KineticSystems Company, LLC
AL10-2PA1
Windows 2000 Device Driver
V122 FOXI
User's Manual

March 13, 2003

KineticSystems Company, LLC

Windows 2000 Device Driver & API Libraries

2962 PCI Grand Interconnect

900 N. State Street, Lockport, Illinois 60441 (815) 838-0005 (815) 838-4424

Windows 2000 Device Driver/API

2962 PCI Grand Interconnect

Document Revision: March, 2003

Software Version: 4.0.0

Operating System: Microsoft Windows 2000

March, 2003

KineticSystems Company, LLC makes no representations that the use of its products in manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of license to make, use, or sell equipment or software in accordance with the description.

Available Monday through Friday 8:00 a.m. to 5:00 p.m. central standard time

Telephone: (815) 838-0005
Fax: (815) 838-4424
E-mail: tech-support@kscorp.com
Web Home page: http://www.kscorp.com

TABLE OF CONTENTS

1 Introduction

This document describes the application programming libraries for the KineticSystems' 2962 PCI Grand Interconnect adapter. Although a cursory overview of the 2962 is provided, the user should reference the 2962 hardware manual for more details.

1.1 2962 PCI Grand Interconnect Adapter

The 2962 provides the host application programs the ability to address CAMAC and VXI chassis on the Grand Interconnect highway. The actual accessing of the modules within the CAMAC and VXI crates is via command lists. These command lists must first be loaded into the command list memory of the 2962 and then requested to execute. The 2962 supports command lists containing CAMAC commands (CNAF), VXI commands, and additional command lists unique to the 2962.

All data transfer operations to or from the 2962 requires a data buffer (except for a command list containing all inline writes). A command list when executed by the 2962 may either supply data or sink data but not both (e.g., a block write and a block read in the same command list will cause an error). Special command list instructions are provided that allows the ability to store data within the list itself. These special instructions can be used for setting up crate registers prior to a read or a write.

The 2962 has the ability to trigger lists by either an internal or external clock. When the 2962 is loaded with a command list the list will begin execution when the clock period expires. Prior to the clock expiring, there must be a data buffer available for the transfer or the clock trigger will be lost.

The 2962 supports multi-buffering using the memory within the host processor. Once set up a multi-buffer interrupt occurs whenever a fixed number of transfers occur. The number of buffers can be from two to four. The host processor must have all of the buffer mapped for DMA transfer prior to the execution of the list. The list must be first loaded into the 2962 command list memory and then triggered (normally by the internal or external clock).

1.2 Windows 2000 Device Driver

The 2962 device driver for Windows 2000 provides a standard Windows interface to the 2962 Grand Interconnect adapter.

The command memory of the 2962 is partitioned into eight partitions (numbered 1 to 8). The size of each of the partition is user selectable via an appropriate API call. The driver defaults the size of the all the partitions to start at zero and to contain the complete command list memory. This means that in effect all partitions are using the same command list memory.

The CAMAC and VISA library functions make use of command list partition number one. The simple single CAMAC commands, such as CAM16 and CAM24 only use about eight memory locations in the 2962. The CAMAC list building routines can generate larger lists that may not fit in partition one. The standard CAMAC list building routines did not provide the ability to specify a partition that may limit the minimum size the user may want to select for the first partition.

The 2962 device driver supports the following functions of the 2962:

- Command List processing
- Demand Message processing

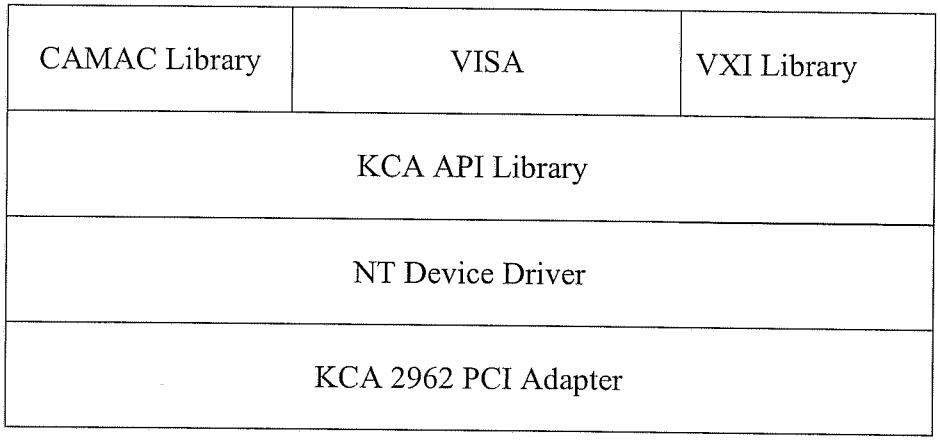
1.3 Demand Messages

The driver will only dequeue the demands from the device when a process posts a read for the demands. The read will complete when one or more demands are present in the FIFO of the 2962. The user will receive from one to the number that can be placed into user's buffer. If there are demands present when the user posts the read, as many as possible are immediately returned to the user. The user's read request will wait only if there are currently no demands in the demand FIFO of the 2962. The maximum number of demands that are dequeued is also limited by the driver as they must be unloaded under interrupt lockout. If the 2962 is reset due to an error, the user will be notified with a status indicating that a reset has occurred. The user should be aware that demands may have been lost.

A special demand process server is provided with the driver to help users integrate demands into their application. The Demand Process is required for LAM support in the KineticSystems CAMAC library. LAMS (CAMAC Look At Me) is a subset of the demand types handled by the 2962 device driver. The Demand Process also services the Demand Messages from VXI chassis.

1.4 Programming Support

An API (Application Programming Interface) is provided with the 2962 CAMAC Serial device driver. The user should use these routines to provide more portability and reduce system dependencies. KineticSystems also provides a list building support for the 2962 specific command lists and for the 3972 CAMAC crate controller. The following describes the provided software layers:



2 Installation

The device driver is installed using the InstallShield product. The user only needs to determine in what directory the device driver this product should be placed into. Normally, this software is placed in: C:\KCAxxx] (xxx= product release and version number). Depending if the kit was acquired via an FTP site or was distributed on floppies, use the normal NT installation procedure.

2.1 Directory Structure

The following documents the contents of each of the sub-directories.

\DRIVER	Contains Driver Image
\API	Contains API Shareable image (KSCAPI.LIB)
\CAMAC_UTIL	CAMAC utilities
\TEST	Simple test programs
\EXAMPLES	Example source code
\DEMAND	Demand Process and configuration file
\INCLUDE	Include files
\DOCUMENTATION	Contains this document and release notes

2.1.1 Include Files

The following is a description of the include files provided in the include directory.

C header files

- CAMAC.H- Contains CAMAC specific parameters
- CAMERR.H- Defines all of the CAMAC library error codes
- CMDLIST.H- Contains the macros and definitions used for building load and go command lists. The user must define the symbol: "KSCADP_SH" to select the generation of the correct command list generation.
- KERRORS_MSG.H- Defines all of the KSC API error codes
- KSC_API.H - Contains prototypes for the KSC API library
- KSC_GENLIST.H- Contains prototypes and definitions for the KSC list building routines
- KSC_HANDLE.H- Contains the definition of the KSC API handle
- KSCIOCTL.H- Defines all of the IOCTL codes for the 2115 NT device driver
- KSCUSER.H- Contains all of the CAMAC library prototypes and list building header

2.2 Post Installation

The 2115 is automatically configured by the POST (Power Up and Self Test) of the PCI based processor upon system bootstrap. This configuration information is then used by the driver to determine IRQ levels and bus address space requirements. The user has a choice of requesting that the device driver be loaded upon bootstrap by placing an entry into the Windows NT startup window. The driver may also be automatically loaded by running REGEDT32.EXE to modify the NT registry. Modify the value of Start from 3 to 1 under the following registry tree.

HKEY_LOCAL_MACHINE
À SYSTEM
À CurrentControlSet
À Services
À KSC2115

The driver may be manually started by entering NET START KSC2115 at the command prompt. The driver may be manually stopped by entering NET STOP KSC2115 at the command prompt.

2.3 Program Groups

The Window Program Group "KSC CAMAC2115" is created in the program manager. The following sub-groups are also created off the main program group.

- **CAMAC UTILITIES**
  CAMAC_COMMANDS- Runs the CAM utility program
  CAMAC CONTROL- Runs the CCTRL utility program
  CRATE STATUS- Runs the CCSTS utility program

- **DEMAND PROGRAM**
  Demand Process- Starts the Demand Process
  Demand Status- Start the Demand Status process
  Edit Configuration File- Calls up note pad to allow user to edit the configuration file
  Start Driver and Demand Process- This will load the 2115 device driver and request the execution of the Demand process
  Stop Driver- Unloads the 2115 device driver

- **TEST PROGRAMS**
  LAM3473- Example LAM program that requires a 3473 change of state card
  TEST_API- Example test program that tests the KSC API library
  TEST_CAMAC- Example test program that tests the CAMAC library calls
  TEST_DEMAND- Test program that tests the demand functioning

- **README**- Calls up note pad to read the read me file.

3 API Library

This chapter explains the use of the existing CAMAC software library available from KineticSystems for CAMAC crates. The CAMAC library routines eventually call the KSC API for the 2962. Users who want to use the extended list building capabilities of the 2962 must use the KSC API list building routines.

3.1 Support Limitations

The existing KSC CAMAC routines were originally developed for RSX11M Plus and Windows NT/VAX environments using older KSC serial highway and bus technologies. These older CAMAC adapters did not support the ability of placing the CAMAC command execution list completely within the adapter as the 2962 does.

The execution of the command list within the adapter results in the inability to return certain status information to the user, No-X, NO-Q, etc. on a per command instance. Every attempt was made to support the existing CAMAC installed base from VMS. The CAMAC library maintained a header that points to five different buffers. The CAMAC compatibility library supplied with the 2962 does not use all of these buffers. The main reason for this change is the design of the 2962 and the changes in the NT device driver. Because the command list is actually loaded into the 2962 and then executed by the 2962, the device driver can only report the following information:

1. Number of bytes actually transferred
2. The location in the command list after last command executed
3. The highway status of the last command
4. Controller CSR
5. The method of LAM handling is different than previous releases of the software

The earlier CAMAC serial highway drivers executed a single CAMAC instruction and the result of each instruction was available to the caller. The CAMAC compatibility library uses the three items above to populate the status array, however, the QXE buffer and the Word Count buffer are not supported. The compatibility library maps the 2962 NT error codes to the existing error codes in the CAMAC library.

The CAMAC library required a channel number passed for each call to the CAMAC library routines. This variable was a 16-bit word value. This value has been replaced by a bit long word value. This variable no longer contains the channel number but is a pointer to an allocated structure.

All buffers used by the CAMAC library must be long word (32-bits) aligned. Additionally, all buffers that are used for write functions must contain additional four 32-bit longwords at the end. These requirements are a result of the DMA pipelining of the 2962.

Users who wish to use the clocked lists or the multibuffer support of the 2962 must use the KSC API as there is no equivalent function in the existing CAMAC library.

The routine CXLMST is no longer supported. The ASTs that were triggered by the LAMs are passed with different arguments.

3.1.1 CAMAC list building

The CAMAC library provided software to build CAMAC lists. This structure has been changed by adding a new longword. If existing users have followed the CAMAC library standards, the size of the header array will be increased with a new compile. This new addition allows for the ability to use either the existing CAMAC list building along with the 2962 specific list building routines found in the KSC API library.

3.2 CAMAC Routines

The High Level Language CAMAC Library Routines supplied in the CAMAC library can be used in conjunction with C, and other high level languages. These routines may also be used with any language that follows the calling standard. The documentation explains each call in a language independent manner.

The routines in this chapter are simple to understand and use. In general, the specified CAMAC I/O operation will be executed before control is returned to the user process, and each call corresponds to a basic CAMAC I/O operation (either adding to the command list being built or actually performing the operation).

Note: Before attempting to issue any CAMAC commands to a Serial Crate the user must insure the crate is on-line. This can be accomplished by a call to CACTRL specifying the 'ONLINE' function.

3.2.1 Performance Considerations

Frequently CAMAC applications may involve time critical program segments. By default Windows is a time sharing system and as a result can lead to very disappointing performance in some time critical situations. The user may need to lock down all data buffers to improve performance. The loading of the 2962 with the command list can also be done once and left within the 2962 for later execution. The software driver provides the ability to partition the command list memory into eight different partitions. All CAMAC routines use the first partition. More control of the 2962 requires the user use the KSC API calls.

The 2962 also supports multi-buffer memory and triggers that can improve performance of lists that are executed more than once. This is achieved by locking the buffer into memory (the I/O request never completes) and loading the command list only once and providing the I/O completion notification via another I/O device.

3.2.2 CAMAC Library Call Summary

The standard CAMAC library routines provide you with a simple direct set of calls to perform I/O operations to CAMAC. The calls are divided into six groups:

Initialization calls
CAOPEN (chan,device,StatusArray)
CACLOS (chan,StatusArray)

Single-Action Data Transfer Calls
CAM16 (chan,C,N,AF,data,StatusArray)
CAM24 (chan,C,N,AF,data,StatusArray)

Block Transfer Calls

CAB16 (chan,C,NA,F,mode,DataArray,TransCount,StatusArray)
CAB24 (chan,C,NA,F,mode,DataArray,TransCount,StatusArray)

Enhanced Serial Highway Block Transfer Calls
CAB16E (chan,C,N,A,F,mode,DataArray,TransCount,StatusArray)
CAB24E (chan,C,NA,Fmode,DataArray,TransCount,StatusArray)

Status and Control Calls
CACTRL (chan,C,func,StatusArray)
CCSTAT (chan,C,CrateStat,StatusArray)
CAMSG (StatusArray)

LAM or Asynchronous Calls
CXLAM (chan,C,LAMid,Type,Prio,ASTadr,StatusArray)
CALAM (handle,C,lam_id,lam_type,priority,ast_addr,user_parm,ClrN,ClrA,ClrF,DsbN,DsbA,DsbF,error)

The CAMAC library routines are called as a longword function subroutine:
```
int    IERROR;
IERROR = camroutine(arguments...);
```

where camroutine is one of the CAMAC routines defined in this manual. In the case of the Function subroutine, the function returns the error status. The error status follows NT conventions and is always odd if the operation was successful. The Function subroutine simplifies the checking of the success or failure of a CAMAC I/O operation, since the call and the test are made in the same line as follows:

```
if((camroutine(args ... ) .and. 1) .eq. 1) THEN "success" ELSE "fail"
```

Examples in this manual use the CALL form, but the Function form can also be used as appropriate.

3.2.3 Initialization Calls

The initialization calls provide a mechanism to open the CAMAC device for I/O by a program. Subroutine CAOPEN should be called once for each CAMAC interface (2962) to be accessed by the program and should not be called again until the channel has been closed. The pointer points to a KSC API handle that is allocated when the 2962 is opened.

3.2.4 Single-Action Data Transfer Calls

The single-action data transfer calls are simple to use. Each call results in a single CAMAC operation and the appropriate data transfer. Two versions of the single-action routines are provided, CAM16 for 16-bit transfers and CAM24 for full 24-bit transfers. These routines are appropriate for applications where single I/0 operations are required or for short blocks of data where the overhead of program-transfer operations can be tolerated. For large blocks of data, the CAMAC block transfer routines are recommended; they take full advantage of the hardware DMA features and only incur the setup overhead once for the entire operation.

3.2.5 Block Transfer Calls

The CAMAC block transfer calls move blocks of data to or from modules in a single operation using the DMA features of the 2962. Use these routines for reading or writing blocks of data between Alpha memory and transient digitizers, FIFO modules, display modules, etc.; for repeated operations to a single module; and for reading or writing a group of modules in a CAMAC crate. Even for a modest-size data block, these routines have less overhead than the equivalent number of single-action calls because they transfer the data block at a DMA rate and incur the software setup overhead only once for the entire operation.

3.2.6 Highway Operations

The CAMAC calls move blocks of data to or from CAMAC modules in a single operation using the Enhanced Block Transfer features of the CAMAC Serial Highway. Since the 2962 does not have a CAMAC Serial Highway, the Enhanced Block Transfer are emulated using normal CAMAC block transfers.

The Enhanced Serial Highway modes are summarized in Table below.

QSTP (mode=0)	Performs a Q-Stop CAMAC block transfer operation. This mode continues to transfer the block of data until the Transfer Count is exhausted or a NO-Q is received.
QIGN (mode=8)	Performs a Q-Ignore CAMAC block transfer operation. This mode transfers the block of data until the Transfer Count is exhausted. The Q response is ignored.
QRPT (mode=16)	Performs a Q-Repeat CAMAC block transfer operation. This mode transfers the block of data until the Transfer Count is exhausted or a (hardware or Software) time-out occurs. Whenever a Q=0 response is received during the block, the Dataway operation is repeated and the data array address pointer is not incremented.
QSCN (mode=24)	Performs a Q-Scan CAMAC block transfer operation. This mode transfers a block of data until the Transfer Count is exhausted or N>23. A represents the starting subaddress and N represents the initial station number for the scan operation. Note that the ending values of A and N are not returned.

3.2.7 Control and Status Calls

With the control and status calls, you can Initialize or Clear a crate, change the state of crate Inhibit, read crate status, and read the status of the last CAMAC operation.

All of the library calls return a status array. This array contains information on the last call to the CAMAC routines. At the simplest level, it indicates whether the I/O request was successfully performed. If StatusArray(ERR) is odd, it indicates a successful completion of the I/O operation (no errors). Additional information on the success or failure of the I/O request in the status array is indicated the following table. The error codes follow the NT standard for error codes as well. The odd codes were selected as successful status as well such that users migrating from Windows NT would not need to modify their software if they tested for odd status. Note that the Subroutine CAMSG can be used to decode the returned error number into ASCII text. Even though the simple CAMAC routines build the lists for the user, the last four items are returned for all CAMAC routines. The symbolic name for the status array element is shown along with its decimal 1-based array index. Remember, arrays in C are 0-based, therefore you must subtract 1 from the given index to obtain the C array index.

STATUS ARRAY

0 ERR	**Error Status**: Contains the returned error code. An odd return status indicates a successful transfer. Any other value indicates an error or warning.
1 StaCSR	**Control and Status Register**: Contains the state of the 2962 Control and Status register. This is copied from the I/O status block. See the I/O status block for the 2962 device driver and the 2962 hardware manual for a more complete description.
2 StaERS	**Error Status Register**: Contains the state of the 2962 Error Status Register. This is copied from the I/O status block. See the I/O status block for the 2962 device driver and the 2962 hardware manual for a more complete description.
3 StaLCS	**List Status Register**: Field is zero and is reserved for future KSC use.
4 StaSum	A variable using bit 1 to indicate the sum of CAMAC NO-X responses and bit 0 to indicate the sum of CAMAC NO-Q responses for all the CAMAC operations in the Command List. If there were any CAMAC NO-Qs, bit zero of StaSum would be set and if there were any CAMAC NO-Xs, bit one of StaSum would be set.
5 StaCnt	A variable returning the number of words not transferred for the last Block Transfer operation. A zero will be returned if the last Block Transfer operation was successful or if there were no Block Transfers in the Command List.
6 StaLis	A variable returning the Fortran index into the CCL of the last command in the Command List that was executed by the driver.
7 StaDat	A variable returning the Fortran index into the Data Buffer of the last Data word read or written by the driver.
8 StaWC	A variable returning the total number of Word Count Buffer errors that occurred. This number can be greater than the number of Word Count Buffer records. Total number of word count errors (not supported on 2962).
9 StaQXE	A variable returning the total number of QXE Buffer errors that occurred. This number can be greater than the number of QXE Buffer records. Total number of QXE buffer errors (not supported on 2962).

3.2.8 Error Status Considerations

There are many places where status information is provided. For compatibility reasons, status information is translated to other existing error codes. However, sometimes this manipulation of the status occludes the real reason of the fault. NT, the 2962 driver, the API library, or the CAMAC library may all return error status.

The 2962 device driver returns its status via the I/O status block to the API level into the structure pointed to by the both the CAMAC library (CAOPEN) and the API library (KSC_INIT). Typically the NT status is returned in the same structure as well. The CAMAC library will attempt to translate this error code to one of the existing CAMAC error codes.

In summary, to acquire the most detailed status information, the user should call the KSC_PRINT_SYMBOLIC passing the address returned from CAOPEN or KSC_INIT. The CAMAC library can be called as function values. While trying to support both the existing CAMAC error codes and still trying to provide as much information as possible, the function value returned and the first word of the status array may be different in the event there is an error status to be returned. A test of odd on either error codes indicates success.

The error status array has been made consistent for all CAMAC library calls. This array is a nine long word array. Some of the entries are not populated for some of the CAMAC calls where a list is not used.

3.2.9 Asynchronous Event Handling (LAMS)

In many real-time applications it is necessary to handle asynchronous events such as events which occur outside the computer and sometimes outside of the CAMAC front-end. For example, an application may require notification when a discrete input from some device changes state, when some amount of data has been stored in a FIFO memory in a module, or when a transient recorder has completed recording a wave form. The LAM or Look-At-Me is the CAMAC mechanism for signaling of asynchronous events. The CAMAC LAM is delivered to the host computer system as a hardware interrupt.

In the computer, the application software must receive notification of the asynchronous event. The operating system mechanism for asynchronous event notification is the Asynchronous Procedure Call (APC). The CXLAM routine is provided to notify the CAMAC driver of the module and crate that will be generating LAMs and the operating system of the address of the routine to be dispatched when the event occurs.

The CXLAM routine with LAM- Types 2 and 3 are new with this release of the driver and is the preferred LAM handling mechanism. The CALAM routine with LAM_Types 0 and 1 continue to be supported for compatibility with previous releases. LAM_Types 2 and 3 are more powered and can handle most modules whose design conform to the IEEE 538 CAMAC Standard. LAM-Types 0 and 1 can only handle LAMs from modules that provide a single control command to clear LAM and disable LAM. Refer to Appendix E on Driver LAM Handling.

In developing software employing LAMs some special care needs to be observed:

1. LAMs typically signal asynchronous real-time events that in turn trigger execution of time critical application software.

2. The CAMAC library is written in the C language and is re-entrant so calls to the library may be made from both the APC routine and the main program.

3. The operating system can only handle a limited number of outstanding (undelivered) APCs at any given time. The delivery of the LAMs to the user process is done with the use of the DEMAND process and NT pipes. All LAMs that are expected to be processed must be configured by the Demand process. The Demand Process is responsible for enabling LAM recognition for any CAMAC crates that are to process LAMs. The actual enabling of a particular CAMAC device in a crate is the responsibility of the user.

4. For recurring LAMs, the demand process will queue LAM messages to the user process as long as the pipe is not full.

3.2.10 Error Codes

Error codes are documented in the appendix. Those errors denoted as "KSC_xxxxx" where xxxxx is a symbolic string are from the KSC API library or the NT device driver. The error codes of the form: "ERRnnn" where nnn is an integer are from the CAMAC library. Errors may also be returned by the NT itself. The function return value and the ERR entry of the status array should both be examined when the value of the function is even. Either may be used for success status (an odd status is successful).

3.2.11 Linker Requirements

All of the CAMAC and KSC API library routines are provided in the library: KGIAPI.LIB object library. Any applications that use the CAMAC or KSC API library routines will need to link to this library.

Include files necessary for application building are detailed in the Installation chapter of this document.

3.3 CAMAC Routines

3.3.1 cab16

Syntax

```
int cab16(      void  **hdlptr,
                short int  *c,
                short int  *n,
                short int  *a,
                short int  *f,
                short int  *mode,
                short int  *data,
                long  int  *dataln,
                int  errarr[]);
```

Purpose

The *cab16* function is used to execute a 16-bit block transfer write or read operation.

Description

The *cab16* function performs block transfer operations to or from a CAMAC module(s) utilizing 16-bit data words. For the 16-bit data transfers, only the lower 16-bits of the 24 bit CAMAC data word is used during the transfer. The array used to move data to or from the module must be longword aligned. Since the PCI bus is organized as 32-bit data words, the array for data must be aligned on a longword boundary for facilitating Direct Memory Access (DMA). If an odd number of 16-bit data words is to be transferred, the application software must allocate an additional 16-bit data entry in host memory to accommodate the re-alignment of data onto a longword boundary. When the 2962 executes a CAMAC block transfer operation with a transfer count specification that is odd, an additional 16-bit data word is sent to memory to force the longword alignment.

The *cab16* function supports all four types of block transfer operations. These four modes consist of Q-Ignore, Q-Stop, Q-Repeat and Q-Scan. Please refer to the *Transfer Mode* section of this manual for details on each operating mode.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
mode	Input	Type of CAMAC block transfer to perform. Please refer to *Transfer Mode* section of this manual for additional information.

Parameter Name	Direction	Description
data	Input/Output	CAMAC Write (Input) or Read (Output) data
dataln	Input	Requested number of CAMAC data 16-bit data words
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

The *mode* parameter in the *cab16* function is used to specify the CAMAC block transfer Q-mode and a specification as to the termination technique when a No-X condition occurs. The following table shows the available selections as *#defines* in the *kscuser.h* include file. Note that only one defined Q-mode can be specified for each block transfer.

#define	Description
QSTP	Selects the Q-Stop Block Transfer Mode
QIGN	Selects the Q-Ignore Block Transfer Mode
QRPT	Selects the Q-Repeat Block Transfer Mode
QSCN	Selects the Q-Scan Block Transfer Mode

Q-mode Block Transfer Selection

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR141 Data buffer not long word aligned.

ERR701 An invalid CAMAC sub-address (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.

ERR703 An invalid CAMAC block transfer type was found. The legal block transfer types are QSTP, QIGN, QRPT, and QSCN with corresponding values of 0, 8, 16, and 24, respectively.

ERR704 An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31.

ERR706 An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30.

ERR709 A CAMAC block transfer control operation was specified which is invalid. Only CAMAC Read or Write block transfers are allowed. The function code (F) for the block transfer was either between 8 and 15 inclusive or between 24 and 31 inclusive ($8 \leq F < 15$ or $24 \leq F \leq 31$).

ERR714 Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
```

```
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"

main()
{
  int     status;                          // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                            // Handle for operations
  int     errstat[STAMAX];                 // array with list of errors
  short n;                                 // slot
  short a;                                 // sub address
  short f;                                 // function
  short c;                                 // crate
  short qmode;                             // q mode for transfer
  short ShortWriteBuffer[8192];            // short write data buffer
  unsigned long TransferCount;             // transfer count for block

//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
exit(status);

  }
//
// Setup the parameters for the block transfer
//
  c=1;
  n=1;
  f=16;
  a=0;
  qmode = QSTP ;
  TransferCount = 100;
  status = cab16(&hdl, &c, &n, &a, &f, &qmode, ShortWriteBuffer,
&TransferCount,errstat);
  if (status & 1) != 1)
  {
        printf("****ERROR**** cab16\n");
        camsg(errstat);
        exit(status);
  }
//
// Close the device
//
```

```
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
            printf("****ERROR**** caclos\n");
            camsg(errstat);
            exit(status);

    }
}
```

3.3.2 cab24

Syntax

```
int cab24(      void  **hdlptr,
                short int  *c,
                short int  *n,
                short int  *a,
                short int  *f,
                short int  *mode,
                short int  *data,
                int  *dataln,
                int  errarr[]);
```

Purpose

The *cab24* function is used to execute a 24-bit block transfer write or read operation.

Description

The *cab24* function performs block transfer operations to or from a CAMAC module(s) utilizing 16-bit data words. For these 24-bit data transfers, the entire 24-bits of the 24-bit CAMAC data word are used during the transfer. The array used to move data to or from the module must be longword aligned. Since the PCI bus is organized as 32-bit data words, the array for data must be aligned on a longword boundary for facilitating Direct Memory Access (DMA). Due to the architecture of the 2962, each 24-bit CAMAC data word is contained in a single 32-bit PCI memory word. The additional 8-bits of the PCI memory data word are ignored. When CAMAC read operations are performed, the upper 8 bits of the 32-bit PCI memory word are padded with zeros.

The *cab24* function supports all four types of block transfer operations. These four modes consist of Q-Ignore, Q-Stop, Q-Repeat and Q-Scan. Please refer to the *Transfer Mode* section of this manual for details on each operating mode.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
mode	Input	Type of CAMAC block transfer to perform. Please refer to *Transfer Mode* section of this manual for additional information.
data	Input/Output	CAMAC Write (Input) or Read (Output) data
dataln	Input	Requested number of CAMAC data 16-bit data words
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

The *mode* parameter in the *cab24* function is used to specify the CAMAC block transfer Q-mode and the termination technique when a No-X condition occurs. The following table shows the available selections as *#defines* in the *kscuser.h* include file. Note that only one defined Q-mode can be specified for each block transfer.

#define	Description
QSTP	Selects the Q-Stop Block Transfer Mode
QIGN	Selects the Q-Ignore Block Transfer Mode
QRPT	Selects the Q-Repeat Block Transfer Mode
QSCN	Selects the Q-Scan Block Transfer Mode

Q-mode Block Transfer Selection

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR141 Data buffer not long word aligned.

ERR701 An invalid CAMAC sub-address (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.

ERR703 An invalid CAMAC block transfer type was found. The legal block transfer types are QSTP, QIGN, QRPT, and QSCN with corresponding values of 0, 8, 16, and 24, respectively.

ERR704 An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31.

ERR706 An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30.

ERR709 A CAMAC block transfer control operation was specified which is invalid. Only CAMAC Read or Write block transfers are allowed. The function code (F) for the block transfer was either between 8 and 15 inclusive or between 24 and 31 inclusive ($8 \leq F < 15$ or $24 \leq F \leq 31$).

ERR714 Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
```

```
#include "cmdlist.h"


main()
{
  int     status;                       // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                         // Handle for operations
  int     errstat[STAMAX];              // array with list of errors
  short n;                              //  slot
  short a;                              //  sub address
  short f;                              //  function
  short c;                              //  crate
  short qmode;                          //  q mode for transfer
  int LongWriteBuffer[8192];            // long write data buffer
  unsigned long TransferCount;          // transfer count for block

//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
exit(status);

  }
//
// Setup the parameters for the block transfer
//
  c=1;
  n=1;
  f=16;
  a=0;
  qmode = QSTP;
  TransferCount = 100;
  status = cab24(&hdl, &c, &n, &a, &f, &qmode, LongWriteBuffer,
&TransferCount,errstat);
  if (status & 1) != 1)
  {
        printf("****ERROR**** cab24\n");
        camsg(errstat);
        exit(status);
  }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
```

```
    {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

    }
}
```

3.3.3 caclos

Syntax

```
int caclos(     void  **hdlptr,
                int *error
```

Purpose

The *caclos* function is used to close the current CAMAC session with the 2962.

