If the device is the AXV11-C set for asynchronous I/O, you must specify the device-specific argument of the LIO$ENQUEUE routine as LIO$K_OUTPUT to signal that data is to be written to the D/A converter.

LIO$_NOINTERP, Device not set for interprocess I/O

Explanation: A memory queue device is not attached for interprocess I/O.

User Action: Specify the value of the **io_type** argument as LIO$K_INTER_PROC when you attach the memory queue device in each process. See Section 2.7.2.4, Setting Up a Memory Queue Device for Interprocess Communications, for information about setting up a memory queue device for interprocess communications.

LIO$_NOLB, No large buffer transaction in progress

Explanation: An attempt to clear a large buffer overflow condition for an AAF01,[1] ADF01,[1] or DRQ11-C[1] device proved that no large buffer transfer was in progress at the time.

User Action: Do one of the following: Correct the data transfer to indicate that a large buffer transfer is to take place by using the LIO$K_LARGEBUF value as part of the user-supplied parameter block in the **device_specific** argument for the LIO$ENQUEUE, LIO$READ, or LIO$WRITE routine. Or, remove the routine call that uses the LIO$K_CLR_LBO parameter to clear the large buffer overflow condition. It is not necessary for other types of data transfers.

LIO$_NOLOCAL, Device not set for local I/O

Explanation: A memory queue device is not attached to manage local memory.

User Action: Specify the value of the **io_type** argument as LIO$K_LOCAL when you attach the memory queue device. See Section 2.7.2.3, Using a Memory Queue Device to Manage Local Memory, for complete information about setting up a memory queue device to manage memory local to a user's process.

[1] This device is available only in Europe.

LIO$_NOMAP, Device not attached with mapped I/O

Explanation: Your program attempted to set up a device using a parameter that is valid only when the device is attached to use mapped I/O.

User Action: Edit your program to attach the device with another I/O type. Or, edit your program to set up the device using a parameter that is valid for the current I/O type. The nature of your application program determines which action is appropriate.

LIO$_NOMIX, Cannot mix LIO$K_EVENT_WAIT and LIO$K_EVENT_AST

Explanation: You set an event AST to be called on IEEE-488 device events, using the LIO$K_EVENT_AST parameter, and then called LIO$SHOW with the LIO$K_EVENT_WAIT parameter. This is an invalid combination of modes.

User Action: Reread the reference descriptions of the LIO$K_EVENT_AST and LIO$K_EVENT_WAIT parameters. These parameters perform different functions. The nature of your application program determines which function is appropriate. Be sure to use only one of these methods for detecting the occurrence of events.

LIO$_NOOUTPUT, Device not set for output

Explanation: A program attempted to write to an input device. If the device is a disk file or one of the parallel I/O devices, it is currently set for input.

User Action: If the device is a disk file or a parallel I/O device, use the LIO$K_DIRECTION parameter to set the device direction to output (LIO$K_OUTPUT).

If the device is an input-only device, such as an A/D converter, use the LIO$READ routine or the LIO$ENQUEUE routine to read from the device. If the device is the AXV11-C set for asynchronous I/O, you must specify the device-specific argument of the LIO$ENQUEUE routine as LIO$K_INPUT to signal that data is to be read (input) from the A/D converter.

LIO$_NOQIO, Device not attached with QIO

Explanation: Certain features, such as asynchronous I/O and buffer forwarding, are only available when a device is attached with QIOs (LIO$K_QIO).

User Action: Check your program to ensure that the set-up parameters are compatible for use with the **io_type** argument in the LIO$ATTACH routine and subsequent I/O routine calls.

LIO$_NORESET, Device has not been reset

Explanation: You did not reset the AAF01,[1] ADF01,[1] or DRQ11-C[1] device before a read or a write operation.

User Action: Use the LIO$K_RESET_DRX parameter to reset the device before issuing a read or a write request.

LIO$_NOROOM, Not enough room in file for operation

Explanation: A disk file set for output is not set up for extension and there is not sufficient space left in the file for a buffer. Or, the extension was not set large enough to accomodate the last write request.

User Action: Use the LIO$K_FILE_EXTENT parameter to set up the file for extension, and to specify the file extension size. Or, use the LIO$K_FILE_SIZE parameter to specify a larger file size.

LIO$_NOSHARE, Device is already owned by another process

Explanation: You attempted to attach a DRB32 device that is currently attached by another process.

User Action: Wait for the other process to detach the DRB32 device before you attempt to attach it again.

LIO$_NOSLOT, All prelocked buffer slots are currently in use

Explanation: A user attempted to lock more than 16 buffers using the LIO$K_BUFFER_LOCK parameter.

User Action: Use the LIO$K_UNLOCK_BUFFER parameter to unlock a locked buffer, thus freeing a slot for a new buffer. Or, reuse a previously locked buffer.

---

[1] This device is available only in Europe.

LIO$_NOSYNCH, Device not set for synchronous I/O

Explanation: The current device is set for asynchronous I/O, but your program is calling synchronous (LIO$READ and LIO$WRITE) I/O routines.

User Action: Use the LIO$K_SYNCH parameter to set the device for synchronous I/O.

LIO$_NOT_CIC, Unit is not controller-in-charge

Explanation: The requested operation cannot be performed because the IEEE-488 device is not currently the controller-in-charge.

User Action: Use LIO$ATTACH with LIO$K_SYS_CTRL when you initially attach the device. This gives the device both system controller and controller-in-charge status.

LIO$_NOTOPEN, Device not open

Explanation: Your program attempted to read from or write to a disk file that is not open.

User Action: Use the LIO$K_OPEN_FILE parameter to open the disk file.

LIO$_NO_TRANS, No translation for logical name

Explanation: A logical device name cannot be translated into a real device name.

User Action: Check your program to ensure that you have specified the correct logical name as the value of the **devspec** argument in the LIO$ATTACH routine call argument list. Check that the logical name is correctly assigned, and that there are no misspellings in either the logical name or the device name assignment.

LIO$_NOTREADY, Device in self test sequence or USER RDY line not deasserted

Explanation: (1) The DRB32 device is currently performing a self-test (which takes approximately 10 seconds to complete) or (2) the DRV11-J USER RDY line is not deasserted.

User Action: For the DRB32, retry the operation after 10 seconds. If this error is returned again, there may be a hardware problem.

For the DRV11-J, deassert USER RDY in the user device to indicate that the device is ready to send data to or receive data from the DRV11-J.

LIO$_NOTSETCDMA, Device not set up for continuous DMA

Explanation: The device is not set up to perform continuous DMA.

User Action: Use the LIO$K_CONT parameter to enable continuous DMA mode for the device.

LIO$_NOT_SETUP, Device not set up

Explanation: A device is not set up sufficiently to perform a specified I/O operation.

User Action: See the device support section in Chapter 2 for information about setting up the device. You may have omitted a required set parameter when you set up the device, such as not specifying file size (LIO$K_FILE_SIZE) when you set up a disk file.

LIO$_ONFREEQ, Buffer is on free queue

Explanation: A program attempted to enqueue a buffer to the memory queue device before the buffer has been dequeued from the free queue.

User Action: Use the LIO$DEQUEUE routine to dequeue the buffer from the free queue. Then, use the LIO$ENQUEUE routine to enqueue the buffer to the memory queue device.

LIO$_ONQ, Buffer is on user queue

Explanation: Your program attempted to enqueue a buffer that is currently on the device's user queue.

User Action: Use the LIO$DEQUEUE routine to dequeue the buffer from the device's user queue. Perform whatever processing may be required for the buffer. Then, enqueue the buffer again to the device's device queue.

LIO$_OVERRUN, Data overrun

Explanation: The user program is not supplying buffers to the device faster enough to keep up with the data flow.

User Action: See Section 1.6.3.4, Double-Buffer DMA, for information about supplying buffers to the ADQ32 device for double-buffer DMA data transfers.

LIO$_PAGEALIGN, Buffer must be page-aligned

Explanation: One of the following has occurred:

- The 64K-bytes block of memory allocated for continuous DMA is not page-aligned.
- The user array supplied for interprocess memory queue is not page-aligned.

User Action: You can page-align the 64K-byte block of memory by placing the block in a PSECT, and then using a linker options file to page-align it. See the language-specific reference manual for the programming language you are using for more information.

LIO$_POLL_STAT, Invalid parallel poll status byte specified for this unit

Explanation: A user-specified parallel poll status byte is out of range. This value range is 1 through 255.

User Action: Check your program to ensure that the value of the status byte is in the valid range.

LIO$_QIOCHAN, Must have A/D channels set up in ascending order

Explanation: The A/D channels are not set up sequentially in ascending order.

User Action: For ADV11-D and AXV11-C devices attached with LIO$K_QIO, the A/D channels must be specified sequentially in **ascending order**. Check the LIO$K_AD_CHAN parameter in your program and correct the channel specifications as necessary.

LIO$_QNEMP, Buffers on device or user queue

Explanation: Your program attempted to switch from using one I/O interface to using the other I/O interface while there are still buffers enqueued to a device.

User Action: Use the LIO$DEQUEUE routine to dequeue all buffers from the device and return them to the main program. Then, use the appropriate set parameter (LIO$K_ASYNCH or LIO$K_SYNCH) to change the I/O interface.

LIO$_REMOTE_DEV, Cannot access device on remote node

Explanation: Your program attempted to attach a device that is installed in a remote node. This occurs if you include a node name (NODE::devspec) in the **devspec** argument of the LIO$ATTACH routine. You can only attach devices installed on your local node.

User Action: Edit your program to remove the node name from the **devspec** argument. Or, copy your program to the remote node and execute it locally there.

LIO$_REQ64K, Buffers must sum to 64K bytes

Explanation: The total number of bytes in the declared buffers for continuous DMA does not equal 64K bytes.

User Action: Declare a two-dimensional array 64K bytes in size. For example, in FORTRAN a 64K-byte block of four buffers can be defined as an 8192 X 4 array of word (2-byte) integers.

LIO$_RUNNING, Illegal operation when device is running

Explanation: A program attempted to set up a device characteristic that cannot be set up while the device is running.

User Action: Edit the program to set up the device characteristic before the device starts running. Or, use the LIO$K_STOP parameter to stop the device, then set up the characteristic, and restart the device using the LIO$K_START parameter. The nature of your application and the characteristic you want to set up determine which action is appropriate.

LIO$_SS_INTERR, LIO system service internal error

Explanation: The LIO software detected an unrecoverable, inconsistent condition.

User Action: Submit a Software Performance Report (SPR) that describes the conditions leading to the error.

LIO$_SUCCESS, Success

Explanation: The operation was successful.

User Action: None.

LIO$_TERM_CHAR, Buffer terminated due to match character detection

Explanation: A buffer transfer to or from an IEEE-488 bus device was terminated on receipt of a termination character or a count previously set up by the user program.

User Action: This is an informational message only. No user action is required.

LIO$_TERM_EOI, Buffer terminated due to EOI assertion

Explanation: An IEEE-488 device sending a data buffer terminated the data transfer by asserting the EOI (end-or-identify) line.

User Action: This is an informational message only. No user action is required.

LIO$_TERM_ERR, Buffer terminated due to an IFC or ERR interrupt

Explanation: The IEEE-488 system controller issued an interface clear (IFC) command, or an error interrupt occurred on the IEEE-488 bus during a data transfer.

User Action: Design your program to reissue the I/O request to obtain the rest of the data. Or, set the device so that it does not terminate data transfer on receipt of an IFC or ERR interrupt.

LIO$_TERM_SRQ, Buffer terminated due to SRQ

Explanation: Another IEEE-488 bus device issued a service request (SRQ) on the bus while a data transfer was in progress.

