VISION Draft Standard

VITA 25-199x

Draft 0.4

May 29, 1997

This standard is being prepared by the VITA Standards Organization (VSO) and is unapproved.

Do not specify or claim conformance to this document.

VITA Standards Organization
7825 East Gelding Drive, Suite 104
Scottsdale, AZ 85260
Ph: 602-951-8866 Fx: 602-951-0720
URL: http://www.vita.com

TABLE OF CONTENTS

At the time this standard was completed, the VISION task group had the following membership:

>>> List to be provided. <<<

The following people were on the sponsor balloting committee:

>>> List to be provided. <<<

When the ANSI Standards Board approved this standard on Xxxx xx, 199x, it had the following membership:

>>> List to be provided. <<<

The following people contributed to the pre-VITA development of this specification (listed by institution):

Brookhaven National Lab: David Connor

CEBAF (Continuous Electron Beam Accelerator Facility): Dave Doughty, Chip Watson

CERN (European Laboratory for High-Energy Physics): Doris Burckhart, Chris Eck, Bert Jost, Douglas Kemp, Martin Merkel, Chris Parkman

Fermi National Accelerator Laboratory: Ed Barsotti, David Berg, Kurt Biery, John DeVoy, Robert Downing, Robert Goodwin, Gene Oleynik, James Pangburn, James Patrick, Ruth Pordes, Kurt Schurecht, Greg Schuweiler, Jay Wightman

Harvard University: Colin Gay

KEK (Japanese Laboratory for High-Energy Physics): Yasu Fukui, Yasishi Nagasaka, Masa Nomachi

LeCroy Corporation: Francois Lamarche, Uwe Uhmeyer

Los Alamos National Laboratory: Thomas Kozlowski

TRIUMF: Renee Poutissou, Graham Waters

University of Chicago: Ryan Pierce, John Wahl

University of New Mexico: Eric Gottlieb, Timothy Thomas

– the following 7 sections will not be in the final standard

Comments, Corrections and/or Additions

Anyone wishing to provide comments, corrections, and/or additions to this proposed standard, please direct them to the task group chair:

Jim Pangburn, VISION Chair & Draft Editor
Fermi National Accelerator Lab
P.O. Box 500 MS 318
Batavia, IL 60510-0500 (USA)
Phone: 630-840-2207 FAX: 630-840-6315
E-mail: pangburn@fnal.gov

The best way to provide corrections and small additions is by marking up the specific pages and FAXing them to the chair. For longer additions, the chair prefers textual information via e-mail. This draft standard is being prepared in HTML via emacs on UNIX, and converted to PostScript via an HTML-to-LaTeX converter and LaTeX on UNIX.

VSO and Other Standards

Anyone wanting information on other standards being developed by VSO, VME Product Directories, VME Handbooks, or general information on the VME market, please contact the VITA office using the information given on the front cover. Copies of the VSO minutes, plus VSO meeting announcements and agenda can be retrieved from the VITA web site listed on the front cover.

Draft Copies on Web Sites

Copies of this draft standard are available on a Fermilab web site, at:

`http://www-b0.fnal.gov:8000/vme/VITA-25-199x/`

in both HTML format for direct browsing, and PostScript format for printing on U.S. letter-size paper. Also, a copy in Adobe PDF (Portable Document Format) is available from the VITA web site, at:

`http://www.vita.com/`

under the ”VITA Standards Organization Activities” hyperlink.

Draft Summary

Added acknowledgements for pre-VITA development. Added second paragraph in 1.2.

Draft History

- 0.3:

  This draft is the first one with the ”Standard Terminology” section, as well as the sections for committee memberships, and the 7 sections (one of which is this one) to be removed upon final approval. The ”dump” methods have been removed. They were very UNIX-centric and proved inflexible in a test implementation. The ”transferProtocol” and ”autoThreshold” slave options and all mention of their behavior was removed. The ”geo32” component has been replaced by the ”vmePnP” component. The ”autoconf” component was removed. This idea is not yet ready for formal specification. The section ”Possible Applications for VISION” was added.

- 0.2:

  Reorganized so each section has: abstract name of method, description of behavior, bindings for four languages; within each binding, name, arguments, return value, and an example. Hopefully clarified much material in chapter 1. Added material to introductory sections in each chapter.

- 0.1:

  Added ”data move” method (Transfer Data Between two VISION Slaves). In Master Access API: changed ”requesterMode” to ”requestMode”, ”requesterType” to ”requestType”, and ”requesterLevel” to ”requestLevel”. Added ”arbitrationTimer” and ”transactionTimer”. Added errorReporter option to Component, Master, and Interrupt APIs.

- 0.0:

  Copied from ”VISION, revision 2” document and added VITA draft standard cover page, etc.

Issues & Concerns to be Resolved

Following are some of the issues and concerns that need to be resolved before final approval of this draft standard:

Task Group Ballots

It is hoped to hold the first task group meeting at the July 1997 VSO meeting in Vancouver, BC, Canada. After any discussion of issues there, the first task group ballot is expected to be called.

1. Introduction

1.1. Description of VISION

VISION (Versatile I/o Software Interface for Open-bus Networks) is a software specification which standardizes the low-level aspects of bus I/O. Although originally intended only for VMEbus, it can, through the use of ”components”, be applied to other similar technologies. A *VISION component* is a software ”plug-in” which encapsulates the support for a particular bus, set of protocols and access mechanisms. It is like a driver, although it may not be implemented as a driver on all platforms. With components, a VISION implementation may be used to integrate hybrid systems under a single software interface. As of this writing, VISION has been successfully applied to VMEbus (IEEE 1014-1987) and VME64 (ANSI/VITA 1-1994), FASTBUS (IEEE 960-1993), IndustryPack (ANSI/VITA 4-1994), and PCI (PMC - IEEE P1386.1).