Description

The *caclos* function is used to unassign a channel from the CAMAC 2962 device and deallocate the per-process space for the controller. This routine is the opposite of the *caopen* routine that opens a device for communication. Once the *caclos* is executed, the device session is closed.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
error		Error Code

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR603 The *caclos* error is unknown.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"

main()
{
  int     status;                          // return status from the functions
  char    devname[] = "kpa00";
```

```
  int     *hdl;                    // Handle for operations
  int     errstat[STAMAX];         // array with list of errors
  int     lwdata;                  // long write data
  short n;                         // slot
  short a;                         // sub address
  short f;                         // function
  short c;                         // crate
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
exit(status);
}
//
// Setup the parameters for the single transfer
//
  c=1;
  n=1;
  f=16;
  a=0;
  lwdata = 0x112233;
  status = cam24(&hdl, &c, &n, &a, &f, &lwdata, errstat);
  if (status & 1) != 1)
  {
        printf("****ERROR**** cam24\n");
        camsg(errstat);
        exit(status);
  }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
   {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

   }
}
```

3.3.4 cactrl

Syntax

```
int cactrl(     void  **hdlptr,
                short int  *c,
                short int *func
                int  errarr[]);
```

Purpose

The *cactrl* function is used to execute various CAMAC Crate Controller functions.

Description

The *cactrl* function generates CAMAC crate-wide operations. These operations include the CAMAC Initialize (Z) cycle, the CAMAC Clear (C) cycle, and setting/clearing the CAMAC Inhibit (I) signal. These operations are addressed to the specified CAMAC Crate Controller by the 2962 with the station number set to 30 (N=30). All CAMAC Crate Controllers have an internal register accessible at N=30 for generating crate-wide operations.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
func	Input	Requested crate-wide operation. (see Note 1)
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

Note 1: There are 4 valid values that can be used with this command. These *#defines* are found in the *kscuser.h* include file and are listed below.

#define	Description
INIT	Execute a CAMAC Initialize (Z) cycle
CLEAR	Execute a CAMAC Clear(C) cycle
SETINH	Set the CAMAC Inhibit (I) signal
CLRINH	Clear the CAMAC Inhibit (I) signal

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR224 Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;                          // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                            // Handle for operations
  int     errstat[STAMAX];                 // array with list of errors
  short ctrlval;                           // cactrl value
  short c;                                 //  crate

//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
//
// Setup the parameters for the control call
//
  c=1;
  ctrlval=INIT;
  status = cactrl (&hdl, &c, &ctrlval, errstat);
  if (status & 1) != 1)
  {
        printf("****ERROR**** cactrl\n");
        camsg(errstat);
        exit(status);
  }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
```

```
    {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

    }
}
```

3.3.5 calam

Syntax

```
int calam(
        void **hdlptr,
        short int *c,
        short int *lam_id,
        short int *lam_type,
        short int *priority,
        void (*apc_addr)(),
        void *parm,
        short *clrN,
        short *clrA,
        short *clrF,
        short *dsbN,
        short *dsbA,
        short *dsbF,
        int *error);
```

Purpose

The *calam* function is used to register a LAM for subsequent asynchronous notification of an application program.

Description

The *calam* function requests the Demand Process to service LAMs for the LAM specified in the call to the function. The process of enabling a LAM to be serviced by the Demand Process is called *booking* a LAM. When the LAM pipe message is received, an Asynchronous Procedure Call (APC) is made to the Demand Process which disables the LAM and calls the user specified APC routine. Prior to executing the user APC routine, the Demand Process APC executes the CAMAC commands passed into the *calam* routine at the time is was booked. The Demand Process APC will execute either the Clear or Disable CAMAC command passed into the routine based on the setting of the *lam_type*. By setting the *lam_type* to a 0 (Type 0), the Demand Process will unbook the LAM and issue the disable CAMAC command to the specified module. When the *lam_type* is set to a 1 (Type 1), the Demand Process will leave the LAM booked and issue the clear CAMAC command to the specified module.

In general, it is up to the user application program to actually enable a module to generate a LAM. The LAM Mask Registers in the crate controller and other associated enables for the LAM are taken care of by the *calam* functions. The command to enable the LAM generation within a module should be placed in the application program after the *calam* function is used. The Demand Process will enable LAMs for a crate if they are not already enabled from a previous request to enable another LAM in the same crate.

If the LAM for a module in not enabled by the user application, a LAM that it generates will never be serviced. If the user application enables the modules LAM prior to calling the *calam* routine, the LAM

could be generated by the module prior to it being booked. This situation MUST be avoided as the results of this sequence may be erroneous.

Once the Demand Process has completed processing its portion of the LAM service, it passes control onto the user specified APC. The user's APC is called with 4 arguments that define specific information regarding the source of the LAM. This information contains the Station Number (N) of the device generating the LAM, the handle returned from the *caopen* routine of the parent program that called the *calam*, the chassis that generated the LAM, and the user specified parameter passed into the *calam* routine when it was booked. Please refer to the *Demands* section of this manual for additional information regarding the APC calling conventions.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
lam_id	Input	Specifies the Station Number (N) of the LAM to be booked
lam_type	Input	Specifies the type of LAM booking. Type 0 indicates an unbook and disable and a Type 1 indicates a remain booked and clear LAM
priority	Input	Not Supported. This parameter is for legacy parameter placeholding.
apc_addr	Input	This parameter specifies the address of the APC to be called once the LAM is generated.
parm	Input	This value is passed onto the APC once the LAM is serviced.
clrN	Input	Specifies the Station Number (N) to be accessed when a Type 1 LAM is serviced
clrA	Input	Specifies the Subaddress (A) to be accessed when a Type 1 LAM is serviced
clrF	Input	Specifies the Function (F) to be performed when a Type 1 LAM is serviced
dsbN	Input	Specifies the Station Number (N) to be accessed when a Type 0 LAM is serviced
dsbA	Input	Specifies the Subaddress (A) to be accessed when a Type 0 LAM is serviced
dsbF	Input	Specifies the Function (F) to be performed when a Type 0 LAM is serviced
error		Error code

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR714 Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"
//
// Global variable definitions....shared between main and APC routine
//
int   *hdl;                                    // Handle for operations
HANDLE hEvent;                                 // So APC can wake up mainline

void ApcRoutine (int *dmd_id,
                 struct KSC_handle *handle,
                 int chassis,
                 void *UserArg);


main()
{
 int   status;                                 // return status from the functions
 int   iStatus;
 int lpcnt=0;
 int UserParm;

 char     devname[] = "kpa00";
 int   errstat[STAMAX];                        // array with list of errors

 short n;                                      // slot
 short a;                                      // subaddress
 short f;                                      // function
 short c;                                      // crate
 short n_clr,n_dis;
 short a_clr,a_dis;
 short f_clr,f_dis;
 short    c_3291 = 1;                          // crate address for 3291
 short n_3291 = 1;                             // slot number for 3291
 short swdata;                                 // short write data
 short srdata;                                 // short read data
 short nLamType;
 short nLamPriority;
 short fix;
 fix = n_3291 - 1;

//
// Open the device
//

 status = caopen(&hdl, devname, errstat);
 if ((status & 1) == 0)
 {
  printf("CAOPEN, error opening device = %s\n", devname);
  camsg(errstat);
  exit(status);
 }
```

```
//
// Crate event flag & check status
//
 hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

 if (NULL == hEvent)
 {
  iStatus = GetLastError();
  printf("Error creating event object: 0x%x\n", iStatus);
  exit(iStatus);
 }

//
// Set up the APC for the LAM (Book the LAM)
//
  n_clr = n_3291;
  f_clr = 10;
  a_clr = 0;
  n_dis = n_3291;                          // filled in but not used for type 1
  f_dis = 24;
  a_dis = 0;

  nLamType = 1;                        // clear but remain booked
  nLamPriority = -1;                       // not used
  iStatus = calam(&hdl,                    // handle from caopen
            &c_3291,                       // crate with our 3291
            &n_3291,                       // slot containing 3291
            &nLamType,                     // forever let us process it
            &nLamPriority,                 // not used, but must be here
            &ApcRoutine,                 // APC to call
             &UserParm,                    // user parameter
             &n_clr,                       // station number for clear
             &a_clr,                       // subaddress for clear
             &f_clr,                       // function for clear
             &n_dis,                       // station number for disable
             &a_dis,                       // subaddress for disable
             &f_dis,                       // function code for disable
            errstat);                      // CAMAC status

 if ((iStatus & 1) == 0)
 {
  printf("Failure to book the 3291 LAM\n");
  camsg(errstat);
  exit(iStatus);
 }
while (TRUE)
 {
  iStatus = ResetEvent(hEvent);

  if (iStatus == FALSE)
  {
   iStatus = GetLastError();
   printf("Error Resetting Event: 0x%x\n", iStatus);
   exit(iStatus);
```

```
    }

//
// Generate a LAM on the 3291
//
   c=1;
   n = n_3291;
   f=14;
   a=0;
   iStatus = cam16(&hdl,                    // handle from caopen
                   &c,                      // crate for 3291
                   &n,                      // slot number for 3291
                   &a,                      // subaddress
                   &f,                      // function code
                   &srdata,                 // data space
                   errstat);                // error array
//
// Waiting for a LAM
//
    printf("Waiting for a LAM\n");
    iStatus = WaitForSingleObject(hEvent, INFINITE);

    if (WAIT_OBJECT_0 != iStatus)
    {
     printf("Error in WaitForSingleObject: 0x%x\n", iStatus);
                        cwait();

     exit(iStatus);
    }
   printf("Object Received \n");
  }
}                       // end of main
// APC procedure.
//
// This routine is called when a LAM is received from the demand process
// as a result of the 3291 getting a LAM.
//
void ApcRoutine(int *dmd_id,
            struct KSC_handle *handle,
            int chassis,
            void *user_arg)
{
//
// Set the event so mainline will continue
//
 printf("APC triggered\n");
 SetEvent(hEvent);

}
```

3.3.6 cam16

Syntax

```
int cam16(      void  **hdlptr,
                short int  *c,
                short int  *n,
                short int  *a,
                short int  *f,
                short int  *data,
                int  errarr[]);
```

Purpose

The *cam16* function is used to execute a 16-bit single transfer write, read or control operation.

Description

The *cam16* function performs a single transfer operation to the CAMAC crate. This command accommodates write, read and control operations. All data words moved using this command are 16-bits in width. Therefore, only the lower 16-bits of the 24-bit CAMAC data word can be accessed using this function.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
data	Input/Output	This parameter either specifies CAMAC write data for write operations to the CAMAC crate or read data returned from executing a CAMAC read operation.
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR701 — An invalid CAMAC subaddress (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.

ERR704 — An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31.

ERR706	An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30.
ERR714	Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;                   // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                     // Handle for operations
  int     errstat[STAMAX];          // array with list of errors
  short n;                          // slot
  short a;                          // sub address
  short f;                          // function
  short c;                          // crate
  short swdata;                     // short write data
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
//
// Setup the parameters for the single transfer
//
  c=1;
  n=1;
  f=16;
  a=0;
  swdata = 0x1122;
  status = cam16(&hdl, &c, &n, &a, &f, swdata,errstat);
  if (status & 1) != 1)
```

```
    {
        printf("****ERROR**** cam16\n");
        camsg(errstat);
        exit(status);
    }
//
// Close the device
//
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

    }
}
```

3.3.7 cam24

Syntax

```
int cam24(      void  **hdlptr,
                short int  *c,
                short int  *n,
                short int  *a,
                short int  *f,
                int  *data,
                int  errarr[]);
```

Purpose

The *cam24* function is used to execute a 24-bit single transfer write, read or control operation.

Description

The *cam24* function performs a single transfer operation to the CAMAC crate. This command accommodates write, read and control operations. All data words moved using this command are 24-bits in width.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
data	Input/Output	This parameter either specifies CAMAC write data for write operations to the CAMAC crate or read data returned from executing a CAMAC read operation.
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR701 — An invalid CAMAC sub-address (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.

ERR704 — An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31.

ERR706	An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30.
ERR714	Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;                   // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                     // Handle for operations
  int     errstat[STAMAX];          // array with list of errors
  short n;                          //  slot
  short a;                          //  sub address
  short f;                          //  function
  short c;                          //  crate
  int lwdata;                       // long write data
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
exit(status);

  }
//
// Setup the parameters for the single transfer
//
  c=1;
  n=1;
  f=16;
  a=0;
  lwdata = 0x112233;
  status = cam24(&hdl, &c, &n, &a, &f, &lwdata, errstat);
  if (status & 1) != 1)
```

```
    {
        printf("****ERROR**** cam24\n");
        camsg(errstat);
        exit(status);
    }
//
// Close the device
//
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

    }
}
```

3.3.8 camsg

Syntax

int camsg(int *error);

Purpose

The *camsg* function is used to translate error codes received from various CAMAC API functions and print a message to the standard output device.

Description

The *camsg* function can be called whenever an error is detected as a result of executing a CAMAC API function. An error code returned from the API functions can be printed to the standard output device. The printed error may be as a result of an error from the device driver, the API, or from the operating system.

Parameters

Parameter Name	Direction	Description
error	Input	Completion status or error code returned from a previous function call to a CAMAC API routine.

Return Values

For a comprehensive list, please refer to the *Error Codes* section of this manual.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;             // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;               // Handle for operations
  int     errstat[STAMAX];    // array with list of errors

//
```

```
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
//
// Close the device
//
  status = caclos (&hdl, errstat);
  if ((status & 1) != 1)
  {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

  }
}
```

3.3.9 caopen

Syntax

```
int caopen(     void  **hdlptr,
                char *device,
                int     *error
```

Purpose

The *caopen* function opens a session with the 2962.

Description

The *caopen* function assigns a channel to a device and initializes the CAMAC library so that subsequent CAMAC operations may be executed. This function must be called at the start of a program before attempting any CAMAC operations. Once the channel has been open to the device, it should not be re-opened until the channel is unassigned by a call to the *caclos* function.

The *caopen* function initializes the handle parameter. The handle is a pointer to a process and controller specific region that has been allocated for the user application. The *caopen* should be called as part of the process's initialization. The handle obtained as a result of the *caopen* function should be passed to any other CAMAC API function requiring use of the handle. The *caclos* function is used to close the channel and release this per process handle and controller space.

The second parameter in this function call is the name of the CAMAC driver associated with the 2962. In order to open a valid connection to the driver, the device name of *kpa00* must be used.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
device	Input	Character string containing the name of the device to be opened. The device name for the 2962 is *kpa00*.
error	Output	Returned value

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

KSC_BAD_ARG One or more of the arguments is not readable or writeable.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
```

```
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;                       // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                         // Handle for operations
  int     errstat[STAMAX];              // array with list of errors

//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
//
// Close the device
//
  status = caclos (&hdl, errstat);
  if ((status & 1) != 1)
  {
        printf("****ERROR**** caclos\n");
        camsg(errstat);
        exit(status);

  }
}
```

3.3.10 ccstat

Syntax

```
int ccstat(     void **hdlptr,
                short int *c,
                int data[]
                int errarr[]);
```

Purpose

The *ccstat* function is used to retrieve the current status of the 3922 Crate Controller.

Description

The *ccstat* function performs a read operation to the 3922 Crate Controller to determine its current status. The status information returned includes the state of the CAMAC Inhibit (I) line, the state of the Service Request Enable (LAM Enable) signal, the current LAM pattern. And the entire 3922 Control/Status Register contents. As a result of the *ccstat* function, the 2962 executes various CAMAC commands directed at Station Number (N) 30 of the target crate. The N=30 commands are internal operations directed at the 3922. Therefore, no CAMAC dataway cycles occur as a result of this command.

The third argument in the *ccstat* function returns a pointer to four 32-bit words. The contents of these words are described below.

Word 1

Bit 31 ----------Bit 1	Bit 0
0	INH

Word 2

Bit 31 ----------Bit 1	Bit 0
0	SRR ENA

Word 3

Bit 31----------Bit 24	Bit 23----------Bit 0
0	LAM Status 24 through 1

Word 4

Bit 15	Bit 14	Bit 13	Bit 12-10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5-3	Bit 2	Bit 1-0
SLP	RSVD	OFF LINE	0	L24	SRR ENA	RSVD	RD INH	0	INH	0

INH – Read as a 1 if the 3922 is asserting the CAMAC Inhibit (I) signal.
SRR ENA – Read back as a 1 if the Service Request on the 3922 is enabled.
LAM Status 24 – 1 is the LAM Status bits reflecting the current state of the CAMAC LAM signals.
RD INH – Read back as a 1 if the CAMAC Inhibit (I) signal is asserted by any module.
RSVD are reserved bits.

L24 – Read back as a one when Internal LAM 24 is set.
OFF LINE – Read back as a 1 when the 3922 front panel switch is in the Off-Line position.
SLP – Read back as a one is a Selected LAM is present in the crate.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
data	Output	Array of four 32-bit words reflecting the current state of the 3922 Crate Controller.
errarr	Output	Returned 10-element status array. Please refer to *Status Array* section of this manual for additional information.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR141	Data buffer not long word aligned.
ERR601	An invalid channel number was specified. The passed handle is invalid.
ERR701	An invalid CAMAC sub-address (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.
ERR704	An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31.
ERR706	An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30.
ERR714	Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int    status;                    // return status from the functions
  char   devname[] = "kpa00";
  int    *hdl;                      // Handle for operations
```

```
    int      errstat[STAMAX];          // array with list of errors
    int CrateStatus[4];                // ccstat returns
    short c;                           //  crate

//
// Open the device
//
    status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
    if ((status & 1) == 0)
    {
            printf("CAOPEN, error opening device = %s\n", devname);
            camsg(errstat);
            exit(status);

    }
//
// Get the current status of the 3922 crate controller
//
    status = ccstat (&hdl, &c, CrateStatus, errstat);
    if (status & 1) != 1)
    {
            printf("****ERROR**** ccstat\n");
            camsg(errstat);
            exit(status);
    }
//
// Close the device
//
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
            printf("****ERROR**** caclos\n");
            camsg(errstat);
            exit(status);

    }
}
```

3.3.11 cxlam

Syntax

```
int cxlam(
        void **hdlptr,
        short int *c,
        short int *lam_id,
        short int *lam_type,
        short int *priority,
        void (*apc_addr)(int  *,  struct KSC_handle  *, int,  void  *),
        int *error);
```

Purpose

The *cxlam* function is used to register a LAM for subsequent asynchronous notification of an application program. This function call is similar in nature to the *calam* function call, except that it performs a more sophisticated LAM processing algorithm in order to service an asynchronous notification. Please refer to the *Demand* section of this manual for additional information.

Description

The *calam* function requests the Demand Process to service LAMs for the LAM specified in the call to the function. The process of enabling a LAM to be serviced by the Demand Process is called *booking* a LAM. When the LAM pipe message is received, an Asynchronous Procedure Call (APC) is made to the Demand Process which disables the LAM and calls the user specified APC routine. Prior to executing the user's APC routine, the Demand Process APC executes a specific sequence of CAMAC commands in order to clear the pending LAM.

Once the Demand Process receives notification of a LAM, it verifies the crate address that generated the LAM. Once this is determined, the Demand Process reads the LAM Status Register of the 3922 generating that generated the LAM. The LAM Status Register is a 24-bit register located on the 3922 that contains a bit that corresponds to each station number within the crate. With this information, the Demand Process can determine which module in the crate is requesting service.

The Demand Process will try to clear the source of the LAM within a module by first using the selective clear operation to the module generating the LAM. The selective clear operation performed is an F(23)A(12) command using the LAM pattern as the data for the selective clear. If this does not clear the LAM, then the Demand Process executes an F(11)A(12) command to clear the LAM. If this is not successful, an F(10)A(0) is then tried. As a last resort, an F(10)A(*i*) command is executed where *i* corresponds to the bit positions set in the LAM Status Register.

In general, it is up to the user application program to actually enable a module to generate a LAM. The LAM Mask Registers and other associated enables for the LAM are taken care of by the *cxlam* functions. The command to enable the LAM generation within a module should be placed in the application program after the *cxlam* function is used. The Demand Process will enable LAMs for a crate if they are not already enabled from a previous request to enable another LAM in the same crate.

If the LAM for a module in not enabled by the user application, a LAM that it generates will never be serviced. If the user application enables the modules LAM prior to calling the *cxlam* routine, the LAM could be generated by the module prior to it being booked. This situation MUST be avoided as the results of this sequence may be erroneous.

Parameters

Parameter Name	Direction	Description
hdlptr	Input	Handle returned by caopen function
c	Input	Address of the chassis to be accessed
lam_id	Input	Specifies the Station Number (N) of the LAM to be booked
lam_type	Input	Specifies the type of LAM booking. Type 0 indicates an unbook and disable and a Type 1 indicates a remain booked and clear LAM
priority	Input	Not Supported. This parameter is for legacy support
apc_addr	Input	This parameter specifies the address of the APC to be called once the LAM is generated.
error	Output	Error Return

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR714 Illegal CAMAC crate number.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"
//
// Global variable definitions....shared between main and APC routine
//
int   *hdl;                                   // Handle for operations
HANDLE hEvent;                                      // So APC can wake up mainline

void ApcRoutine (int *dmd_id,
                 struct KSC_handle *handle,
                 int chassis,
                 void *UserArg);
main()
{
```

```
  int   status;                          // return status from the functions
  int   iStatus;
  char     devname[] = "kpa00";
  int   errstat[STAMAX];                 // array with list of errors

  short n;                          // slot
  short a;                          // sub address
  short f;                          // function
  short c;                          // crate
  short c_3296 = 1;                      // crate address for 3296
  short n_3296 = 6;                      // slot number for 3296
  short swdata;                          // short write data
  short srdata;                          // short read data
  short nLamType;
  short nLamPriority;

//
// Open the device
//
  status = caopen(&hdl, devname, errstat);
  if ((status & 1) == 0)
  {
   printf("CAOPEN, error opening device = %s\n", devname);
   camsg(errstat);
   exit(status);
  }
//
// Create event flag for notification of main process
//
  hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
//
// Check status of event creation
//
  if (NULL == hEvent)
  {
   iStatus = GetLastError();
   printf("Error creating event object: 0x%x\n", iStatus);
   exit(iStatus);
  }
//
// Set up the APC for the LAM (Book the LAM)
//
  nLamType = 3;
  nLamPriority = -1;
  iStatus = cxlam(&hdl,                          // handle from caopen
            &c_3296,                             // crate with our 3296
            &n_3296,                             // slot containing 3296
            &nLamType,                           // forever let us process it
            &nLamPriority,                       // not used, but must be here
            &ApcRoutine,                    // APC to call
            errstat);                            // CAMAC status

  if ((iStatus & 1) == 0)
  {
```

```
    printf("Failure to book the 3296 LAM\n");
    camsg(errstat);
   exit(iStatus);
  }
 //
 // Enable the LS switch on the 3296 to generate a LAM
 //
    c=1;
    n = n_3296;
    f=26;
    a=0;

  iStatus = cam16(&hdl,                                    // handle from caopen
              &c,
              &n,
              &a,
              &f,
              &srdata,
              errstat);


  while (TRUE)
  {
   iStatus = ResetEvent(hEvent);

   if (iStatus == FALSE)
   {
    iStatus = GetLastError();
    printf("Error Resetting Event: 0x%x\n", iStatus);
                        cwait();
    exit(iStatus);
   }
//
// Waiting for a LAM
//
   printf("Waiting for a LAM\n");
   iStatus = WaitForSingleObject(hEvent, INFINITE);

   if (WAIT_OBJECT_0 != iStatus)
   {
    printf("Error in WaitForSingleObject: 0x%x\n", iStatus);
    exit(iStatus);
   }
  printf("Object Received  #%d\n",++lpcnt);
 }
}                   // end of main

// APC procedure.
//
// This routine is called when a LAM is received from the demand process
// as a result of the 3296 getting a LAM through its front panel switch.
//
void ApcRoutine(int *dmd_id,
          struct KSC_handle *handle,
```

```
            int chassis,
            void *user_arg)
{
//
// Set the event so mainline will continue
//
 printf("APC triggered\n");
 SetEvent(hEvent);

}
```

3.3.12 camlookupmsg

Syntax

```
void camlookupmsg (int *returnCode,
                   char *szSeverityBuffer,
                   int sizeSeverityBuffer,
                   char *szNameBuffer,
                   int sizeNameBuffer,
                   char *szDescBuffer,
                   int sizeDescBuffer);
```

Purpose

The camlookupmsg function is used to lookup descriptive strings associated with an error code.

Description

The camlookupmsg function takes an error code returned from other API functions and populates 3 user-supplied buffers with strings that describe the severity of the return code, the name of the return code, and a description of the return code. The user also specifies the size of each of the supplied buffers; this call will truncate any message larger than the specified size.

This function is similar to function *camsg*, that passes the same information to standard output.

Parameters

Parameter Name	Direction	Description
returnCode	Input	Return Code from previous API call
szSeverityBuffer	Input/Output	User Buffer to hold the Severity Description
sizeSeverityBuffer	Input	Size of the Severity Buffer
szNameBuffer	Input/Output	User Buffer to hold the Name Description
sizeNameBuffer	Input	Size of the Name Buffer
szDescBuffer	Input/Output	User Buffer to hold the Return Code Description
sizeDescBuffer	Input	Size of the Return Code Description Buffer

For each of the three return strings (severity, name, and description), the caller supplies a buffer to be filled in, and the size of the buffer. Return code strings that exceed the specified size of the buffer will be truncated. It is valid to pass a NULL for any of the 3 buffers; any buffer specified as NULL will be ignored, and no string will be returned for that category.

Return Values

None

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;                    // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                      // Handle for operations
  int     errstat[STAMAX];           // array with list of errors
  char severity[256];
  char name[256];
  char description[256];
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        camlookupmsg (&status,
                      severity,
                      sizeof(severity),
                      name,
                      sizeof(name),
                      description,
                      sizeof(description));
        printf("CAOPEN failure:\n");
        printf("\tName %s\n",name);
        printf("\tSeverity %s\n",severity);
        printf("\tDescription %s\n",description);
        exit(status);
 }
```

3.4 CAMAC List Generation Routines

This chapter describes the routines provided that will allow the user to build CAMAC command lists (CCL). The CAMAC command list provides an efficient mechanism to predetermine a sequence of CAMAC operations to be performed and executed with a single function call. This functionality results in an increase in performance since the application software does not have to make as many calls to the CAMAC driver or the operating system.

The CAMAC command list generated by the application software must be unidirectional. This means that all data transfers must be either transfer data to the CAMAC crate or from the CAMAC crate, but not both. There is one exception to this rule, and that is the use of the Single Inline Write (*cainaf*) list instruction. This instruction can be embedded in either a write list or a read list. The reason that this instruction is allowed to be embedded in a list is that the associated write data for the command is contained in the list itself. Therefore, a transfer of data from the write or read buffer is not required in order to obtain the write data.

The 2915 requires that block transfers return multiples of thirty-two bit long words. Therefore, if a user does a block transfer of five sixteen bit words, the list building routines will also store an instruction that will return an additional sixteen zero bits to round the transfer up to a long word boundary. All data and command list buffers must be long word aligned, even if their data type is a sixteen-bit integer.

The following table summarizes the available list building and support functions.