User Action: Design your program to reissue the I/O request to obtain the rest of the data. Or, set the device so that it does not terminate data transfer on receipt of an SRQ. See the reference description of the LIO$K_AUX_COMMAND parameter in Chapter 4 for more information.

LIO$_TOOFEWARGS, Insufficient arguments

Explanation: Too few arguments were included in a routine call argument list. Most of the LIO routine calls contain optional arguments in the argument list. If you omit an optional argument from a routine call argument list, you must use the convention specific to your programming language to account for the argument.

User Action: Check your program to ensure that all arguments in the routine call argument list are either specifically included, or are defaulted appropriately. See Chapter 3 in *Getting Started with VAXlab* for information about how to default routine call arguments.

LIO$_TOOFEWVALS, Too few set values

Explanation: Your program did not specify the required number of parameter values for an LIO$SET_x parameter.

User Action: See the reference description of the parameter in Chapter 4 for information about the required number of parameter values.

LIO$_TOOMANYPROCS, Too many processes mapped to the global section

Explanation: A memory queue device attached for interprocess data transactions supports only two processes mapped to the global section.

User Action: Both memory queue devices can be set up for interprocess data transfers (LIO$K_TRANSFER). Or, one memory queue device can be set up to display data (LIO$K_DISPLAY) to the other memory queue device that is set up to read the data (LIO$K_READ_ONLY).

LIO$_TOOMANYVALS, Too many set values

Explanation: Your program specified greater than the maximum allowable number of parameter values for an LIO$SET_x parameter.

User Action: See the reference description of the parameter in Chapter 4 for information about the required number of parameter values.

LIO$_UNKDEV, Unknown device

Explanation: The LIO$ATTACH routine did not recognize a **devspec** value.

User Action: See the reference description of the LIO$ATTACH routine, and check your program to be sure you used the appropriate value for the **devspec** argument. You can also use the SHOW DEVICES command to see if the device to which you are trying to attach is installed in your VAXlab system.

LIO$_UNKPARAM, Unknown set or show parameter

Explanation: Your program attempted to set up a device with a parameter that is unrecognized by the system, or that is not supported for use with the device.

User Action: See the device support description in Chapter 2 for the parameters appropriate for use with the device. Also, make sure you have entered the parameter name correctly.

LIO$_VALTOOBIG, Set parameter value too large

Explanation: A specified parameter value is greater than the maximum allowable value for the parameter.

User Action: See the reference description of the parameter in Chapter 4 for the acceptable values for this parameter.

LIO$_VALTOOSMALL, Set parameter value too small

Explanation: A specified parameter value is less than the minimum allowable value for the parameter.

User Action: See the reference description of the parameter in Chapter 4 for the acceptable values for this parameter.

LIO$_WORDALIGN, Buffer must be word-aligned

Explanation: Data buffers are not word-aligned. When using the AAV11-D, the ADV11-D, the DRQ3B, and the Preston I/O subsystem interfaced through the DRQ3B, data buffers must be word-aligned.

User Action: Declare data buffers as word (INTEGER*2) arrays.

SYSTEM-F-EXQUOTA, exceeded quota

Explanation: An image could not continue executing because the process exceeded one of its resource quotas or limits. This system condition is signalled during an LIO$ENQUEUE, LIO$READ, or LIO$WRITE routine call to a non-DMA device attached with LIO$K_QIO if the requested buffer is larger than the the user process's quota or is larger than the maximum size for a buffered I/O transfer (MAXBUF) of the system.

User Action: Run the AUTHORIZE utility and SHOW USERNAME to determine the byte limit (BYTLM) quota.

Chapter 6

Online Sample Programs

This chapter provides an overview of sample VSL application programs showing how to use the Laboratory I/O routines. These programs are shipped with your VAXlab software kit and are put on-line during the VAXlab software installation procedure in a directory with the logical name LIO$EXAMPLES. The logical name of this directory is defined at installation time by a command in one of the VAXlab startup command files.

The sample program source file names incorporate the facility code, LIO, and in most cases, an abbreviation of the device the program uses and a keyword describing the function the program performs. The sample program source file name extensions indicate the programming language in which each sample program is written. For example, the program LIO_AXV_MAPPED.BAS uses the AXV11-C device to perform memory-mapped I/O, and is written in VAX BASIC.

Table 6-1, LIO Online Sample Programs, does the following:

- Lists the sample program names
- Lists the devices used in each program
- Lists the LIO set parameters used to set up the devices
- Describes briefly what each program does

The table does not list the LIO routines each program uses because the LIO facility is designed with the following general rules for using the routines:

- All application programs using the LIO routines must use the LIO$ATTACH and LIO$DETACH routines to attach and detach devices supported by LIO.
- To set up LIO devices, you use the LIO parameters in conjunction with the LIOSET_I, LIOSET_R, and LIO$SET_S routines. The data type of a parameter determines which of the LIO$SET_X routines you use to set up that parameter.
- LIO devices set up to use the synchronous I/O interface must use the LIO$READ and/or LIO$WRITE routines to perform data transfers.
- LIO devices set up to use the asynchronous I/O interface must use the LIO$ENQUEUE and one or more of the asynchronous synchronization methods, such as an AST routine, buffer forwarding, or the LIO$DEQUEUE routine, to receive completed buffers.

Study Table 6-1 to determine which of the sample programs will be helpful to you in learning how to use the LIO routines. Then, copy the programs to your own directory.

To copy a sample program, in this case LIO_AXV_MAPPED.BAS, to your directory, enter the following command line:

```
$ COPY LIO$EXAMPLES:LIO_AXV_MAPPED.BAS *.* Return
```

Once a sample program is successfully copied to your directory, you can read it, print it, or edit it for your own purposes.

6.1 Programs for European Devices

Online sample programs LIO_AAFBIG.C, LIO_AAF_SINGLE.C, LIO_AAF_DOUBLE.C, and LIO_AAF_SEL_OUT.C require the use of a special include file. This include file, AAFDEF.H, is located in SYS$EXAMPLES. Be sure to copy AAFDEF.H to your directory before you run any of the programs listed above.

The online sample program LIO_AAF_RW_ACS.C requires the use of a special include file. This include file, AXFOFF.H, is located in SYS$EXAMPLES. Be sure to copy AXFOFF.H to your directory before you run LIO_AAF_RW_ACS.C.

To determine the exact logical device names of the devices configured in your system, see your system manager. You can assign, or define, these logical names interactively before you begin to run LIO_ADF sample programs or LIO_AAF sample programs, for example:

```
$ DEFINE UA UUA0:
$ DEFINE UB UUB0:
```

Or, you can include these command lines in your LOGIN.COM file. Then, each time you log in to the system, these logical name assignments are made automatically. Once you assign these logical names, you are ready to execute the sample programs.

Table 6–1: LIO Online Sample Programs

Devices	Parameters	Description
	LIO_AAFBIG.C[2]	
AAF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_CLR_LBO LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_ED_CTT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the large buffer transfer capability of the AAF01 D/A device.
	LIO_AAF_CALIB.C	
AAF01[1] DRQ11-C[1]	LIO$K_ANA_OUT LIO$K_DATA_PATH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DRX_STAT LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_SYNCH LIO$K_TIMEOUT	Uses the synchronous I/O interface and QIOs to test the self-calibration feature of the AAF01 D/A device.

[1]This device is available only in Europe.

[2]This sample program requires include file AAFDEF.H, which contains the high/low limit definitions for the AAF01 device.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO$K_AAF_DOUBLE.C[2]	
AAF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_CAL_CC LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DRX_STAT LIO$K_EVENT_AST LIO$K_READ_STAT LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the alternate buffer transfer capability of the AAF01 D/A device.
	LIO_AAF_RW_ACS.C[3]	
AAF01[1] DRQ11-C[1]	LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DRX_STAT LIO$K_FUNCTION_BITS LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_SYNCH LIO$K_TIMEOUT	Uses the synchronous I/O interface and QIOs to perform an ACS read/write test using the AAF01 D/A device.

[1]This device is available only in Europe.

[2]This sample program requires include file AAFDEF.H, which contains the high/low limit definitions for the AAF01 device.

[3]This sample program requires include file AXFOFF.H, which contains the offset definitions for AXF01 devices.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_AAF_SEL_OUT.C[2]	
AAF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_CAL_CC LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DRX_STAT LIO$K_EVENT_AST LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to perform selectable output using AAF01 D/A device.
	LIO_AAF_SINGLE.C[2]	
AAF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_CAL_CC LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DRX_STAT LIO$K_EVENT_AST LIO$K_READ_STAT LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the single buffer transfer capability of the AAF01 D/A device.

[1]This device is available only in Europe.

[2]This sample program requires include file AAFDEF.H, which contains the high/low limit definitions for the AAF01 device.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_ADFBIG.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_CLR_LBO LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_ED_CTT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_READ_STAT LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the large buffer transfer capability of the ADF01 A/D device.
	LIO_ADF_CALIB.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_REA_ADC LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to perform an ADF01 DAC calibration.

[1]This device is available only in Europe.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_ADF_DAC_CALIB.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_REA_ADC LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to perform an ADF01 DAC calibration.
	LIO$K_ADF_DOUBLE.C	
ADF01[1] DRQ11-C[1]	LIO$K_AST_RTN LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_ED_CTT LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the alternate buffer transfer capability of the ADF01 A/D device.

[1]This device is available only in Europe.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_ADF_DOUBLE_AST.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNC LIO$K_DATA_PATH LIO$K_DRX_AST_RTN LIO$K_ED_CTT LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Performs an alternate buffer transfer using a DRX AST routine.
	LIO_ADF_DOUBLE_SAST.C	
ADF01[1] DRQ11-C[1]	LIO$K_AST_RTN LIO$K_ASYNC LIO$K_DATA_PATH LIO$K_ED_CTT LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Performs an alternate buffer transfer using a standard AST routine.

[1]This device is available only in Europe.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_ADF_LOOPBACK.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_BAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to perform an ADF01 DAC/ADC loop test.
	LIO$K_ADF_SINGLE.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_ED_CTT LIO$K_EVENT_AST LIO$K_READ_STAT LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_RW_CWT LIO$K_SET_DAC LIO$K_TIMEOUT LIO$K_WRITE_CTA LIO$K_WRITE_PCR	Uses the asynchronous I/O interface and QIOs to test the single buffer transfer capability of the ADF01 A/D device.

[1]This device is available only in Europe.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_ADF_TEST_SEQ.C	
ADF01[1] DRQ11-C[1]	LIO$K_ASYNCH LIO$K_DATA_PATH LIO$K_DEVICE_EF LIO$K_DIS_STE LIO$K_RESET_AXF LIO$K_RESET_DRX LIO$K_TIMEOUT LIO$K_WRITE_ST0_1	Uses the asynchronous I/O interface and QIOs to perform an ADF01 sequence timer test.
	LIO_ADQ_ASYNCH.FOR	
ADQ32	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_BUFF_SIZE LIO$K_CLK_RATE LIO$K_DBL_BUF LIO$K_DIAG_CHAN LIO$K_TRIG	Uses the asynchronous I/O interface and double buffering to read 10 buffers of data from the ADQ32 device.
	LIO_ADQ_SYNCH.FOR	
ADQ32	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_BUFF_SIZE LIO$K_DIAG_CHAN LIO$K_SYNCH LIO$K_TRIG	Uses the synchronous I/O interface to read 12 A/D values (24 bytes) from the ADQ32 device.