1.2. The VISION standard for VME software

Reliability, reusability, and convenience of software for VMEbus systems will greatly benefit from a standard software interface. This is especially true with the most recent VME64 extensions proposed by VITA. Without a standard, thousands of programmers must independently implement the same support for the new hardware capabilities. While individual programmers keep the freedom to write application-specific algorithms, VISION provides them a simple framework for writing their software in a mutually-compatible way.

From the software viewpoint, there are two basic ways to use the VMEbus: transparently or opaquely. Using the bus transparently means making resources on the bus appear local by mapping them into the CPU address space. This is typically done at the Operating System (OS) level, but is sometimes needed at the application level. At the operating system level, it is convenient to have a hardware-independent way of configuring the maps. At the application level, it is convenient to have a hardware- and OS- independent mapping API. Using the bus opaquely means acknowledging the existence of the bus as an I/O channel. This requires a software interface. It also allows the use of hardware optimizations such as DMA (Direct Memory Access) which can drastically improve performance.

1.3. What VISION Provides

VISION specifies interfaces for software running on bus *masters*, which can initiate data transfers with bus *slaves*, possibly generate and/or respond to bus *interrupts*, and possibly act as slaves. This specification provides:

- A standard order of arguments and naming convention:

  ESONE (the committee for European Studies On Norms for Electronics) has provided such a framework for CAMAC (IEEE 583-1982).

- An Internet access standard:

  A program can be written on one platform, do local bus access, and then, without recompilation, do the same accesses on a completely different platform which can even sit elsewhere on an IP network, simply by changing one of the arguments. The building block of this Internet access capability is the ISO standard **Basic Encoding Rules** (ISO-8825), so universal compatibility is guaranteed.

- A standard way to add functionality:

  The ”component” approach permits run-time linking and an optimal usage of computer resources. VISION components can be shared, or potentially sold, just like PC drivers, because they have a standard fit.

1.4. Possible Applications for VISION

VISION is primarily intended as an application-level interface, and so it is well-suited to environments requiring custom driver-like software. However, it could also be implemented in a basic kernel I/O library, becoming part of a driver-kernel interface. This would enable the development of highly portable drivers, because VISION abstracts the details of bus interface hardware, and the remainder of the driver-kernel interface could abstract the operating system services.

1.5. Organization of this Document

This specification is grouped into five APIs: slave access, component management, master access, list management, and interrupt access. Each API is designated by a chapter in this document. These APIs correspond to classes in object-oriented bindings, and related functions for non-OO bindings. The reason for the division into five APIs is to allow implementors to implement, and users to use, only the functionality they need. The APIs allow for client/server support, but implementation of client/server is optional.

The Slave Access API (page 9) provides access to slaves (some of which may also be masters) from the master on which the software runs. Component Management (page 35) provides management of components. The Master Access API (page 48) provides access to features specific to the master. The List Management API (page 59) provides a ”list” facility (delayed execution of a series of operations). The Interrupt Access API (page 71) allows customization of the master’s response to bus interrupts.

Sections in each chapter describe the API’s methods. A one-line description of each method’s function comprises the section heading. This is followed by a detailed definition of the method’s behavior. Specific language bindings are then given, with each containing subsections showing:

- how to invoke the method, including argument descriptions;
- how the method denotes success or failure; and
- an example usage of the method.

1.6. Standard Terminology

To avoid confusion and make clear what the requirements for compliance are, many of the paragraphs in this standard are labeled with keywords that indicate the type of information they contain. The keywords are listed below:

- Rule
- Recommendation
- Suggestion
- Permission
- Observation

These keywords are used as follows:

Rule <chapter>.<number>:

Rules form the basic framework of this standard. They are sometimes expressed in text form and sometimes in the form of figures, tables or drawings. All rules shall be followed to ensure portability of user software. Rules use the ”shall” or ”shall not” words to emphasize the importance of the rule. The ”shall” and ”shall not” words are reserved exclusively for stating rules in this standard and are not used for any other purpose.

Recommendation <chapter>.<number>:

Recommendations offer useful advice for implementors. Failure to take the advice might result in awkward problems.

Suggestion <chapter>.<number>:

Suggestions contain advice which is helpful but not vital. The reader is encouraged to consider the advice before discarding it. Some design decisions are difficult until experience with a standard has been gained. Suggestions are included to aid the implementor who has not yet gained this experience.

Permission <chapter>.<number>:

In some cases, a rule does not specifically prohibit a certain approach, but the reader might be left wondering if that approach might violate the spirit of the rule or lead to a subtle problem. Permissions reassure the reader that a certain approach is acceptable and should cause no problems. The lower-case word ”may” is reserved exclusively for stating permissions in this standard and is not used for any other purpose.

Observation <chapter>.<number>:

Observations do not offer any specific advice. They usually follow naturally from what has just been discussed. They spell out implications of certain rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules so that the reader understands why the rule must be followed.

I.7. Technical Terminology

This section is to familiarize the reader with some of the less common or non-standard terminology used throughout this text.

method: This term is applied to methods in object-oriented (OO) languages, AND functions (or procedures) in non-OO languages.

component: A software ”plug-in” which adds functionality to a ”framework” designed to accept it. A VISION component (typically) implements a particular set of access mechanisms and protocols for I/O with slaves.