List Building Function	Function Category	Description
cainit	Initialization	This function is used to initialize a CAMAC command list prior to adding the instruction to be performed once the list is executed.
cablk	Command	This function adds a CAMAC block transfer operation to the CAMAC command list.
cainaf	Command	This function adds a single inline write CAMAC transfer operation to the CAMAC command list. This command does not transfer data from the data buffer, but embeds the write data in the list.
canaf	Command	This function adds a single CAMAC transfer operation to the CAMAC command list. This command does transfer one 16 or 32-bit data word from the specified buffer.
cahalt	Command	This function places the end-of-list marker in the CAMAC command list. The CAMAC command list processor executes the elements contained in a list until this special halt command is encountered.
caexew	Execute	This function executes a preloaded CAMAC command list and waits until the requested operation is complete.
caexec	Execute	This function executes a preloaded CAMAC command list and does not wait until the operation is complete. Instead, this function requires the use of an event flag to communicate completion information to the calling process.

3.4.1 cablk

Syntax

```
long int cablk(   struct s_header *header,
                  short *c,
                  short *n,
                  short *a,
                  short *f,
                  short *mode,
                  int *datcnt,
                  int *datind,
                  int *error
               );
```

Purpose

The *cablk* function adds a command to the CAMAC command list which when executes results in a CAMAC block transfer operation.

Description

The *cablk* function performs a block transfer operation to or from a CAMAC module(s) utilizing either 16-bit or 24-bit data words. A portion of the *mode* parameter for this function is used to indicate the CAMAC data word size. For the 16-bit data transfers, only the lower 16-bits of the 24 bit CAMAC data word are used during the transfer.

The *cablk* function supports all four types of block transfer operations. These four modes consist of Q-Ignore, Q-Stop,Q-Repeat and Q-Scan. Please referto the *Transfer Mode* section of this manual for details on each operating mode. A portion of the *mode* parameter for this function is used to indicate the block transfer-operting mode.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
mode	Input	Type of CAMAC block transfer to perform. Please refer to *Transfer Mode* section of this manual for additional information. (See below)
datcnt	Input	Number of CAMAC operations to perform

Parameter Name	Direction	Description
datind	Output	This parameter is returned with the index in to the data buffer marking the starting location for the block of data used for the operation.
error	Output	Returned error code

The *mode* parameter in the *cablk* function is used to specify the CAMAC block transfer Q-mode, the CAMAC data word size, and a specification as to the termination technique when a No-X condition occurs. The following table shows the available selections as *#defines* in the *kscuser.h* include file. Note that only one defined Q-mode or word size can be specified for each block transfer.

#define	Description
QSTP	Selects the Q-Stop Block Transfer Mode
QIGN	Selects the Q-Ignore Block Transfer Mode
QRPT	Selects the Q-Repeat Block Transfer Mode
QSCN	Selects the Q-Scan Block Transfer Mode

Q-mode Block Transfer Selection

#define	Description
WTS16	Selects 16-bit CAMAC Data Word Size
WTS24	Selects 24-bit CAMAC Data Word Size

CAMAC Data Word Size Selection

#define	Description
AD	Inclusion of the *AD* in the mode description causes the CAMAC command processor to ignore No-X conditions during processing.

CAMAC Data Word Size Selection

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR702 Invalid Mode specification

ERR701 An invalid CAMAC sub-address (A) was found. The CAMAC subaddress was either less than 0 or greater than 15.

ERR703 An invalid CAMAC block transfer type was found. The legal block transfer types are QSTP, QIGN, QRPT, and QSCN with corresponding values of 0, 8, 16, and 24, respectively.

ERR712 The CAMAC command list is not large enough to hold all the commands.

ERR715 Direction error, the CAMAC command list should be unidirectional.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;           // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                                   // Handle for operations
  int     errstat;                                // array with list of errors
  short n;                                        // slot
  short a;                                        // sub address
  short f;                                        // function
  short c;                                        // crate
  short qmode;                                    // q mode for transfer
  int LongReadBuffer[8192];                       // short write data buffer
  unsigned long TransferCount;                    // transfer count for block
  int camac_list[8192];                           // list for camac processing
  int DataIndex;                                  // data index
  int listmax;                                    // max size of list
  int datamax;                                    // max size of data
  int     indicator;
  int     zero=0;                                 // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

  n=1;
  f=16;
  a=0;
  qmode = QIGN | WTS24 ;
```

```
  status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax,
                  &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
  }
  c=1;
  n=1;
  f=0;
  a=0;
  qmode = QIGN | WTS24 ;
  TransferCount = 100;
  status = cablk (&header, &c, &n, &a, &f, &qmode, &TransferCount, &DataIndex, errstat);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cablk\n");
   camsg(errstat);
  }
  status = cahalt(&header, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cahalt\n");
   camsg(errstat);
  }
  status = caexew(&header, &hdl, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** caexew\n");
   camsg(errstat);
   exit(status);
  }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
   {
    printf("****ERROR**** caclos\n");
    camsg(errstat);
    exit(status);
   }
}
```

3.4.2 caexec

Syntax

```
int caexec(    struct s_header *header,
               void  **hdl,
               int *error,
               HANDLE event
           );
```

Purpose

The *caexec* function loads and executes a CAMAC command list without waiting for the routine to complete. An event flag must be used with this function in order to provide a notification mechanism to the main application on completion.

Description

The *caexec* function executes a CAMAC command list built using the CAMAC list building routines. Control is returned to the user process after the operation is queued to the driver. The user application must then check the *event* flag to determine when the requested operation is complete.

The execution of a CAMAC command list is beneficial when an application program needs to perform computations or other activity while the command list is executed.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
hdlptr	Input	Handle returned by caopen function
error	Output	Error Code
event	Input	Event to be signaled when the operation is complete.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR143 The CAMAC header is not initialized.

ERR144 The CAMAC header is initialized, but not correctly.

ERR601 An invalid channel number is specified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"

HANDLE hEvent;                                  // event for caexe (w/o wait)
main()
{
  int     status;           // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                                 // Handle for operations
  int     errstat[STAMAX];                      // array with list of errors
  short n;                                      //  slot
  short a;                                      //  sub address
  short f;                                      //  function
  short c;                                      //  crate
  short qmode;                                  //  q mode for transfer
  int LongReadBuffer[8192];                     // short write data buffer
  unsigned long TransferCount;                  // transfer count for block
  int camac_list[8192];                         // list for camac processing
  int DataIndex;                                // data index
  int listmax;                                  // max size of list
  int datamax;                                  // max size of data
  int     indicator;
  int     zero=0;                               // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);      // create event for done indication
  status = ResetEvent(hEvent);

  if (status == FALSE)
  {
   status = GetLastError();
   printf("Error Resetting Event: 0x%x\n", status);
```

```
}
if (NULL == hEvent)
{
 status = GetLastError();
 printf("Error creating event object: 0x%x\n", status);
}

 listmax = 1024;
 datamax = 1024;

 n=1;
 f=16;
 a=0;
 qmode = QIGN | WTS24 ;
 status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
                 &zero, &zero, &zero, &zero, errarr);
 if ((status & 1) != 1)
 {
   printf("****ERROR**** cainit\n");
   camsg(errstat);
 }
 c=1;
 n=1;
 f=0;
 a=0;
 qmode = QIGN | WTS24 ;
 TransferCount = 100 ;
 status = cablk (&header, &c, &n, &a, &f, &qmode, &TransferCount, &DataIndex, errarr);
 if ((status & 1) != 1)
 {
  printf("****ERROR**** cablk\n");
  camsg(errstat);
 }
 status = cahalt(&header, errarr);
 if ((status & 1) != 1)
 {
  printf("****ERROR**** cahalt\n");
  camsg(errstat);
 }
 status = caexec(&header, &hdl, errarr, hEvent);
 if ((status & 1) != 1)
 {
  printf("****ERROR**** caexec\n");
  camsg(errstat);
  exit(status);
 }
 status = WaitForSingleObject(hEvent, INFINITE);

 if (WAIT_OBJECT_0 != status)
 {
   printf("Error in WaitForSingleObject: 0x%x\n", status);
 }
```

```
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
   {
    printf("****ERROR**** caclos\n");
    camsg(errstat);
    exit(status);
  }
}
```

3.4.3 caexew

Syntax

```
int caexew(    struct s_header *header,
               void  **hdl,
               int *error,
          );
```

Purpose

The *caexew* function loads and executes a CAMAC command list and waits for the routine to complete. If processing needs to continue while the CAMAC command list is being processed, use the *caexec* function that incorporates use of event notification for completion indication.

Description

The *caexew* function executes a CAMAC command list built using the CAMAC list building routines. Control is not returned to the user process until the operation is complete.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
hdlptr	Input	Handle returned by caopen function
error	Output	Error Code

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR143	The CAMAC header is not initialized.
ERR144	The CAMAC header is initialized, but not correctly.
ERR601	An invalid channel number is specified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
```

```
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int     status;           // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                                   // Handle for operations
  int     errstat[STAMAX];                        // array with list of errors
  short n;                                        // slot
  short a;                                        // sub address
  short f;                                        // function
  short c;                                        // crate
  short qmode;                                    // q mode for transfer
  int LongReadBuffer[8192];                       // short write data buffer
  unsigned long TransferCount;                    // transfer count for block
  int camac_list[8192];                           // list for camac processing
  int DataIndex;                                  // data index
  int listmax;                                    // max size of list
  int datamax;                                    // max size of data
  int     indicator;
  int     zero=0;                                 // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

  n=1;
  f=16;
  a=0;
  qmode = QIGN | WTS24 ;
  status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
                  &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
```

```
    }
    c=1;
    n=1;
    f=0;
    a=0;
    qmode = QIGN | WTS24 ;
    TransferCount = 100 ;
    status = cablk (&header, &c, &n, &a, &f, &qmode, &TransferCount, &DataIndex, errarr);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** cablk\n");
     camsg(errstat);
    }
    status = cahalt(&header, errarr);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** cahalt\n");
     camsg(errstat);
    }
    status = caexew(&header, &hdl, errarr);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** caexew\n");
     camsg(errstat);
     exit(status);
    }
//
// Close the device
//
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** caclos\n");
     camsg(errstat);
     exit(status);
    }
}
```

3.4.4 cahalt

Syntax

```
int cahalt(   struct s_header *header,
              int   *error
          );
```

Purpose

The *cahalt* function adds a command to the CAMAC command list that marks the end of the CAMAC list. This function must always be called in order to provide a terminating entry in the list.

Description

The *cahalt* function adds a command to the CAMAC command list. This command is used to indicate the termination of a CAMAC command list. The CAMAC command processor executes list instructions until this halt is encountered.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
error	Output	Error Code

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR143 The CAMAC header is not initialized.

ERR144 The CAMAC header is initialized, but not correctly.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"
```

```
main()
{
  int     status;           // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                                   // Handle for operations
  int     errstat[STAMAX];                        // array with list of errors
  short n;                                        //  slot
  short a;                                        //  sub address
  short f;                                        //  function
  short c;                                        //  crate
  short qmode;                                    //  q mode for transfer
  int LongReadBuffer[8192];                       // short write data buffer
  unsigned long TransferCount;                    // transfer count for block
  int camac_list[8192];                           // list for camac processing
  int DataIndex;                                  // data index
  int listmax;                                    // max size of list
  int datamax;                                    // max size of data
  int     indicator;
  int     zero=0;                                 // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

  n=1;
  f=16;
  a=0;
  qmode = QIGN | WTS24 ;
  status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
              &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
  }
  c=1;
  n=1;
  f=0;
```

```
  a=0;
  qmode = QIGN | WTS24 ;
  TransferCount = 100 ;
  status = cablk (&header, &c, &n, &a, &f, &qmode, &TransferCount, &DataIndex, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cablk\n");
   camsg(errstat);
  }
  status = cahalt(&header, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cahalt\n");
   camsg(errstat);
  }
  status = caexew(&header, &hdl, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** caexew\n");
   camsg(errstat);
   exit(status);
  }
//
// Close the device
//
  status = caclos (&hdl, errstat);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** caclos\n");
   camsg(errstat);
   exit(status);
  }
}
```

3.4.5 cainaf

Syntax

```
int cainaf(   struct s_header *header,
              short           *c,
              short           *n,
              short           *a,
              short           *f,
              short           *mode,
              int             *data,
              int             *error
          );
```

Purpose

The *cainaf* function adds a command to the CAMAC command list which when executed results in single CAMAC write operation. The data for this operation is included in the CAMAC command list and not extracted from the data buffer.

Description

The *cainaf* function adds a single inline write operation to the CAMAC command list. The write data for the *cainaf* instruction is contained in the CAMAC command list. Therefore, this instruction does not require data transfer from the data buffer for the list. This instruction is beneficial for setting up CAMAC module parameters that do not vary from list execution to list execution. Since this command does not transfer any data from the list data buffer, one can embed this instruction in either a write CAMAC command list or a read CAMAC command list.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
c	Input	Address of the chassis to be accessed
n	Input	Slot number of the module to be accessed
a	Input	Subaddress within the module to be accessed
f	Input	Function code to be performed
mode	Input	Type of CAMAC operation to perform. Please refer to *Transfer Mode* section of this manual for additional information. (See below)
data	Input	32-bit word reserved for specification of the CAMAC write data embedded in the list.
error	Output	Error Code

The *mode* parameter in the *cainaf* function is used to specify the CAMAC transfer Q-mode, the CAMAC data word size, and the termination technique used when a No-X condition occurs. Even though the *cainaf* only executes a single write operation, the CAMAC Q and X returns are evaluated and continued list processing is based on their values as it relates to the selected Q-mode. The following table shows the

available selections as *#defines* in the *kscuser.h* include file. Note that only one defined Q-mode or word size can be specified for each transfer.

#define	Description
QSTP	Selects the Q-Stop Block Transfer Mode
QIGN	Selects the Q-Ignore Block Transfer Mode
QRPT	Selects the Q-Repeat Block Transfer Mode
QSCN	Selects the Q-Scan Block Transfer Mode

Q-mode Transfer Selection

#define	Description
WTS16	Selects 16-bit CAMAC Data Word Size
WTS24	Selects 24-bit CAMAC Data Word Size

CAMAC Data Word Size Selection

#define	Description
AD	Inclusion of the *AD* in the mode description causes the CAMAC command processor to ignore No-X conditions during processing.

CAMAC Data Word Size Selection

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR143 The CAMAC header is not initialized.

ERR144 The CAMAC header is initialized, but not correctly.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int    status;          // return status from the functions
  char   devname[] = "kpa00";
```

```
  int     *hdl;                          // Handle for operations
  int     errstat[STAMAX];               // array with list of errors
  short n;                               // slot
  short a;                               // sub address
  short f;                               // function
  short c;                               // crate
  short qmode;                           // q mode for transfer
  int lwdata;                            // write data
  int camac_list[8192];                  // list for camac processing
  int DataIndex;                         // data index
  int listmax;                           // max size of list
  int datamax;                           // max size of data
  int     indicator;
  int     zero=0;                        // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
                &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
  }
  c=1;
  n=1;
  f=16;
  a=0;
  qmode = QIGN | WTS24 ;
  lwdata = 0x112233;
  status = cainaf (&header, &c, &n, &a, &f, &qmode, &lwdata, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cainaf\n");
   camsg(errstat);
  }
  status = cahalt(&header, errarr);
  if ((status & 1) != 1)
```

```
    {
     printf("****ERROR**** cahalt\n");
     camsg(errstat);
    }
    status = caexew(&header, &hdl, errarr);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** caexew\n");
     camsg(errstat);
     exit(status);
    }
//
// Close the device
//
    status = caclos (&hdl, errstat);
    if ((status & 1) != 1)
    {
     printf("****ERROR**** caclos\n");
     camsg(errstat);
     exit(status);
    }
}
```

3.4.6 cainit

Syntax

```
int cainit(    struct s_header *header,
               void    *control_list,
               int     *control_list_size,
               short   *data_buffer,
               int     *data_buffer_size,
               int     *status_buffer,
               int     *WC_buffer,
               int     *WC_buffer_size,
               int     *QXE_buffer,
               int     *QXE_buffer_size,
               int     *error);
```

Purpose

The *cainit* function initializes the CAMAC list building header. This function must be called prior to building a CAMAC command list.

Description

The *cainit* function is used to initialize the header and other data structures for the CAMAC command lists. This function should be called whenever a new CAMAC command list is built. The header holds the sizes, lengths, and pointers to other data structures used during processing time. The header is a parameter for all subsequent calls to the list building functions.

The *cainit* function requires array parameters that should be declared sufficiently large enough to contain all the list processing elements and data buffers. The data buffer used for the CAMAC command list must be large enough to "hold" all the data for a CAMAC list operation. All data transferred throughout the list operation is moved through this buffer. When generating a list of operations, a tally must be made to ensure that the total number of data words transferred for each individual list element is accounted for when determining the total data buffer size.

Parameters

Parameter Name	Direction	Description
header	Input	The header information contains pointer to other data structures, lengths of structures, and other vital information regarding the operation of the CAMAC command processor.
control_list	Input	This array is used to hold the CAMAC command list. The control list should be declared as a longword array with a size of *control_list_size*.
control_list_size	Input	This parameter specifies the number of elements available in the CAMAC command list.
data_buffer	Input/Output	This array holds all the data for the associated data transfers contained in the CAMAC command list. The shortword array should be declared with a size of *data_buffer-size*. The address of this array must be longword aligned and is initialized when

Parameter Name	Direction	Description
		the *cainit* function is used.
data_buffer_size	Input	This parameter specifies the size of the *data_buffer*. The buffer must be declared by the user sufficiently large so that the array can hold all requests from the CAMAC command list.
status_buffer	Input	This parameter is not used but here for legacy support. It may be set to a pointer to a value of zero.
WC_buffer	Input	This parameter is not used but here for legacy support. It may be set to a pointer to a value of zero.
WC_buffer_size	Input	This parameter is not used but here for legacy support. It may be set to a pointer to a value of zero.
QXE_buffer	Input	This parameter is not used but here for legacy support. It may be set to a pointer to a value of zero.
QXE_buffer_size	Input	This parameter is not used but here for legacy support. It may be set to a pointer to a value of zero.
error	Output	Error Code

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR103 Th header size does not match the header size of the current version.

ERR141 Data buffer not longword aligned.

ERR142 Control list buffer not longword aligned.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"


main()
{
  int    status;          // return status from the functions
  char   devname[] = "kpa00";
  int    *hdl;                          // Handle for operations
  int    errstat[STAMAX];               // array with list of errors
  short n;                              // slot
  short a;                              // sub address
  short f;                              // function
  short c;                              // crate
```

```
  short qmode;                          //  q mode for transfer
  int LongReadBuffer[8192];             // short write data buffer
  unsigned long TransferCount;          // transfer count for block
  int camac_list[8192];                 // list for camac processing
  int DataIndex;                        // data index
  int listmax;                          // max size of list
  int datamax;                          // max size of data
  int      indicator;
  int      zero=0;                      // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

  n=1;
  f=16;
  a=0;
  qmode = QIGN | WTS24 ;
  status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
                &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
  }
  c=1;
  n=1;
  f=0;
  a=0;
  qmode = QIGN | WTS24 ;
  TransferCount = 100 ;
  status = cablk (&header, &c, &n, &a, &f, &qmode, &TransferCount, &DataIndex, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cablk\n");
   camsg(errstat);
  }
  status = cahalt(&header, errarr);
  if ((status & 1) != 1)
  {
```

```
    printf("****ERROR**** cahalt\n");
    camsg(errstat);
   }
   status = caexew(&header, &hdl, errarr);
   if ((status & 1) != 1)
   {
    printf("****ERROR**** caexew\n");
    camsg(errstat);
    exit(status);
   }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
   {
    printf("****ERROR**** caclos\n");
    camsg(errstat);
    exit(status);
   }
}
```

3.4.7 canaf

Syntax

```
int canaf(    struct s_header *header,
              short           *c,
              short           *n,
              short           *a,
              short           *f,
              short           *mode,
              int             *DatInd,
              int             *error
          );
```

Purpose

The *canaf* function adds a command to the CAMAC command list which when executed results in a single CAMAC transfer.

Description

The *canaf* function is used to add a single CAMAC transfer operation to the CAMAC command list. This function will only execute a single transfer. For block transfer operations, the *cablk* list instruction should be used.

This command will allocate one element in the CAMAC command list. If the CAMAC operation is a read or write operation, then space in the data buffer will also be allocated for the command entry. The parameter *Datind* will be returned with a value corresponding to the index into the data buffer where the commands' data is located. Note that the data buffer used to transfer data for these single operations is specified in the *cainit* function.

The *canaf* function supports all four types of transfer operations. These four modes consist of Q-Ignore, Q-Stop, Q-Repeat and Q-Scan. Please refer to the *Transfer Mode* section of this manual for details on each operating mode. A portion of the *mode* parameter for this function is used to indicate the block transfer-operating mode.

Parameters

Parameter Name	Direction	Description
header	Input	Header array that is built by the *cainit* function and contains pointers to the CAMAC command list and data buffer. This is updated as additional list elements are added.
C	Input	Address of the chassis to be accessed
N	Input	Slot number of the module to be accessed
A	Input	Subaddress within the module to be accessed
F	Input	Function code to be performed
mode	Input	Type of CAMAC block transfer to perform. Please refer to *Transfer Mode* section of this manual for additional information. (See below)

Parameter Name	Direction	Description
DatInd	Output	This parameter is returned with the index in to the data buffer marking the starting location for the word of data used for the operation.
error	Output	Error Code

The *mode* parameter in the *canaf* function is used to specify the CAMAC transfer Q-mode, the CAMAC data word size, and the termination technique used when a No-X condition occurs. Even though the *canaf* only executes a single write operation, the CAMAC Q and X returns are evaluated and continued list processing is based on their values as it relates to the selected Q-mode. The following table shows the available selections as *#defines* in the *kscuser.h* include file. Note that only one defined Q-mode or word size can be specified for each transfer.

#define	Description
QSTP	Selects the Q-Stop Block Transfer Mode
QIGN	Selects the Q-Ignore Block Transfer Mode
QRPT	Selects the Q-Repeat Block Transfer Mode
QSCN	Selects the Q-Scan Block Transfer Mode

Q-mode Transfer Selection

#define	Description
WTS16	Selects 16-bit CAMAC Data Word Size
WTS24	Selects 24-bit CAMAC Data Word Size

CAMAC Data Word Size Selection

#define	Description
AD	Inclusion of the *AD* in the mode description causes the CAMAC command processor to ignore No-X conditions during processing.

CAMAC Data Word Size Selection

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the *Error Codes* section of this manual.

ERR143 The CAMAC header is not initialized.

ERR144 The CAMAC header is initialized, but not correctly.

ERR202 A CAMAC in-line read operation was specified. Only CAMAC write and control functions can be specified as in-line operations.

Example

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#include "ksc_api.h"
#include "kscuser.h"
#include "camerr.h"
#include "strfunc.h"
#include "cmdlist.h"

main()
{
  int     status;           // return status from the functions
  char    devname[] = "kpa00";
  int     *hdl;                                   // Handle for operations
  int     errstat[STAMAX];                        // array with list of errors
  short n;                                        // slot
  short a;                                        // sub address
  short f;                                        // function
  short c;                                        // crate
  short qmode;                                    // q mode for transfer
  int LongReadBuffer[8192];                       // short write data buffer
  int camac_list[8192];                           // list for camac processing
  int DataIndex;                                  // data index
  int listmax;                                    // max size of list
  int datamax;                                    // max size of data
  int     indicator;
  int     zero=0;                                 // zero value
  struct s_header header;
//
// Open the device
//
  status = caopen(&hdl, devname, errstat);

//
// Check if device opened properly
//
  if ((status & 1) == 0)
  {
        printf("CAOPEN, error opening device = %s\n", devname);
        camsg(errstat);
        exit(status);

  }
  listmax = 1024;
  datamax = 1024;

status = cainit(&header, camac_list, &listmax, LongReadBuffer, &datamax, errarr,
              &zero, &zero, &zero, &zero, errarr);
  if ((status & 1) != 1)
  {
    printf("****ERROR**** cainit\n");
    camsg(errstat);
  }
  c=1;
  n=1;
```

```
  f=0;
  a=0;
  qmode = QIGN | WTS24 ;
  TransferCount = 100 ;
  status = canaf (&header, &c, &n, &a, &f, &qmode,  &DataIndex, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** canaf\n");
   camsg(errstat);
  }
  status = cahalt(&header, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** cahalt\n");
   camsg(errstat);
  }
  status = caexew(&header, &hdl, errarr);
  if ((status & 1) != 1)
  {
   printf("****ERROR**** caexew\n");
   camsg(errstat);
   exit(status);
  }
//
// Close the device
//
   status = caclos (&hdl, errstat);
   if ((status & 1) != 1)
   {
    printf("****ERROR**** caclos\n");
    camsg(errstat);
    exit(status);
  }
}
```

3.5 VXI List Generation Interface Library

3.5.1 Library Usage

The List Generation Library is implemented as a set of linkable routines in the KSCAPI library. The list building routines are prototyped in the "kscapi.h" file. Additionally, a set up "C" macros is also available to create inline lists.

The list generation routines are provided to help in the creation of lists using a more structured convention. Creating a list involves first allocating memory to store the list and then calling KSC_init_list. This routine will return back a pointer to a structure of type ksc_list that will be used by all of the other list generating routines. If the user is building multiple lists, the user must provide storage for each of the lists and call KSC_init_list for each list. The user may build multiple lists concurrently as all information about the current state of each list is maintained by the structure allocated by KSC_init_list. The list must be allocated on a long word boundary.

The user calls the individual functions to "compile" the instruction list into the user provided list memory. Each callable function in the library is usually associated with one particular command instruction. There exist functions that implement standard IF...ELSE...ENDIF and SWITCH...CASE...ENDCASE properties found in most high-level languages. The list-generating library keeps track of calculating offsets and inserting the proper commands into the list, making such IF and CASE blocks much easier to develop.

Upon completion of making a list, KSC_finish should be called to clean up the list and check for any possible errors in the list. The routine KSC_dump_list can be called to display the compiled list to standard output.

A sample program that creates a list follows. This list does not perform any real functionality, and is provided merely as an example for list creation. Do not attempt to actually execute the list!