[1]This device is available only in Europe.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
LIO_ADV_AST.BAS, LIO_ADV_AST.C, LIO_ADV_AST.FOR, LIO_ADV_AST.PAS		
ADV11-D	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_AST_RTN LIO$K_ASYNCH LIO$K_SGL_BUF LIO$K_TRIG	Shows how to use the asynchronous I/O interface and single-buffer DMA using an AST routine, instead of the LIO$DEQUEUE routine, to receive and process completed buffers from the ADV11-D device.
LIO_ASYNCH_CLK_TRIG.FOR		
AXV11-C KWV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_CLK_RATE LIO$K_FUNCTION LIO$K_TRIG	Shows how to use the asynchronous I/O interface to control the start and stop of the KWV11-C clock device to prevent trigger slivering. The program enqueues a data buffer to the AXV11-C device and starts the clock. Then, the program waits for the buffer to complete, dequeues it from the device, and stops the clock.
LIO_AXV_CTI.FOR		
AXV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_CTI_BUF LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and connect-to-interrupt I/O to read 20 analog-to-digital values from the AXV11-C device.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_AXV_DIRECTION.FOR	
AXV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_DA_CHAN LIO$K_TRIG	Shows how to use the asynchronous I/O interface to read 20 analog-to-digital values from the AXV11-C A/D and write them to the AXV11-C D/A. Specifically, this program shows the use of the **device_specific** argument of the LIO$ENQUEUE routine to specify the direction of asynchronous routine calls to the AXV11-C.
	LIO_AXV_MAPPED.BAS, LIO_AXV_MAPPED.C, LIO_AXV_MAPPED.FOR, LIO_AXV_MAPPED.PAS	
AXV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and memory mapped (polled) I/O to read 20 analog-to-digital values from the AXV11-C device.
	LIO_AXV_QIO.FOR	
AXV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and QIOs to read 20 analog-to-digital values from the AXV11-C device.
	LIO_AXV_RTPLOT.FOR	
AXV11-C R/T plotting	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_N_BUFFS LIO$K_PO_CHAN LIO$K_START LIO$K_SYNCH LIO$K_TITLE_N LIO$K_X_RANGE LIO$K_Y_MAX LIO$K_Y_MIN	Uses the synchronous I/O interface and QIOs to read 360 analog-to-digital values from each of the two AXV11-C A/D channels and plots them on the terminal screen using the LIO plotting device in a continuous real-time display. This sample program only runs on a VAXstation-based VAXlab system.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_BUF_FWD.FOR	
ADV11-D Disk file	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_DEVICE_EF LIO$K_DIRECTION LIO$K_FILE_SIZE LIO$K_FORWARD LIO$K_NAME LIO$K_OPEN_FILE LIO$K_SGL_BUF LIO$K_TRIG	Shows how to use the asynchronous I/O interface and single-buffer DMA with buffer forwarding to read analog-to-digital values from the ADV11-D device and forward the data buffers to a disk file.
	LIO_BUF_INX.FOR	
ADV11-D	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_SGL_BUF LIO$K_TRIG	Shows how to use the asynchronous I/O interface and single-buffer DMA with buffer indexing to read analog-to-digital values from the ADV11-D device.
	LIO_CONT_DMA.FOR	
ADV11-D	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_ASYNCH LIO$K_CONT LIO$K_TRIG	Shows how to use the asynchronous I/O interface and continuous DMA to read data values from the ADV11-D device.
	LIO_DRJ_SETUP.FOR	
DRV11-J	LIO$K_DIRECTION LIO$K_HANDSHAKE LIO$K_POLARITY	Shows how to set up the DRV11-J device. Specifically, this program sets ports A and B for input, ports C and D for output, and disables handshaking. Then, the program shows the values of the set parameters and the polarity of the device for which the default value is used.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_DRQ3B_LOOP.FOR	
DRQ3B	LIO$K_ASYNCH LIO$K_BUFF_SIZE LIO$K_ERR_HANDLE	Demonstrates how to use the DRQ3B with the asynchronous I/O interface. Using a loopback cable, the program transfers 1024 words of data from HXA1 to HXA0.
	LIO_DRV_LOOP.PAS, LIO_DRV11J_LOOP.FOR	
DRV11-J	LIO$K_DIRECTION LIO$K_HANDSHAKE LIO$K_SYNCH	Uses the synchronous I/O interface to perform a loopback test. Ports A and B are set for input and ports C and D are set for output. Handshaking is disabled.
	LIO_FILE_POS.FOR	
Disk file	LIO$K_DIRECTION LIO$K_FILE_POS LIO$K_FILE_SIZE LIO$K_NAME LIO$K_OPEN_FILE LIO$K_SYNCH	Uses LIO$K_FILE_POS to set a file pointer to a given block within the file.
	LIO_FILTER_EVENT.FOR	
AXV11-C KWV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_CLK_SRC LIO$K_FUNCTION LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and the KWV11-C clock device to filter external events. Specifically, the KWV11-C clock device is used to enable and disable external events to protect the AXV11-C A/D from trigger slivering.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_HX_EXAMPLE.C	
DRQ3B	LIO$K_AST_RTN LIO$K_BUFF_SIZE LIO$K_STOP	Shows how to do continuous DMA using the DRQ3B.
	LIO_IEEE_LOOP.FOR	
IEQ11	LIO$K_ASYNCH LIO$K_EOI LIO$K_IEEE_ADDR LIO$K_SYNCH	Attaches IXA0 as system controller at IEEE-488 bus address 0, and IXA1 as an instrument at IEEE-488 bus address 1. The system controller addresses the instrument as a listener and outputs a string of characters to it. The IX instrument reads the characters and displays them on the screen. This program requires a loopback cable between IXA0 and IXA1.
	LIO_IEX_ASYNC.C	
IEQ11	LIO$K_AST_RTN LIO$K_ASYNCH LIO$K_AUX_COMMAND LIO$K_COMMAND LIO$K_IEEE_ADDR LIO$K_SYNCH LIO$K_TERM_CHAR LIO$K_TIMEOUT	Attaches an IEQ11 device as the system controller and uses the asynchronous I/O interface to read data from the HP 3455A Digital Voltmeter.
	LIO_IEX_SYNC.C	
IEQ11	LIO$K_AUX_COMMAND LIO$K_COMMAND LIO$K_IEEE_ADDR LIO$K_SYNCH LIO$K_TERM_CHAR LIO$K_TIMEOUT	Attaches an IEQ11 device as the system controller and uses the synchronous I/O interface to read data from the HP 3455A Digital Voltmeter.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_IEZ_SYNC.C	
IEZ11	LIO$K_AUX_COMMAND LIO$K_COMMAND LIO$K_IEEE_ADDR LIO$K_SYNCH LIO$K_TERM_CHAR LIO$K_TIMEOUT	Attaches an IEZ11 device as the system controller and uses the synchronous I/O interface to read data from the HP 3455A Digital Voltmeter.
	LIO_KWV_AST.FOR	
KWV11-C	LIO$K_ASYNCH LIO$K_CLK_RATE LIO$K_EVENT_AST LIO$K_FUNCTION LIO$K_TRIG	Shows how to use the asynchronous I/O interface and an event AST routine to illustrate the ability of the KWV11-C clock device to call an AST routine on every clock tick.
	LIO_MQ_DISPLAY.FOR	
Memory queue	LIO$K_BUFF_SIZE LIO$K_BUFF_SOURCE LIO$K_DEVICE_EF LIO$K_DISPLAY_ONLY LIO$K_NAME LIO$K_N_BUFFS	Shows how to use the display function of the interprocess memory queue to display data buffers acquired by one memory queue device to a memory queue device running in a second process that is set up to read the displayed buffers.
	LIO_MQ_READONLY.FOR	
Memory queue	LIO$K_BUFF_SIZE LIO$K_BUFF_SOURCE LIO$K_DEVICE_EF LIO$K_NAME LIO$K_N_BUFFS LIO$K_READ_ONLY LIO$K_SYNCH	Shows how to use the read-only function of the interprocess memory queue to read data buffers acquired by a display-only memory queue device.

Table 6-1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_MQ_XFER.FOR	
Memory queue	LIO$K_BUFF_SIZE LIO$K_BUFF_SOURCE LIO$K_DEVICE_EF LIO$K_NAME LIO$K_N_BUFFS	Shows how to use the transfer function of the interprocess memory queue to transfer data buffers from one memory queue device to a second memory queue device running in a second process.
	LIO_PRESTON_AST_PLOT.C	
Preston (DRQ3B)	LIO$K_AST_RTN LIO$K_CLK_RATE LIO$K_UPDATE	Uses the asynchronous I/O interface to read buffers from the Preston A/D. The program includes an AST routine that receives completed buffers from the device and plots them in a continuous real-time display using several of the Laboratory Graphics Package routines.
	LIO_PRESTON_READ.C	
Preston	LIO$K_AD_CHAN LIO$K_BURST_RATE LIO$K_CLK_BASE LIO$K_CLK_RATE LIO$K_N_AD_CHAN LIO$K_UPDATE	Shows how to set up and then display Preston device characteristics. Specifically, this program sets up and then displays the number of A/D channels, the clock rate, the clock base frequency, and the burst rate.
	LIO_RTC01_COUNTER.FOR	
Simpact RTC01	LIO$K_ASYNCH LIO$K_CLK_RATE LIO$K_COUNTER LIO$K_FUNCTION LIO$K_START LIO$K_TRIG	Shows how to use the Simpact RTC01 clock device as a counter and how to read the clock register while the clock is running. The program uses the asynchronous I/O interface, the repeat count function, and a software start.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_RTC01_SET.FOR	
Simpact RTC01	LIO$K_INTERRUPT_LEVEL LIO$K_SCHMITT_TRIGGER	Shows how to set the Schmitt trigger mode and interrupt level for the Simpact RTC01 clock device.
	LIO_SERIAL.C	
Serial line	LIO$K_DEVICE_ACK_NAK_BUFF LIO$K_PROTOCOL LIO$K_USER_READ_PROTOCOL_AST LIO$K_USER_WRITE_NAK_HANDLING	Demonstrates the user-defined protocol feature of LIO serial devices. The program enables the protocol feature and sends a buffer to a serial line device. The program ends execution when the receiving device acknowledges (ACKs) receipt of the buffer.
	LIO_SGLBUF_DMA.FOR	
AAV11-D ADV11-D	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_DA_CHAN LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and single-buffer DMA to read 20 analog-to-digital values from the ADV11-D device using QIOs, and then write the values to the digital-to-analog converter on the AAV11-D device using QIOs. This example also shows the use of the buffer overrun area required by single-buffer DMA on both the AAV11-D and ADV11-D devices.
	LIO_SYNCH_CLK_TRIG.FOR	
AXV11-C KWV11-C	LIO$K_AD_CHAN LIO$K_AD_GAIN LIO$K_CLK_RATE LIO$K_FUNCTION LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface to control the start and stop of the KWV11-C clock device to prevent trigger slivering. The analog-to-digital converter on the AXV11-C device reads one channel on each KWV11-C clock tick and continues cycling through the channels until the buffer is full.

Table 6–1 (Cont.): LIO Online Sample Programs

Devices	Parameters	Description
	LIO_TIME_EVENT.FOR	
KWV11-C	LIO$K_CLK_SRC LIO$K_FUNCTION LIO$K_SYNCH LIO$K_TRIG	Shows how to use the synchronous I/O interface and the KWV11-C clock device to time external events.
	LIO_UQ_LOOP.C	
DRB32	LIO$K_INPUT LIO$K_OUTPUT LIO$K_DATA_WIDTH	Uses the asynchronous I/O interface to transfer data between two DRB32 devices connected in loopback.

Appendix A

ADQ32 Triggering and Clock Modes

This appendix describes the clock modes available for use on the ADQ32.