2. Slave Access API

The primary intention of the Slave Access API is to provide a flexible way for applications running on a bus master (or running over the network via a bus master) to perform generic I/O with arbitrary slaves. The basic idea is that an application creates a ”VISION slave object” (which usually corresponds to a single slave board) by specifying some unique identification, like a logical address space, slot number, etc. This object models the slave by providing methods to:

- Move data between the slave and local memory;
- Move data between two slaves;
- Establish a mapping of the slave to a range of memory addresses for direct, programmed access to the slave;
- Set new values for certain parameters which allow ”fine-tuning” of the object’s behavior;
- Get the current values of these parameters; AND
- Destroy the object

Rule 2.1:

An implementation of the Slave Access API for C, C++, Java and/or Microsoft Visual BASIC shall provide the precise class/method names, argument order, argument data types, and return data types given in the sections in this chapter. In other words, the remaining sections of this chapter form rules for implementations wishing to provide bindings to these languages.

Observation 2.1:

In C, C++, Java and Microsoft Visual BASIC, any changes in class/method names, argument order, argument data types, or return data types will render user code non-portable.

Permission 2.1:

Implementations for programming languages not mentioned above may choose other calling conventions.

Recommendation 2.1:

Implementations for programming languages not mentioned above should attempt to adhere to the basic spirit of the calling conventions for the language most similar to it.

Observation 2.2:

The order of the arguments obeys a convention; the first argument is a software identification of the slave module (in object-oriented languages like C++, this argument is implicit). The next arguments follow in order from the most compulsory to the most optional. The arguments contain only large and/or varying numbers, such as addresses, lengths, and pointers. The choices of access methods, or other limited-choice numbers, are specified at the time of creation of the slave object. As an example, a module that can be accessed with two distinct address modifiers must be accessed through the creation of two distinct slave objects.

2.1. Create a VISION slave object

This method shall attempt to create a new object based on the name specified. If successfully created, this object shall provide access to the specified bus slave until the object is destroyed. If a remote server is specified and the implementation supports client/server, the object shall provide access to the specified slave on the server's bus. The new object shall be initialized with default values for slave-specific options (page 33) .

C Binding

int **VISIONopen**(VISION_SLAVE *slave, char *name, char *server);

(OUT) slave: a reference (pointer) to an application-allocated variable of type "VISION_SLAVE". This type is defined by the inclusion of *VISION.h* (see "Compile-time Conventions" (page 85)). Upon success, the implementation shall fill the variable with a "handle" for the application's subsequent use with the other methods in this API.

(IN) name: a name which represents a slave entity (module, logical address space, etc.). For maximum flexibility with present and future bus standards, various forms of this argument may be accepted, depending upon which components are available. The first part of the name represents the component to be used for accessing the entity. Please refer to "Introduction to VISION Components" (page 35) for further information.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server's bus. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
/* use "vmePnP" component for the first function of the
module in slot 4 of the local crate. */
status = VISIONopen( &mySlave, "vmePnP:/slot=4", "" );
```

C++ Binding

(in class VISIONslave):
VISIONslave(int *status, char *name, char *server);

(OUT) status: status is returned in this way due to the semantics of object construction.

(IN) name: a name which represents a slave entity (module, logical address space, etc.). For maximum flexibility with present and future bus standards, various forms of this argument may be accepted, depending upon which components are available. The first part of the name represents the component to be used for accessing the entity. Please refer to "Introduction to VISION Components" (page 35) for further information.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server's bus. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall fill *status with 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave *mySlave;
// use "vmePnP" component for the first function of the
// module in slot 4 of the local crate.
mySlave = new VISIONslave( &status, "vmePnP:/slot=4", "" );
```

Java Binding

(in public class VISIONslave):

public **VISIONslave**(String name, String server) throws VISIONexception;

(IN) name: a name which represents a slave entity (module, logical address space, etc.). For maximum flexibility with present and future bus standards, various forms of this argument may be accepted, depending upon which components are available. The first part of the name represents the component to be used for accessing the entity. Please refer to "Introduction to VISION Components" (page 35) for further information.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server's bus. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
VISIONslave mySlave = null;
// use "vmePnP" component for the first function of the
// module in slot 4 of the local crate.
try {
  mySlave = new VISIONslave( "vmePnP:/slot=4", "" );
  }
catch( VISIONexception error ) {
  System.err.println( "Error creating VISION slave!" );
  System.exit(-1);
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONopen**(ByRef slave As Long, ByVal name As String, ByVal server As String) As Long

(OUT) slave: a reference to an application-allocated variable of type "Long". Upon success, the implementation shall fill this variable with a "handle" for the application's subsequent use with the other methods in this API.

(IN) name: a name which represents a slave entity (module, logical address space, etc.). For maximum flexibility with present and future bus standards, various forms of this argument may be accepted, depending upon which components are available. The first part of the name represents the component to be used for accessing the entity. Please refer to "Introduction to VISION Components" (page 35) for further information.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server's bus. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long mySlave
REM use "vmePnP" component for the first function of the
REM module in slot 4 of the local crate.
status& = VISIONopen( ByRef mySlave, ByVal "vmePnP:/slot=4",
        ByVal "" )
```

2.2. Transfer data between local memory and a VISION slave

The "read" method shall attempt to transfer data from the slave to local memory.

The "write" method shall attempt to transfer data from local memory to the slave.

The "MCAS" (Masked Compare-And-Set) method shall attempt to:

- read the data;
- perform a bitwise AND with the given mask;
- and if the result matches the given compare value, write the new data given in the set value.

The read data shall always be placed in the buffer. This operation shall be performed while the bus is locked so that no other master can access the bus. It is intended for implementing software synchronization mechanisms (like semaphores) over the bus.

By default, these methods shall use the fastest transfer type (programmed I/O, DMA, etc.) the hardware provides. By default, these methods shall execute synchronously; waiting for their I/O to complete before returning to the caller. See the slave configuration method (page 29) for information on modifying these behaviors.

C Bindings

int **VISIONread**(VISION_SLAVE slave, unsigned int offset, unsigned int req_bytes, unsigned int *real_bytes, void *data);

int **VISIONwrite**(VISION_SLAVE slave, unsigned int offset, unsigned int req_bytes, unsigned int *real_bytes, void *data);

int **VISIONmcas**(VISION_SLAVE slave, unsigned int offset, unsigned int req_bytes, void *data, unsigned int mask_value, unsigned int compare_value, unsigned int set_value);

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) offset: the offset to read from or write to, relative to the base address specified when the object was created.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference (pointer) to application-allocated space the size of an unsigned int. The method shall fill the space with the number of bytes successfully transferred (unless the reference is NULL).

(IN/OUT) data: reference (pointer) to a data buffer allocated by the application. The data will be copied out of or into this array to/from the slave.

(IN) mask_value: the bitmask to apply to the data read.

(IN) compare_value: the value to compare the masked data against.

(IN) set_value: the new value to write if the comparison succeeds.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
int status;
unsigned int real_bytes;
unsigned long buffer[0x100];
status = VISIONopen( &mySlave, ... );
```

```
.
.
/* read data into my buffer from slave at its offset 0x2000 */
status = VISIONread( mySlave, 0x2000, sizeof(buffer),
        &real_bytes, (void *)buffer);
```