```
/*
        TEST PROGRAM
        This program will demonstrates the use of the List
Generation functions
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "../include/ksc_genlist.h"

main()
{
        /* Variable defs */

        short   *mem;           /* Our memory buffer */
        struct ksc_list *list;  /* Our list definition structure
*/
```

```
        int             size;   /* Our value of how big list is
*/
        /*
        * Begin here
        */

        mem = malloc(1024);             /* Allocate a 1024 byte
buffer */
        KSC_init_list(mem,1024,&list);

        /*
        * List code begins here
        */

        KSC_bdcast_trigger(list);
        KSC_block_rw(list,ABORT,WS8,DECADR,15,33,READ,INTERNAL,
                                0x7F7F7F,0x252525);

        KSC_if(list,EQ,0xFFFFFF,0x353535);
                KSC_execute_msg_dev(list,0x75,0,1,20,50,"A simple
text block");
                KSC_gen_demand(list,200);
        KSC_endif(list);


KSC_inline_rw(list,ABORT,WS8,DECADR,15,33,WRITE,INTERNAL,0x13300)
;

KSC_inline_w(list,ABORT,WS8,DECADR,15,33,INTERNAL,0x22222,0x53535
3);
        KSC_if(list,EQ,0xFFFF,0x616161);
                KSC_load_test_val(list, 15,WS16,0x7A7A7A);
                KSC_mark_list(list);
        KSC_else(list);
                KSC_slave_trigger(list, 33, 1,1,0,1,0);
                KSC_if(list,EQ,0xFFFF,0x616161);
                        KSC_load_test_val(list,
15,WS16,0x7A7A7A);
                        KSC_mark_list(list);
                KSC_else(list);
                        KSC_slave_trigger(list, 33, 1,1,0,1,0);
                        KSC_store_flag(list,0x5050);
                KSC_endif(list);
                KSC_store_flag(list,0x5050);
        KSC_endif(list);

        KSC_time_stamp(list);

        KSC_switch(list,0xFAFAFA);
                KSC_case(list,0x101010);
                KSC_load_test_val(list, 15,WS16,0x7A7A7A);
```

```
                KSC_mark_list(list);
                KSC_if(list,EQ,0xFFFFFF,0x353535);

KSC_execute_msg_dev(list,0x75,0,1,20,50,"A simple text block");
                        KSC_gen_demand(list,200);
                KSC_endif(list);

                KSC_case(list,0x202020);
                KSC_load_test_val(list, 15,WS16,0x717171);
                KSC_mark_list(list);

                KSC_case(list,0x303030);
                KSC_load_test_val(list, 15,WS16,0x2b2b2b);
                KSC_mark_list(list);
        KSC_endcase(list);

        KSC_end_list(list);

        /*
        * List code ends here
        */

        KSC_finish(list);
        /*
        * Write the list out in a symbolic fashion (see following
output)
        */
        KSC_dump_list(mem,0,1);  /* Display the built list */
}
```

This code creates the following output list:

```
LOC  DATA CODE
---- ---- ------------------------------------------------------------
0000 8041 BRDCST_TRIG
     0000
     0000
     0000
0008 47AE BLK_RW ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:1 int:1
               addr:007F7F7F tr_cnt:00252525
     C021
     7F7F
     007F
     2525
     0025
0014 8084 IF cond:0 mask:00FFFFFF test:00353535
     0000
     FFFF
```

```
     00FF
     3535
     0035
     002A
0022 8090   EXEC_MSG_DEV addr:75 term:0 rply:1 time_out:0014
rply_lng:32
                        cmd_lng:14 [A simple text block]
     8075
     0014
     1432
     2041
     6973
     706D
     656C
     7420
     7865
     2074
     6C62
     636F
     006B
003E 8091   RESUME_MSG_DEV
     0000
0042 8102   GEN_DEMAND pattern:C8
     00C8
0046 8083   END_OF_SUBLIST
     0000
          END_IF
004A 478E INL_RW ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:0 int:1
               addr:00013300
     8021
     3300
     0001
0052 47CE INLN_W ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:0 int:1
               addr:00022222 data:00535353
     8021
     2222
     0002
     5353
     0053
005E 8085 IF(ELSE) cond:1 mask:0000FFFF test:00616161
     0001
     FFFF
     0000
     6161
     0061
     0014
006C 8082   LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
     7A7A
     007A
0074 8080   MRK_LST_ADR
```

```
     0000
0078 8083    END_OF_SUBLIST
     0000
007C 0042 ELSE
007E 8040    ADDR_SLV_TRIG chas_adr:21 TTL: 1 ECL:1 FP:0 list:1
timst:0
     0021
     1101
     0000
0086 8085    IF(ELSE) cond:1 mask:0000FFFF test:00616161
     0001
     FFFF
     0000
     6161
     0061
     0014
0094 8082      LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
     7A7A
     007A
009C 8080      MRK_LST_ADR
     0000
00A0 8083      END_OF_SUBLIST
     0000
00A4 0012    ELSE
00A6 8040      ADDR_SLV_TRIG chas_adr:21 TTL: 1 ECL:1 FP:0 list:1
timst:0
     0021
     1101
     0000
00AE 80F8      STO_FLG flag:5050
     5050
00B2 8083      END_OF_SUBLIST
     0000
           END_IF
00B6 80F8    STO_FLG flag:5050
     5050
00BA 8083    END_OF_SUBLIST
     0000
         END_IF
00BE 8002 READ_TIME_STAMP
     0000

00C2 8086 SWITCH mask:00FAFAFA
     007E
     FAFA
     00FA
00CA 1010    CASE test_val:00101010
     0010
     004A
00D0 8082    LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
```

```
     7A7A
     007A
00D8 8080    MRK_LST_ADR
     0000
00DC 8084    IF cond:0 mask:00FFFFFF test:00353535
     0000
     FFFF
     00FF
     3535
     0035
     002A
00EA 8090      EXEC_MSG_DEV addr:75 term:0 rply:1 time_out:0014
rply_lng:32
                        cmd_lng:14 [A simple text block]
     8075
     0014
     1432
     2041
     6973
     706D
     656C
     7420
     7865
     2074
     6C62
     636F
     006B
0106 8091      RESUME_MSG_DEV
     0000
010A 8102      GEN_DEMAND pattern:C8
     00C8
010E 8083      END_OF_SUBLIST
     0000
             END_IF
0112 8083    END_OF_SUBLIST
     0000
0116 2020    CASE test_val:00202020
     0020
     0014
011C 8082    LD_TEST_VAL add_mod:0F ws:2 addr:00717171
     800F
     7171
     0071
0124 8080    MRK_LST_ADR
     0000
0128 8083    END_OF_SUBLIST
     0000
012C 3030    CASE test_val:00303030
     0030
     0000
0132 8082    LD_TEST_VAL add_mod:0F ws:2 addr:002B2B2B
```

```
     800F
     2B2B
     002B
013A 8080    MRK_LST_ADR
     0000
013E 8083    END_OF_SUBLIST
     0000
          END_CASE
0142 8081 END_OF_LIST
     0000
0146 8000 HALT
     0000
```

3.5.2 KSC_bdcast_trigger

Syntax
int KSC_bdcast_trigger (struct ksc_list *list_base);

Purpose
This adds a Broadcast Trigger instruction to the passed list.

Description
This routine adds the Broadcast Trigger instruction to the end of the list defined by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.3 KSC_block_rw

Syntax

```
int KSC_block_rw (struct ksc_list *list_base,
                  int abort,
                  int word_sz,
                  int acc_mod,
                  int ch_addr,
                  int addr_mod,
                  int rw,
                  int it_cmd,
                  int address,
                  int trans_count);
```

Purpose

This routine adds a Block Read/Write instruction to the passed list.

Description

This routine will insert a Block Read/Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.
trans_count	input	32-bit transfer count.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.4 KSC_dump_list

Syntax

```
int KSC_dump_list (mem,
                   int size,
                   int dump);
```

Purpose

This routine displays an already built list in a readable format.

Description

This routine will display an already built list stored in memory. The list should end with a HALT instruction.

The display will give for each instruction its location (as a byte offset), instruction code, and the actual instruction and data. If the data value is set to a non-zero value, you will also receive each additional word of data for the instruction.

IF and CASE blocks will be indented accordingly. Currently, no nesting of CASE blocks is supported, and up to 10 nested IF blocks are supported.

If the routine encounters an invalid opcode, it will be displayed and the routine will continue, attempting to parse the next word as an opcode.

Parameters

Parameter Name	Direction	Description
mem	Input	This should be a pointer to the start of the list to be displayed.
size	input	This is set to the maximum size of the buffer, in bytes. The routine will display all bytes up to and including mem+size bytes. If size is specified as zero, the routine will display all instructions up to a HALT instruction.
dump	input	This is a flag used to set the display format for the instruction list. If set to a value of zero, the display will only show the beginning of each command. If set to any other value, the display will also include all additional words of data for each command.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.

3.5.5 KSC_end_list

Syntax
int KSC_end_list(struct ksc_list *list_base);

Purpose
This will add an EOL (End of List) instruction to the list.

Description
This routine will insert an End of List (EOL) instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.6 KSC_finish

Syntax

int KSC_finish(struct ksc_list *list_base);

Purpose
End the creation of a list and free allocated list building resources.

Description
This routine should be called at the completion of creating a list. It will check to insure that all IF and CASE blocks are properly completed.

A halt instruction is automatically added to the end of the list and the list_base memory is then released back to the system. You cannot use the list_base value after calling this routine.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.7 KSC_gen_demand

Syntax

```
int KSC_gen_demand(struct ksc_list *list_base,
                   int pattern);
```

Purpose

This places a Generate Demand instruction into the list.

Description

This routine will insert a Generate Demand instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
pattern	Input	Demand pattern value (0-255).

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.8 KSC_init_list

Syntax

```
int KSC_init_list (*mem_base,
        int size,
        struct ksc_list *list_base);
```

Purpose

Prepare allocated memory for list generation.

Description

This routine must be called before using any of the other list generating routines. It will allocate a structure of type ksc_list, initialize it, and then return its location back in list_base. You will need this value for calls to any of the other list generating functions.

You may work on more than one list at a time. Each list will have its own list_base value.

At the end of creating a list, you must call KSC_finish to cleanup the list and remove the allocated structure from memory.

Parameters

Parameter Name	Direction	Description
mem_base	Input	This is a pointer to the start of memory where you want the list to be built. The memory must already be allocated.
size	Input	This is the size, in bytes, of the allocated memory starting at mem_base.
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_BAD_ARD	Bad arguments passed.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.9 KSC_inline_rw

Syntax

```
int KSC_inline_rw (struct ksc_list *list_base,
                   int abort,
                   int word_sz,
                   int acc_mod,
                   int ch_addr,
                   int addr_mod,
                   int rw,
                   int it_cmd,
                   int address);
```

Purpose

This places an Inline Read/Write VXI/VME instruction into the list.

Description

This routine will insert an Inline Read/Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.10 KSC_inline_w

Syntax

```
int KSC_inline_w (struct ksc_list *list_base,
        int abort,
        int word_sz,
        int acc_mod,
        int ch_addr,
        int addr_mod,
        int rw,
        int it_cmd,
        int address,
        int data);
```

Purpose

This places an Inline Write VXI/VME instruction into the list.

Description

This routine will insert an Inline Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.
data	input	The actual data to be written.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.5.11 KSC_slave_trigger

Syntax

```
int KSC_slave_trigger (struct ksc_list *list_base,
                       int ch_addr,
                       int ttl_trig,
                       int ecl_trig,
                       int fp_trig,
                       int list,
                       int timst);
```

Purpose

This places an Addressed Slave Trigger instruction into the list.

Description

This routine will insert an Addressed Slave Trigger instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
ch_addr	input	Chassis address. Valid values are 0-127.
ttl_trig	input	Generate VXI TTL trigger line.
ecl_trig	input	Generate VXI ECL trigger line.
fp_trig	input	Generate V160 front panel trigger.
list	input	Trigger list execution.
timst	input	Clear time stamp counter.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

3.6 VXI Routines

The KSC API (Application Programming Interface) provides all of the functions of the 2962 to the user by actually do the QIO calls to the Windows NT or 2000 device driver. The users are encouraged to use these routines to limit the changes resulting from changes in the Driver QIO interface and the Windows operating system (Note: Many of the KSC API calls are available on Digital UNIX).

Besides providing the basic functioning the 2962, the KSCAPI also supports the building of command lists. These command lists are for both VXI and CAMAC chassis. The CAMAC library calls these routines to build its lists and to function the 2962. The next chapter describes the KineticSystem CAMAC Application Library.

Programmers that work in "C", may also use command list generation macros written in "C". These macros generate runtime code that initializes a command list. The use of these macros or the command list generation routines are encouraged in the event there are changes to the command list instructions.

3.6.1 API Usage

The API is implemented as a set of linkable routines in a linkable object library, giapi.lib. All of the routines are prototyped in the ksc_api.h file. This file may be included using the following in a normal "C" and programs:

```
#include <ksc_handle.h>
#include <ksc_api.h>
```

3.6.2 API and Driver Errors

The API and driver error codes are described in a later chapter along with their recovery procedures. By Windows convention, all of the interface routines return an odd value if the call was successful and even if the call was not. The error returned by the Windows kernel is returned in the status argument of the handle. This status argument is only valid if the interface returns an even error code. The value of the status entry may be from the device driver or from Windows. The user should also reference Windows error codes. All of the error codes are defined in the kerrors.h header file. The user should reference the KSC 2962 Hardware documentation for the most of the device driver errors.

The routine KSC_print_symbolic will translate and print the error code to the symbolic English description to the standard output. Additional error information may also be displayed by using KSC_lasterror, which will return the last known API routine and driver error status.

3.6.3 API Handle

The API allocates a handle and returns its address to the caller when the user calls the KSC_Init interface. All particulars of the API are then maintained within this allocated region. The definition of this handle is maintained in the ksc_handle.h file. The handle is passed to all of the routines (except for the KSC_init routine). The actual byte count completed for a request is returned in the handle element: "xfsize". The device driver status is returned in the handle element: "status". The status should be examined along with the return value from the interface routine. The status may be either a status from the device driver or another error from Windows NT. The routine KSC_print_symbolic will translate and print the returned

English status code to standard output. All of the API routines return a 32-bit integer for status similar to the Windows NT system services. If the returned status is odd, then the call is completely successfully.

3.6.4 Command List Generation

There are a set of macros that are provided for the user to create lists which can be executed by the Grand Interconnect Host adapter or a slot zero crate controller. These are documented in the command list macros chapter. Additionally, the user may use the runtime routines to initialize a command list. The runtime routines will function in any language while the macros are only valid for "C" programmers.

3.6.5 Partition Contention

The Grand Interconnect device driver allows the user to load any of the partitions. The user must use mutexes, semaphore, or locks to protect the use of the partitions by multiple threads or processes. The KSC_loadgo always uses partition one and will both load the command partition and execute it autonomously.

3.6.6 Program TEST_API

The KGI_EXAMPLES:TEST_API program provides a menu to execute many of the KSC routines. Originally its purpose was to test the various KSC routines. It can also be used to control events for debugging application programs and testing hardware configurations. The logical name kgi_examples is defined by the KGI_STARTUP.COM command file.

The menu presented on the screen/window is:

KSC Application Library Test Utility

1.	KSC_init	2.	KSC_set_partitions
3.	KSC_display_partitions	4.	KSC_diag
5.	KSC_load_cmdlist	6.	KSC_read_cmdlist
7.	KSC_exec_rlist	8.	KSC_exec_wlist
9.	KSC_read_counters	10.	KSC_reset
11.	KSC_demand_read	12.	
13.	Init command list (read)	14.	KSC_loadgo (write)
15.	Init command list (write)	16.	KSC_loadgo (read)
17.	KSC_get_failure	18.	Change Crate number
19.	KSC_v160_loadcmd (& build)	20.	KSC_v160_trigger
21.	KSC_v160_readcmd	22.	KSC_v160_readbuf
23.	KSC_v160_readreg	24.	KSC_exec_clocked_list
25.	KSC_v160_writereg	26.	KSC_read_multibuf
27.	KSC_mbuf_done	28.	
99.	Exit		

Enter selection [1]:

Always execute selection 1 before any other.

As an example, to generate a demand for testing a user application program that expects to see a demand_id of zero in Chassis one (user program must have registered for this demand id from the chassis using KSC_enable_demand), execute:

#1	To open initialize the library
#19	To place a command list into the target V160 (default is chassis 1, see option 18)

#20 To actually trigger the list in the V160

3.6.7 KSC_demand_read

Syntax

```
int KSC_demand_read (struct KSC_handle *handle,
                     int demand_list[],
                     int demand_list_size,
                     int *demands_found);
```

Purpose

Post a read for one or more demand commands.

Description

The driver allows the ability to receive demand messages from the highway and list interrupts while the 2962 is executing a command list. These demands are stored in a demand FIFO and they are dequeued with this read. If there are no demands currently pending, the calling thread will be suspended until such time that a demand message or a list interrupt is processed.

In the event of a device reset this routine will return with status indicating that demand messages might have been lost (those that might have been in the FIFO when the reset was executed).

The host "in list" interrupts can be distinguished from the demand messages from the 2962 by a non-zero upper word. The content of the lower demand message is as it was read from the 2962. The user should reference the 2962 manual for this information.

There is no included support to provide access control for different processes using this call. The Demand Process uses this call to acquire the demands from the driver. This call is provided for users who wish to develop their own demand servicing applications.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
demand_list	Input	A long word array to receive the demand interrupts. Must be long word aligned.
demand_list_size	Input	Size in bytes of the demand list array.
demands_found	Output	Represents the location of an integer that will be set to the number of demands actually transferred.

Return Values

3.6.8 KSC_display_partitions

Syntax

```
int KSC_display_partitions(struct KSC_handle *handle,
                           struct KSC_driver_ptable *partition_table);
```

Purpose

Read command list partition table of the KSC 2962.

Description

This call will call the kscgi device driver to return the current partition table of the KSC 2962. The KSC_driver_ptable structure contains both the starting address and the length of each partition in bytes.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition_table	Input	Returns the current partition table of the KSC 2962 as maintained by the KSCGI device driver.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_IOCTL	Error from device driver or Windows NT. Examine status in KSC_handle.

3.6.9 KSC_enable_demand

Syntax

```
int KSC_enable_demand(struct KSC_handle *handle,
                      int Chassis,
                      int demand_id,
                      int demand_type,
                      long *AstAdr,
                      long *AstPrm);
```

Purpose

Enable reception of a single demand from a particular chassis.

Description

This routine Enables Demands by sending a message to the Demand Process requesting notification when the particular demand id for the indicated chassis occurs. The Demand Process must be running or this process will wait forever.

If a new Demand Process is started, any processes that have registered for a demand are forced to exit. All processes that desire to connect for Demands must be running in the same group as the Demand Process as a shared group global section is used between the user process and the demand process. A lock is captured when the first demand enable is called which is used by the demand process to exit any processes that may have enabled demands using the old demand region.

This routine communicates with the demand process using mailboxes. If the user process fails to empty its demand mailbox in a timely fashion, the demand process will disconnect the user's process for any demands that the process may have registered for.

For one shot demands, this routine may be called multiple times to re-enable the one shot. Any process that registers for a demand supersedes any previous process for the particular demand id for a particular chassis.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
Chassis	Input	Chassis number that the demand is expected from. Chassis number zero is the host grand interconnect adapter.
demand_id	Input	This is the ID of the demand. For command lists that generate the demand, this is the demand id that is coded into the instruction. If the demand is a demand from a VXI crate (V160), the demand id is the encoding of the IRQ or multi-buffer bit. If the demand is a demand from a CAMAC chassis (2952), the demand id is the encoding of the LAM lines or the multi-buffer bit.
demand_type	Input	The demands can either be set up to generate a single one shot when the demand occurs or a reoccurring demand. A

Parameter Name	Direction	Description
		value of one (1) is a single one shot and a value of two (2) is a repeating demand.
AstAdr	Input	AST routine to be called when the demand occurs. The AST parameter is the demand id and the chassis that generated the demand passed by reference. The demand id is in the lower sixteen bits and the chassis is in the upper sixteen bits.
AstPrm	Input	A pointer to a user defined value passed along to the Ast Routine.

Return Values

KSC_CHASSIS	Invalid user chassis number entered.
KSC_NOTCFG	All demands that are to be enabled must be configured. See the Demand Process chapter.
KSC_DMDTBLFULL	The system allows for a maximum number of demands that single process can connect for. The caller has exceeded this value. Examine the KSC_handle.h for this maximum.

3.6.10 KSC_exec_rlist

Syntax

```
int KSC_exec_rlist (struct KSC_handle *handle,
                    int partition,
                    byte buf[],
                    int buffer_size);
```

Purpose

Execute a read command list.

Description

The command list currently stored in the KSC 2962 will be executed. The user-supplied buffer will receive any data that is sourced by the KSC 2962. If the user buffer is too large, or the KSC 2962 fails to source sufficient data, the request will only complete via a device timeout or an embedded command list interrupt. If the command list contains a list interrupt, the driver will consider such an interrupt as a completion of the list. The kscgi device driver always attempts to store a list completion interrupt at the end of the loaded command list partition.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Command list partition within the KSC 2060 that is to be executed.
buf	Input	Data to be returned to user as generated by the execution of the command list stored in the indicated partition.
buffer_size	Input	Total size of the buffer in bytes.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status in KSC_handle.

3.6.11 KSC_exec_wlist

Syntax

```
int KSC_exec_wlist (struct KSC_handle *handle,
        word buf[],
        int buffer_size,
        int *used_size,
        int partition);
```

Purpose

Executes a write command list.

Description

This call will request the execution of the command list that has already been loaded into the command list partition of the KSC 2962. The actual completion of the request depends on whether the user supplied the correct number of bytes for the KSC 2962. A command list may require more data than what was provided by the user, however, the DMA will complete regardless. If the user embedded a command list interrupt, this will terminate the command list.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
buf	Input	Data source for the command list that was loaded in the indicated partition.
buffer_size	Input	Size of the user supplied buffer.
used_size	Output	The returned number of bytes that requested by the KSC 2962.
partition	Input	The command list partition that contains the command list that is to be executed.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status in KSC_handle.

3.6.12 KSC_get_failure

Syntax

```
int KSC_get_failure (struct KSC_handle *handle,
                     int partition,
                     struct KSC_error_exc failure_array[]);
```

Purpose

Executes a write command list.

Description

This call will request the execution of the command list that has already been loaded into the command list partition of the KSC 2962. The actual completion of the request depends on whether the user supplied the correct number of bytes for the KSC 2962. A command list may require more data than what was provided by the user, however, the DMA will complete regardless. If the user embedded a command list interrupt, this will terminate the command list.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Partition to return the last failure. This should be the same as the partition passed to the KSC_exec_list, KSC_exec_rlist, or KSC_exec_wlist.
failure_array	Input	An array to receive the last failure that the grand interconnect device driver encountered. The following are returned in the eight "ints" at the time of the I/O completion. The user should reference the KSC 2962 device manual for a description of the bits contained in the device registers.

[0]- CSR (Control and Status Register)
[1]- ESR (Error Status Register)
[3]- Word count low (from last block transfer in list)
[4]- Word count high (from last block transfer in list)
[5]- CMA (Command Memory Address)
[6]- CSR (At start of list execution)
[7]- Actual byte count transferred on last list execution.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status

	in KSC_handle.
KSC_IOCTL	Driver error or Windows NT error. Examine status in KSC_handle.
KSC_PARTIONERR	Desired partition number is not valid.

3.6.13 KSC_init

Syntax

```
int KSC_init (struct KSC_handle *handle,
        int ctrl);
```

Purpose

Initialize driver access routines.

Description

This routine initializes the application library. The KSC 2962 devices are opened and a pointer to the KSC_handle is returned to the caller for future calls to the API. The controller number is used to open the "/dev/ks<ctrl>0" device. Upon a successful call, the KSC_handle->ksa field will be filled in with the device descriptor for the device.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
ctrl	Input	Controller number of the KSC 2962 Grand Interconnect (range: 0-4).

Return Values

KSC_ALLOC	Unable to allocate the KSC handle.
KSC_OPENERROR	Unable to open KSC devices. The status of the Windows NT "open" call is returned in handle->status.
KSC_SUCCESS	Successful completion.

3.6.14 KSC_lasterror

Syntax

```
int KSC_lasterror (struct KSC_handle *handle,
        long api_status,
        long sys_status);
```

Purpose

Display the last known API and Driver error conditions.

Description

This routine will return the most recent API and Driver status information. The API status is the last status received from any non-listbuilding API routine. The driver status information is from actual device QIO calls or from Windows NT.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
api_status	Input	Last status returned by the API routines.
sys_status	Input	Last status returned from Windows NT or the device driver.

Return Values

KSC_HANDLE	Handle not initialized. Need to call KSC_init first before using this function.

3.6.15 KSC_loadgo

Syntax

```
int KSC_loadgo (struct KSC_handle *handle,
                word buf[],
                int buf_size,
                int direction);
```

Purpose

Load a command list and execute it.

Description

This entry provides an ability for the user to both load a command list partition and then execute the loaded command list. The user indicates the direction of the data transfer. This routine simply calls the command partition load function and then either the read or write command list function. The completion of the command list is identical to that for the execute read or write command lists.

Because this routine is completed in a single operation, there is no need to control access to a particular partition. This routine always uses command list partition one of the Grand Interconnect host adapter.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
buf	Input	Source or sink of the data for the command list loaded into the indicate partition. The buffer contains both the command list and the data buffer. The buffer contains in the first "int" the size of the command list in bytes, followed by the command list and the data buffer. The command list macros provide a convenient way to create this combined buffer.
buf_size	Input	Size of user buffer.
direction	Input	Direction of the transfer (0= user sources the data, 1= 2962 sources the data for the command list).

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status in KSC_handle.
KSC_PARTIONERR	Desired partition number is not valid.

3.6.16 KSC_load_cmdlist

Syntax

```
int KSC_load_cmdlist (struct KSC_handle *handle,
                      int partition,
                      int list[],
                      int list_len);
```

Purpose

Load a KSC 2962 command list into a partition.

Description

Before a command list can be executed it must be loaded into the command list memory of the KSC 2962. The programmer should reference the KSC 2962 documentation with regard to the actual content of the command list.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which partition the command list should be loaded into.
list	Input	Command list to be loaded into the indicated partition of the KSC 2962.
list_len	Input	Length of the command list to be loaded into the indicated partition of the KSC 2962. The length of the list must be less than the length of the indicated partition.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status in KSC_handle.
KSC_IOCTL	Driver error or Windows NT error. Examine status in KSC_handle.
KSC_PARTIONERR	Desired partition number is not valid.

3.6.17 KSC_print_symbolic

Syntax
int KSC_print_symbolic (int status);

Purpose
Convert and print symbolic text for a status code.

Description
This routine calls the sys$getmsg and prints to standard output the symbolic description of the status code. The user must have linked the message file: kgdriver_msg from the library.

Parameters

Parameter Name	Direction	Description
status	Input	Windows NT, API, or driver status to be converted to text.

Return Values

3.6.18 KSC_read_cmdlist

Syntax

```
int KSC_read_cmdlist (struct KSC_handle *handle,
                      int partition,
                      int list[],
                      int list_len);
```

Purpose

Load a KSC 2962 command list into a partition.

Description

Before a command list can be executed it must be loaded into the command list memory of the KSC 2962. The programmer should reference the KSC 2962 documentation with regard to the actual content of the command list.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	The desired command list partition to read the contents.
list	Input	Buffer to receive the current contents of the indicated command list partition. The number of bytes returned depends on the list size passed and the current size of the requested partition.
list_len	Input	list_len bounds the number command list words from the command list memory that can be returned from the partition. If the size of the partition currently is larger than user's buffer, only list_len bytes will be returned along with a informational status indicating that the user's buffer was too small. The actual size is returned in xsize in the KSC_handle.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_READERR	Error from device driver or Windows NT. Examine status in KSC_handle.
KSC_PARTIONERR	Desired partition number is not valid.

3.6.19 KSC_read_counters

Syntax

```
int KSC_read_counters(struct KSC_handle *handle,
                      struct KSC_counters_gi *Counters);
```

Purpose

Return Driver statistic counters.

Description

This routines returns the current counters from the 2962 device driver. These may be used for user written diagnostic programs.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
Counters	Input	Buffer to receive the counters. The array contains the following in long word integers in order as follows:

Number of timeouts
Number of writes
Number of reads
Number of command list loads
Number of interrupts
Number of list_interrupts
Number of demand_interrupts

Return Values

KSC_HANDLE	Handle is invalid.

3.6.20 KSC_set_partitions

Syntax
int KSC_set_partitions (struct KSC_handle *handle,
struct KSC_partition_table *partition_table);

Purpose
Load partition boundaries of the KSC 2962.

Description
This routine calls the device driver to partition the 32K command memory of the KSC 2962. The command memory may be divided in up to eight different partitions. Each of the partitions is assigned a pair of devices. Any of the partitions may be from zero to the remaining size of the command list memory. Partitions cannot be overlapped or concatenated..

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition_table	Input	The desired partitioning of the command list memory of the 2962. This table contains the starting address in bytes for each of the partitions. A zero terminates the table. Specifying all zeroes will result in the last partition being allocated all of the 32KB of the command list memory. To allocate all of the command list memory to the first partition, specify (0,0x8000,0,0,0,0,0,0).

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.
KSC_IOCTL	Driver error or Windows NT error. Examine status in KSC_handle.

3.6.21 KSC_set_timeouts

Syntax
int KSC_set_timeouts (struct KSC_handle *handle,
long time_array[]);

Purpose
Set partition command list timeout values.

Description
The device driver requires that all lists complete within a specific time limit. This value can be specified for each partition. When the device driver is loaded a default value is used for each partition.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
time_array	Input	This array contains in seconds the timeout for each partition. The minimum value is two seconds as Windows NT driver timeout routines are called only at one-second intervals plus or minus one second.

Return Values

KSC_HANDLE	Handle is invalid.
KSC_NOTOPEN	Handle has not been initialized.

3.6.22 KSC_v160_loadcmd

Syntax

```
int KSC_v160_loadcmd (struct KSC_handle *handle,
                      int partition,
                      int cmd_list[],
                      int list_len,
                      int v160,
                      int list_addr);
```

Purpose

This routine loads the user passed command list into the command V160 VXI slot zero controller.

Description

This routine loads the user passed command list into the command V160 VXI slot zero controller. This is done by first loading a command list into the host Grand Interconnect (ksc2962) that will set up the V160, and then writing the command list into the V160. The CSR of the V160 is read and modified for loading the list.

All command lists that are loaded into the V160 must be set for 32-bit address space.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which host partition to use.
cmd_list	Input	Command for V160.
list_len	Input	Length of list in bytes.
v160	Input	VXI crate address.
list_addr	Input	Word address where to load.

Return Values

KSC_HANDLE	Handle is invalid.

3.6.23 KSC_v160_readbuf

Syntax

```
int KSC_v160_readbuf (struct KSC_handle *handle,
                      int partition,
                      int cmd_list[],
                      int list_len,
                      int v160,
                      int list_addr);
```

Purpose

This routine loads the user passed command list into the command V160 VXI slot zero controller.

Description

This routine loads the user passed command list into the command V160 VXI slot zero controller. This is done by first loading a command list into the host Grand Interconnect (ksc2962) that will set up the V160, and then writing the command list into the V160. The CSR of the V160 is read and modified for loading the list.

All command lists that are loaded into the V160 must be set for 32-bit address space.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which host partition to use.
cmd_list	Input	Command for V160.
list_len	Input	Length of list in bytes.
v160	Input	VXI crate address.
list_addr	Input	Word address where to start read.

Return Values

KSC_HANDLE Handle is invalid.

3.6.24 KSC_v160_readcmd

Syntax