A.1 Clock Mode Summary

The ADQ32 supports the following clock modes:

1. Burst
2. Burst with edge gate
3. Burst with delayed edge gate
4. Burst, activated by external trigger
5. Timed triggers
6. Timed triggers with edge gate
7. Timed triggers with delayed edge gate
8. Timed triggers with level gate
9. Timed trigger activated by external trigger
10. Burst sweep
11. Burst sweep, with edge gate
12. Burst sweep, with level gate
13. Burst sweep, activated by external trigger
14. Burst sweep, sweep controlled by external trigger
15. Timed sweep
16. Timed sweep, with edge gate
17. Timed sweep, with level gate

18. Timed sweep, activated by external trigger
19. Timed sweep, sweep controlled by external trigger
20. External trigger
21. External trigger, with edge gate
22. External trigger, with delayed edge gate

A.2 Definition of Terms Used to Describe Clock Modes

The term "ADC" refers to the analog-to-digital converter on the ADQ32. For a complete description of how the ADQ32 converts a voltage into a digital value, refer to the *ADQ32 A/D Converter Module User's Guide.*

The term "burst" refers to the rate used by the ADQ32 state machine. The timing is actually done by the state machine, not the clock chip. The clock chip is still used, in some modes, to provide gating, triggering, and sweep functions, but each sample is triggered at a rate determined by the state machine. The actual rates used depend upon the gain selected, and whether the gain is changed between samples. Table A–1 shows the rates that are actually used.

Table A–1: Burst Rates

Gain	Throughput
1	166 KHz
2	115 KHz
4	135 KHz
8	135 KHz
Changing Gains	100 KHz

The term "timed triggering" refers to the pulses generated by the clock and routed to the ADC to serve as triggers. The clock rate used is selected with the LIO$K_CLK_RATE parameter. This clock is called the "primary clock." The primary clock rate is always shown in the following diagrams as t_1. When each pulse from the clock output occurs, the ADC acquires the signal and converts it.

The term "sweep" refers to modes in which two timing sources are used.

The first timing source controls the rate at which individual samples are acquired. This primary clock rate is selected with the LIO$K_CLK_RATE parameter.

A second timing source controls the rate at which one pass through the specified number of channels is made. This clock rate is selected using the LIO$K_SWEEP_RATE parameter. This clock is called the "sweep clock." The sweep clock rate is always shown in the following diagrams as t_2.

The term "burst sweep" refers to clock modes where each sample in a sweep is acquired at the burst rate of the ADQ32 state machine. Each pass of the sweep is controlled either by the sweep clock or by an external trigger. This discussion uses the term "burst sweep, sweep controlled by external trigger" for the latter case.

The term "timed sweep" refers to clock modes where each sample in a sweep is acquired at a rate specified to the primary clock. Each pass of the sweep is controlled either by a second rate specified to the sweep clock or by an external trigger. This discussion uses the term "timed sweep, sweep controlled by external trigger" for the latter case.

The burst sweep modes are essentially the same as the timed sweeps, except that for timed sweeps, you must specify the rate at which samples within a sweep are taken.

The term "gate" refers to an external signal used to control the ADQ32 clock in some way. The gate signal is always connected to the external gate/trigger input.

An "edge gate" refers to a signal which acts as an ON/OFF switch for the trigger mode. A negative transition pulse on an edge gate signal turns on the clock mode. A subsequent negative transition turns off the clock mode.

A "level gate" also acts as an ON/OFF switch. With a level gate, the clock mode begins whenever the signal on the external gate/trigger input is HIGH. (TTL convention is used. HIGH is a signal greater than 2.0 volts. LOW is a signal lower than 0.5 volts.) This means that if the gate input is HIGH when the clock mode is programmed, data collection starts immediately. The clock mode continues until the signal falls to the LOW state. The gate input can remain HIGH for as long as is desired.

The term "delayed edge gate" is essentially the same as the edge gate, except that the ON or OFF function does not occur immediately when the negative transition is detected. The clock mode is turned ON or OFF after a short delay time. The delay time is one clock tick of the sweep clock.

You select a clock rate and specify it using the LIO$K_SWEEP_RATE parameter. One clock tick of this rate is used as the delay time. Delayed edge gates are not available in sweep modes.

The term "external trigger" refers to an external signal used to notify the ADQ32 to take samples. In some clock modes the triggering signal is connected to the external gate/trigger input. In modes where a gating signal is used with an external trigger, the external trigger signal is connected to the external frequency input and the gate signal is connected to the external gate/trigger input.

In the simplest case, a negative transition on the external trigger input causes a sample to be acquired. The external trigger can also be gated by an edge gate or a delayed edge gate. In this mode, however, the actual triggering signal is connected to the external frequency input and the gating signal is connected to the external gate/trigger input. The gate then controls when the external frequency input is routed to the ADC.

In sweep modes, the term "activated by external trigger" refers to a signal which has a one-time-only ON function. The clock mode does not begin until a negative transition is detected on the external gate/trigger input. In other words, the mode is similar to an edge gate signal, except that subsequent pulses on the external gate/trigger input generate a clock overrun error (modes 4 and 9) or are ignored (modes 13 and 18).

The "activated by external trigger" modes perform slightly differently depending on whether the ADQ32 is attached for single-buffer DMA or double-buffer DMA. In single buffer DMA, each external trigger starts data acquisition and transfer of a single buffer's worth of DMA. When the buffer completes, the next buffer is not started until the next external trigger occurs. In double-buffer DMA, each external trigger starts data acquisition and transfer. This continues as long the ADQ32 remains double-buffered. In other words, transitions from one DMA buffer to the next do not affect the ADQ32 ADC. The ADQ32 continues to acquire data and place it in the data FIFO.

In sweep modes, the term "sweep controlled by external trigger" refers to modes in which each pass through a sweep of the specified channels is controlled by an external signal connected to the external gate/trigger input. A negative transition on this input causes the ADQ32 to acquire the number of specified samples at the rate specified (if a timed sweep) or as fast as possible (if a burst sweep). The next negative transition causes another pass through the specified channels. However, if a subsequent negative pulse occurs on the external gate/trigger input before the ADQ32 has acquired the specified number of channels, a clock overrun error occurs.

NOTE

There is no requirement that any external trigger be a regularly timed pulse.

The term "clock tick" refers to pulses issued by the clock logic to control either data acquisition or sweeps through a specified number of conversions. The number of clock ticks in a second is usually given as the clock "rate" or clock "frequency." A clock rate of 50 Hz provides 50 clock ticks per second, or one clock tick every 20 milliseconds.

A.3 Channel Specification

The channels to be sampled are specified with the LIO$K_AD_CHAN set parameter. For the ADQ32, channels can be sampled in any order. For example,

```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 3, 5, 2, 9)
```

specifies channels 5, 2, and 9. They are sampled in that order.

Channels can be specified more than once. For example,

```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 5, 1, 1, 1, 1, 1)
```

specifies channel 1 five times. This might be useful in a sweep mode where you want to take several samples on a channel and do this at specified time intervals. For example, if you want to take 50 samples on channel 1 at the burst rate, and do this every second, you can select the burst sweep mode (mode 10) and specify channel 1 50 times in the LIO$K_AD_CHAN set parameter. The flexibility of the LIO$K_AD_CHAN parameter allows it to be used in scenarios other than the straightforward multiple channel scenario.

The maximum specifiable channel is 64.

A.4 Gain Specification

Every sample in the ADQ32 can be acquired at a specified gain. Amplification gains of 1, 2, 4, and 8 are available. The gain is specified the LIO$K_GAIN set parameter. The order of gain specifications in this call must parallel the channel specifications in the LIO$K_AD_CHAN set parameter. For example,

```
status = LIO$SET_I (adq_id, LIO$K_AD_CHAN, 4, 0, 3, 5, 7)
status = LIO$SET_I (adq_id, LIO$K_GAIN, 4, 1, 1, 2, 8)
```

sets unity gain on channels 0 and 3, a gain of 2 on channel 5, and a gain of 8 on channel 7.

A.5 Buffer Specification

Data from the ADQ32 is transferred into your user buffer. The buffer address and size is specified in the LIO$ENQUEUE or LIO$READ call.

The ADQ32 uses single buffer DMA transfers by default. Double buffering can be selected by specifying LIO$K_DBL_BUF in an LIO$SET call.

A.5.1 Single Buffer Transfers

In single buffer transfers, the ADQ32 collects data until the buffer is filled. The event flag or AST routine then is set or executes. The ADQ32 ADC becomes disabled and the ADC FIFO is cleared. The clock is restarted in its current mode for subsequent buffers.

For example in single buffer mode, clock modes that are "activated by an external trigger" begin data acquisition when the external trigger occurs. Data acquistion stops when the buffer is filled. The FIFO is cleared, the ADC is disabled, and the clock stops. When the next external trigger occurs, the clock is restarted, the ADC is enabled, and data collection begins filling the next buffer.

A.5.2 Double Buffer Transfers

In double buffer mode, the ADQ32 continues to acquire and convert samples without regard to buffer transitions. Some thought should be given to the buffer sizes selected. It is possible for the first part of a sweep's data to be in the end of one buffer and the rest of the sweep to be at the beginning of the next buffer. It is your responsibility to keep track of what data values are in which buffer.

In double buffered mode, external triggers activate buffers differently than in single buffer mode. In single buffer mode, clock modes that are "activated by an external trigger" begin data acquisition when the external trigger occurs. Data acquistion stops when the buffer is filled. In double buffer mode, the external trigger starts data acquisition, but when the first buffer is filled, the ADQ32 starts filling the second buffer. The event flag or AST routine signaling the filling of the first buffer does occur, but the ADC is not disabled, nor is the ADC FIFO cleared. The ADQ32 just continues to fill all subsequent buffers.

If the ADQ32 becomes non-double buffered when set for double buffered mode, data acquisition stops. The ADQ32 could become non-double buffered if buffer sizes are small and the clock rate is fast. If the time it takes to set up for the third buffer is greater than the entire time to transfer the second buffer, the ADQ32 will lose double buffered status. See Section 1.6.3.4, Double-Buffer DMA.

When enqueuing multiple buffers for double buffering, enqueue the buffers in the desired order of filling. Each buffer except the last should be enqueue with the LIO$M_HOLD_DMA argument. Enqueue the last buffer with the LIO$M_DONE_DBL_BUF argument. The ADQ32 then starts data collection when the last buffer is enqueued (if a timed or burst clock mode is enqueued; if a clock mode started by an external trigger event was selected, the ADQ32 waits for the external trigger event).

A.6 Start of Data Acquisition

For modes which do not use an external trigger or gate (modes 1, 5, 10, and 15), data acqusition starts when the LIO$READ or LIO$ENQUEUE routines are called. Data acquisition does not start when the triggering mode is set up with the LIO$SET parameter calls.

In double buffered mode, when buffers are enqueued with the LIO$ENQUEUE call using the LIO$M_HOLD_DMA qualifier, data acquisition does not start until the last buffer is queued without the LIO$M_HOLD_DMA qualifier and with the LIO$M_DONE_DBL_BUF qualifier. See Section 1.6.3.4, Double-Buffer DMA, for more information on the LIO$K_HOLD_DMA qualifier. See also LIO$ENQUEUE in Chapter 3.

For modes using external triggers or gates, data acquisition starts when the external gate or trigger event occurs, assuming that the LIO$READ or LIO$ENQUEUE call has previously executed. In double buffered mode, all buffers must have been enqueued (that is, the last buffer must have been queued without the LIO$M_HOLD_DMA qualifier) before the external trigger or gate occurs.

A.7 Clock Overrun Errors

The term clock overrun error is used in four situations:

- A trigger signal occurs while the ADC is currently converting a sample.
- A second sweep trigger signal occurs before all of the specified channels in a sweep have been converted.
- A trigger signal occurs but the ADC is currently disabled.
- In modes 4 and 9, an external trigger occurred, but the ADQ32 is still filling the current buffer.