C++ Bindings

(in class VISIONslave):

int **Read**(unsigned int offset, unsigned int req_bytes, unsigned int *real_bytes, void *data);

int **Write**(unsigned int offset, unsigned int req_bytes, unsigned int *real_bytes, void *data);

int **MCAS**(unsigned int offset, unsigned int req_bytes, void *data, unsigned int mask_value, unsigned int compare_value, unsigned int set_value);

(IN) offset: the offset to read from or write to, relative to the base address implied when the object was created.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference (pointer) to application-allocated space the size of an unsigned int. The method shall fill the space with the number of bytes successfully transferred (unless the reference is NULL).

(IN/OUT) data: reference (pointer) to a data buffer allocated by the application. The data will be copied out of or into this array to/from the slave.

(IN) mask_value: the bitmask to apply to the data read.

(IN) compare_value: the value to compare the masked data against.

(IN) set_value: the new value to write if the comparison succeeds.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave *mySlave;
int status;
unsigned int real_bytes;
unsigned long buffer[0x100];
mySlave = new VISIONslave( ... );
.
.
// write my buffer to slave at its offset 0x10000
status = mySlave->Write( 0x10000, sizeof(buffer),
        &real_bytes, (void *)buffer );
```

Java Bindings

(in public class VISIONslave):

public int **Read**(int offset, int req_bytes, int data[]) throws VISIONexception;

public int **Write**(int offset, int req_bytes, int data[]) throws VISIONexception;

public int **MCAS**(int offset, int req_bytes, int data[], int mask_value, int compare_value, int set_value) throws VISIONexception;

(IN) offset: the offset to read from or write to, relative to the base address implied when the object was created.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(IN/OUT) data: reference (pointer) to a data buffer allocated by the application. The data will be copied out of or into this array to/from the slave.

(IN) mask_value: the bitmask to apply to the data read.

(IN) compare_value: the value to compare the masked data against.

(IN) set_value: the new value to write if the comparison succeeds.

Execution Status The method shall return the number of bytes successfully transferred. In the event of an error, it shall throw a VISIONexception (page 96) .

Example

```
VISIONslave mySlave = null;
int bytes_read;
int data[] = new int[0x1000];
mySlave = new VISIONslave( ... );
.
.
// read 4KB into array, starting at offset 4KB
try {
  bytes_read = mySlave.Read( 0x1000, 0x1000, data );
  }
catch( VISIONexception error ) {
  System.err.println( "Error reading VISION slave!" );
  System.exit(-1);
  }
```

Microsoft Visual BASIC Bindings

Public Declare Function **VISIONread**(ByVal slave As Long, ByVal offset As Long, ByVal reqbytes As Long, realbytes As Long, data As Any)

Public Declare Function **VISIONwrite**(ByVal slave As Long, ByVal offset As Long, ByVal reqbytes As Long, realbytes As Long, data As Any)

Public Declare Function **VISIONmcas**(ByVal slave As Long, ByVal offset As Long, ByVal reqbytes As Long, data As Any, ByVal mask_value As Long, ByVal compare_value As Long, ByVal set_value As Long)

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) offset: the offset to read from or write to, relative to the base address implied when the object was created.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference to an application-declared Long. The method shall fill the Long with the number of bytes successfully transferred (unless the reference is NULL).

(IN/OUT) data: a reference to a data buffer allocated by the application. The data will be copied out of or into this array to/from the slave.

(IN) mask_value: the bitmask to apply to the data read.

(IN) compare_value: the value to compare the masked data against.

(IN) set_value: the new value to write if the comparison succeeds.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
REM read 4KB into array, starting at offset 0
DIM data&[4096]
Long status, mySlave, bytes
status& = VISIONopen( ByRef mySlave, ... )
.
.
status& = VISIONread( mySlave, 0, 4096, ByRef bytes, data )
```

2.3. Create mapping between local memory and a VISION slave

This method shall attempt to create a mapping between the specified slave and part of the caller's address space. The mapping shall be valid starting at the given offset within the slave, and continuing for at least the given number of bytes.

C Binding

int **VISIONmap**(VISION_SLAVE slave, unsigned int offset, unsigned int bytes, void **address);

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) offset: specifies the offset within the slave.

(IN) bytes: specifies the length, in bytes, of the address region to map.