```
int KSC_v160_readcmd (struct KSC_handle *handle,
                      int partition,
                      int cmd_list[],
                      int list_len,
                      int v160,
                      int list_addr);
```

Purpose

This routine read the command list from the V160 into the user passed buffer starting at the passed initial address.

Description

This routine read the command list from the V160 into the user passed buffer starting at the passed initial address. This is done by first loading a command list into the host grand interconnect (ksc2962 or ksc29062) that will set up the V160, and then reading the command list from the V160.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which host partition to use.
cmd_list	Input	Command for V160.
list_len	Input	Length of list in bytes.
v160	Input	VXI crate address.
list_addr	Input	Word address where to start read.

Return Values

KSC_HANDLE	Handle is invalid.

3.6.25 KSC_v160_readreg

Syntax

```
int KSC_v160_readcmd (struct KSC_handle *handle,
                      int partition,
                      int v160,
                      int reg_offset,
                      int reg_value);
```

Purpose

This routine reads the V160 register at the offset passed.

Description

This routine reads the V160 register at the offset passed. The user should use the offsets defined in "cmdlist.h".

This routine builds a simple list to read the desired register, loads it into the KSC2962 command memory, and executes it. The user should see the KSC 2962 hardware documentation for the device register documentation.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which host partition to use.
v160	Input	VXI crate address.
reg_offset	Input	Word address where to load.
reg_value	Input	Value from register.

Return Values

KSC_HANDLE	Handle is invalid.

3.6.26 KSC_v160_trigger

Syntax

```
int KSC_v160_trigger (struct KSC_handle *handle,
                      int v160chassis,
                      int partition,
                      int v160cmdaddr);
```

Purpose

This routine reads the V160 register at the offset passed.

Description

This routine reads the V160 register at the offset passed. The user should use the offsets defined in "cmdlist.h".

This routine builds a simple list to read the desired register, loads it into the KSC2962 command memory, and executes it. The user should see the KSC 2962 hardware documentation for the device register documentation.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
v160chassis	Input	Chassis address.
partition	Input	Which host partition to use.
v160cmdaddr	Input	Command address.

Return Values

KSC_HANDLE	Handle is invalid

3.6.27 KSC_v160_writereg

Syntax

```
int KSC_v160_writereg (struct KSC_handle *handle,
                       int partition,
                       int v160,
                       int reg_offset,
                       int reg_value);
```

Purpose

This routine writes the V160 internal register at the offset passed.

Description

This routine writes the V160 internal register at the offset passed. The user should use the offsets defined in "cmdlist.h". This routine builds a command list using the passed parameters and calls KSC_loadgo to execute the list thereby writing the register in the V160. The user should consult the KSC 2962 hardware manual for effects and documentation of the V160 device registers.

Parameters

Parameter Name	Direction	Description
handle	Input	Used by all device driver access routines. Returned by KSC_INIT.
partition	Input	Which host partition to use.
v160	Input	VXI crate address.
reg_offset	Input	Word address where to load.
reg_value	Input	Value to write to register.

Return Values

KSC_HANDLE	Handle is invalid.

4 CAMAC Command Line Utilities

This chapter describes the general-purpose CAMAC utilities available for simple testing. These utilities may be called from a DOS prompt, a batch file, or from the WINDOWS icon. Features included in the commands are single 24/16-bit CAMAC data transfers, control operations to the crate controller, and return of the crate controller status. Using the commands allow the user to verify that a given CAMAC module can be addressed, and that it is operating properly. In addition, it is a convenient way to become familiar with how the module functions before developing application code.

4.1 Command Summary

Command parameters define what the utility will act upon. All parameters are optional as indicated by brackets "[...]". The user will be prompted for any parameters that are not specified on the command line.

The following describes the execution of the utilities from the DOS prompt or a batch file with parameters.

4.1.1 CACTRL Utility

This utility does control functions to CAMAC chassis on the CAMAC Serial highway. CACRTL performs a crate wide CAMAC control operation (i.e., Init, Clear, Set Inhibit, Clear Inhibit, Online).

The syntax for the CACTRL utility is:

CACTRL [/C=] [/INIT] [/CLEAR] [/SETINH] [/CLRINH] [/ONLINE]

Note - All parameters may be omitted or when specified may be entered in any order.

Qualifier	Description
C	Chassis number of the crate (0 to 7). The default value is chassis one.
INIT	Assert the init line in the CAMAC chassis
CLEAR	Performs a CAMAC clear operation
SETINH	Set the dataway inhibit line in the CAMAC chassis
CLRINH	Clear the dataway inhibit line in the CAMAC chassis
ONLINE	Put the chassis online

CACTRL Examples

Example 1:
In this example, the first CACTRL command specifies the crate number and performs a control operation (crate online). The second CACTRL command will prompt the user for the crate number and sets the inhibit bit in the crate controller. As a result of the inhibit bit being set the LED on the crate controller is turned on and the inhibit dataway signal true.

CACTRL /ONLINE /C=3
CACTRL /SETINH

Example 2:
In this example, the first CACTRL command specifies the crate number and performs a control operation (crate online). The second CACTRL command prompts for the crate number and sets the inhibit bit in the crate controller. As a result of the inhibit bit being set the LED on the crate controller is turned on and the inhibit data way signal true. The third CACTRL command prompts for the crate and performs a CAMAC clear operation.

```
CACTRL /ONLINE /C=2
CACTRL /ONLINE /SETINH
CACTRL /CLEAR
```

4.1.2 CAM Utility

The CAM utility allows the user to do simple CAMAC operations. This utility should be used with caution as it performs commands to the target crate without any regard to the current applications running on the system. The syntax for the CAM utility is:

CAM [/C=] [/N=] [/A=] [/F=] [/DATA=]

Note - All parameters may be omitted or when specified may be entered in any order.

Qualifier	Description
C	Chassis number of the crate (0 to 7). The default value is chassis one.
N	Station number within the CAMAC chassis of the module to be selected.
A	Sub address to be selected within the CAMAC module. The default value is zero.
F	The CAMAC function code to be performed to the device. The default value is zero.
DATA	Optional Write data if the function requires data. The user may indicate hexadecimal by prepending a "X" to the value.

CAM executes a single 24-bit CAMAC data transfer. This command reads or writes 24 bits of data to or from a CAMAC module.

CAM Examples

Example 1:
In this example, the first CAMAC command performs a read function, F(0), from sub-address zero, A(0), of crate one, C(1) directed to slot 1, N(l). The second command also performs a read function from the same slot and address, but the user will be prompted for the crate number. The third CAMAC command will prompt the user for all parameters. The output for a read operation displays the data in both decimal and hexadecimal format. Although the output is listed only once in the following example, it would actually be produced by each of the read operations as they were executed.

```
CAM /C=1 /N=l /A=O /F=O
CAM /N=1 /A=O /F=O
CAM
```

Data returned from CAM24 in decimal = 32, in hex = 0x20

Example 2:
In this example, the first CAMAC command performs a write function, F(16), to sub-address zero, A(O), with a value of 10 directed to crate 2, slot 3. The second command also performs a write function however the data value is specified in hexadecimal format. The "x" is used to represent hex notation with a value "20".

CAM /C=2 /N=3 /A=O /F=16 /DATA=32
CAM /C=2 /N=3 /A=O /F=16 /DATA=x20

4.1.3 CCSTAT Utility

Displays the crate controller status (i.e., Inhibit status, L-SUM status, LAM register status, Crate Controller Status register, and Error Status register). The first two values are displayed in decimal, the remaining three values are in hexadecimal format. Refer to the crate controller manual for the meaning of the bits in the crate controller registers.

CCSTAT [/C=]

Qualifier	Description
C	Chassis number of the crate (0 to 7). The default value is chassis one.

Example 1:
In this example, the first CCSTAT command specifies the crate number and displays all crate controller status registers.

CCSTAT /C=1

Output -

Crate status for crate: 1
Inhibit Status = 1
LSUM status = 0
Lam Register (Box) = 0x40
Crate Controller Status Register = 0x 44
Error Status Register = 0x0

5 Resource Manager

5.1 Resource Manager Functionality

The Grand Interconnect (GI) VXI Resource Manager (RM) configures all devices in GI nodes with the KSC Model V160 GI-VXI Slot 0 Controller (hereafter called GI-VXI nodes). It fully complies with revision 1.4 of the VXIbus specification and revision 1.0 of the MXIbus specification. It operates properly with devices that comply with revisions 1.2, 1.3, or 1.4 of the VXIbus specification, particularly message-based devices designed to the earlier revisions. Any GI-VXI node may be configured with any of the following types of VXI devices:

- Static Configuration (SC) devices
- Dynamic Configuration (DC) devices
- VXI-MXI extenders with (or without) INTX

Non-VXI (pseudo-VXI and VME) devices and MXI A16 address windows are not supported in RM version 2.0.

Systems integrators familiar with other VXI Slot 0 Controllers and their associated Resource Manager software should note that the GI RM differs in only one major respect. The GI allows up to 126 distinct nodes, and each VXI node can have 256 devices. The GI RM must configure all GI-VXI nodes, and to do so it sequences through all possible nodes from 1 to 126 looking for V160s. Once found, the RM procedure for that node is equivalent to the RM run on those other controllers.

The GI RM will normally be run once for all nodes during system initialization, which is usually within seconds of power being applied to the VXI mainframes. However, there may be times when one or more nodes may be subsequently powered off while others remain powered on. (If at least one node in a GI system is powered off without backup power to its fiber-optic components, thus logically breaking the fiber-optic loop, it will not be possible to communicate with any of the remaining nodes.) To allow this, the RM can be run on a single node after its power has been restored, and the configuration data previously recorded for that node alone will be replaced with the current data. Data for other nodes will not be affected in this mode of operation.

It is important to note that the VXIbus specification requires a RM to be run exactly once after power-on or system reset. Therefore, if it appears to be necessary to re-run RM on a particular node, either SYSRESET* must be asserted in that node, or power to that node must be turned off momentarily. Failing to follow this requirement may place some VXI devices in an indeterminate state.

The GI RM software is also provided in archive library (lib.a) format, so that the system integrator may link the core RM with a portion of the data acquisition application. In this case, a call to the rmShell function replaces the command-line invocation of the RM. A "C" language header file is provided and contains the function prototype with which the RM call in an application must comply.

5.2 Resource Manager Files

The RM software and the RM input table files are copied to disk by the kit installation when the 2962 device driver is installed. The system integrator should verify that the RM executable, resman, has been copied to disk, and that the Resman Tables directory contains the following files:

1. gidevice.tbl
2. intcfg.tbl
3. mfnameid.tbl
4. model.tbl
5. trigutil.tbl

These are all plain comma delimited ASCII text files with one entry per line. Detailed information on these files, and the resman.tbl output file, will be found in the File Formats section below. In the following paragraphs, set-up comments for each of these files are provided in the (alphabetical) order in which the files are listed above.

The device name file, gidevice.tbl, is optional; the RM will assign device names based on the hexadecimal logical address (of the form "DEVICE_XX") unless a device's characteristics match one of the entries in this file. This file is unique in that its fields may contain "wildcard" values that match any corresponding device value. With this capability, for example, all the V160's can be given the same device name, regardless of node number. System integrators can defer the modification of this file until unique device names are to be assigned.

The interrupt configuration file, intcfg.tbl, serves two purposes in a GI system. The first purpose is to support MXI extenders and static interrupters. The second purpose is to support the static interrupt handlers of the V160. The hardware design of the V160 requires the use of this file if any V160 must handle interrupts on one or more of the VME IRQ lines. This file must contain one entry (line) for each interrupt-handling V160. System integrators should edit the supplied copy of this file as necessary to fulfill this requirement.

The system integrator should verify that all relevant manufacturers and models are listed in mfnameid.tbl and model.tbl, respectively. The RM will assign names of the forms "MFR_XXX" and "MODEL_XXX" if there is no corresponding entries in these table files for a device's manufacturer ID or model code, respectively. In both these forms, the XXX will be replaced with the hexadecimal representation of the value read from the device's configuration registers.

Only MXI extended systems require the trigger and utility bus configuration file, trigutil.tbl. The RM will write the values found in this file to the designated MXI extenders in the designated GI nodes. For information on how to determine the correct values to be entered in this file, consult the MXIbus specification and the user's manuals for the particular MXI extenders.

5.2.1 RM Path

The files described above can be placed anywhere on the system's disk. The RM must then be provided with the directory path to those tables so that it can access the files. Both the command- line interface and the core RM function (rmShell) have three methods to get this directory path: a path argument, an environment variable, and a default (the current directory path). The GIRM_TABLES environment variable should be set to the full path to the Resman Tables directory. This is particularly recommended in applications where the command-line interface is primarily used, as setting the environment variable in a start-up script (that should also load the device driver) is not as error-prone as requiring users to enter the directory path on the command line.

5.2.2 VXI Configuration

There are only two restrictions on the Logical Address (LA) settings of devices in each GI-VXI node. First, the V160 must be set to Logical Address 0 in mainframe slot 0, and the V160 Slot 0 functionality must be enabled. This allows the V160 to be the Resource Manager device in each GI-VXI node. Second,

consistent with the VXIbus specification, if there will be any DC devices in a GI-VXI node, no SC devices can be assigned to Logical Address 255. If RM detects an SC device at Logical Address 255, it will issue a warning and will not configure any DC devices in that node.

5.2.3 Highway Integrity

All nodes must be powered up and a complete fiber-optic loop must be established before the RM is run on the GI system. If at least one node in a GI system is powered off without backup power to its fiber-optic components, thus logically breaking the fiber-optic loop, it will not be possible to communicate with any of the other nodes. The green Sync LEDs on all GI components (2962, 3972s and V160s) will all be flashing or continuously on when the loop is complete. These LEDs will be of assistance in locating improper fiber-optic connections. If Sync is out on any node, it will also be out on the 2962. In this case, trace the loop starting with the 2962 fiber OUT connection, and proceed down the loop to the first node without Sync. Power up this node, or swap the fiber-optic cables on that node if power is already on. For further assistance, refer to the GI component's hardware instruction manual.

5.2.4 General

No access should be made to any GI-VXI node until the RM has been run on that node. Failure to do this will prevent access to A24 or A32 address space, and will prevent the use of operational commands to message-based devices.

5.2.5 COMMAND-LINE INTERFACE

The resman command has six optional flags that may be issued in any order. If a flag is not specified, the default value or behavior will be used. Flags can appear one or more times on the command line, but since flags are processed left-to-right, the rightmost value will take effect. It is important that the command must have "whitespace" (tabs or spaces) between all elements: the command name "resman," the flags, and the flag values. Unlike some Unix programs, it is not possible to merge a number of flags together. Each must be preceded by at least one space and the hyphen ("-") character. (For compatibility with other operating systems, the RM will also allow the slash ("/") character in place of the hyphen.)

On-line help is available for the resman command with the -h (help) flag. If this flag appears anywhere on the command line, the following message is displayed but the RM itself will not be started:

Usage:
resman [-c #] [-h] [-n #] [-q] [-t #] [-T <path>]
Flags:

-c	2962 controller number (default: 0)
-h	help - print this message
-n	GI node to manage (default: all nodes 1..126)
-q	quiet mode - suppress all messages (default: verbose)
-t	Sysfail delay time in seconds (default: 5)
-T	<path> Path to RM table directory (default: envir. GIRM_TABLES or current dir.)

The -c (controller) flag directs the RM to use a particular GI controller card in multi-highway systems. The flag must be followed by a non-negative integer designating the desired controller. The first 2962 Interconnect Highway Driver (IHD) in a system is always controller zero (0), which is the default; the

second IHD is controller one (1), and so on. The command "resman -c 1" would select the second controller.

The -n (node) flag allows the user to manage VXI resources in only one node, as would be necessary if power to a node was interrupted or if the node was reset. This flag must be followed by the base 10 (decimal) non-negative integer corresponding to the address of the node to be resource-managed. Note that the address switches on the V160 (and 3972 GI CAMAC crate controller) are base 16 (hexadecimal) and therefore the node address must be converted to its base 10 equivalent for use with this flag. The command "resman -n 32" will manage resources for the node whose address switch reads 20. The commands "resman -n 0" (zero) and "resman" are equivalent, as the default is zero (0) and will direct the RM to manage all GI-VXI nodes (from 1 to 126).

System integrators should note that each time the resman command is issued with the -n flag only one node will be managed. If it is necessary to manage 2 or more (but not all) nodes on the highway, the command must be repeated with a new node value each time.

The -q (quiet) flag turns off all standard output, but does not suppress error and warning messages. When this option is selected (as in "resman -q"), redirecting standard output to a file will result in an empty file.

The -t (time) flag sets the maximum time for the RM to wait for the SYSFAIL* line to be de-asserted. This flag must be followed by a non-negative integer representing the number of seconds to wait. The default is 5 seconds, which is consistent with the VXIbus specification. However, the resman command will allow values from zero to 4, which are not recommended; a value of zero will effectively disable the wait-for-SYSFAIL* logic. The command "resman -t 30" will set the delay time to 30 seconds.

System integrators should use the default unless the system includes a device that takes longer than 5 seconds to complete its self-test. In this case, the longest such time should be used on the command line. The RM starts its timer before the first node is managed, and does not reset it between nodes. If the longest delay is used, the slowest device will be ready when or before the RM's timer expires.

The -T (tables) flag indicates the path to the RM tables directory in which the RM configuration files are located. If supplied, it overrides the value associated with the GIRM_TABLES environment variable. This flag must be followed by a string of no more than PATH_MAX (currently) characters. Note: that if neither the -T flag nor the GIRM_TABLES variable are used, the resman command will look for table files in the current working directory.

5.2.6 THEORY OF OPERATION

This multi-frame RM performs the following steps:

1. Configure the Logical Address map if MXI extenders are present
2. Identify the devices in the system
3. Manage the system self-tests
4. Configure the A24 and A32 address maps
5. Establish the initial system hierarchy
6. Configure the interrupt, trigger, and utility resources if MXI extenders are present
7. Initiate normal system operation

Initially, RM will monitor the SYSFAIL* line via a V160 internal register on a periodic basis until it is de-asserted or the specified time delay expires, whichever occurs first. The following paragraphs describe each of the above steps in more detail.

5.2.7 FILE FORMATS

All numeric values are in hexadecimal, except Logical Address in resman.tbl, and are 16-bit values unless otherwise indicated. All strings are a maximum of 15 characters (vice 12 in NI files); the matching logic in the GI RM only compares as much of the string as is supplied. Therefore, a name may be truncated in one file and right-padded with blanks in another and the logic will treat them as a match. For example, 14- char "KineticSystems" in one file will still match 15-char "KineticSystems " in another. Edit the files with any text editor.

5.2.8 Grand Interconnect Device Table

The gidevice.tbl is an optional user supplied file. If a device does not support the MODID line, the RM will not be able to determine what slot it occupies (see the troubleshooting section below). For such modules, use FFFF in the slot field so the matching logic won't care which slot the device is in. Null string ("") for name entries matches anything.

Device Name (string)
Manufacturer Name (string)
Model Name (string)
Logical Address
Mainframe
Slot
GI Node

5.2.9 Interrupt Configuration File

The intcfg.tbl file is a user-supplied file. An example is provided. There must be an entry in this file for every V160 that will be handling interrupts. Only the (Extender) Logical Address, Interrupter Levels, and GI Node fields are used for V160s; the Levels field indicates which IRQs the V160 will monitor. This value is written to the V160s Interrupt Handler Mask register, and the appropriate IRQ lines are assigned to the V160. If the V160 will not be handling interrupts, and there are no MXI extenders in the same node, the Levels field should be zero (0) so the V160 handlers are not enabled. There is no need to put zeroes in the appropriate Handler Logical Address fields; the RM does this itself. The Interrupter Directions field has no effect on the V160 and can have any value; 0 is recommended.

For MXI extenders, the Interrupter Levels and Interrupter Directions fields will be concatenated and written to the MXI Interrupt Configuration register.

The Handler Logical Address fields should be set to FF unless the system requirements dictate that a specific handler must be assigned to that IRQ line. This supports devices whose interrupts are enabled on the module with switches or jumpers, not via configuration registers. In these cases, the logical address of the interrupt handler (not the interrupter) should be placed in the appropriate field. The value FF allows the RM assign IRQ lines to handlers (see Interrupt Allocation under Theory of Operation, above).

Extender Logical Address (0 for V160)
Interrupter Levels (7-bit bit vector)
Bits 6-0 correspond to IRQs 7-1, respectively
Set to 1 to enable the IRQ, 0 to disable it
Interrupter Directions (7-bit bit vector)
Bits 6-0 correspond to IRQs 7-1, respectively
Set to 1 to route into mainframe, 0 to route out of mainframe
VME IRQ 1 Handler Logical Address
VME IRQ 2 Handler Logical Address
VME IRQ 3 Handler Logical Address
VME IRQ 4 Handler Logical Address
VME IRQ 5 Handler Logical Address
VME IRQ 6 Handler Logical Address
VME IRQ 7 Handler Logical Address
GI Node

5.2.10 Manufacturer Name Table

The mfnameid.tbl file documents known manufactures of different VXI devices.

Manufacturer Name (string)
Manufacturer ID

5.2.11 Model Table

The model.tbl file (no GI node # field) usable as-is; add devices as necessary (may want to keep separate master file & extract devices that actually appear in system)

Model Name (string)
Manufacturer Name (string)
Model Code

5.2.12 Resource Table

The resman.tbl is the only file generated by the resource manager. This file is used by the VISA when the default resource table is opened.

1. Logical Address (expressed in base 10)
2. Device Name (string)
3. Commander's Logical Address
4. Mainframe (NI uses Extender Logical Address)
5. Slot
6. Manufacturer ID
7. Manufacturer Name (string)
8. Model Code
9. Model Name (string)
10. VXI Device Class
11. VXI Extended Subclass
12. VXI Address Space Code
13. VXI Address Base for memory (32-bit integer)
14. VXI Address Size for memory (32-bit integer)
15. Memory Class Attributes
16. VXI Interrupter Levels
17. VXI Interrupt Handler Levels
18. Extender/Controller data
19. Word-Serial "Async Mode Control" response
20. Word-Serial "Control Response" response
21. Contents of Protocol Register
22. Miscellaneous Flags
23. Status Register state
24. GI Node

5.2.13 Trigger Table

The trigutil.tbl file is an optional user supplied file. An example is provided.

Extender Logical Address
TTL Trigger Lines (8-bit bit vector)
    Bits 7-0 correspond to triggers 7-0, respectively
    Set to 1 to enable the trigger, 0 to disable it
TTL Trigger Directions (8-bit bit vector)
    Bits 7-0 correspond to triggers 7-0, respectively
    Set to 1 to route into mainframe, 0 to route out of mainframe
ECL Trigger Lines (6-bit bit vector)
    Bits 5-0 correspond to triggers 5-0, respectively
    Set to 1 to enable the trigger, 0 to disable it
ECL Trigger Directions (6-bit bit vector)
    Bits 5-0 correspond to triggers 5-0, respectively
    Set to 1 to route into mainframe, 0 to route out of mainframe
Utility Bus Register value (6-bit integer)
GI Node

6 VISA Library

KineticSystems is a member of the VXI Plug and Play Systems Alliance. KineticSystems provides this library to allow users to interact with devices in the VXI chassis on the Grand Interconnect Highway using the VISA (Virtual Instrument Software Architecture). It should be noted that most VXI implementations are done with the VXI chassis mapped directly to I/O space of the host processor. KineticSystems implementation is via a high-speed data highway, the Grand Interconnect.

6.1 VISA Overview

The VISA Library (VISA) for the KSC 2962 device driver is designed to implement simple functioning of VXI cards in crates that exist on the Grand Interconnect Highway. These routines make calls to the Application Programming Interface (API) which in turn makes the actual system calls to the KSC 2962 device driver. All VISA calls and error codes are defined by the VISA header files.

VISA only provides a subset of the functionality that is available using the KSC 2962 device and the V160 Slot0 controller. To truly use the functionality of these devices, the API must be used.

The key to much of the VISA Plug and Play is based on the functioning of the Resource Manager. The Resource Manager is executed as part of system startup. The Resource Manager identifies all devices, instruments, or resources available within chassis on the Grand Interconnect highway. The execution of the Resource Manager results in the generation of the resman.tbl file. This file is used by all processes that use the VISA library. This file is read by the VISA routine: viOpenDefaultRM.

The current release of the VISA is implemented as a linkable object library. Future releases may be implemented as a shared library. Users must relink their VISA applications with subsequent releases of this software.

6.1.1 VISA Routines Overview

The VISA specification states a minimum of set of function calls that each device must support. It also amplifies certain mapping functions for directly mapped devices. This direct mapping does not fit into the Grand Interconnect architecture. The functions that are part of the VISA library are:

- viAssertTrigger - Assert a software trigger
- viClear - Clear a device
- ViClose - Closes a session with a device
- viFindNext - Returns the next device found during a previous call to viFindRsrc
- viFindRsrc - Queries a VISA system to locate the devices associated with a specified interface.
- viGetAttribute - Retrieves the state of a resource attribute.
- viIn8 - Reads an eight bit value from the specified memory space (assigned memory-base + offset).
- viIn16 - Read a sixteen bit word from the specified memory space (assigned memory space plus offset)
- viIn32 - Reads in a thirty-two bit word from the specified memory space (assigned memory space + offset)
- ViMapAddress - Maps memory space
- ViMove - Moves a block of data
- ViMoveIn8 - Moves a block of data from the specified memory space (assigned memory space + offset) to local memory
- ViMoveIn16 - Moves a block of data from the specified memory space (assigned memory space + offset) to local memory

- ViMoveIn32 - Moves a block of data from the specified memory space (assigned memory space + offset) to local memory
- ViMoveOut8 - Moves a block of data from local memory to the specified address space and offset
- ViMoveOut16 - Moves a block of data from local memory to the specified address space and offset
- ViMoveOut32 - Moves a block of data from local memory to the specified address space and offset
- viOpen - Opens a session to the specified device
- viOpenDefaultRM - Creates a session with the Default Resource Manager
- viOut8 - Writes an 8-bit word to the specified memory space
- viOut16- Writes a 16-bit word to the specified memory space
- viOut32 - Write a 32-bit word to the specified memory space
- viPeek8 - Reads an 8-bit value from the specified address location
- viPeek16 - Reads a 16-bit value from the specified address location
- viPeek32- Reads a 32-bit value from the specified address location
- viPoke8 - Writes an 8-bit value to the address location pointed to by address
- viPoke16 - Writes a 16-bit value to the address location pointed to by address
- viPoke32 - Writes a 32-bit value to the address location pointed to by address
- viPrintf - Write data to a device using formatting of data
- viQueryf - Performs a formatted write and read through a single operation invocation
- ViRead - Reads data from a device synchronously
- viReadSTB - Reads a status byte of the service request
- viScanf - Reads data from a device using formatted input
- viSetAttribute - Sets the state of a resource attribute for a device
- viSPrintf - Formats write to a user-specified buffer using a variable number of arguments
- viSScanf - Formats read from a user-specified buffer using a variable number of arguments
- viStatusDesc - Returns a user readable string that describes the passed status code
- viUnmapAddress - UNmaps memory space
- viVPrintf - Write data to a device using formatting of data
- viVQueryf - Performs a formatted write and read through a single operation invocation
- viVScanf - Read data from a device using formatted input
- viVSPrint - Formats write to the device using a variable argument list
- viVSScanf - Formats read from a user-specified buffer using a variable argument list
- viVxiCommandQuery - Sends the device a miscellaneous command or query and/or retrieves the response to a previous query.
- viWrite - Writes Data to a device synchronously

The first time that any of the VISA calls are made, an initialization must be done. The VISA standard provides no interface for opening the VISA library or for initializing the VISA on a particular platform. Further, the standard calls provide no way to provide a pointer to any VISA global structure (such as a file handle). This limitation suggest that the library must provide dynamic storage within itself which will limit the ability to generate a shareable reentrant library. The "sesn" argument is used only for the session manager calls (viFindRsrc, ViOpen, ViFindNext, viFindFirst).

Users programming with VISA must use the visa32.lib file to link to the visa32.dll. Users must #include visa.h.

6.1.2 viAssertTrigger

Syntax
int viAssertTrigger (viSession vi,
viUInt16 protocol);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to assert a software or hardware trigger.

Description
This routine is part of the VISA Library (VISA) and its purpose is to assert a software or hardware trigger. This function will source a software or hardware trigger dependent on the interface type. For a GPIB device, the device is addressed to listen, and then the GPIB 'GET' command is sent. For a VXI device, if VI_ATTR_TRIG_ID is VI_TRIG_SW, then the device is sent the Word Serial Trigger command. For a VXI device, if VI_ATTR_TRIG_ID is any other value, a hardware trigger is sent on the line corresponding to the value of that attribute.

For a VXI device, if VI_ATTR_TRIG_ID is any other value, a hardware trigger is sent on the line corresponding to the value of that attribute.

VI_TRIG_SW	(-1) (Use word serial command "GET")
VI_TRIG_TTL0	(0)
VI_TRIG_TTL1	(1)
VI_TRIG_TTL2	(2)
VI_TRIG_TTL3	(3)
VI_TRIG_TTL4	(4)
VI_TRIG_TTL5	(5)
VI_TRIG_TTL6	(6)
VI_TRIG_TTL7	(7)
VI_TRIG_ECL0	(8)
VI_TRIG_ECL1	(9)

For GPIB and VXI software triggers, VI_TRIG_PROT_DEFAULT is the only valid protocol. For VXI hardware registers, VI_TRIG_PROT_DEFAULT is equivalent to VI_TRIG_PROT_SYNC.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
protocol	input	Trigger protocol to use during assertion. Valid values are: VI_TRIG_PROT_DEFAULT, VI_TRIG_PROT_ON, VI_TRIG_PROT_OFF, VI_TRIG_PROT_SYNC

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.
VI_ERROR_INV_MASK	Unknown trigger type mask.

6.2 viClear

Syntax
int viClear (viSession vi);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to clear a device.

Description
This routine is part of the VISA Library (VISA) and its purpose is to clear a device.
This function performs an IEEE 488.2-style clear of the device (for VXI, the Word Serial Clear command should be used; for GPIB systems, the Selected Clear command should be used).

Note an invocation of the viClear function on an INSTR resource will discard the read and write buffers used by the formatted I/O services for that session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.

6.3 viClose

Syntax
int viClose (viSession vi);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to close the specified session.

Description
This routine is part of the VISA Library (VISA) and its purpose is to close the specified session.

This function closes a session to a device, event queue or a "find session". By calling this function a process frees all the data structures that had been allocated for the session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

6.4 viFindNext

Syntax
int viFindNext (viSession vi,
viPRsrc instrDesc);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to return the next device found in the linked list of devices that match the user specified search string during a previous call to viFindRsrc().

Description
This routine is part of the VISA Library (VISA) and its purpose is to return the next device found in the linked list of devices that match the user specified search string during a previous call to viFindRsrc(). The list is referenced by the handle that was returned by viFindRsrc().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
instrDesc	Input	Returns a string identifying the location of a device. Strings can then be passed to viOpen() to establish a session to the given device.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in system.

6.5 viFindRsrc

Syntax

int viFindRsrc (viSession vi,
viString expr,
viFindList findList_ptr,
viUInt_32 retCnt,
viRsrc desc);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to query a VISA system to locate the devices associated with a specific interface.

Description

This routine is part of the VISA Library (VISA) and its purpose is to query a VISA system to locate the devices associated with a specific interface. It accomplishes this by matching the value specified in the 'expr' parameter with the devices available for a particular interface. On successful completion, it returns the first device found in the list along with a count to indicate if there were more devices found for the designated interface.

This function also returns a handle to a FIND session. This handle, 'vi' points to the list of devices and it must be used as an input to viFindNext(). When this handle is no longer needed, it should be passed to viClose().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
expr	Input	When an incomplete name of class identifier of a resource is specified in this parameter, this operation searches for all resources with names matching 'expr'.
findList_ptr	Input	Returns a handle identifying this search session. This handle will be used as input in viFindNext(). It should be initialized to VI_NULL.
retCnt	Input	Number of resources that match 'expr'.
desc	Input	Returns a string identifying the location of a device. Strings can then be passed to viOpen() to establish a session to the given device.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid

VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ALLOC	Insufficient system resources.
VI_ERROR_INV_EXPR	Invalid expression specified for search.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in system.

6.6 viGetAttribute

Syntax

int viGetAttribute (viSession vi,
viAttr attribute,
viPAttrState attrState);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to retrieve the state of a resource attribute.

Description

This routine is part of the VISA Library (VISA) and its purpose is to retrieve the state of a resource attribute.

This function retrieves the state of a specified attribute from the specified session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
attribute	Input	Resource attribute for which the state query is made.
attrState	Input	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced session, event, or find list.

6.7 viIn16

Syntax

```
int viIn16 (viSession vi,
        viUInt16 space,
        viBusAddress offset,
        viPUInt16 value);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read in a 16-bit word from the specified memory space (assigned memory space + offset).