In all of these situations, a trigger signal is whatever current signal is being used to trigger the ADC. This could be a pulse from the clock chip, an external trigger signal, or a gated external frequency signal.

When the clock overrun error occurs, the ADC is disabled. This ensures that only valid data is written to the FIFO. The DMA machine however, does not stop and continues transferring any data that is in the FIFO. This is data that was acquired before the overrun error occurred.

A.8 Important Points About the Clock Logic

In all of the following diagrams and discussions, t_1 is used to designate the rate at which samples are acquired. This rate is specified using the LIO$K_CLK_RATE parameter and is referred to as the primary clock.

The rate at which sweeps are taken is designated as t_2. This rate is specified using the LIO$K_SWEEP_RATE parameter and is called the sweep clock. Because of modes such as burst sweep, where you specify only the rate at which sweeps are taken, t_2 is sometimes specified even though t_1 is not used.

The clock rate you select should be at least twice the frequency of the highest component sine wave of the signal you are acquiring. This is called the Nyquist frequency.

All signals can be defined as the product of any number of sine waves. If you can identify the frequency of the highest component sine wave you are interested in, your sampling rate should be at least twice that frequency. For example, if you know that your signal contains no frequency higher than 20 KHz, a sampling rate of 40 KHz is sufficient to capture the signal data without aliasing.

Aperture delay is the time between the occurrence of an ADC triggering pulse and the acquisition of the sample voltage by the track-and-hold circuit. Effective aperture delay is the complete time difference between the occurrence of an external pulse and the acquisition of the sample voltage by the track-and-hold circuit. This includes the multiplexer and preamplifier settling times, and the clock offset referred to previously.

For applications that require the minimum effective aperture delay time, the external trigger mode is recommended. In this mode, the amount of time that elapses between the occurrence of the external trigger and the actual acquisition of the sample is as small as can be for the ADQ32.

A.9 Clock Mode 1, Burst

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1. Figure A-1 illustrates the clock output in this mode. Each upward spike in this drawing indicates one ADC sample taken.

Data acquisition starts as soon as the LIO$READ or LIO$ENQUEUE call executes. See Section A.6, Start of Data Acquisition, for more information.

Figure A-1: Clock Mode 1, Burst

You select clock mode 1 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST, LIO$K_SAME,
1                    LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)
```

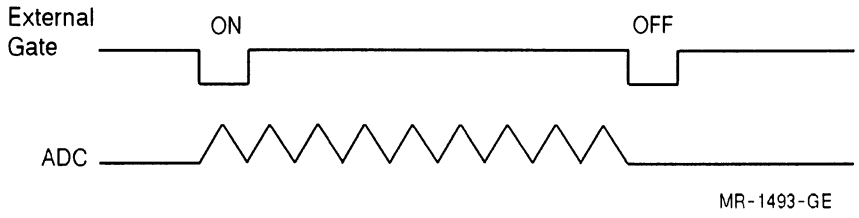
The GATE set parameter is not needed unless you previously enabled gating and are now switching to a new clock mode.

A.10 Clock Mode 2, Burst, with Edge Gate

In this mode, the ADQ32 acquires data at the burst rate, but data collection is controlled by an external gate signal. Negative transitions on the external gate toggle the ADQ32 in and out of burst mode. The first negative transition on the external gate input starts data acquisition. The second negative transition on the external gate stops data acquisition.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1. Figure A-2 illustrates the clock output in this mode.

Figure A-2: Clock Mode 2, Burst, with Edge Gate



You select clock mode 2 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST, LIO$K_SAME,
1                    LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE)
```

A.11 Clock Mode 3, Burst, with Delayed Edge Gate

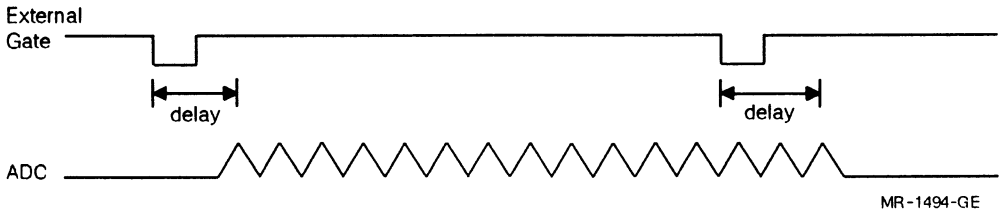
In this mode, the ADQ32 acquires data at the burst rate, but data collection is controlled by an external gate signal. In addition, data collection is delayed from the occurrence of the external gate signal by a specified time delay. You specify the time delay by selecting a clock rate with the LIO$K_SWEEP_RATE parameter. The time delay is one clock tick of the rate you select.

Negative transitions on the external gate toggle the ADQ32 in and out of burst mode. The first negative transition on the external gate input causes the delay counters to begin. After the delay time elapses, the ADQ32 starts data acquisition. The second negative transition on the external gate stops data acquisition, after the delay time elapses.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A–1.

Figure A–3 illustrates the clock output in this mode.

Figure A–3: Clock Mode 3, Burst, with Delayed Edge Gate



You select clock mode 3 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST, LIO$K_SAME,
1                    LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE_DELAY)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_rate_for_delay)
```

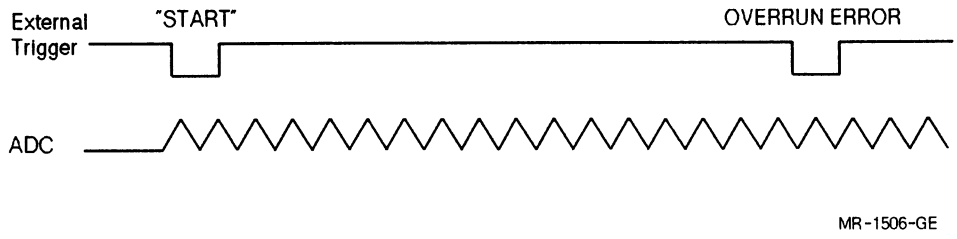
A.12 Clock Mode 4, Burst, Activated by External Trigger

In this mode, the ADQ32 acquires data at the burst rate, but data collection is started by an external trigger signal. The first negative transitions on the external gate/trigger input puts the ADQ32 into burst mode.

If the ADQ32 is attached for single buffer DMA, each external trigger causes one buffer of data to be acquired and transferred. If the ADQ32 is attached for doubled buffered DMA, each external trigger causes all queued buffers to be acquired and transferred. In both cases, if a subsequent external trigger occurs while a buffer is currently being filled, the subsequent external trigger causes a clock overrun error.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1. Figure A-4 illustrates the clock output in this mode.

Figure A-4: Clock Mode 4, Burst, Activated by External Trigger



You select clock mode 4 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST, LIO$K_SAME,
1                   LIO$K_EXTERNAL)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)
```

The GATE set parameter is not needed unless you previously enabled gating and are now switching to a new clock mode.

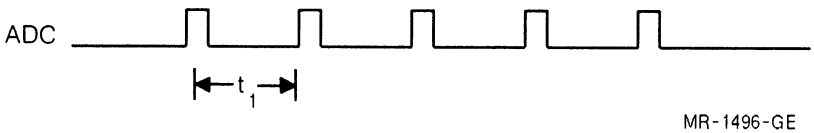
A.13 Clock Mode 5, Timed Triggers

In this mode, the ADQ32 acquires data at a specified clock rate. The clock rate is selected with the LIO$K_CLK_RATE parameter.

Data acquisition starts as soon as the LIO$READ or LIO$ENQUEUE call executes. See Section A.6, Start of Data Acquisition, for more information.

Figure A-5 illustrates the clock output in this mode.

Figure A-5: Clock Mode 5, Timed Triggers



You select clock mode 5 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK, LIO$K_SAME,
1                    LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_trigger_rate)
```

The GATE set parameter is not needed unless you previously enabled gating and are now switching to a new clock mode.

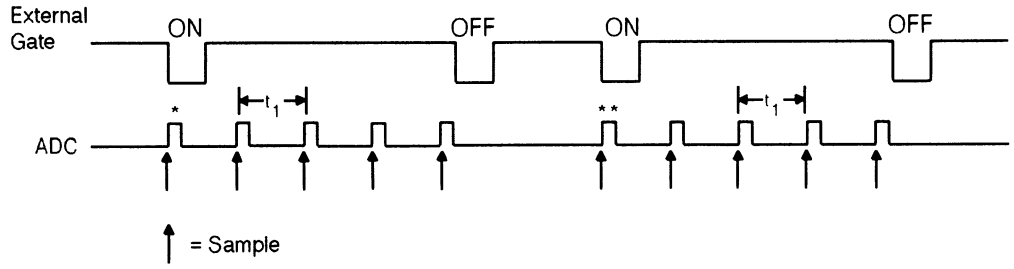
A.14 Clock Mode 6, Timed Triggers, with Edge Gate

In this mode, the ADQ32 acquires data at the clock rate specified by you. The clock rate is selected with the LIO$K_CLK_RATE parameter.

Data collection at the specified clock rate is controlled by an external gate signal. Negative transitions on the external gate turn the clock logic on and off. The first negative transition on the external gate input starts the clock counters. The second negative transition on the external gate stops data acquisition.

Figure A-6 illustrates the clock output in this mode.

Figure A-6: Clock Mode 6, Timed Triggers, with Edge Gate



↑ = Sample

* The first sample occurs slightly after the gate ON signal.

** Subsequent first samples occur between 0 and t_1 time after the gate edge. ($0 < \text{sample} \leq t_1$).

MR-3573-GE

You select clock mode 6 by using the following LIO routine calls:

```
status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK, LIO$K_SAME,
1                   LIO$K_SAME)

status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE)

status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_trigger_rate)
```

A.15 Clock Mode 7, Timed Triggers, with Delayed Edge Gate

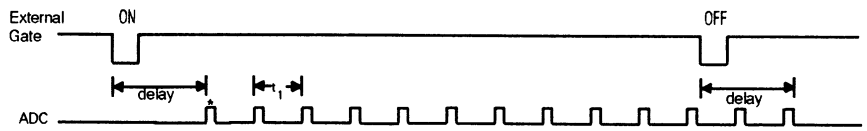
In this mode, the ADQ32 acquires data at the clock rate specified by you. The clock rate is selected with the LIO$K_CLK_RATE parameter.

Data collection at the specified clock rate is controlled by an external gate signal. In addition, data collection is delayed from the occurrence of the external gate signal by a specified time delay. You specify the time delay with the sweep clock, using the LIO$K_SWEEP_RATE parameter. The time delay is one clock tick of the rate you select.

Negative transitions on the external gate turn the clock logic on and off. The first negative transition on the external gate input causes the delay counters to begin. After the delay time elapses, the clock counters for the timed triggers begin. The second negative transition on the external gate stops data acquisition, after the delay time elapses.

Figure A-7 illustrates the clock output in this mode.

Figure A-7: Clock Mode 7, Timed Triggers, with Delayed Edge Gate



* The first sample occurs between 0 and t_1 time after the delay ends, ($0 < \text{sample} \leq t_1$).

MR-3574-GE

You select clock mode 7 by using the following LIO routine calls:

```
status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK, LIO$K_SAME,
1                   LIO$K_SAME)

status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE_DELAY)

status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_trigger_rate)

status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_rate_for_delay)
```

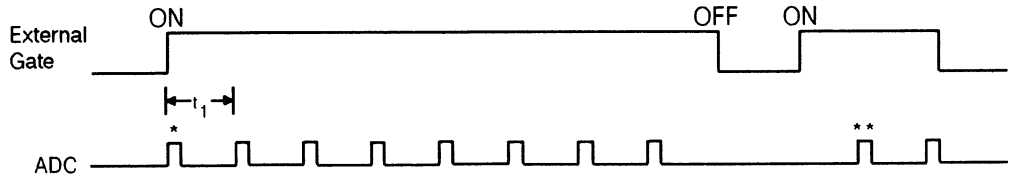
A.16 Clock Mode 8, Timed Triggers, with Level Gate

In this mode, the ADQ32 acquires data at the clock rate specified by you. The clock rate is selected with the LIO$K_CLK_RATE parameter.

Data collection at the specified clock rate is controlled by an external gate signal. The level of the signal controls when data collection occurs. Data collection at the specified rate occurs whenever the external gate input is HIGH. Data collection stops when the external gate input is LOW. Note that this means data collection starts immediately if the external gate input is HIGH when the data buffers are enqueued.

Figure A-8 illustrates the clock output in this mode.

Figure A-8: Clock Mode 8, Timed Triggers, with Level Gate



* The first sample occurs slightly after the external gate ON signal.

** Subsequent first samples occur between 0 and t_1 time after the gate signal. ($0 < \text{sample} \leq t_1$).

MR-3575-GE

You select clock mode 8 by using the following LIO routine calls:

```
status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK, LIO$K_SAME,
1                   LIO$K_SAME)

status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_LEVEL)

status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_trigger_rate)
```

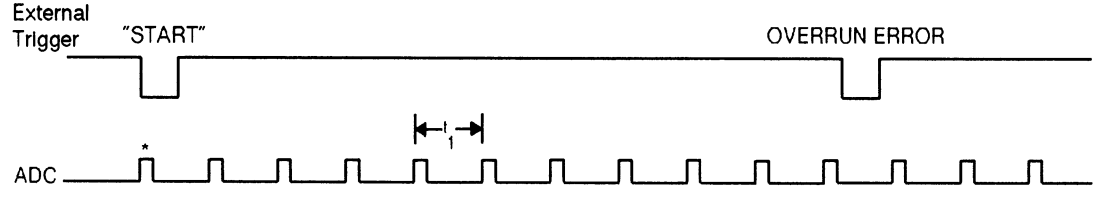
A.17 Clock Mode 9, Timed Triggers, Activated by External Trigger

In this mode, the ADQ32 acquires data at the clock rate specified by you, but data collection is controlled by an external trigger signal. The clock rate is selected with the LIO$K_CLK_RATE parameter.

The first negative transitions on the external gate/trigger input starts data collection at the specified rate. If the ADQ32 is attached for single buffer DMA, each external trigger causes one buffer of data to be acquired and transferred. If the ADQ32 is attached for doubled buffered DMA, each external trigger causes all queued buffers to be acquired and transferred. In both cases, if a subsequent external trigger occurs while a buffer is currently being filled, the subsequent external trigger causes a clock overrun error.

Figure A-9 illustrates the clock output in this mode.

Figure A-9: Clock Mode 9, Timed Triggers, Activated by External Trigger



* The first sample occurs slightly after the external trigger.

MR-3579-GE

You select clock mode 9 by using the following LIO routine calls:

```
status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK, LIO$K_SAME,
1                   LIO$K_EXTERNAL)

status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_trigger_rate)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.18 Clock Mode 10, Burst Sweeps

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter at the burst rate. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

NOTE

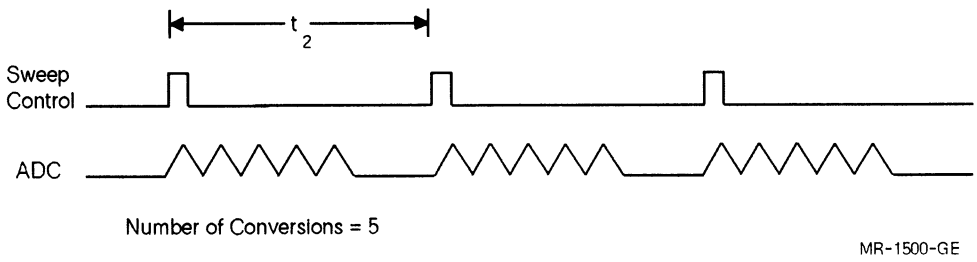
The sweep rate, t_2, must be greater than the product of the burst rate times the number of conversions. If it is not, a clock overrun error occurs.

Data acquisition starts as soon as the LIO$READ or LIO$ENQUEUE call executes. See Section A.6, Start of Data Acquisition, for more information.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1.

Figure A-10 illustrates the clock output in this mode.

Figure A-10: Clock Mode 10, Burst Sweeps



You select clock mode 10 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_R (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

A.19 Clock Mode 11, Burst Sweeps, with Edge Gate

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter at the burst rate. Each pass through the specified channels is controlled by a clock rate, t_2. You select this clock rate with the sweep clock, using the LIO$K_SWEEP_RATE parameter.

The entire clock mode is controlled by an external gate signal. A negative transition on the external gate starts the clock counters for t_2. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the burst rate.

The second negative transition on the external gate stops data acquisition. Note, however, that the external gate controls only the sweep timing signal, t_2. If a second negative transition occurs before a specified sweep has finished, that sweep of the specified channels is allowed to finish. The sweep control timing is shut off by the gate.

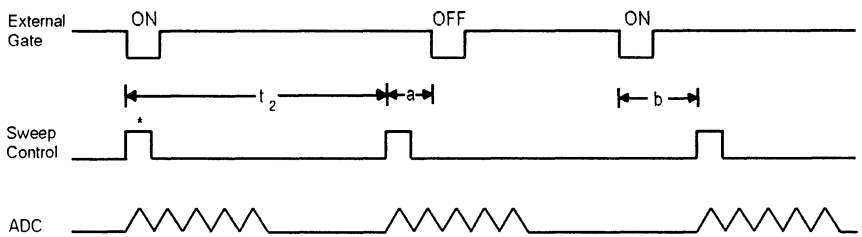
The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A–1.

NOTE

The sweep rate, t_2, must be greater than the product of the burst rate times the number of conversions. If it is not, a clock overrun error occurs.

Figure A–11 illustrates the clock output in this mode.

Figure A–11: Clock Mode 11, Burst Sweeps, with Edge Gate



Number of conversions = 5

* The first sweep control signal occurs slightly after the external gate ON. External gate OFF stops the t_2 counters and a subsequent gate ON restarts the t_2 counters so that a + b = t_2.

MR-3721-GE

You select clock mode 11 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

A.20 Clock Mode 12, Burst Sweeps, with Level Gate

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter at the burst rate. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

The entire clock mode is controlled by an external gate signal. The level of the signal controls when data collection occurs. The sweep control timing signal runs at the specified rate whenever the external gate input is HIGH. The sweep control timing signal stops when the external gate input is LOW.

A HIGH signal on the external gate starts the sweep clock counters for t_2. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the burst rate.

The LOW signal on the external gate stops the sweep clock counters. It is important to note, however, that the external gate controls only the sweep timing signal, t_2. If the external gate signal goes LOW before a specified sweep has finished, that sweep of the specified channels is allowed to finish. The sweep control timing is shut off by the gate.

Note that this clock mode starts immediately if the external gate input is HIGH when the buffers are enqueued.

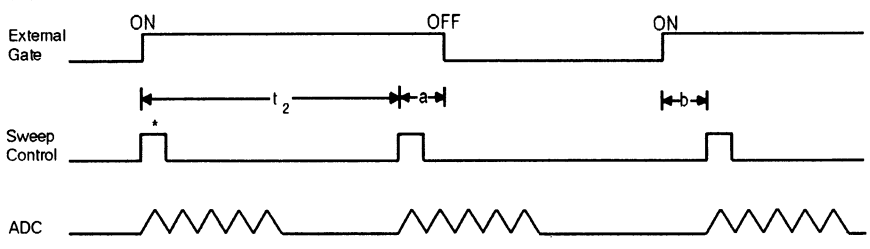
The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1.

NOTE

The sweep rate, t_2, must be greater than the product of the burst rate times the number of conversions. If it is not, a clock overrun error occurs.

Figure A–12 illustrates the clock output in this mode.

Figure A–12: Clock Mode 12, Burst Sweeps, with Level Gate



Number of conversions = 5

* The first sweep control signal occurs slightly after the external gate ON. External gate OFF stops the t_2 counters and a subsequent gate ON restarts the t_2 counters so that a + b = t_2.

MR-3722-GE

You select clock mode 12 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_LEVEL)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

A.21 Clock Mode 13, Burst Sweeps, Activated by External Trigger

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter at the burst rate. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

The clock mode is activated by an external trigger. The clock counters that control the sweep timing are not started until a negative transition occurs on the external trigger input. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the burst rate.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1.

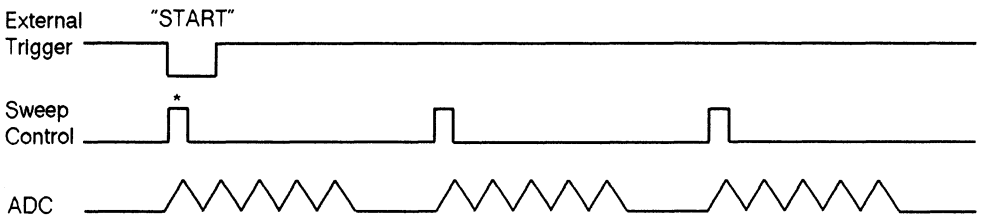
NOTE

The sweep rate, t_2, must be greater than the product of the burst rate times the number of conversions. If it is not, a clock overrun error occurs.

If the ADQ32 is attached for single buffer DMA, each external trigger causes one buffer of data to be acquired and transferred. If the ADQ32 is attached for doubled buffered DMA, each external trigger causes all queued buffers to be acquired and transferred. In both cases, if a subsequent external trigger occurs while a buffer is currently being filled, the subsequent external trigger is ignored.

Figure A–13 illustrates the clock output in this mode.

Figure A–13: Clock Mode 13, Burst Sweeps, Activated by External Trigger



* The first sweep control signal occurs slightly after the external trigger.

MR-3723-GE

You select clock mode 13 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1                    LIO$K_SWEEP_CLOCK, LIO$K_EXTERNAL)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.22 Clock Mode 14, Burst Sweeps, Sweep Controlled by External Trigger

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter at the burst rate. Each pass through the specified channels is controlled by a an external trigger signal connected to the external trigger input. When a negative transition on the external trigger input occurs, the ADQ32 makes one sweep at the burst rate of the specified channels.

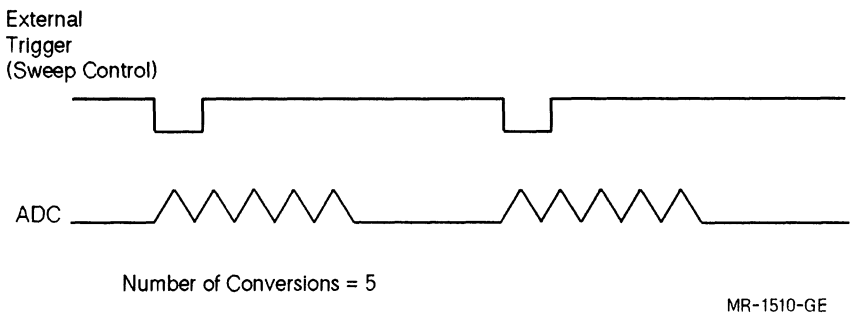
NOTE

The sweep through the specified channels must finish before the next external trigger occurs. If the next trigger occurs during a sweep, a clock overrun error occurs.

The burst clock mode provides data acquisition at a rate controlled by the ADQ32 state machine. The rate selected is the fastest possible rate that can be used and still guarantee data precision within 1/2 LSB. The actual rates used are shown in Table A-1.