(OUT) *address: the method shall fill *address with the mapping's address upon successful execution.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
int status;
void *my_addr;
status = VISIONopen( &mySlave, ... );
.
.
/* map 8MB from my module into my address space */
status = VISIONmap( mySlave, 0, 0x800000, &my_addr );
/* write there */
*(int *)my_addr = 0xabadcafe;
```

C++ Binding

(in class VISIONslave):

int **Map**(unsigned int offset, unsigned int bytes, void **address);

(IN) offset: specifies the offset within the slave (usually 0 is most useful).

(IN) bytes: specifies the length, in bytes, of the address region to map.

(OUT) *address: the method shall fill *address with the mapping's address upon successful execution.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave *mySlave;
int status;
mySlave = new VISIONslave( ... );
.
```

```
.
void *myAddr;
// map 256KB from my module into my address space
status = mySlave->Map( 0, 0x40000, &myAddr );
/* read from it */
test = *(int *)myAddr;
```

Java Binding

Implementing this functionality in Java would require more than simply a binding; it would have to be added to the Java Virtual Machine. This would be in direct opposition to the principles of Java. It is therefore considered out of the scope of this specification.

Microsoft Visual BASIC Binding

Implementing this functionality in Visual BASIC would require more than simply a binding; it would have to be added to the Visual BASIC run-time environment. It is therefore considered out of the scope of this specification.

2.4. Transfer data between two VISION slaves

This method shall attempt to move data directly from one VISION slave to another, without being copied into local memory. Naturally, the two slaves must have a direct path to one another in the appropriate direction for this to be possible.

By default, this method shall use the fastest transfer type (programmed I/O, DMA, etc.) the hardware provides. By default, this method shall execute synchronously; waiting for the I/O to complete before returning to the caller. See the slave configuration method (page 29) for information on modifying these behaviors.

C Binding

int **VISIONdataMove**(VISION_SLAVE dest_slave, VISION_SLAVE source_slave, unsigned int dest_offset, unsigned int source_offset, unsigned int req_bytes, unsigned int *real_bytes);

(IN) dest_slave/source_slave: specifies handles as filled by VISIONopen(). The first is the slave to which data should be written, and the second is the slave from which data should be read.

(IN) dest_offset/source_offset: the offset to which to write in the destination slave, and the offset from which to read in the source slave, respectively.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference (pointer) to application-allocated space the size of an unsigned int. The method shall fill the space with the number of bytes actually transferred (unless the reference is NULL).

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
VISION_SLAVE myPCIslave;
int status;
unsigned int real_bytes;
status = VISIONopen( &mySlave, ... );
.
.
/* open first PMC's first I/O space */
status = VISIONopen( &myPCIslave,
        "pmc:/position=a/space=io/spaceNumber=0/base=0", "" );
if( 0 == status )
  {
  /* move 8KB to PCI slave from VME slave (offset 0x10000) */
  status = VISIONdataMove( myPCIslave, mySlave, 0x10000,
        0, 0x2000, &real_bytes );
  }
```

C++ Binding

(in class VISIONslave):

int **DataMove**(VISIONslave source_slave, unsigned int offset, unsigned int source_offset, unsigned int req_bytes, unsigned int *real_bytes);

(IN) source_slave: specifies a VISIONslave from which data should be read.

(IN) offset/source_offset: the offset to which to write in the destination slave, and the offset from which to read in the source slave, respectively.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference (pointer) to application-allocated space the size of an unsigned int. The method shall fill the space with the number of bytes actually transferred (unless the reference is NULL).

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave mySlave;
int status;
unsigned int real_bytes;
mySlave = new VISIONslave( ... );
.
.
// open first PMC's first I/O space
VISIONslave *myPCIslave;
myPCIslave = new VISIONslave( &status,
        "pmc:/position=a/space=io/spaceNumber=0/base=0", "" );
if( 0 == status )
  {
  // move 8KB to PCI slave from VME slave (offset 0x10000)
  status = myPCIslave->DataMove( mySlave, 0x10000, 0,
        0x2000, &real_bytes );
  }
```

Java Binding

(in public class VISIONslave):

public int **DataMove**(VISIONslave source_slave, int offset, int source_offset, int req_bytes) throws VISIONexception;

(IN) source_slave: specifies VISIONslave from which to read data.

(IN) offset/source_offset: the offset to which to write in the destination slave, and the offset from which to read in the source slave, respectively.

(IN) req_bytes: the number of bytes the application wishes to transfer.

Execution Status The method shall return the number of bytes successfully transferred. In the event of an error, it shall throw a VISIONexception (page 96) .

Example

```
VISIONslave mySlave = null;
VISIONslave myPCIslave = null;
int bytes_moved;
mySlave = new VISIONslave( ... );
```

```
.
.
// open first PMC's first I/O space
try {
  myPCIslave = new VISIONslave(
        "pmc:/position=a/space=io/spaceNumber=0/base=0", "" );
  }
catch( VISIONexception error ) {
  System.err.println( "Couldn't create VISION slave!" );
  System.exit(-1);
  }
// move 8KB from VME slave (offset 0x10000) to PMC slave */
try {
  bytes_moved = mySlave.DataMove( myPCIslave,
        0x10000, 0, 0x2000 );
  }
catch( VISIONexception error ) {
  System.err.println( "Error performing VISION data move!" );
  System.exit(-1);
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONdataMove**(ByVal dest_slave As Long, ByVal source_slave As Long, ByVal dest_offset As Long, ByVal source_offset As Long, ByVal reqbytes As Long, ByRef realbytes As Long)

(IN) dest_slave/source_slave: specifies handles as filled by VISIONopen(). The first is the slave to which data should be written, and the second is the slave from which data should be read.

(IN) dest_offset/source_offset: the offset to which to write in the destination slave, and the offset from which to read in the source slave, respectively.

(IN) req_bytes: the number of bytes the application wishes to transfer.

(OUT) real_bytes: a reference (pointer) to application-allocated space the size of an unsigned int. The method shall fill the space with the number of bytes actually transferred (unless the reference is NULL).

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long myPCIslave, mySlave, status, real_bytes
status = VISIONopen( ByRef mySlave, ... )
.
.
REM open first PMC's first I/O space
status = VISIONopen( ByRef myPCIslave,
        ByVal "pmc:/position=a/space=io/spaceNumber=0/base=0",
        ByVal "" )
If status = 0
  REM move 8KB to PCI slave from VME slave (offset 65536)
  status = VISIONdataMove( myPCIslave, mySlave, 65536,
        0, 8192, ByRef real_bytes )
```

```
End If
```