Description

(Register based function.)

This routine is part of the VISA Library (VISA) and its purpose is to read in a 16-bit word from the specified memory space (assigned memory space + offset).

This function uses the specified address space to read in 16 bits of data from the specified offset of the device associated with this INSTR Resource.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

offset	Input	Offset (in bytes) of the device to read from.
value	Input	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.

VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_INV-SETUP	Invalid setup.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.

6.8 viIn8

Syntax

```
int viIn8 (viSession vi,
        viUInt16 space,
        viBusAddress offset,
        viPUInt8 value);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read an 8-bit value from the specified memory space (assigned memory-base + offset).

Description

This routine is part of the VISA Library (VISA) and its purpose is to read an 8-bit value from the specified memory space (assigned memory-base + offset).

This function, by using the specified address space, reads in 8 bits of data from the specified offset of the device associated with this INSTR resource. This operation does not require viMapAddress() to be called prior to its invocation.

Note an invocation of the viClear function on an INSTR resource will discard the read and write buffers used by the formatted I/O services for that session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

offset	Input	Offset (in bytes) of the device to read from.
value	Input	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid

VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_INV_SETUP	Invalid setup.

6.9 viMapAddress

Syntax

```
int viMapAddress (viSession vi,
        viUInt16 mapSpace,
        viBusAddress mapBase,
        viBusSize mapSize,
        viBoolean Access,
        viAddr suggested,
        viPAddr address);
```

Purpose

If suggested parameter is not VI_NULL, the operating system attempts to map the memory to the address specified in suggested.

Description

(Register based functions.)

This routine is part of the VISA Library (VISA) and its purpose is to map memory space.

This function maps in a specified memory space. The memory space that is mapped is dependent on the type of interface specified by the vi parameter and mapSpace parameter (refer to the following table).

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
mapSpace	Input	If the vi is a VXI or MXI session, mapSpace is used to specify the address space to map.
mapBase	Input	Offset of the memory to be mapped.
mapSize	Input	Amount of memory to map.
Access	Input	Specifies whether to request owner privileges with this mapping. Having owner privileges lets you modify the hardware context.
suggested	Input	If suggested parameter is not VI_NULL, the operating system attempts to map the memory to the address specified in suggested. There is no guarantee, however, that the memory will be mapped to that address. This operation may map the memory into an address region different from suggested.
address	Input	Address in your process space where the memory was mapped.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid

VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_INV-OFFSET	Invalid offset specified.

6.10 viMove

Syntax

```
int viMove (ViSession vi,
        ViUInt16 srcSpace,
        ViBusAddress srcOffset,
        ViUInt16 srcWidth,
        ViUInt16 destSpace,
        ViBusAddress destOffset,
        ViUInt16 destWidth,
        ViBusSize length);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

This operation moves data from the specified source to the specified destination. The source and the destination can either be local memory or the offset of the interface with which this MEMACC Resource is associated. This operation uses the specified data width and address space. In some systems, such as VXI, users can specify additional settings for the transfer, like byte order and access privilege, by manipulating the appropriate attributes.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
srcSpace	Input	
srcOffset	Input	
srcWidth	Input	
destSpace	Input	
destOffset	Input	
destWidth	Input	
length	Input	

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.

VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.11 viMoveIn8

Syntax

```
int viMoveIn8 (ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt8 buf8);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

This function uses the specified address space to read in 8 bits of data from the specified offset.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf8	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.12 viMoveIn16

Syntax

```
int viMoveIn16 (ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt16 buf16);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

This function uses the specified address space to read in 16 bits of data from the specified offset.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf16	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.13 viMoveIn32

Syntax

```
int viMoveIn32 (ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt16 buf32);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

This function uses the specified address space to read in 32 bits of data from the specified offset.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf32	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.14 viMoveOut8

Syntax

```
int viMoveOut8 (ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt8 buf8);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from local memory to the specified address space and offset.

This function, by using address space, writes 8 bits of data to the specified offset. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf8	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.15 viMoveOut16

Syntax

```
int viMoveOut16(ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt16 buf16);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from local memory to the specified address space and offset.

This function, by using address space, writes 16 bits of data to the specified offset. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf16	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.16 viMoveOut32

Syntax

```
int viMoveOut32(ViSession vi,
        ViUInt16 space,
        ViBusAddress offset,
        ViBusSize length,
        ViAUInt32 buf32);
```

Purpose

This routine is part of the VISA Library and its purpose is to move a block of data from the specified memory space (assigned memory space + offset) to local memory.

Description

(Register based function)

This routine is part of the VISA Library and its purpose is to move a block of data from local memory to the specified address space and offset.

This function, by using address space, writes 32 bits of data to the specified offset. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	
offset	Input	Offset of the starting address to read
length	Input	Number of elements to transfer
buf32	Input	Data read from bus

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.17 viOpen

Syntax

```
int viOpen(ViSession sesn,
        ViRsrc rsrcName,
        ViAccessMode accessMode,
        ViUInt32 timeout,
        ViSession vi);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to open a session to the specified device.

Description

This routine is part of the VISA Library (VISA) and its purpose is to open a session to the specified device (creates the necessary data structures to communicate with the device.)

This operation opens a session to the specified device and returns a session identifier that will be used to call other routines to perform operations on that device.

The address string should look like one of the following:

VXI[board]::LOGICAL ADDR[::INSTR]
ie. VXI0::1::INSTR

GPIB-VXI[board][::GPIB-VXIprimary addr]::VXI logical addr::[INSTR]
ie. GPIB-VXI::9::INSTR

GPIB[board]::primary addr[::secondary addr][::INSTR]
ie. GPIB::1::0

Parameters

Parameter Name	Direction	Description
sesn	Input	Resource Manager session (should always be viDefaultRM for VISA).
rsrcName	Input	Unique symbolic name of a resource, with the appropriate initialization.
accessMode	Input	VI_NULL for VISA .
timeout	Input	VI_NULL for VISA .
vi	Output	Unique logical identifier reference to a session.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.

VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_INV-ACC_MODE	Invalid access mode.
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
VI_ERROR_ALLOC	Insufficient system resources.
VI_WARN_CONFIG_NLOADED	The specified configuration either does not exist or could not be loaded; using VISA-specified defaults.
VI_SUCCESS_DEV_NPRESENT	Session opened successfully, but the device at the address is not responding.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in system.
VI_ERROR_SYSTEM_ERROR	Unknown system error.

6.18 viOpenDefaultRM

Syntax
int viOpenDefaultRM(ViSession *sesn_ptr);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to create a session with the Default Resource Manager.

Description
This routine is part of the VISA Library (VISA) and its purpose is to create a session with the Default Resource Manager.

This function must be called before any VISA operations can be invoked. The first call to this function initializes the VISA system, including the Default Resource Manager resource, and also returns a pointer to that resource. Subsequent calls to this function return unique session pointers to the same Default Resource Manager resource.

This routine uses the following functions:

ksc_init	allocate/populate the device handle structure
ksc_set_partitions	define the size of the device partitions

Parameters

Parameter Name	Direction	Description
sesn_ptr	Output	Unique pointer to the default resource manager.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_ALLOC	Insufficient system resources.
VI_ERROR_SYSTEM_ERROR	Unknown system error

6.19 viOut8

Syntax

```
int viOut8 (viSession vi,
        viUInt16 space,
        viBusAddress offset,
        viUInt8 value);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 8-bit word to the specified memory space.

Description

(Register based function.)

This routine is part of the VISA Library (VISA) and its purpose is to write a 8-bit word to the specified memory space (assigned memory space + offset.)

This function, by using address space, writes 8 bits of data to the specified offset of the device associated with this INSTR resource. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

offset	Input	Offset (in bytes) of the device to read from.
value	Input	8 bit value written to the bus.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.

VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.20 viOut16

Syntax

```
int viOut16 (viSession vi,
        viUInt16 space,
        viBusAddress offset,
        viUInt16 value);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 16-bit word to the specified memory space.

Description

(Register based function.)

This routine is part of the VISA Library (VISA) and its purpose is to write a 16-bit word to the specified memory space (assigned memory space + offset.)

This function, by using address space, writes 16 bits of data to the specified offset of the device associated with this INSTR resource. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

offset	Input	Offset (in bytes) of the device to read from.
value	Input	16-bit value written to the bus.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.

VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.21 viOut32

Syntax

```
int viOut32 (viSession vi,
        viUInt16 space,
        viBusAddress offset,
        viUInt32 value);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 32-bit word to the specified memory space.

Description

(Register based function.)

This routine is part of the VISA Library (VISA) and its purpose is to write a 32-bit word to the specified memory space (assigned memory space + offset.)

This function, by using address space, writes 32 bits of data to the specified offset of the device associated with this INSTR resource. This operation does not require viMapAddress() to be called prior to its invocation.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

offset	Input	Offset (in bytes) of the device to read from.
value	Input	32-bit value written to the bus.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.

VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_INV_SETUP	Invalid setup.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.

6.22 viPeek8

Syntax

viPeek8 (viSession vi,
viUInt16 space,
viPUInt8 val8);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read a 8-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read a 8-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

va18	Input	Value read from the bus.

Return Values

No Errors

6.23 viPeek16

Syntax

viPeek16(viSession vi,
viUInt16 space,
viPUInt16 val16);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read a 16-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read a 16-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

val16	Input	Value read from the bus.

Return Values

No Errors

6.24 viPeek32

Syntax

viPeek32(viSession vi,
viUInt16 space,
viPUInt32 val32);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read a 32-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read a 32-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

val32	Input	Value read from the bus.

Return Values

No Errors

6.25 viPoke8

Syntax

viPoke8 (viSession vi,
viUInt16 space,
viUInt8 val8);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 8-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to write a 8-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

val8	Input	Value written to the bus.

Return Values

No Errors

6.26 viPoke16

Syntax

viPoke16(viSession vi,
viUInt16 space,
viUInt16 val16);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 16-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to write a 16-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

val16	Input	Value written to the bus.

Return Values

No Errors

6.27 viPoke32

Syntax

viPoke32(viSession vi,
viUInt16 space,
viUInt32 val32);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write a 32-bit value from the specified address location.

Description

This routine is part of the VISA Library (VISA) and its purpose is to write a 32-bit value from the specified address location. The address must be a valid memory address in the current process previously mapped by a call to viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
space	Input	Specifies the address space:

Value	*Description*
VI_A16_SPACE	A16 address space of VXI/MXI bus
VI_A24_SPACE	A24 address space of VXI/MXI bus
VI_A32_SPACE	A32 address space of VXI/MXI bus

val32	Input	Value written to the bus.

Return Values

No Errors

6.28 viPrintf

Syntax
int viPrintf(viSession vi,
viString writeFmt,
[arg1,arg2,...]);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to convert, format and send the parameters arg1, arg2,...

Description
This routine is part of the VISA Library (VISA) and its purpose is to convert, format, and send the parameters arg1, arg2,... to the device as specified by the format string.

Functionally, this routine works much like the standard C PRINTF routine, but instead will send the output to the specified device. Within the format string, the following characters can be used to insert special characters into the output stream:

Format String	Description
\n	Sends the ASCII LF character. The END identifier is also sent.
\r	Sends the ASCII CR character.
\t	Sends the ASCII TAB character.
\###	Sends the ASCII character represented by the octal value ###.
\"	Sends the ASCII double quote.
\\	Sends a backslash.

Format conversion of the passed arguments is specified by including a percent symbol (%), followed by any modifying arguments or flags, and then ended with a format specified character. Format conversion of arguments is taken in order from left to right, matching each conversion with one argument. Note that some arguments may also take variable arguments, which are taken from the argument list before the actual value of the argument. Modifiers occurring between a % symbol and the format character may include:

Format Code Modifier	Description
-	Left justify the output within its field length.
+	Requests that an explicit sign be placed before any numerical conversions. Normally, only negative signs are placed if the number is already negative.
space	Prefix a space to the front of a numerical conversion. This is ignored if the + flag is specified.
0	Use zeros rather than spaces to pad the output field.
field width	A number specifying the size of the field width. If this is an asterisk, then the next argument in the argument list is converted to represent the size of the input field.

.precision	A number of an asterisk is used to define the number of digits to appear for d,i,o,u,x and X conversions; the number of digits to appear after a decimal point for e,E, and f conversions; the number of significant digits for g and G conversions; or the maximum number of characters written is a s conversion. If an asterisk is specified, this value is taken from the next argument in the passed argument list.
,array size	A comma followed by a number or an asterisk indicates that the next argument in the argument list points to an array of integers. This array is then output using *array size* to indicate how many elements in the array to output. A comma separates each element in the array. If an asterisk is specified, the *array* size value is taken from the next argument in the passed argument list.
h,l, or L	Convert the passed argument as a short (h), unsigned long (l), or double long (L) integer (AXP only).

Valid conversion specifiers are:

Format Conversion	Description
d,i	Converts signed int to a decimal format.
o	Converts and unsigned int to octal format.
u	Converts and unsigned int to an unsigned decimal.
x,X	Converts and unsigned into a hexadecimal number. Lowercase 'x' will use lowercase letters in the output, uppercase 'X' will use uppercase letters in the output.
f	Converts a float or double into a decimal floating point number.
e,E	Converts a float or double into a decimal number represented in scientific notation.
c	Converts an int argument into a single ASCII character.
s	Converts the pointer to a string argument and outputs the string pointed to by the argument.
%	Writes the percent symbol.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
writeFmt	Input	String describing the format for arguments to be output.
arg1,arg2,...	Input	List of parameters used by the format string.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.

VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code

6.29 viQueryf

Syntax

```
int viQueryf(viSession vi,
        viString readFmt,
        viString writeFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to perform write and read through a single operation invocation.

Description

This operation provides a mechanism of "Send, then receive" typical to a command sequence from a commander device. In this manner, the response generated from the command can be read immediately.

This operation is a combination of the viPrintf() and viScanf () operations. The n arguments corresponding to the first format string are formatted by using the writeFmt string and then sent to the device. The write buffer is flushed immediately after the write portion of the operation completes. After these actions, the response data is read from the device into the remaining parameters (starting from parameter n+1) using the readFmt string.

Format String	Description
\n	Sends the ASCII LF character. The END identifier is also sent.
\r	Sends the ASCII CR character.
\t	Sends the ASCII TAB character.
\###	Sends the ASCII character represented by the octal value ###.
\"	Sends the ASCII double quote.
\\	Sends a backslash.

Format conversion of the passed arguments is specified by including a percent symbol (%), followed by any modifying arguments or flags, and then ended with a format specifier character. Format conversion of arguments is taken in order from left to right, matching each conversion with one argument. Note that some arguments may also take variable arguments, which are taken from the argument list before the actual value of the argument. Modifiers occurring between a % symbol and the format character may include:

Format Code Modifier	Description
-	Left justify the output within its field length.
+	Requests that an explicit sign be placed before any numerical conversions. Normally, only negative signs are placed if the number is already negative.
space	Prefix a space to the front of a numerical conversion. This is ignored if the + flag is specified.
0	Use zeros rather than spaces to pad the output field.
field width	A number specifying the size of the field width. If this is an asterisk, then the next argument in the argument list is converted

	to represent the size of the input field.
.precision	A number of an asterisk is used to define the number of digits to appear for d,i,o,u,x and X conversions; the number of digits to appear after a decimal point for e,E, and f conversions; the number of significant digits for g and G conversions; or the maximum number of characters written is a s conversion. If an asterisk is specified, this value is taken from the next argument in the passed argument list.
,array size	A comma followed by a number or an asterisk indicates that the next argument in the argument list points to an array of integers. This array is then output using *array size* to indicate how many elements in the array to output. A comma separates each element in the array. If an asterisk is specified, the *array* size value is taken from the next argument in the passed argument list.
h,l, or L	Convert the passed argument as a short (h), unsigned long (l), or double long (L) integer (AXP only).

Valid conversion specifiers are:

Format Conversion	Description
d,i	Converts signed int to a decimal format.
o	Converts and unsigned int to octal format.
u	Converts and unsigned int to an unsigned decimal.
x,X	Converts and unsigned into a hexadecimal number. Lowercase 'x' will use lowercase letters in the output, uppercase 'X' will use uppercase letters in the output.
f	Converts a float or double into a decimal floating point number.
e,E	Converts a float or double into a decimal number represented in scientific notation.
c	Converts an int argument into a single ASCII character.
s	Converts the pointer to a string argument and outputs the string pointed to by the argument.
%	Writes the percent symbol.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
writeFmt	Input	String describing the format for arguments to be output.
readFmt	Input	String describing the format for arguments to be input.
[arg1,arg2,...]	Input	Value written to the bus.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid

VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

6.30 viRead

Syntax

```
int viRead (viSession vi,
        viPBuf buf,
        viUInt32 cnt,
        viPUInt32 retCnt);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read data from a device synchronously.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read data from a device synchronously.

The synchronous read operation synchronously transfers data. The data read is to be stored in the buffer represented by buf. This operation returns only when the transfer terminates. Only one synchronous read operation can occur at any one time.

A viRead() operation can complete successfully if one or more of the following conditions are met: A) END indicator received. B) Termination character read. C) Number of bytes read is equal to count. It is possible to have one, two, or all three of these conditions satisfied at the same time.

IF an END indicator is received, THEN viRead() SHALL return VI_SUCCESS, regardless of whether the termination character is received or number of bytes read is equal to count.

IF no END indicator is received and the termination character is read, THEN viRead() SHALL return VI_SUCCESS_TERM_CHAR, regardless of whether the number of bytes read is equal to count.

IF no END indicator is received, no termination character is read and the number of bytes read is equal to count, THEN viRead() SHALL return VI_SUCCESS_MAX_CNT.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Output	Represents the location of a buffer to receive data from a device.
cnt	Input	Number of bytes to be read.
retCnt	Input	Represents the location of an integer that will be set to the number of bytes actually transferred.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

6.31 viReadSTB

Syntax
int viReadSTB (viSession vi,
viPUInt16 protocol);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to read a status byte of the service request.

Description
This routine is part of the VISA Library (VISA) and its purpose is to read a status byte of the service request.

This function reads a service request status from a service requester (the message-based device). For example, on the IEEE 488.2 interface, the message is read by polling devices; for other types of interfaces, a message is sent in response to a service request to retrieve status information. If the status information is only one byte long, the most significant byte is returned with the zero value. If the service requester does not respond in the actual timeout period, VI_ERROR_TMO is returned.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
protocol	Input	Service request status byte.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.

6.32 viScanf

Syntax
int viScanf (viSession vi,
viString readFmt,
[arg1,arg2,...]);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to read, convert and format data using the format specifier.

Description
This routine is part of the VISA Library (VISA) and its purpose is to read, convert and format data using the format specifier. Store the formatted data in the arg1, arg2... parameters.

This function receives data from a device, formats it by using the format string, and stores the resultant data in the arg parameter list. The format string can have format specifier sequences, white characters, and ordinary characters. The white characters-blank, vertical tabs, horizontal tabs, form feeds, new line/linefeed, and carriage return- are ignored except in the case of %c and %[ ]. All other ordinary characters except % should match the next character read from the device.

The format string consists of a %, followed by optional modifier flags, followed by one of the format codes in that sequence. It is of the form

%[modifier]format code

where the optional modifier describes the data format, while format code indicates the nature of data (data type). One and only one format code should be performed at the specifier sequence. A format specification directs the conversion to the next input arg. The results of the conversion are placed in the variable that the corresponding argument points to, unless the * assignment-suppressing character is given. In such a case, no arg is used and the results are ignored.

The viScanf() operation accepts input until the END indicator is read or all the format specifiers in the readFmt string are satisfied. Thus, detecting an END indicator before the readFmt string is fully consumed will result in ignoring the rest of the format string. Also, if some data remains in the buffer after all format specifiers in the readFmt string are satisfied, the data will be kept in the buffer and will be used by the next viScanf operation.

The viRead() operation is used for the actual low-level read from the device. Therefore, viRead() should not be used in the same session with formatted I/O operations. Also, if multiple sessions using formatted I/O resources are connected to the same device, the actual low-level reads must be synchronized between themselves.

Some of the format modifiers that are allowed include:

Format Code Modifier	Description
*	Assignment suppressing character.
field width	A number specifying the size of the field width. If this is an asterisk, then the next argument in the argument list is converted to represent the size of the input field.
,array size	A comma followed by a number or an asterisk indicates that the next argument in the argument list points to an array of integers. This array is then filled using *array size* numbers from the input stream. If an asterisk is specified, the *array* size value is taken from the next argument in the passed argument list.
[code]	Indicates that code is to be used as a terminating character for the input string. It must be a single character enclosed by square brackets.
h,l, or L	Convert the passed argument as a short (h), unsigned long (l), or double long (L) integer (AXP only).

Valid conversion specifiers are:

Format Conversion	Description
d	Converts a decimal integer. The argument is a pointer to an int.
i	Converts an integer according to its digits. A leading 0 will result in octal conversion, a 0X will result in hex conversion, otherwise a default of decimal conversion is used. The conversion type must be a pointer to an int.
o	Converts an octal integer. Argument is a pointer to an int.
x	Converts a hexadecimal number. Leading '0X' is ignored if it exists. Argument must be a pointer to an int.
e,f,g	Convert a floating point or scientific notation number. Argument must be a pointer to a float.
c	Converse a single character. Argument must be a pointer to a char.
s	Converts a character string. Argument must be a pointer to a char with enough space to hold the entire string.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
readFmt	Input	String describing the format for arguments.
[arg1,arg2,...]	Input	A list with the variable # of parameters into which the data is read and the format string is applied.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.

VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

6.33 viSetAttribute

Syntax

```
int viSetAttribute (ViSession vi,
                    ViAttr attrName,
                    ViAttrState attrValue);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to set the state of a resource attribute.

Description

This routine is part of the VISA Library (VISA) and its purpose is to set the state of a resource attribute.

This function modifies the state of a specified attribute on the specified session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
attrName	Input	Resource attribute for which the state query is made.
attrValue	Input	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_ATTR_STATE	Although the specified attribute state is valie, it is not supported by this implementation.
VI_ERROR_ATTR_READONLY	The specified attribute is read-only.
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced session, event, or find list.

6.34 viSPrintf

Syntax

```
int viSPrintf(viSession vi,
        viBuf buf,
        viString writeFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to format write to a user-specified buffer using a variable number of arguments.

Description

This operation is similar to viPrintf (), except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Output	resulting output string
writeFmt	Input	String describing the format for arguments to be output.
arg1,arg2,...	Input	List of parameters used by the format string.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_FMT	A format specifier in the string is invalid.

6.33 viSetAttribute

Syntax

```
int viSetAttribute (ViSession vi,
                    ViAttr attrName,
                    ViAttrState attrValue);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to set the state of a resource attribute.

Description

This routine is part of the VISA Library (VISA) and its purpose is to set the state of a resource attribute.

This function modifies the state of a specified attribute on the specified session.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
attrName	Input	Resource attribute for which the state query is made.
attrValue	Input	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_ATTR_STATE	Although the specified attribute state is valie, it is not supported by this implementation.
VI_ERROR_ATTR_READONLY	The specified attribute is read-only.
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced session, event, or find list.

6.34 viSPrintf

Syntax

int viSPrintf(viSession vi,
viBuf buf,
viString writeFmt,
[arg1,arg2,...]);

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to format write to a user-specified buffer using a variable number of arguments.

Description

This operation is similar to viPrintf (), except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Output	resulting output string
writeFmt	Input	String describing the format for arguments to be output.
arg1,arg2,...	Input	List of parameters used by the format string.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_FMT	A format specifier in the string is invalid.

6.35 viSScanf

Syntax

```
int viSScanf (viSession vi,
        viBuf buf,
        viString readFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to format read from a user-specified buffer using a variable number of arguments.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read from a user-specified buffer rather than a device..

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Output	input string to be parsed
readFmt	Input	String describing the format for arguments to be input.
arg1,arg2,...	Input	List of parameters used by the format string.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV__FMT	A format specifier in the string in invalid.

6.36 viStatusDesc

Syntax

```
int viStatusDesc (viSession vi,
                  viStatus status,
                  viPString desc);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to return a user readable string that describes the passed status code.

Description

This routine is part of the VISA Library (VISA) and its purpose is to return a user readable string that describes the passed status code.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
status	Input	Status code to interpret.
desc	Input	The text interpretation of the status code passed to this function.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_WARN_UNKOWN_STATUS	Unknown status.

6.37 viUnmapAddress

Syntax
int viUnmapAddress (viSession vi);

Purpose
This routine is part of the VISA Library (VISA) and its purpose is to UNmap memory space.

Description
This routine is part of the VISA Library (VISA) and its purpose is to UNmap memory space.

This function unmaps the memory previously mapped by viMapAddress().

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_WINDOW_NMAPPED	There is no window mapped to this session.

6.38 viVPrintf

Syntax

```
int viVPrintf (viSession vi,
        viString writeFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to convert, format and send params to the device as specified by the format string.

Description

This routine is part of the VISA Library (VISA) and its purpose is to convert, format and send params to the device as specified by the format string.

This function is identical to viPrintf, except that the viVAList parameters list provides the parameters rather than separate arg parameters. For a complete description of the use of this function, please see the section regarding viPrintf.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
writeFmt	Input	String describing the format for arguments.
arg1,arg2,...	Input	A list containing the variable number of parameters on which the format string is applied The formatted data is written to the specified device.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_INV_OBJECT	(both are the same value).
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_INV_PROT	The protocol specified is invalid.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_LINE_IN_USE	The specified trigger line is currently in use.
VI_ERROR_NCIC	The interface associated with the given vi is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).

6.39 viVQueryf

Syntax

```
int viVQueryf (viSession vi,
        viString writeFmt,
        viString readFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to perform a formatted write and read through a single operation invocation.