Figure A-14 illustrates the clock output in this mode.

Figure A-14: Clock Mode 14, Burst Sweeps, Sweep Controlled by External Trigger



NOTE

All of the samples in a sweep must finish before the occurrence of the next external trigger. If this is not true, a clock overrun error occurs.

You select clock mode 14 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_BURST,
1                    LIO$K_EXTERNAL, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.23 Clock Mode 15, Timed Sweeps

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter using two specified rates. The primary clock rate, t_1, controls the rate at which each sample in the sweep is converted. This clock rate is specified using the LIO$K_CLK_RATE parameter. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

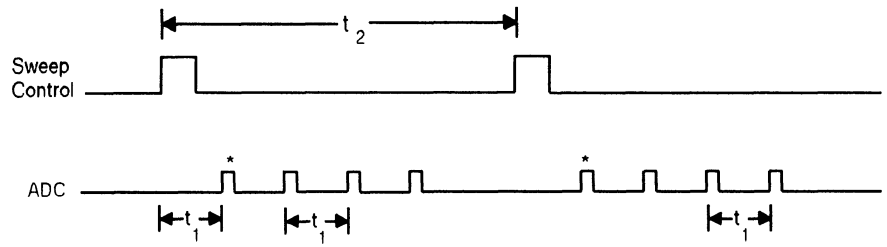
NOTE

The sweep rate, t_2, must be greater than the product of t_1 times the number of conversions. If it is not, a clock overrun error occurs.

Data acquisition starts as soon as the LIO$READ or LIO$ENQUEUE call executes. See Section A.6, Start of Data Acquisition, for more information.

Figure A-15 illustrates the clock output in this mode.

Figure A-15: Clock Mode 15, Timed Sweeps



* The first sample occurs t_1 time after the sweep control signal.

MR-3576-GE

You select clock mode 15 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_conversion_rate)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.24 Clock Mode 16, Timed Sweeps, with Edge Gate

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter using two specified rates. The primary clock rate, t_1, controls the rate at which each sample in the sweep is converted. This clock rate is specified using the LIO$K_CLK_RATE parameter. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

The entire clock mode is controlled by an external gate signal. A negative transition on the external gate starts the clock counters for t_2. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the primary clock rate.

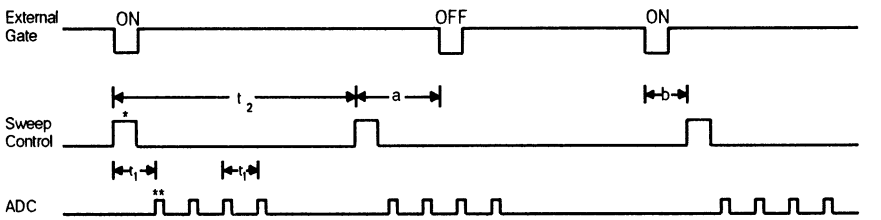
The second negative transition on the external gate stops data acquisition. It is important to note, however, that the external gate controls only the sweep timing signal, t_2. If a second negative transition occurs before a specified sweep has finished, that sweep of the specified channels is allowed to finish. The sweep control timing is shut off by the gate. The primary clock stops after the sweep completes.

NOTE

The sweep rate, t_2, must be greater than the product of the t_1 times the number of conversions. If it is not, a clock overrun error occurs.

Figure A–16 illustrates the clock output in this mode.

Figure A–16: Clock Mode 16, Timed Sweeps, with Edge Gate



Number of conversions per sweep = 4

* The first sweep control signal occurs slightly after the external gate ON. External gate OFF stops the t_2 counters and a subsequent gate ON restarts the t_2 counters so that a + b = t_2.

** The first signal in each sweep is t_1 time after the sweep control signal.

MR-3577-GE

You select clock mode 16 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_conversion_rate)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

A.25 Clock Mode 17, Timed Sweeps, with Level Gate

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter using two specified rates. The primary clock rate, t_1, controls the rate at which each sample in the sweep is converted. This clock rate is specified using the LIO$K_CLK_RATE parameter. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

The entire clock mode is controlled by an external gate signal. The level of the signal controls when data collection occurs. The sweep control timing signal runs at the specified rate whenever the external gate input is HIGH. The sweep control timing signal stops when the external gate input is LOW.

A HIGH signal on the external gate starts the clock counters for t_2. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the sweep clock rate.

The LOW signal on the external gate stops the clock counters. It is important to note, however, that the external gate controls only the sweep timing signal, t_2. If the external gate signal goes LOW before a specified sweep has finished, that sweep of the specified channels is allowed to finish. The sweep control timing is shut off by the gate. The primary clock stops after the sweep completes.

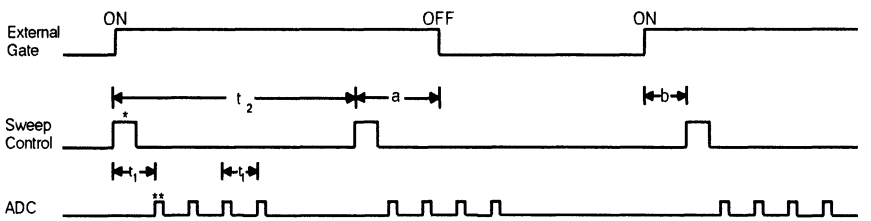
Note that this clock mode starts immediately if the external gate input is HIGH when the clock mode is programmed.

NOTE

The sweep rate, t_2, must be greater than the product of t_1 times the number of conversions. If it is not, a clock overrun error occurs.

Figure A–17 illustrates the clock output in this mode.

Figure A–17: Clock Mode 17, Timed Sweeps, with Level Gate



Number of conversions per sweep = 4

* The first sweep control signal occurs slightly after the external gate ON. External gate OFF stops the t_2 counters and a subsequent gate ON restarts the t_2 counters so that a + b = t_2.

** The first signal in each sweep is t_1 time after the sweep control signal.

MR-3578-GE

You select clock mode 17 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                    LIO$K_SWEEP_CLOCK, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_LEVEL)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_conversion_rate)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

A.26 Clock Mode 18, Timed Sweeps, Activated by External Trigger

In this mode, the ADQ32 makes each pass through the channels specified in the LIO$K_AD_CHAN parameter using two specified rates. The primary clock rate, t_1, controls the rate at which each sample in the sweep is converted. This clock rate is specified using the LIO$K_CLK_RATE parameter. Each pass through the specified channels is controlled by the sweep clock rate, t_2, which you specify using the LIO$K_SWEEP_RATE parameter.

The clock mode is activated by an external trigger. The clock counters that control the sweep timing are not started until a negative transition occurs on the external trigger input. When the first clock tick for this clock rate occurs, the ADQ32 samples all of the specified channels at the primary clock rate.

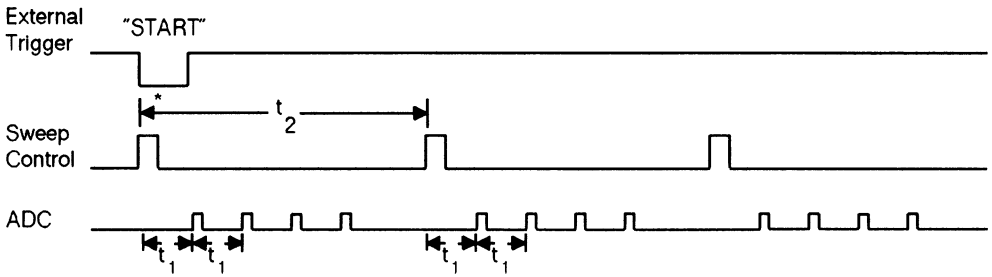
NOTE

The sweep rate, t_2, must be greater than the product of t_1 times the number of conversions. If it is not, a clock overrun error occurs.

If the ADQ32 is attached for single buffer DMA, each external trigger causes one buffer of data to be acquired and transferred. If the ADQ32 is attached for doubled buffered DMA, each external trigger causes all queued buffers to be acquired and transferred. In both cases, if a subsequent external trigger occurs while a buffer is currently being filled, the subsequent external trigger is ignored.

Figure A–18 illustrates the clock output in this mode.

Figure A–18: Clock Mode 18, Timed Sweeps, Activated by External Trigger



* The first sweep control signal occurs slightly after the external trigger.

MR-3724-GE

You select clock mode 18 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                    LIO$K_SWEEP_CLOCK, LIO$K_EXTERNAL)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_conversion_rate)

 status = LIO$SET_I (adq_id, LIO$K_SWEEP_RATE, 1, desired_sweep_rate)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.27 Clock Mode 19, Timed Sweeps, Sweep Controlled by External Trigger

In this mode, the ADQ32 samples within a sweep at a specified clock rate. The primary clock rate, t_1, controls the rate at which each sample in the sweep is converted and is specified using the LIO$K_CLK_RATE parameter.

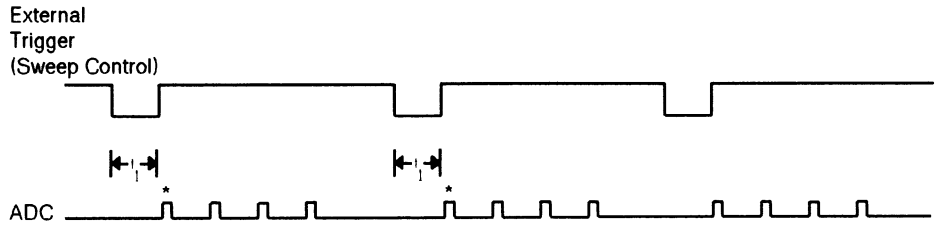
Each pass through the channels specified in the LIO$K_AD_CHAN parameter is controlled by a an external trigger signal connected to the external gate/trigger input. When a negative transition on the external gate/trigger input occurs, the ADQ32 makes one sweep at the specified clock rate of the specified channels.

NOTE

The sweep through all of the specified channels must finish before the next external trigger occurs. If the next trigger occurs during a sweep, a clock overrun error occurs.

Figure A-19 illustrates the clock output in this mode.

Figure A-19: Clock Mode 19, Timed Sweeps, Sweep Controlled by External Trigger



Number of conversions = 4

* The first sample in each sweep occurs t_1 time after the sweep control signal.

MR-3580-GE

You select clock mode 19 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_AD_CLOCK,
1                        LIO$K_EXTERNAL, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)

 status = LIO$SET_I (adq_id, LIO$K_CLK_RATE, 1, desired_conversion_rate)
```

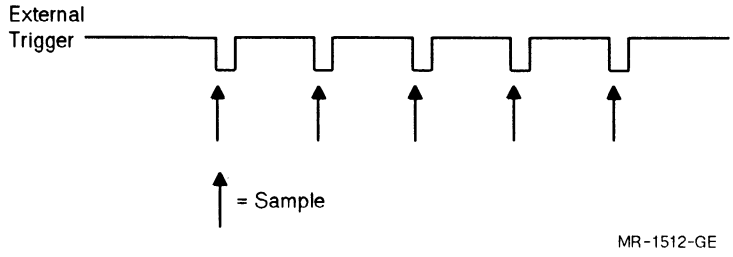
The GATE set parameter is not needed unless you previously enabled gating.

A.28 Clock Mode 20, External Triggers

In this mode, the external triggering signal is connected to the external gate/trigger input. A single conversion is taken each time a negative transition occurs on this input. This mode provides the minimum effective aperture delay for the ADQ32. In other words, the offset time between the occurrence of the triggering signal and the acquisition of the sample voltage is as small as is possible.

Figure A–20 illustrates the clock output in this mode.

Figure A–20: Clock Mode 20, External Triggers



You select clock mode 20 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_EXTERNAL,
1                    LIO$K_SAME, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_OFF)
```

The GATE set parameter is not needed unless you previously enabled gating.