2.5. Ensure completion of outstanding I/O requests

NOTE: this method is pertinent only when using asynchronous mode or memory-mapping over a network. This method allows an application to ensure that all I/O associated with a slave is complete (with or without errors). If the I/O has not completed when the given timeout expires, the method shall return the error ETIMEDOUT. This behavior is different if the timeout is given as zero, in which case, the method shall indicate the status of the I/O and return immediately.

C Binding

int **VISIONsync**(VISION_SLAVE slave, unsigned int timeout);

(IN) slave: specifies a handle as returned by VISIONopen().

(IN) timeout: specifies a timeout, in microseconds, after which the method will return.

Execution Status If a timeout of zero is specified:

- if the slave's I/O is complete, 0 (zero) shall be returned;
- otherwise, EINPROGRESS shall be returned.

If a non-zero timeout is specified:

- the implementation shall poll the completion status of the slave's outstanding I/O for the specified time. If, at the end of this time, the I/O is still not complete, EINPROGRESS shall be returned. Otherwise, 0 (zero) shall be returned.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
int status;
status = VISIONopen( &mySlave, ... );
{I/O requests (read/write calls, memory-mapped operations)}
/* ensure completion of the above within 10 ms */
status = VISIONsync( mySlave, 10000 );
```

C++ Binding

(in class VISIONslave):

int **Sync**(unsigned int timeout);

(IN) timeout: specifies a timeout, in microseconds, after which the method will return.

Execution Status If a timeout of zero is specified:

- if the slave's I/O is complete, 0 (zero) shall be returned;
- otherwise, EINPROGRESS shall be returned.

If a non-zero timeout is specified:

- the implementation shall poll the completion status of the slave's outstanding I/O for the specified time. If, at the end of this time, the I/O is still not complete, EINPROGRESS shall be returned. Otherwise, 0 (zero) shall be returned.

Example

```
#include "VISION.h"
VISIONslave mySlave;
int status;
mySlave = new VISIONslave( ... );
{I/O requests (read/write calls, memory-mapped operations)}
// ensure completion of the above within 10 ms
status = mySlave->Sync( 10000 );
```

Java Binding

(in public class VISIONslave):

public void **Sync**(int timeout) throws VISIONexception;

(IN) timeout: specifies a timeout, in microseconds, after which the method will return.

Execution Status If a timeout of zero is specified:

- if the slave's I/O is complete, the method shall complete.
- otherwise, a VISIONexception (page 96) shall be thrown (it shall be a VISIONtimeoutException).

If a non-zero timeout is specified:

- the implementation shall poll the completion status of the slave's outstanding I/O for the specified time. If, at the end of this time, the I/O is still not complete, a VISIONexception (page 96) shall be thrown (it shall be a VISIONtimeoutException). Otherwise, the method shall return.

Example

```
VISIONslave mySlave = null;
mySlave = new VISIONslave( ... );
{I/O requests (read/write calls)}
// ensure completion of the above within 10 ms
try {
  mySlave.Sync( 10000 );
  }
catch( VISIONexception error ) {
  System.err.println( "Error on sync!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONsync**(ByVal slave As Long, ByVal timeout As Long) As Long

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) timeout: specifies a timeout, in microseconds, after which the method will return.

Execution Status If a timeout of zero is specified:

- if the slave's I/O is complete, 0 (zero) shall be returned;
- otherwise, EINPROGRESS shall be returned.

If a non-zero timeout is specified:

- the implementation shall poll the completion status of the slave's outstanding I/O for the specified time. If, at the end of this time, the I/O is still not complete, EINPROGRESS shall be returned. Otherwise, 0 (zero) shall be returned.

Example

```
Long mySlave, status
status = VISIONopen( ByRef mySlave, ... )
{I/O requests (read/write calls)}
REM ensure completion of the above within 10 ms
status& = VISIONsync( mySlave, 10000 )
```

2.6. Get slave-related information

This method will return information related to the slave. The application invokes it, specifying an "option" which represents some type of information. The method returns to the application the current value of the option. For object-oriented languages, an implementation may provide this functionality in the form of accessors with the same names as the options.

C Binding

int **VISIONinfo**(VISION_SLAVE slave, char *option, void *value, int value_size);

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(OUT) value: A reference (pointer) to memory allocated by the application to contain the information returned by the method.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value. The method will, however, denote a lack of sufficient buffer space by returning E2BIG.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
int status;
unsigned int datasize;
status = VISIONopen( &mySlave, ... );
.
.
/* check the current dataSize */
status = VISIONinfo( mySlave, "dataSize",
        (void *)&datasize, sizeof(datasize) );
```

C++ Binding

(in class VISIONslave):

int **Info**(char *option, void *value, int value_size);

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(OUT) value: A reference (pointer) to memory allocated by the application to contain the information returned by the method.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value. The method will, however, denote a lack of sufficient buffer space by returning E2BIG.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave *mySlave;
int status;
mySlave = new VISIONslave( &status, ... );
.
.
// check the current data size
int dataSize;
status = mySlave->Info( "dataSize",
        (void *)&dataSize, sizeof(dataSize) );
```

Java Binding

(in public class VISIONslave):

public int **Info**(String option) throws VISIONexception;

public String **StringInfo**(String option) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Info() should be used on options whose values are integral;
- StringInfo() should be used on options whose values are strings.