Description

This operation is similar to viQueryf(), except that the viVAList parameters list provides the parameters rather than the separate arg parameter list.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
writeFmt	Input	String describing the format for arguments.
readFmt	Input	String describing the format for arguments to be output.
arg1,arg2,...	Input	List of parameters used by the format string.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

6.40 viVScanf

Syntax

```
int viVScanf (viSession vi,
        viString readFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to read, convert and format data using the format specifier.

Description

This routine is part of the VISA Library (VISA) and its purpose is to read, convert and format data using the format specifier. Store the formatted data in params.

This function is similar to viScanf, except that the viVAList parameters list provides parameters rather than separate arg parameters. For more detailed information on viScanf.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
readFmt	Input	String describing the format for arguments.
arg1,arg2,...	Input	A list with the variable # of parameters into which the data is read and the format string is applied.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

6.41 viVSPrintf

Syntax

```
int viVSPrintf (viSession vi,
        viPBuf buf,
        viString writeFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to format write to the device using a variable

Description

This operation is similar to viVPrintf(), except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Output	Output string
writeFmt	Input	String describing the format for arguments.
arg1,arg2,...	Input	A list containing the variable number of parameters on which the format string is applied The formatted data is written to the specified device.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code

6.42 viVSScanf

Syntax

```
int viVSScanf (viSession vi,
        viPBuf buf,
        viString readFmt,
        [arg1,arg2,...]);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to format read from a user-specified buffer using a variable argument list arguments.

Description

This operation is similar to viVScanf (), except that the data is read from a user-specified buffer rather than a device.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Input	Input string to parse
readFmt	Input	String describing the format for arguments.
arg1,arg2,...	Input	A list containing the variable number of parameters on which the format string is applied The formatted data is written to the specified device.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code

6.43 viVxiCommandQuery

Syntax

```
int viVxiCommandQuery (viSession vi,
                       viUInt16 mode,
                       viUInt32 cmd,
                       viUInt32 response);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to send the device a miscellaneous command or query and/or retrieve the response to a previous query.

Description

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
mode	Input	
cmd	Input	
response	Input	

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.
VI_ERROR_IO	Unknown error code.

6.44 viWrite

Syntax

```
int viWrite (viSession vi,
        viBuf buf,
        viUInt32 cnt,
        viPUInt32 retcnt);
```

Purpose

This routine is part of the VISA Library (VISA) and its purpose is to write data to a device synchronously.

Description

This routine is part of the VISA Library (VISA) and its purpose is to write data to a device synchronously.

The write operation synchronously transfers data. The data to be written is in the buffer represented by buf. This operation returns only when the transfer terminates. Only one synchronous write operation can occur at any one time.

Parameters

Parameter Name	Direction	Description
vi	Input	Unique logical identifier to a session.
buf	Input	Represents the location of a data block to be sent to a device (constant).
cnt	Input	Number of bytes to be written.
retcnt	Output	Represents the location of an integer that will be set to the number of bytes actually transferred.

Return Values

VI_SUCCESS	The specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session or object reference is invalid
VI_ERROR_NSUP_OPER	The given vi does not support this operation.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_IO	Unknown error code.

7 KSC List Generation Interface Library

7.1 Library Usage

The List Generation Library is implemented as a set of linkable routines in the KSCAPI library. The list building routines are prototyped in the "kscapi.h" file. Additionally, a set up "C" macros is also available to create inline lists.

The list generation routines are provided to help in the creation of lists using a more structured convention. Creating a list involves first allocating memory to store the list and then calling KSC_init_list. This routine will return back a pointer to a structure of type ksc_list that will be used by all of the other list generating routines. If the user is building multiple lists, the user must provide storage for each of the lists and call KSC_init_list for each list. The user may build multiple lists concurrently as all information about the current state of each list is maintained by the structure allocated by KSC_init_list. The list must be allocated on a long word boundary.

The user calls the individual functions to "compile" the instruction list into the user provided list memory. Each callable function in the library is usually associated with one particular command instruction. There exist functions that implement standard IF...ELSE...ENDIF and SWITCH...CASE...ENDCASE properties found in most high-level languages. The list-generating library keeps track of calculating offsets and inserting the proper commands into the list, making such IF and CASE blocks much easier to develop.

Upon completion of making a list, KSC_finish should be called to clean up the list and check for any possible errors in the list. The routine KSC_dump_list can be called to display the compiled list to standard output.

A sample program that creates a list follows. This list does not perform any real functionality, and is provided merely as an example for list creation. Do not attempt to actually execute the list!

```
/*
        TEST PROGRAM
        This program will demonstrates the use of the List
Generation functions
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "../include/ksc_genlist.h"

main()
{
        /* Variable defs */

        short   *mem;           /* Our memory buffer */
        struct ksc_list *list;  /* Our list definition structure
*/
        int             size;   /* Our value of how big list is
*/
```

```
        /*
        * Begin here
        */

        mem = malloc(1024);             /* Allocate a 1024 byte
buffer */
        KSC_init_list(mem,1024,&list);

        /*
        * List code begins here
        */

        KSC_bdcast_trigger(list);
        KSC_block_rw(list,ABORT,WS8,DECADR,15,33,READ,INTERNAL,
                                0x7F7F7F,0x252525);

        KSC_if(list,EQ,0xFFFFFF,0x353535);
                KSC_execute_msg_dev(list,0x75,0,1,20,50,"A simple
text block");
                KSC_gen_demand(list,200);
        KSC_endif(list);


KSC_inline_rw(list,ABORT,WS8,DECADR,15,33,WRITE,INTERNAL,0x13300)
;

KSC_inline_w(list,ABORT,WS8,DECADR,15,33,INTERNAL,0x22222,0x53535
3);
        KSC_if(list,EQ,0xFFFF,0x616161);
                KSC_load_test_val(list, 15,WS16,0x7A7A7A);
                KSC_mark_list(list);
        KSC_else(list);
                KSC_slave_trigger(list, 33, 1,1,0,1,0);
                KSC_if(list,EQ,0xFFFF,0x616161);
                        KSC_load_test_val(list,
15,WS16,0x7A7A7A);
                        KSC_mark_list(list);
                KSC_else(list);
                        KSC_slave_trigger(list, 33, 1,1,0,1,0);
                        KSC_store_flag(list,0x5050);
                KSC_endif(list);
                KSC_store_flag(list,0x5050);
        KSC_endif(list);

        KSC_time_stamp(list);

        KSC_switch(list,0xFAFAFA);
                KSC_case(list,0x101010);
                KSC_load_test_val(list, 15,WS16,0x7A7A7A);
                KSC_mark_list(list);
                KSC_if(list,EQ,0xFFFFFF,0x353535);
```

```
KSC_execute_msg_dev(list,0x75,0,1,20,50,"A simple text block");
                        KSC_gen_demand(list,200);
                KSC_endif(list);

                KSC_case(list,0x202020);
                KSC_load_test_val(list, 15,WS16,0x717171);
                KSC_mark_list(list);

                KSC_case(list,0x303030);
                KSC_load_test_val(list, 15,WS16,0x2b2b2b);
                KSC_mark_list(list);
        KSC_endcase(list);

        KSC_end_list(list);

        /*
        * List code ends here
        */

        KSC_finish(list);
        /*
        * Write the list out in a symbolic fashion (see following
output)
        */
        KSC_dump_list(mem,0,1);  /* Display the built list */
}
```

This code creates the following output list:

```
LOC  DATA CODE
---- ---- ------------------------------------------------------
------------
0000 8041 BRDCST_TRIG
     0000
     0000
     0000
0008 47AE BLK_RW ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:1 int:1
                 addr:007F7F7F tr_cnt:00252525
     C021
     7F7F
     007F
     2525
     0025
0014 8084 IF cond:0 mask:00FFFFFF test:00353535
     0000
     FFFF
     00FF
     3535
```

```
     0035
     002A
0022 8090   EXEC_MSG_DEV addr:75 term:0 rply:1 time_out:0014
rply_lng:32
                        cmd_lng:14 [A simple text block]
     8075
     0014
     1432
     2041
     6973
     706D
     656C
     7420
     7865
     2074
     6C62
     636F
     006B
003E 8091   RESUME_MSG_DEV
     0000
0042 8102   GEN_DEMAND pattern:C8
     00C8
0046 8083   END_OF_SUBLIST
     0000
          END_IF
004A 478E INL_RW ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:0 int:1
               addr:00013300
     8021
     3300
     0001
0052 47CE INLN_W ab:0 ws:3 am:1 chas_adr:0F adr_mod:21 rw:0 int:1
               addr:00022222 data:00535353
     8021
     2222
     0002
     5353
     0053
005E 8085 IF(ELSE) cond:1 mask:0000FFFF test:00616161
     0001
     FFFF
     0000
     6161
     0061
     0014
006C 8082   LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
     7A7A
     007A
0074 8080   MRK_LST_ADR
     0000
0078 8083   END_OF_SUBLIST
```

```
     0000
007C 0042 ELSE
007E 8040   ADDR_SLV_TRIG chas_adr:21 TTL: 1 ECL:1 FP:0 list:1
timst:0
     0021
     1101
     0000
0086 8085   IF(ELSE) cond:1 mask:0000FFFF test:00616161
     0001
     FFFF
     0000
     6161
     0061
     0014
0094 8082     LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
     7A7A
     007A
009C 8080     MRK_LST_ADR
     0000
00A0 8083     END_OF_SUBLIST
     0000
00A4 0012   ELSE
00A6 8040     ADDR_SLV_TRIG chas_adr:21 TTL: 1 ECL:1 FP:0 list:1
timst:0
     0021
     1101
     0000
00AE 80F8     STO_FLG flag:5050
     5050
00B2 8083     END_OF_SUBLIST
     0000
            END_IF
00B6 80F8   STO_FLG flag:5050
     5050
00BA 8083   END_OF_SUBLIST
     0000
          END_IF
00BE 8002 READ_TIME_STAMP
     0000
```

```
00C2 8086 SWITCH mask:00FAFAFA
     007E
     FAFA
     00FA
00CA 1010   CASE test_val:00101010
     0010
     004A
00D0 8082   LD_TEST_VAL add_mod:0F ws:2 addr:007A7A7A
     800F
     7A7A
```

```
     007A
00D8 8080    MRK_LST_ADR
     0000
00DC 8084    IF cond:0 mask:00FFFFFF test:00353535
     0000
     FFFF
     00FF
     3535
     0035
     002A
00EA 8090      EXEC_MSG_DEV addr:75 term:0 rply:1 time_out:0014
rply_lng:32
                        cmd_lng:14 [A simple text block]
     8075
     0014
     1432
     2041
     6973
     706D
     656C
     7420
     7865
     2074
     6C62
     636F
     006B
0106 8091      RESUME_MSG_DEV
     0000
010A 8102      GEN_DEMAND pattern:C8
     00C8
010E 8083      END_OF_SUBLIST
     0000
             END_IF
0112 8083    END_OF_SUBLIST
     0000
0116 2020    CASE test_val:00202020
     0020
     0014
011C 8082    LD_TEST_VAL add_mod:0F ws:2 addr:00717171
     800F
     7171
     0071
0124 8080    MRK_LST_ADR
     0000
0128 8083    END_OF_SUBLIST
     0000
012C 3030    CASE test_val:00303030
     0030
     0000
0132 8082    LD_TEST_VAL add_mod:0F ws:2 addr:002B2B2B
     800F
```

```
     2B2B
     002B
013A 8080    MRK_LST_ADR
     0000
013E 8083    END_OF_SUBLIST
     0000
          END_CASE
0142 8081 END_OF_LIST
     0000
0146 8000 HALT
     0000
```

7.2 KSC_bdcast_trigge r

Syntax
int KSC_bdcast_trigger (struct ksc_list *list_base);

Purpose
This adds a Broadcast Trigger instruction to the passed list.

Description
This routine adds the Broadcast Trigger instruction to the end of the list defined by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.3 KSC_block_rw

Syntax

```
int KSC_block_rw (struct ksc_list *list_base,
                  int abort,
                  int word_sz,
                  int acc_mod,
                  int ch_addr,
                  int addr_mod,
                  int rw,
                  int it_cmd,
                  int address,
                  int trans_count);
```

Purpose

This routine adds a Block Read/Write instruction to the passed list.

Description

This routine will insert a Block Read/Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.
trans_count	input	32-bit transfer count.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.4 KSC_dump_list

Syntax

```
int KSC_dump_list (mem,
                   int size,
                   int dump);
```

Purpose

This routine displays an already built list in a readable format.

Description

This routine will display an already built list stored in memory. The list should end with a HALT instruction.

The display will give for each instruction its location (as a byte offset), instruction code, and the actual instruction and data. If the data value is set to a non-zero value, you will also receive each additional word of data for the instruction.

IF and CASE blocks will be indented accordingly. Currently, no nesting of CASE blocks is supported, and up to 10 nested IF blocks are supported.

If the routine encounters an invalid opcode, it will be displayed and the routine will continue, attempting to parse the next word as an opcode.

Parameters

Parameter Name	Direction	Description
mem	Input	This should be a pointer to the start of the list to be displayed.
size	input	This is set to the maximum size of the buffer, in bytes. The routine will display all bytes up to and including mem+size bytes. If size is specified as zero, the routine will display all instructions up to a HALT instruction.
dump	input	This is a flag used to set the display format for the instruction list. If set to a value of zero, the display will only show the beginning of each command. If set to any other value, the display will also include all additional words of data for each command.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.

7.5 KSC_end_list

Syntax
int KSC_end_list(struct ksc_list *list_base);

Purpose
This will add an EOL (End of List) instruction to the list.

Description
This routine will insert an End of List (EOL) instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.6 KSC_finish

Syntax

int KSC_finish(struct ksc_list *list_base);

Purpose
End the creation of a list and free allocated list building resources.

Description
This routine should be called at the completion of creating a list. It will check to insure that all IF and CASE blocks are properly completed.

A halt instruction is automatically added to the end of the list and the list_base memory is then released back to the system. You cannot use the list_base value after calling this routine.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.7 KSC_gen_demand

Syntax

```
int KSC_gen_demand(struct ksc_list *list_base,
                   int pattern);
```

Purpose

This places a Generate Demand instruction into the list.

Description

This routine will insert a Generate Demand instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
pattern	Input	Demand pattern value (0-255).

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.8 KSC_init_list

Syntax

```
int KSC_init_list (*mem_base,
        int size,
        struct ksc_list *list_base);
```

Purpose

Prepare allocated memory for list generation.

Description

This routine must be called before using any of the other list generating routines. It will allocate a structure of type ksc_list, initialize it, and then return its location back in list_base. You will need this value for calls to any of the other list generating functions.

You may work on more than one list at a time. Each list will have its own list_base value.

At the end of creating a list, you must call KSC_finish to cleanup the list and remove the allocated structure from memory.

Parameters

Parameter Name	Direction	Description
mem_base	Input	This is a pointer to the start of memory where you want the list to be built. The memory must already be allocated.
size	Input	This is the size, in bytes, of the allocated memory starting at mem_base.
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_BAD_ARD	Bad arguments passed.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.9 KSC_inline_rw

Syntax

```
int KSC_inline_rw (struct ksc_list *list_base,
        int abort,
        int word_sz,
        int acc_mod,
        int ch_addr,
        int addr_mod,
        int rw,
        int it_cmd,
        int address);
```

Purpose

This places an Inline Read/Write VXI/VME instruction into the list.

Description

This routine will insert an Inline Read/Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.10 KSC_inline_w

Syntax

```
int KSC_inline_w (struct ksc_list *list_base,
                  int abort,
                  int word_sz,
                  int acc_mod,
                  int ch_addr,
                  int addr_mod,
                  int rw,
                  int it_cmd,
                  int address,
                  int data);
```

Purpose

This places an Inline Write VXI/VME instruction into the list.

Description

This routine will insert an Inline Write VXI/VME instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
abort	input	Abort Disable flag. Set this to one of ABORT (regular abort) or ABORT_D (disable the abort).
word_sz	input	Word size. Set this to one of WS8, WS16, or WS32.
acc_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
ch_addr	input	Chassis address. Set this to a valid chassis address number (0-127).
addr_mod	input	Access mode. Set this to one of INCADR, DECADR, or RETADR.
rw	input	Read/Write mode. Set this to the type of operation to be performed, READ for a read, or WRITE for a write.
it_cmd	input	VXI bus command or slot-0 command. Set this to one of INTERNAL for a slot-0 command or EXTERNAL for a bus command.
address	input	A 32 bit VXI address.
data	input	The actual data to be written.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

7.11 KSC_slave_trigger

Syntax

```
int KSC_slave_trigger (struct ksc_list *list_base,
                       int ch_addr,
                       int ttl_trig,
                       int ecl_trig,
                       int fp_trig,
                       int list,
                       int timst);
```

Purpose

This places an Addressed Slave Trigger instruction into the list.

Description

This routine will insert an Addressed Slave Trigger instruction at the end of the passed list given by list_base.

Parameters

Parameter Name	Direction	Description
list_base	Input	Used by all of the List Generation routines. It is created by calling KSC_init_list.
ch_addr	input	Chassis address. Valid values are 0-127.
ttl_trig	input	Generate VXI TTL trigger line.
ecl_trig	input	Generate VXI ECL trigger line.
fp_trig	input	Generate V160 front panel trigger.
list	input	Trigger list execution.
timst	input	Clear time stamp counter.

Return Values

The most common error codes are listed here. For a comprehensive list, please refer to the Error Codes section of this manual.

KSC_SUCCESS	Normal, successful return.
KSC_NOLISTMEM	Not enough list memory for this instruction.

8 Demands

8.1 The Demand Process

The Demand Process is a high priority process that acts as a server to application processes for handling demands from the KSC2962 NT device driver. Application processes send registration requests to the Demand Process for all demands received from the CAMAC highway they wish to service. Demands are enabled by the Demand Process for each CAMAC or VXI chassis if demands are not currently enabled on the chassis. When the device driver receives demands from the CAMAC highway, the Demand Process immediately dispatches the demand to the registered process. Demands that are received for which there are no processes registered are ignored by the Demand Process (only a statistic is kept).

8.2 Demand Configuration File

On startup, the Demand Process creates a temporary group global section called "DMDREGION". It then populates this region with demand entries read from a configuration file pointed located in the same directory as the Demand Process (DMDPROC.EXE). If the Demand Process receives an error that the region already exists, it knows that another Demand Process is currently servicing demands. The Demand Process exits under these conditions.

The maintenance of this file is through a normal text editor. The information contained in the configuration file is:

Chassis Number	Demand ID	Chassis Type	Demand Queue Length	Description
1 to 63	0 to 255	1 or 2	10	English comment

The syntax of the demand configuration file is:

chassis, id, type, qlength, desc

Where:

chassis	Decimal Chassis number on the Grand Interconnect highway
id	Demand Id generated by the Chassis. See the V160 and 3972 slot zero controllers for description.
type	VXI (1) or CAMAC (2)
qlength	Queue length
desc	User description displayed by dmdsts utility

The following is an example configuration file. Any line beginning with an exclamation mark is ignored.

```
! Sample configuration file. This file is input to the demand process.
! Exclamation points at the beginning of a line denote comment lines.
! Commas are used to separate the columns of information. The columns are
! defined below. Commas used in the description field will simply
! truncate the description at the position of the comma. The use of spaces
! before and after columns will be considered valid input.
!Chassis
```

!	Demand Id	Type	Queue Length	Description
1,	1,	2,	11,	Crate 1 / Demand 1
1,	2,	2,	12,	Crate 1 / Demand 2
1,	3,	2,	13,	Crate 1 / Demand 3
1,	4,	2,	14,	Crate 1 / Demand 4
2,	4,	2,	15,	Crate 2 / Demand 4
2,	5,	2,	16,	Crate 2 / Demand 5
2,	6,	2,	17,	Crate 2 / Demand 6

8.2.1 Application Registration for Demands

The Demand Process establishes a single system-wide pipe "\\.\pipe\dmdproc". The Demand Process reads registration requests from user processes (see KSC_ENABLE_EVENT). The user receives the status of the Demand notification via a unique pipe created by the user process for the particular demand. The demand must be defined within the demand configuration file prior to the startup of the demand process. Adding new demands requires the restart of the Demand Process and the stopping of all processes currently registered for demands.

8.2.2 Demand Processing

When a demand is received by the Demand Process, the Chassis number is used to traverse the demand entries associated with it. This should reduce the search time for the matching Demand ID. Any demands that are received and are not in the table, will be logged to the Demand Process's log file, and the unknown-demand counter incremented. If the Application process that should receive the demand is no longer present, or if its pipe is closed, the demand event will be logged and the not registered counter incremented. Otherwise, the Demand Process sends the following information to the registered application:

Function = DEMAND_MSG
Chassis Number of Demand
Demand ID in Chassis
User Index
Time of Demand

If this demand was a one-shot, the demand entry is cleared. When there are no longer any Application processes registered for a Chassis, the Demand Process will disable demand recognition for that Chassis.

It is possible that the process may be still active but the image that requested the demand registration may have been run down. The Demand Process will consider a pipe write error to be the same as a process no longer being available.

Each time a demand is processed, the Demand Process also increments statistic counters and stores the time stamp of the event in the group global region. (The utility DMDSTS can display this information.) If at any time the Demand Process gets a failure writing to a pipe it will disable the demand .

The number of demand messages in the demand FIFO, the frequency at which they arrive, and activity caused by other processes on the highway at the time will influence the speed at which a demand is delivered to an application process. For CAMAC crates, the Demand Process must read the LAM status register within the Crate to determine which of the slots within the Crate are asserting their LAM lines. This required read competes with all other requests to the device driver and will effect the response of the demand delivery to the registered process.

The Demand Process can not determine if multiple LAMs have been presented within a chassis. It is the user's responsibility to determine a redundant demand notification that can be created under the following circumstances:

1. Slot 1 in Crate 1 asserts LAM
2. Demand is sent to 2962
3. Demand process determines that a LAM is present in Crate 1 and read the LAM status register
4. The Demand is sent to the registered user
5. Slot 2 in Crate 1 asserts LAM
6. Demand process determines that a LAM is present in Crate 1 and reads the LAM status register that shows two slots asserting a LAM.
7. The Demand Process sends a redundant demand for slot 1 and the demand for slot 2
8. The requesting process for Slot 1runs and tries to clears the LAM in slot 1
9. The requesting process for Slot 1 receives the redundant Demand
10. The requesting process for Slot 2 receives the LAM

8.3 User Application Program

There may be more than one Application program that receives demands, but a single Demand ID in a Chassis can be registered to only one Application.

All Application programs must contain the following elements (see program \KCA001\EXAMPLES\TEST_DMD.C):

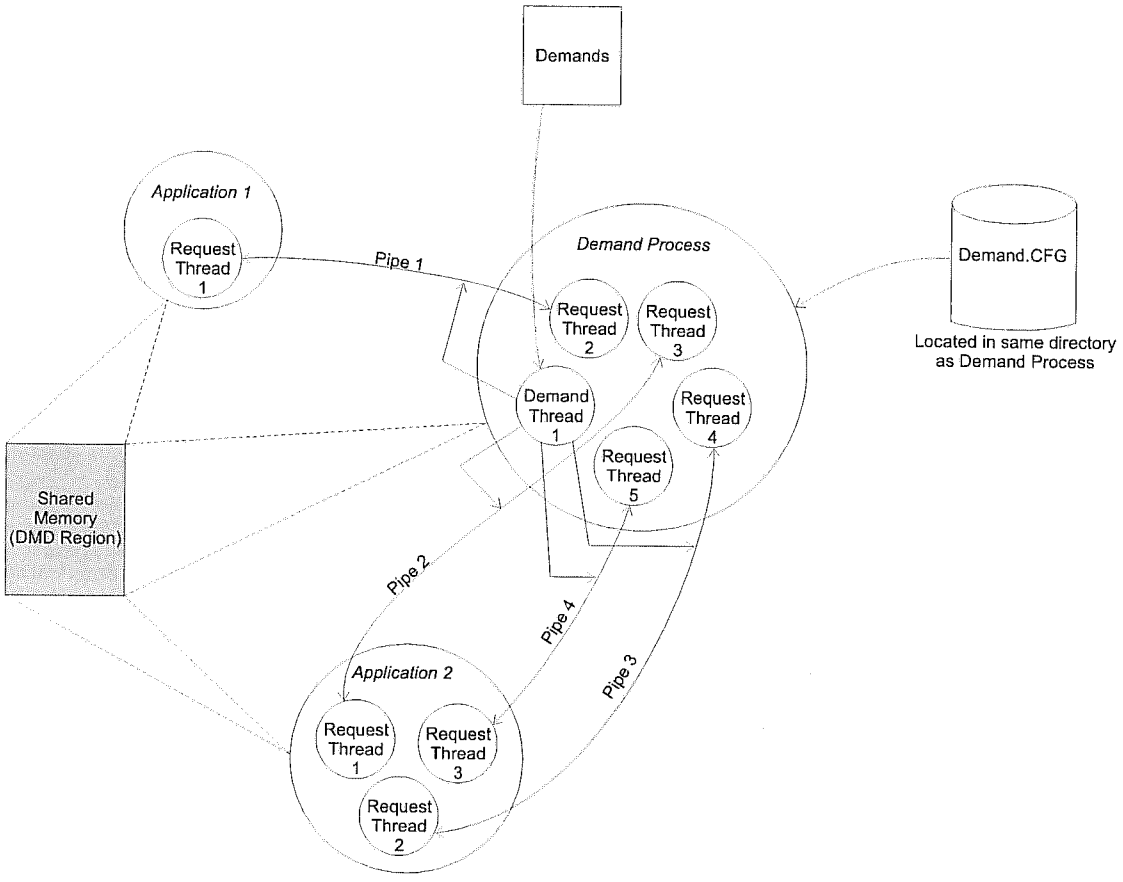
- Call KSC_init to create the structure KSC_handle required by all other KSC and CAM module
- Call KSC_enable_demand for each demand to be received. The application maps the demand region to ensure the Demand Process is running, and another process is not currently capturing the demand. This module creates a pipe then sends a registration request to the Demand Process using the Demand Process's registration pipe.. The registration reply is received in the APC routine, which it also sets up. Demands received are dispatched to a user-written APC routine that should appropriately process each demand received. Finally, this module reposts another read on the pipe for the subsequent demand.
- The developer must create a read APC routine to examine each demand received and take appropriate action.

The following diagram shows an overview of the Demand Process, a process registering for demands via the VISA library, and a process registering for LAMS (Demands) using the CAMAC library.

8.4 Demand Process Dataflow

The design of the Demand Message Process is a combination of the Windows NT, NT and UNIX device drivers for the KSC2962 and KSC2962. This design allows for both CAMAC and VISA (only available on 2962) demands to be supported.

The NT version of the Demand Process utilizes named pipes for its method of communication between application programs and itself. A named pipe is a one-way or two-way pipe for communicating between a server process and one or more client processes. Named pipes allow for multiple instances of a single pipe, however an instance of the pipe may only be opened by one client at a time. Due to the fact that an instance of a pipe may only be opened once, multiple threads are often used to create multiple instances of the pipe to communicate with multiple clients. The following drawing is an overview of demand request flow.



The demand process creates 1 named pipe with the name \\.\PIPE\DMDPROC. The multiple pipes shown in the drawing are multiple instances of the pipe \\.\PIPE\PMDPROC.

The arced lines above each represent an instance of the named pipe \\.\PIPE\DMDPROC. These are two - way pipes. The application program will send to the Demand Process a request for a particular demand,

and will receive from the Demand Process demand information. For every demand requested by an application program a thread and pipe instance will be created by both the application program and Demand Process. There will be one thread and pipe instance per demand request. In addition the Demand Process creates one additional thread (Thread 1 in the drawing above) to read demands from the KSC2962 or KSC2962. Thread 1 will be able to write to Pipes 1, 2, 3, 4 since all it needs is the pipe handle which it will be able to obtain from the shared memory region. The Demand Process is a Windows program with one window to display the error and status messages

8.5 Demand Utilities

8.5.1 Program DMDSTS

The DMDSTS program is a diagnostic that maps the Demand Process's demand global section. It allows a user to display the demand registration, pipes, and demand statistics. DMDSTS has read-only access to the Demand Process's global section.

The program presents information in one of two mutually exclusive modes:

- Continuous mode - Displays and updates every 5 seconds information on currently active demands. Output is only to the CRT screen in 132-column mode.

- Dump mode - Displays various amounts of information to the CRT screen or (optionally) to a file. The amount of information displayed depends on which command line switch is used.