A.29 Clock Mode 21, External Triggers, with Edge Gate

In this mode, an external triggering signal controls when each sample is converted. In addition, an external gating signal controls when the external trigger signals are routed to the ADC.

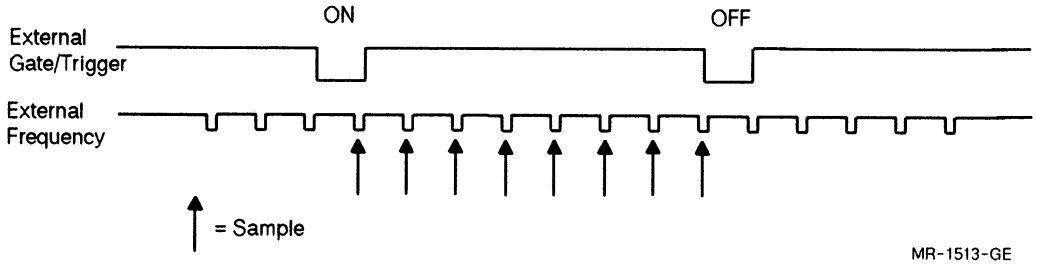
Negative transitions on the external gate input (labelled gate/trigger input) control when the triggering signal is routed to the ADC. The first negative transition on the external gate input causes the triggering signals (connected to the external frequency input) to be routed to the ADC. When the triggering signals are routed to the ADC, a single conversion is taken each time a negative transition occurs on the external frequency input. The second negative transition on the external gate input causes the triggering signals to be ignored.

NOTE

The external triggering signal is connected to the external frequency input and the gating signal is connected to the external gate/trigger input.

Figure A–21 illustrates the clock output in this mode.

Figure A–21: Clock Mode 21, External Triggers, with Edge Gate



You select clock mode 21 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_EXTERNAL,
1                    LIO$K_SAME, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE)
```

A.30 Clock Mode 22, External Triggers, with Delayed Edge Gate

In this mode, an external triggering signal controls when each sample is converted. An external gating signal controls when the external trigger signals are routed to the ADC. In addition, a delay time is specified using the clock, which delays the action of the gating signal.

You specify the time delay with the sweep clock, using the LIO$K_SWEEP_RATE parameter. The time delay is one clock tick of the rate you select.

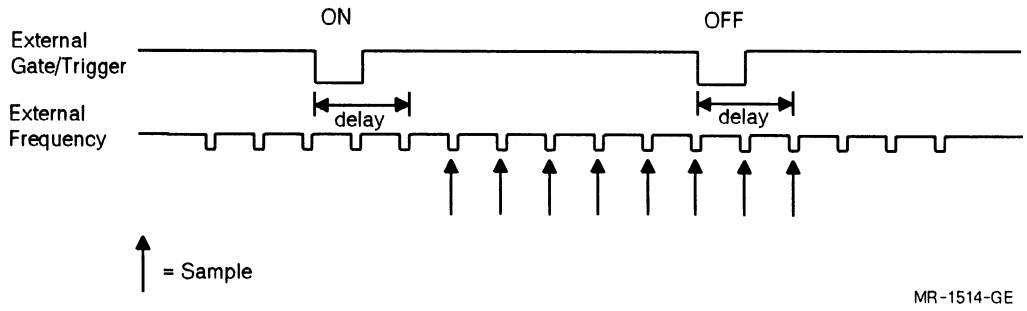
Negative transitions on the external gate input (labelled gate/trigger input) control when the triggering signal is routed to the ADC. The first negative transition on the external gate input causes the delay counters to begin. After the delay time elapses, negative transitions on the external frequency input (the triggering signals) are routed to the ADC. When the triggering signals are routed to the ADC, a single conversion is taken each time a negative transition occurs on the external frequency input. The second negative transition on the external gate causes the delay counters to run again. After the delay time elapses, signals on the external triggering input are ignored.

NOTE

The external triggering signal is connected to the external frequency input and the gating signal is connected to the external gate/trigger input.

Figure A–22 illustrates the clock output in this mode.

Figure A–22: Clock Mode 22, External Triggers, with Delayed Edge Gate



You select clock mode 22 by using the following LIO routine calls:

```
 status = LIO$SET_I (adq_id, LIO$K_TRIG, 3, LIO$K_EXTERNAL,
1                    LIO$K_SAME, LIO$K_SAME)

 status = LIO$SET_I (adq_id, LIO$K_GATE, 1, LIO$K_EDGE_DELAY)

 status = LIO$SET_I( adq_id, LIO$K_SWEEP_RATE, 1, desired_rate_for_delay)
```

Appendix B

Using CTI I/O with the AXV11-C

This appendix contains the procedure you need to follow to use connect-to-interrupt I/O with the AXV11-C. Also included are procedures for reloading and reconnecting the QIO driver to the AXV11-C or to another AXV11-C device, if you have more than one AXV11-C configured in your VAXlab system.

B.1 Connecting the CTI Driver to the AXV11-C

During the VAXlab Software Library installation procedure, the QIO driver, AXDRIVER.EXE, is installed and connected to the AXV11-C. To use CTI I/O with the LIO routines, you need to connect the CTI driver to the AXV11-C.

Do the following to connect the CTI driver to the AXV11-C:

1. Log in to the SYSTEM account.
2. Run the SYSGEN utility and enter the SHOW/CONFIGURATION command to obtain the CSR and vector1 addresses of the AXV11-C, for example:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN>SHOW/CONFIGURATION

        System CSR and Vectors on 23-NOV-1989 13:49:26.86
.
.
.
```

```
Name: AXA  Units: 1 Nexus: 0 (UBA) CSR: 776400 Vector1: 140 Vector2: 0
    .
    .
    .
SYSGEN>EXIT
$
```

All AXV11-C devices configured in your system are prefixed with a device type of AX, followed by a variable controller letter (A, B, C ...) and the unit number 0.

If more than one AXV11-C is configured in your system, be aware of the controller letter of the AXV11-C to which you want to connect the CTI driver. See the description of the LIO$ATTACH routine in Chapter 4 for more information about the LIO-supported devices and their corresponding device types.

Make a note of the CSR and vector1 addresses. You need this information in step 6 of this procedure.

3. Set default to the directory where the drivers are located and rename the QIO driver AXDRIVER.EXE. For example:

```
$ SET DEFAULT SYS$COMMON:[SYS$LDR]
$ RENAME AXDRIVER.EXE AXDRIVER.QIO
```

4. Reboot the system by entering the following command:

```
$ REBOOT
```

An orderly shutdown of the system begins. Several informational messages are displayed on your terminal as the shutdown proceeds. After shutdown completes, the system is rebooted. This procedure takes several minutes to complete.

When you reboot the system, the QIO driver is disabled. That is, the driver is not connected to any device. If you have more than one AXV11-C configured in your system you must do one or both of the following:

- Connect the CTI driver to those AXV11-Cs configured in your system for which you want CTI support. To do this, follow the procedure outlined in this section.
- Reload the QIO driver and connect it to those AXV11-Cs configured in your system for which you want QIO support. See Section B.2, Reloading the QIO Driver, for more information.

5. Log back in to the SYSTEM account.
6. Run SYSGEN again and enter the following command line, specifying the CSR and Vector1 addresses that are specific to your system. (You obtained this information in step 2 of this procedure.)

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN>CONNECT AXA0:/ADAPTER=0/CSR=%0776400/DRIVER=CONINTERR -
SYSGEN>/NUMVEC=2/VEC=%0140
SYSGEN>EXIT
$
```

where:

AXA0	is the AXV11-C device. If more than one AXV11-C is configured in your system, make sure you use the correct device mnemonic and that you obtained the appropriate CSR and vector1 addresses for that particular device in step 2 of this procedure.
ADAPTER	is the bus adapter to which the device is attached. Note that there is no bus adapter on a MicroVAX, so you specify ADA=0.
CSR	is the address of the first addressable location on the controller (usually the status register) for the device. Enter the CSR address you obtained in step 2 of this procedure. The CSR address must be prefixed by %O (percent sign and the letter O) to signal the system that it is an octal number.
DRIVER	is the name of the driver.
CONINTERR	is the name of the CTI driver.
NUMVEC	is the number of interrupt vectors for the device.
VEC	is the value of the interrupt vector for the device, or the lowest vector, if there is more than one. Enter the Vector1 address you obtained in step 2. This address must be prefixed by %O (percent sign and the letter O) to signal the system that it is an octal number.

The CTI driver is now connected to the AXV11-C. If you have more than one AXV11-C configured in your system and you want to connect the CTI driver to them, you must repeat the above procedure for each AXV11-C.

You must also repeat the procedure each time the system is rebooted. Or, you can create a command procedure, called from SYSCONFIG.COM, to reconnect the CTI driver each time the system is rebooted. For example:

```
$ EDIT AXALOAD.COM
$ RUN SYS$SYSTEM:SYSGEN
CONNECT AXA0:/ADAPTER=0/CSR=%0776400/DRIVER=CONINTERR/NUMVEC=2/VEC=%0140
EXIT
$EXIT
```

This sample command procedure is called AXALOAD.COM. To call this procedure from SYSCONFIG.COM to reconnect the CTI driver to the AXV11-C on system reboot, edit SYSCONFIG.COM to contain the following command line:

```
$ EDIT SYSCONFIG.COM
   .
   .
   .
$ @AXALOAD
   .
   .
   .
$EXIT
```

B.2 Reloading the QIO Driver

If you have more than one AXV11-C configured in your system, you can reload the QIO driver and connect it to another AXV11-C, if you choose to connect the CTI driver to only one AXV11-C device.

Do the following to reload the QIO driver and connect it to an AXV11-C device:

1. Log into the SYSTEM account.
2. Run the SYSGEN utility. Enter the SHOW/CONFIGURATION command (discussed in step 2 in Section B.1 to obtain the CSR and vector1 addresses of the appropriate AXV11-C. Then, reload and connect the QIO driver. Be sure to use the same name for the driver as you did when you renamed it in step 3 of the previous procedure. For example:

```
SYSGEN>RELOAD SYS$COMMON:[SYS$LDR]AXDRIVER.QIO
SYSGEN>CONNECT AXB0:/ADAPTER=0/CSR=%0776430/DRIVER=AXDRIVER
SYSGEN>/NUMVEC=2/VEC=%0170
SYSGEN>EXIT
```

3. Reboot the system.

The QIO driver is now reloaded and connected to a second AXV11-C device, which in this example is the device AXB0.

B.3 Reconnecting the QIO Driver

If, at any time, you want to reconnect the QIO driver to an AXV11-C that you previously connected to the CTI driver, do the following:

1. Rename AXDRIVER.QIO to AXDRIVER.EXE.
2. Edit SYSCONFIG.COM to remove the command line @AXALOAD.
3. Reboot the system.

The QIO driver is now reconnected to the AXV11-C.

Index

A

B

C

D

E

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

V

W

X

Reader's Comments

Guide to the VAXlab Laboratory I/O Routines
AA-KN99C-TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

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

What I like best about this manual: _____

_____

What I like least about this manual: _____

_____

My additional comments or suggestions for improving this manual:

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Please indicate the type of user/reader that you most nearly represent:

☐ Administrative Support
☐ Computer Operator
☐ Educator/Trainer
☐ Programmer/Analyst
☐ Sales
☐ Scientist/Engineer
☐ Software Support
☐ System Manager
☐ Other (please specify) _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

10/87

Fold Here and Tape

digital™

Please
Affix Stamp
Here

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications
P.O. BOX 1001
MARLBOROUGH, MA 01752–9840

Fold Here

Cut Along Dotted Line