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

Execution Status Upon success, the appropriate information shall be returned. A VISIONexception (page 96) shall be thrown upon failure.

Example

```
VISIONslave mySlave = null;
mySlave = new VISIONslave( ... );
.
.
// check the current data size
int dataSize;
try {
  dataSize = mySlave.Info( "dataSize" );
  }
catch( VISIONexception error ) {
  System.err.println( "Error getting dataSize!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONinfo**(ByVal slave As Long, ByVal name As String, value As Any, ByVal valSize As Long) As Long

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(OUT) value: A reference (pointer) to memory allocated by the application to contain the information returned by the method.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value. The method will, however, denote a lack of sufficient buffer space by returning E2BIG.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long mySlave, status, datasize
status = VISIONopen( ByRef mySlave, ... )
.
.
REM check the current dataSize
status& = VISIONinfo( mySlave, ByVal "dataSize",
        ByRef datasize, 4 )
```

2.7. Set slave-related configuration

This method allows an application to control the behavior of the implementation by setting new values for ”options”. The application invokes it, specifying an option and a new value for it. For object-oriented languages, an implementation may also provide this functionality in the form of modifiers with the same names as the options.

C Binding

int **VISIONconfig**(VISION_SLAVE slave, char *option, void *value, int value_size);

(IN) slave: specifies a handle as filled by VISIONopen().

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(IN) value: a reference (pointer) to the value to be set for the specified option. If NULL, no attempt to modify the option’s value will be made.

(IN) value_size: size, in bytes, of data referenced by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_SLAVE mySlave;
int status;
status = VISIONopen( &mySlave, ... );
.
.
/* set the current transfer protocol to BLT */
status = VISIONconfig( mySlave, "transferProtocol",
         "blt", strlen("blt") );
```

C++ Binding

(in class VISIONslave):

int **Config**(char *option, void *value, int value_size);

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) (2.10) lists the options defined by this revision of the API.

(IN) value: a reference (pointer) to the value to be set for the specified option. If NULL, no attempt to modify the option’s value will be made.

(IN) value_size: size, in bytes, of data referenced by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONslave *mySlave;
int status;
mySlave = new VISIONslave( &status, ... );
.
.
// set the current transfer protocol to BLT
status = mySlave->Config( "transferProtocol", "blt",
        strlen("blt") );
```

Java Binding

(in public class VISIONslave):

public void **Config**(String option, int value) throws VISIONexception;

public void **StringConfig**(String option, String value) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Config() should be used on options whose values are integral;
- StringConfig() should be used on options whose values are strings.

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(IN) value: the value to be set for the specified option.

Execution Status Upon success, the method shall complete normally. A VISIONexception (page 96) shall be thrown upon failure.

Example

```
VISIONslave mySlave = null;
mySlave = new VISIONslave( ... );
.
.
// set the current transfer protocol to BLT
try {
  mySlave.StringConfig( "transferProtocol", "blt" );
  }
catch( VISIONexception error ) {
  System.err.println( "Error setting transferProtocol!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONconfig**(ByVal slave As Long, ByVal name As String, value As Any, ByVal valSize As Long) As Long

(IN) slave: specifies a handle as filled by VISIONopen(). For requests to be fulfilled by the basic API, its value will be ignored; otherwise, the component corresponding to the slave will execute the request.

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Slave Options Table (page 33) lists the options defined by this revision of the API.

(IN) value: a reference (pointer) to the value to be set for the specified option. If NULL, no attempt to modify the option's value will be made (this behavior is convenient for some options).

(IN) value_size: size, in bytes, of data referenced by "value".

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long mySlave, status
status = VISIONopen( ByRef mySlave, ... )
.
.
REM set the current transfer protocol to BLT
status& = VISIONconfig( mySlave,
        ByVal "transferProtocol", ByVal "blt", 3 )
```

2.8. Terminate a VISION slave object

This method terminates the specified object - freeing any resources allocated (by the implementation) on behalf of the application. Any pending I/O requests are effectively aborted. NOTE: in some environments, if the application terminates without invoking this method for each created object, it may not be possible for the implementation to free the associated resources.

C Binding

int **VISIONclose**(VISION_SLAVE slave);

(IN) slave: specifies a handle as filled by VISIONopen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = VISIONclose( mySlave );
```

C++ Binding

(in class VISIONslave):
~VISIONslave();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
delete mySlave;
```

Java Binding

(garbage collected)