Continuous Mode

Usage:	$ DMDSTS /CONT	Switch explicitly specified
	$ DMDSTS	Invokes/CONT, by default

The screen/window is put into 132 column mode. For each enabled demand found in the system, the following are displayed and updated every 5 seconds.

ACTIVE DEMANDS

chassis	type id	last demand	count	mbx	letters	pid	proc name	image name
1	CAMAC 10	20-FEB-1996 10:03:52.13	12	_MBA275:	0	89	DMD_PROC_1	DKA0:[KGI]DMD_1.EXE;52
1	CAMAC 11	20-FEB-1996 10:07:43.19	432	_MBA275:	0	89	DMD_PROC_1	DKA0:[KGI]DMD_1.EXE;52
2	CAMAC 20	20-FEB-1996 10:05:07.00	53	_MBA318:	4	92	DMD_PROC_2	DKA0:[KGI]DMD_2.EXE;21

To exit this screen, push RETURN. The screen/window should return to the original size (80 or 132 columns).

If the /OUT=file switch is present on the command line with the /CONT switch, it is ignored. Output is only to screen/window.

Dump Mode

Usage:	$ DMDSTS /CHASSIS=x	Region header plus chassis "x", enabled or not
	$ DMDSTS /ENABLED	Region header plus all enabled demands
	$ DMDSTS /CONFIG	Region header plus all configured demands

$ DMDSTS /ALL	Region header plus all demands, even those not enabled or configured.
$ DMDSTS /OUT=file	Output goes to "file", not screen

The first four switches are mutually exclusive, and, if more than one is present on the command line, the one with highest precedence is used.

Switch Precedence:

/ALL	Highest, overrides all below
/CONFIG	Overrides all below
/ENABLED	Overrides all below
/CHASSIS=x	Lowest, overrides no other switches

For each switch, the information displayed is:

Switch	Region Header	Chassis Table	Demand Entry
/ALL	X	All, configured or not	All that are config
/CONFIG	X	Only those configured	All for config
/ENABLED	X	Only those enabled	All for enabled
/CHASSIS=x	X	Only chassis "x", even if not config or not enabled	All for the chassis

The /OUT switch can be used with any of the above four switches to direct output to the indicated "file" instead of to the screen/window.

9 NT KCDRIVER

This chapter describes the implementation of the KSC 2115 PCI adapter under Windows NT version 4.0. The reader should reference the KSC hardware documentation for specifics about this device. Of the functionality provided by the 2115, the following are supported:

- Support for DMA of large memory buffers. The maximum transfer size is 1M bytes (however, depending on configuration, the number of mapping pages given to the device may limit the actual transfer size).
- Storage of command lists within the 2115
- Support for Demands/LAMS
- Support for segmented multibuffers
- Support for the Clock on the 2115 to trigger execution of command lists (Second Release)

The 2115 is a high performance device that differs from other previously manufactured CAMAC adapters from KSC. These differences allow for higher throughput, but provide less knowledge about No-X and No-Q on a per CAMAC instruction execution. Either the list can stop or ignore these conditions with the total status being the or of all of the CAMAC commands contained within the list.

9.1 Driver Interface

The NT driver is called using the following NT native system calls:

- **DeviceIoControl**
- **ReadFile**
- **WriteFile**

The DeviceIoControl service uses IOCTRL codes reserved for user and customer devices. These are defined in the header file provide with the provided kit. The KSC 2115 does not fit well into the NT file and I/O subsystem. It is not a real file structured device. To utilize its functionality and to acquire the maximum benefit from the device, a mapping of functionality was done.

9.2 NT devices

There are twenty (0 through 19) NT devices created when the NT driver is loaded. The purpose of sixteen of the devices is to map the KSC 2115 command list memory into eight partitions. This allows the user to store a command list in any of the partitions and then execute it. The purpose and mapping of each of the devices is as follows:

- KCA00- Used for control functions in particular the setting and reading partitions and status information. It is also used for small buffers that utilized programmed buffered I/O.
- KCA01- Used for reading Demands

- KCA02- Used for getting Buffer Completion flags
- KCA03- For loading command list into partition one
- KCA04- Executing command lists, and normal transfers using command list in partition one
- KCA05- For loading command list into partition two
- KCA06- Executing command lists, and normal transfers using command list in partition two
- KCA07- For loading command list into partition three
- KCA08- Executing command lists, and normal transfers using command list in partition three
- KCA09- For loading command list into partition four
- KCA10- Executing command lists, and normal transfers using command list in partition four
- KCA11- For loading command list into partition five
- KCA12- Executing command lists, and normal transfers using command list in partition five
- KCA13- For loading command list into partition six
- KCA14- Executing command lists, and normal transfers using command list in partition six
- KCA15- For loading command list into partition seven
- KCA16- Executing command lists, and normal transfers using command list in partition seven
- KCA17- For loading command list into partition eight
- KCA18- Executing command lists, and normal transfers using command list in partition eight
- KCA19- Load and Execute a Load and GO command list

9.3 DeviceIoControl functions

The file KSCIOCTL.H contains the IOCTL codes that may be passed to the NT driver. The functionality and buffer contents are defined below. These particular IOCTL codes are only valid for the KCA0, KCA1, and KCA 2 devices. The DeviceIoControl function takes the input buffer and moves it to a system non-paged pool buffer and calls the driver. The driver does the operation and if it is to return data, places the data into another non-paged pool buffer that is then copied to the user buffer.

This buffer copying is not desirable for large high speed transfers but is actually faster for small data transfers as it takes less time to set up the DMA and lock down user buffers than to use those already locked down in the system space. The driver uses this method for programmed I/O of the KSC 2115. In particular the API uses this for single small transfers.

Device KCA00- Control device

- KSC_PARTITION- Set the partition table
- KSC_TIMEOUT- Set the time out for a partition
- KSC_TIMERSET- Sets the timer for a clocked command list
- KSC_2115RESET- Reset the device
- KSC_ID- Return the current release of the driver
- KSC_COUNTERS- Return counters for the driver
- KSC_RDPARTABLE- Read the current partition table
- KSC_ERRREG- Read the last status and error information for a partition

Device KCA01- Demand device

- KSC_DMDREAD- Read any demands currently in the device adapter.

Device KCA02- Buffer Complete device

- KSC_BUFCOMPLETE- Read any buffer completion flags from the driver. The user must have a repeating buffer function active on KCA03.

9.3.1 ReadFile and WriteFile Operations

To transfer large amounts of data, the remaining KCA(03 to 19) devices should be used. These devices are accessed using the NT ReadFile and WriteFile system service calls. The KCA19 device is unique in that it assumes that the user has built a combined buffer that contains the command list and the actual buffer (see earlier description of the LoadGo buffer). Access to these devices results in the user buffer being locked into memory and DMA being transferred to and from the KSC 2115 directly. This method provides the maximum through put for the device. Additionally, it is the only method that will support the segmented buffered mode.

9.3.2 Buffers

All buffers must be long word aligned (e.g. on a 32-bit boundary). Additionally, all output buffer must have space for pipeline requirements of the device (8 long words or 32 bytes).

Some of the IOCTL calls indicate a Read operation when in effect they are being used for a write operation. Since the Read operation means that the driver must be able to write to indicated address space, this of less protection than a read, therefore, the buffer should also be readable by the driver.

The LoadGo buffer is a combined buffer that contains both the command list to be loaded and the buffer. The first long word of the buffer contains the size of the command list in the lower 24 bits and the partition number in the upper 8 bits. If the user does not specify a partition, then partition one is used by the driver.

The command list follows the command list size, followed by storage for the actual buffer which is either read from for a write from the host to the 2115 or written to for a read from the 2115.

9.3.3 KSC_PARTITION- Set the partition table

The user buffer will be buffered. The IOCTL dispatch code can simply populate the partition table within the CBF. The buffer will contain an array of eight long words that specify the length of each of the partitions.

9.3.4 KSC_TIMEOUT- Set the time out for a partition

The user buffer will be buffered. The IOCTL dispatch code can simply populate the partition table for the indicated partition. The buffer will contain the timer value and the partition number both as long words.

9.3.5 KSC_TIMERSET- Set the device internal timer

This will set the value to be used when the user uses the internal clock of the device. The buffer contains the timer value to load into the device.

9.3.6 KSC_2115 RESET- Reset the device

This control code contains no data. It simply does a reset on the device. Any outstanding I/O is terminated. This dispatch code will queue the request to the driver as the controller must be acquired.

9.3.7 KSC_ID- Return the current release of the driver

The IOCTL dispatch code will populate the user's buffer with a long word containing the current release of the driver.

9.3.8 KSC_COUNTERS- Return counters for the driver

The IOCTL dispatch code will populate the user's buffer with the current counters from the CBF.

9.3.9 KSC_RDPARTABLE- Read the current partition table

The IOCTL dispatch code will populate the user's buffer with eight long words describing the current partition layout of the command list memory of the device

9.3.10 KSC_ERRREG[1-8]- Read the last status and error information for a partition

The IOCTL dispatch code will populate the user's buffer with the status information maintained for a particular partition.

9.3.11 KSC_DMDREAD- Read any demands currently in the device adapter.

The DeviceIoControl dispatch code will queue the IRP to the particular demand device. The demand device will then execute the STARTIO entry of the demand device. The STARTIO entry of the demand device will then indicate that there is an user buffer for demands and store the size of the user's demand buffer. It will then enable demand interrupts on the device and wait on a semaphore. If there are demands or when demands arrive, a demand interrupt will be triggered. The DPC (Deferred Procedure Call) for the demands will be requested. It will unload as many demands as possible into the user's demand buffer and trigger the semaphore. Due to timing, it is possible that semaphore may be already signaled before the STARTIO routine begins the wait if there were demands already within the device Demand FIFO.

The 2115 only generates an interrupt when the demand FIFO goes from empty to non-empty. Therefore, if the FIFO is not empty, the Demands must be unloaded.

9.3.12 KSC_BUFCOMPLETE- Read any buffer completion flags

The IOCTL dispatch code will queue the IRP to the buffer device if there are no current buffers completed, otherwise the IOCTL dispatch code will return the flags immediately. The buffer device STARTIO entry will then wait for a semaphore to be triggered. The DPC for buffered I/O will then set the semaphore when a new buffer has been filled. Due to timing, it is possible that semaphore may be already signaled before the STARTIO routine begins the wait if a buffered operation fills a new buffer segment.

9.3.13 KSC_ACKBUFCOMPETE- Acknowledge the processing of the buffer completion

The IOCTL dispatch code will queue the IRP to the buffer device. The buffer device STARTIO will raise IPL, capture a device spinlock, and mark the particular buffers as free. It will then do the I/O completion.

9.3.14 DMA Considerations

NT will provide mapping registers to the device driver. The number of map registers available may limit the size of a DMA transfer. Normally a driver would then execute multiple DMA transfers to accommodate the complete buffer. However, due to the design of the device, this is not feasible. Therefore, the user may receive a DMA size too large status code.

For the LoadGo buffers that are sent to KCA19, the buffers are mapped in both system space and mapped for DMA. This is required since the command list is loaded into the device using programmed I/O.

9.4 Status Returns

The NT WriteFile and ReadFile system service calls provide either a TRUE or FALSE return status and a byte count regarding the read or write operation. If an error is encountered, the application thread may call GetLastError to return the status from the device driver. If the user is doing overlapped I/O and has multiple threads, it is unclear if this is sufficient to get the desired error. The driver supports additional calls to get more detailed information from the driver.

For the CAMAC library, if the list generates a fault, the CAMAC library API will attempt to request the driver for the status of the last execution. The CSR and the current memory address of the list when the list faulted can be useful to determine list faults. The 2115 typically points to the next location of the command list after the list instruction that caused the stop. Depending on the coding of the command list, the stop may be a result of a No-Q, No-X, a Halt Instruction, or a faulty list. Because there is not a good way to acquire sufficient status from the driver with each NT native call, the CAMAC APIs must do a second request from the driver to acquire the status. It is possible and likely if more than a single process is using the device that the status will be overwritten by a subsequent request before the extended status is acquired. Therefore, users who have used the Status buffer will find that the buffer will not be accurate. The overall status and the first word of the Status buffer will always be accurate as it is a reflection of the current I/O. This has impact on the CAB16, CAB24, CACTRL, CAM24, CAM16, and their variants.

Lists that generate or sink less data than expected require examination of the command list itself. The 2115 does not give a status buffer which was available on other KSC devices.

9.5 Demands and LAMS

The CAMAC crates on the CAMAC highway have the ability to generate LAMS that are translated to a demand and stored in the Demand FIFO of the 2115. The 2115 can also generate demands as a result of list execution. Because the servicing of a LAM requires that a read be done to determine what has card within the chassis is requesting the LAM, the processing of the LAMs has been migrated to a Demand Process. The user is notified that a LAM is present using a read from an NT Pipe.

The actual read of the demands as documented earlier is done using the device: kca01. The demands are queued within the 2115 until a user process does a device control function. The driver captures the spinlock for the access to the device registers. It then check to see if there are any currently pending. If not, a flag is set indicating that a demand interrupt is expected and the demand interrupt is enabled. If there are demands, then a maximum number of demands will be removed from 2115 and returned to the user. The extraction of the demand interrupts is done with interrupt lockout and therefore, a maximum of twenty-five demands will be extracted at any one time such that the system does not experience any degradation.

9.6 MultiBuffer Considerations

The multibuffer functions of the 2115 allow the user to create lists that are can be clocked by either an external or the 2115 internal clock. The advantage of the multibuffer is that the user's DMA buffer need not be locked down more than once. The notification of each segment of the user buffer is via returned to the user via the KCA02 device. As each segment is processed by the user the driver must be informed. Failure to do so, will result in a multibuffer overflow. The driver may report more than a single multibuffer segment completion per read on KCA02 device.

9.7 NT Limitations

When NT delivers an IRP packet to the device driver, the packet is not cancelable. There are two special IOCTRL codes that will cancel either a Demand Read or a Multibuffer read request. The special utility RESETDRIVER is provided to reset the driver and to cancel these types of requests. Users should add this to their process exit handling.

10 Error Codes

This appendix documents error generated by the device driver and the different interface libraries. The user may use different header or include files to provide individual values for the error codes. This document does not document their actual value as they may change between releases. The header files and symbolic references should be used to minimize this effect.

10.1 Driver and KSC API Success Codes

There are only two driver and KSC API success codes are returned either as the value of the NT system service or KSC API function call.

- KSC_SUCCESS The function has completed successfully
- KSC_IOQUEUED The driver successfully queued the request

10.2 Driver and KSC API Error Codes

The driver and KSC API error codes are returned either as the value of the NT system service errors (GetLastError), the KSC API function call, or in the status block condition field. Some of the following errors may not be returned for the 2115 as this error messages are shared between the CAMAC (2115) and Grand Interconnect highway devices (2962).

- KSC_ACCESSVIO User buffer cannot be accessed
- KSC_ALIGNMENT Buffer not longword aligned
- KSC_ALLOC Failure to allocate storage
- KSC_ANR List generated no slave on highway recognized addr
- KSC_BADARG Bad argument in list command
- KSC_BADHEADER Unable to read header or not formatted
- KSC_BADTIMEOUT Invalid time-out value specified
- KSC_BADUNIT Function not supported on this unit
- KSC_BUFACCESS Unable to read/write buffer
- KSC_BUFMODULUS Buffers must be of even size integer
- KSC_BUFTOOLARGE User buffer is too large
- KSC_BUFTOOSMALL Read/write buffer is too small
- KSC_CHASSIS Invalid Chassis number

- KSC_CLKINTERVAL Invalid clock interval
- KSC_CLOCKED Trying to do async command list while clocked
- KSC_DEVICEMAJOR Device major number mismatch
- KSC_DEVNOTOPEN Device is not open
- KSC_DEVTIMEOUT Device timed out
- KSC_DMDRSP Bad response from demand process for registration
- KSC_DMDTBLFULL Process demand table full
- KSC_DRIVERERROR Driver internal error
- KSC_DRIVER_FAULT Internal device driver error
- KSC_EMPTYPAR Partition is empty, contains no list
- KSC_HANDLE User has not called ksc_init
- KSC_ILGCMD Illegal command non-existent CAMAC/VXI
- KSC_INVLARG Invalid argument
- KSC_IOCTLERR Ioctl error
- KSC_LOOPSPIN Driver loop problem
- KSC_MBFNOTACTIVE Multi-buffer not setup
- KSC_MBUFALIGNMENT Multi-buffer not evenly divisible
- KSC_MOREDATA List needs more data
- KSC_NOENDIF Missing ENDIF in IF clause in list
- KSC_NOENDCASE Missing ENDCASE in CASE clause in list
- KSC_NOLISTMEM Not enough free list memory for instruction
- KSC_NOMEM Not enough system memory to execute function
- KSC_NOQ List generated a No-Q response
- KSC_NOQTO Q-repeat timed out
- KSC_NOREPLY Command sent, to reply after 200Ms
- KSC_NOSUPPORT Request not supported

- KSC_NOSYNC No sync on highway
- KSC_NOTALLXFER Not all of user buffer transferred
- KSC_NOTCFG Demand id or Chassis not configured
- KSC_NOTINITIALIZED Device not correctly initialized
- KSC_NOTOPEN Device not open
- KSC_NOTPARTITIONED Command partition not initialized
- KSC_NOTUSED Unsupported error condition
- KSC_NOX List generated a No-X response
- KSC_N23 List requested a CAMAC slot > 23
- KSC_NULLCOMANDLIST Command list empty
- KSC_NUMBUFFERS Invalid number of multi-buffers
- KSC_OPEN Device already open
- KSC_OSFERROR NT returned error
- KSC_OPENERROR Failure to open device
- KSC_PARERR Parity error on incoming packet
- KSC_PARTIONERR Partition number is invalid
- KSC_PARTIONSIZE Invalid partition size
- KSC_PARTNUMBER Partition number is invalid
- KSC_READERR Device read error
- KSC_REMPARERR Remote addr. slave parity error
- KSC_RESET The KSC device has been reset
- KSC_RDWTMIX Invalid mix of read/write commands in list
- KSC_UNKNOWNERR Error bit set, but error not defined
- KSC_UNSUPPORTEDFUNC Function not supported on this device
- KSC_VXITMO List generated VXI/VME time-out occurred
- KSC_WRITEERR Device write error

11 Camac Error Codes

The driver and language interface routines perform various checks on both the parameters passed by the calling program and the operation of the hardware. When an error is detected, these routines return an error code to the calling program. This appendix contains a list of error numbers and an explanation of the error.

101. The version number of the driver does not match the version number found in the Header. Check to make sure all software is at the same version number.

102. The length of the Data Buffer is greater than the specified size of the Data Buffer

103. The Header size does not match the Header size of the current version

104. The length of the CAMAC Control List is greater than the specified size of the CAMAC Control List

105. The Status Buffer size does not match the Status Buffer size of the current version

106. The process does not have either read or write access to the Data Buffer. Check that the Data Buffer has been properly declared.

107. The System does not have enough contiguous Real Time Page Table Entries to double map the Data Buffer. The number of Real Time Page Table Entries can be changed by modifying the Sysgen parameter REALTIME_SPTS.

108. The process does not have a big enough Working Set to lock down the Data Buffer. The Working Set size can be changed by modifying the Authorize parameter WSquo.

109. Unknown VMS error while trying to lock the CAMAC Control List into memory.

110. Unknown VMS error while trying to lock the Data Buffer into memory.

111. Unknown VMS error while trying to lock the Status Buffer into memory.

112. The CAMAC Control List does not have enough space at the end for the CAMAC driver to insert a number of halt instructions. The length of the CAMAC Control List must be four long words less than the size of the CAMAC Control List so four Halt instructions can be added.

113. The Data Buffer has a length of zero but must have a length of at least one. A dummy word must be entered into the Data Buffer (Header(DatLen)=1)

114. The driver does not have read access to the Header. Check that the Header has been properly declared.

115. The size of the Header is over 64K words. Check that the size of the Header has been declared as a long word (IINTEGER*4 variable)

116. The process does not have either read or write access to the CAMAC Control List. Check that the CAMAC Control List has been properly declared.

117. The System does not have enough contiguous Real Time Page Table Entries to double map the CAMAC Control List. The number of Real Time Page Table Entries can be changed by modifying the Sysgen parameter REALTIME_SPTS.

118. The process does not have a big enough Working Set to lock down the CAMAC Control List. The Working Set size can be changed by modifying the Authorize parameter WSquo.

119. The length of the CAMAC Control List is over 64K words. Check that the variable specifying the length of the CAMAC Control List has been declared as a long word (INTEGER*4 variable)

120. The CAMAC Control List does not fit in one segment. The CAMAC Control List plus the CAMAC Control List offset cannot fit within one segment (IBM PC only).

121. The size of the CAMAC Control List is over 64K words. Check that the variable specifying the size of the CAMAC Control List has been declared as a long word (INTEGER*4 variable)

122. The length of the CAMAC Control List is over 32K-1 words. The largest CAMAC Control List allowed is 32K-1 words

123. The CAMAC Control List has a size of zero but must have a size of at least one

124. The process does not have either read or write access to the QXE Buffer. Check the address and the size of the QXE Buffer in the Header.

125. The System does not have enough contiguous Real Time Page Table Entries to double map the QXE Buffer. The number of Real Time Page Table Entries can be changed by modifying the Sysgen parameter REALTIME_SPTS.

126. The process does not have a big enough Working Set to lock down the QXE Buffer. The Working Set size can be changed by modifying the Authorize parameter WSquo.

127. The QXE Buffer does not fit in one segment. The QXE Buffer plus the QXE Buffer Offset cannot fit within one segment (IBM PC only).

128. The size of the QXE Buffer is over 64K words. Check that the variable specifying the size of the QXE Buffer has been declared as a long word (INTEGER*4 variable)

129. The size of the QXE Buffer is over 32K-1 words. The largest QXE Buffer allowed is 32K-1 words

130. The process does not have either read or write access to the Status Buffer. Check that the Status Buffer has been properly declared.

131. The System does not have enough contiguous Real Time Page Table Entries to double map the Status Buffer' The number of Real Time Page Table Entries can he changed by modifying the Sysgen parameter REALTIME-SPTS.

132. The process does not have a big enough Working Set to lock down the Status Buffer. The Working Set size can be changed by modifying the Authorize

parameter WSquo.

133. The size of the Status Buffer is over 64K words. Check that the variable specifying the size of the Status Buffer has been declared a long word (INTEGER*4 variable) (advanced Fortran routines).

134. The process does not have either read or write access to the Word Count Buffer. Check the address and the of the Word Count Buffer in the Header.

135. The System does not have enough contiguous Real Time Page Table Entries to double map the Word Count Buffer. The number of Real Time Page Table Entries can be changed by modifying the Sysgen parameter REALTIME_SPTS.

136. The process does not have a big enough Working Set to lock down the Word Count Buffer. The Working Set size can be changed by modifying the Authorize parameter WSquo.

137. The WC Buffer does not fit in one segment. The WC Buffer plus the WC Buffer Offset cannot fit within one segment (IBM PC only).

138. The size of the WC Buffer is over 64K words. Check that the variable specifying the size of the WC Buffer has been declared as a long word (INTEGER*4 variable).

139. The size of the WC Buffer is over 32K-1 words. The largest WC Buffer allowed is 32K-1 words.

140. Unknown VMS error while trying to lock the Word Count Buffer into memory.

141. Unknown VMS error while trying to lock the M Buffer into memory.

201. An illegal command was found in the CAMAC Control List

202. An In-Line CAMAC read was specified. Only CAMAC write and control functions can be specified in an In-Line CAMAC Control List command.

203. An illegal LAM type was specified, the command types are zero through seven.

204. A block transfer CAMAC control function was specified. Only CAMAC read and write functions can be specified for block transfer CAMAC Control List commands

205. The remainder of the Data Buffer is too small to hold the data for the CAMAC block transfer

206. An illegal CAMAC word size for the CAMAC device was encountered

207. Block transfer timeout. The CAMAC software driver has timeout because the CAMAC hardware has not responded.

208. Block transfer timeout. The CAMAC software driver has timeout because the CAMAC hardware has not responded.

209. Bad interrupt mode

210. The QIO request was in some way canceled.

211. Out of data error. The Data Buffer was not big enough to hold or accept the data for the single naf.

212. Error in purging the data-path.

213. Single transfer timeout. The CAMAC software driver has timeout because the CAMAC hardware has not responded.

214. Single transfer timeout. The CAMAC software driver has timeout because the CAMAC hardware has not responded.

215. Error in allocating a data-path

216. Error in allocating mapping registers.

217. Error in purging the data-path.

218. Error in purging the data-path.

219. No PHYIO privileges, PHYIO privileges are needed for the operation.

220. Error in purging the data-path.

221. Power failure error.

222. The CAMAC Control List could not hold the enter LAM command.

223. The CAMAC driver could not allocate enough system memory to book the LAM request.

224. Illegal CAMAC crate. The CAMAC crate is probably off-line.

301. Invalid crate number during a CAMAC block transfer operation. The specified crate is not online.

302. An N greater than 23 error has occurred during a CAMAC block transfer operation.

303. A CAMAC NO-Q error has occurred during a CAMAC block transfer operation.

304. CAMAC no-sync error during a CAMAC block transfer operation.

305. A CAMAC NO-X error has occurred during a CAMAC block transfer operation.

306. A CAMAC non-existent memory error has occurred during a block transfer operation.

307. A CAMAC STE-error has occurred during a CAMAC block transfer operation.

308. A CAMAC timeout error has occurred during a CAMAC block transfer operation.

309. An undefined CAMAC error has occurred during a CAMAC block transfer operation.

310. Invalid crate number during a CAMAC single transfer operation. The specified crate in not online.

311. An N greater than 23 error has occurred during a CAMAC NAF operation.

312. A CAMAC NO-Q error has occurred during a CAMAC NAF operation.

313. A CAMAC STE - error during a CAMAC single transfer operation.

314. A CAMAC NO-X error has occurred during a CAMAC NAF operation.

315. A CAMAC non-existent memory error has occurred during a single transfer operation.

316. A CAMAC STE-error has occurred during a CAMAC single transfer operation.

317. A CAMAC timeout error has occurred during a CAMAC NAF operation.

318. An undefined CAMAC error has occurred during a CAMAC NAF operation.

401. Access violation, either the I/O status block cannot be written by the caller, or the parameters for device-dependent function codes are incorrectly specified.

402. The specified device is offline and not currently available for use.

403. Insufficient system dynamic memory is available to complete the service. There are probably no free IRPS, use SHOW MEMORY to see the number of free IRPS.

404. An invalid channel number was specified.

405. The specified channel does not exist, was assigned from a more privileged access mode, or the process does not have the necessary privileges to perform the specified functions on the device.

406. The QIO error is unknown to the CAMAC software.

501. Access violation, the device string cannot be read by the caller, or the channel number cannot be written by the caller.

502. The CAMAC device is allocated to another process.

503. Illegal device name. No device name was specified, the logical name translation failed, or the device string contains invalid characters.

504. The device name string has a length of 0 or has more than 63 characters.

505. No I/O channel is available for assignment.

506. The specified CAMAC device does not exist. Check the device string for misspellings or a missing colon and check that the device driver has been loaded.

507. The process tried to assign a CAMAC device on a remote node. CAMAC operations cannot be performed over a network.

508. The CAOPEN error is unknown to the CAMAC software.

601. An invalid channel number was specified.

602. The specified channel is not assigned or was assigned from a more privileged mode.

603. The CACLOS error is unknown to the CAMAC software.

701. An invalid CAMAC subaddress (A) was found. The CAMAC subaddress was either less than 0 or greater than 15 ($A < 0$ or $A > 15$).

702. Invalid mode byte.

703. A invalid CAMAC block transfer type was found. The legal block transfer types are QSTP, QIGN, QRPT, and QSCN with corresponding values of 0, 8, 16, and 24, respectively.

704. An invalid CAMAC function code (F) was found. The CAMAC Function code was either less than 0 or greater than 31 ($F < 0$ or $F > 31$).

705. An invalid CAMAC crate controller function was found. The valid CAMAC crate controller functions are INIT, CLEAR, SETINH, CLRINH, and ONLINE with corresponding values of 0, 1, 2, 3, and 4, respectively.

706. An invalid CAMAC slot number (N) was found. The slot number was either less than 1 or greater than 30 ($N < 1$ or $N > 30$).

707. Invalid LAM type

708. Invalid priority

709. A CAMAC block transfer control operation was specified which is invalid. Only CAMAC Read or Write block transfers are allowed. The function code (F) for the block transfer was either between 8 and 15 inclusive or between 24 and 31 inclusive ($8 < F < 15$ or $24 < F < 31$).

710. An in-line CAMAC read was specified. Only in-line CAMAC control and write operations are legal (F8 through F31)

711. The Data Buffer is not big enough to hold all the data for the CAMAC Control List

712. The CAMAC Control List is not big enough to hold all the commands

713. A CAMAC block transfer with a block size of zero was found. A CAMAC block transfer must have a size of at least one word.

714. Illegal CAMAC crate number.

INDEX

C

K

S

V