Microsoft Visual BASIC Binding

Public Declare Function **VISIONclose**(ByVal slave As Long) As Long

(IN) slave: specifies a handle as filled by VISIONopen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status& = VISIONclose( mySlave )
```

2.10. Slave Options Table

TABLE 2.1: Slave Options

Option Name	Data Type	Description
addrSize	INTEGER	Address size (number of bytes asserted simultaneously on the address bus).
dataSize	INTEGER	Data size (number of bytes asserted simultaneously on the data bus).
transferType	STRING	The transfer type to use: • pio: programmed (CPU-controlled) I/O • dma: DMA (Direct Memory Access) • best (DEFAULT): the fastest of the above supported by the hardware
FIFO	INTEGER	A non-zero value indicates that the implementation should treat a transfer request as if it were addressing a FIFO (meaning the bus address should NOT be incremented during the I/O). A zero value (DEFAULT) means normal sequential behavior (the address is incremented during I/O).
localFIFO	INTEGER	A non-zero value indicates that the implementation should treat a transfer request as if the local memory were a FIFO (meaning the memory address should NOT be incremented during the I/O). A zero value (DEFAULT) means normal sequential behavior (the address is incremented during I/O).
wordSize	INTEGER	This specifies (for purposes of automatic byte-swapping) how many bytes are in a word on the local CPU (e.g. 4 for 32-bit CPUs). It defaults to sizeof(long).
syncMode	STRING	The synchronization mode to use: • sync (DEFAULT): I/O requests block the caller • async: I/O requests are non-blocking (requiring subsequent "sync" call)
errorCode (R)	INTEGER	The error code from the last operation performed on this object (possibly 0, denoting no error). This is useful in asynchronous mode.
realBytes (R)	INTEGER	The number of bytes actually transferred in the last data transfer operation performed on this object.
errorReporter	FUNCTION	The function to invoke when slave errors occur.
name (R)	STRING	The name with which the object was created.
server (R)	STRING	The server with which the object was created.

(R) = read-only / (W) = write-only

Slave Options Table Notes

These notes are intended to clarify or give further information about the items in the table:

- **syncMode**:

  Using asynchronous mode implies that an application must take the following extra steps for reliable operation:

– maintain any allocated space in which a VISION method is to return data (e.g. a ”bytes_written” argument) until the associated I/O is known to be complete;
– not switch back to synchronous mode and expect synchronous results until outstanding asynchronous I/O is known to be complete.

Note that when hardware does not support truly asynchronous operation, the software must still allow this option to be set, and otherwise give the appearance of asynchronous operation, for application portability.

- **errorCode**:

  When using ”async” (non-blocking) mode, errors may occur during the transfer which cannot be returned by the transfer method because it has already returned. This can be used to obtain the error code associated with the last transfer.

- **errorReporter**:

  NOTE: this is only applicable in languages with ”function pointers” (variables which can point to functions to allow run-time dispatch). This specifies a function the implementation shall invoke automatically whenever an error occurs. This allows an application to automatically take some action whenever a VISION method returns an error, e.g. printing a message to the controlling terminal, via some network messaging facility, etc. The reporter function should return an integer value, and take two arguments: a numeric error code and a string which describes the error in human terms. If the function is not so declared, a fatal error may occur (e.g. a segmentation violation). If the function returns 0 (zero), the implementation shall perform no further action. If the function returns a non-zero value, the implementation shall assume the application wants the default error reporting behavior to occur, and shall perform it. A NULL in ”value” denotes disabling of the automatic error reporting mechanism.

  C example:

```
int myReporter( int code, char *message )
  {
  .
  .
  return( 0 );
  }
main()
  {
  VISION_SLAVE mySlave;
  int status;
  int (*errorFunc)(int,char *);
  status = VISIONopen( &mySlave, ... );
  .
  .
  errorFunc = myReporter;
  status = VISIONconfig( mySlave, "errorReporter",
        &errorFunc, sizeof(errorFunc) );
  .
  .
  }
```

3. Component Management API

3.1. Introduction to VISION Components

Optional ”components” allow the Slave API to be more flexible and adaptable. A ”component”, as defined here, is a specific software ”plug-in” which provides access to slaves in a specific way. A component is usually associated with a specific type of hardware environment (for example, a certain bus, or with VMEbus, a certain backplane type, support of VME64, VME64x, etc.). In other words, a component provides the ”how to get there” aspect of I/O. As new functionality is added to a bus hardware specification, new components can be created to implement the new functionality. Also, sites can maintain only the components they require, thus reducing their overhead in terms of computing resources, expertise and maintenance.

Rule 3.1:

An implementation of the Component Management API for C, C++, Java and/or Microsoft Visual BASIC shall provide the precise class/method names, argument order, argument data types, and return data types given in the sections in this chapter. In other words, the remaining sections of this chapter form rules for implementations wishing to provide bindings to these languages.

Observation 3.1:

In C, C++, Java and Microsoft Visual BASIC, any changes in class/method names, argument order, argument data types, or return data types will render user code non-portable.

Permission 3.1:

Implementations for programming languages not mentioned above may choose other calling conventions.

Recommendation 3.1:

Implementations for programming languages not mentioned above should attempt to adhere to the basic spirit of the calling conventions for the language most similar to it.

3.2. Application View of Components

3.2.1. Introduction

An application sees VISION components in terms of:

- the ”name” argument specified when creating a new VISION slave object (page 10) .
- the Component Management API, which allows component ”housekeeping” and access to any component-specific functions;

Beyond this, an application need not be written in a manner specific to any particular VISION component.

3.2.2. Name Format

The ”name” argument uses the following general format:

```
<component-name>:[/]<parameter-name>[= <value>][/ ..]
```

where <component-name> specifies a component to use, <parameter-name> specifies a ”parameter” needed by the component, and the optional <value> specifies a value needed by some parameters. Components may require more than one parameter, in which case a slash (”/”) character should be used to separate parameters. The slash preceding the first parameter name is optional. Components may also assume default values for parameters which are not specified by the application. Examples are shown in the table below.

Naming Examples

Name	Meaning
vme:addr=std/base=0xffff00	Use the "vme" component (standard VME addressing), accessing standard (A24) non-privileged data space, with a "base address" of 0xffff00
vmePnP:/slot=4	Use the "vmePnP" component (VME64x address space relocation using geographically-addressed CR/CSR) to access the first function of the module in slot 4
pmc:position=b/space=mem	Use the "pmc" component to access the first memory space on the second ("b" position) PMC
vme:/addr=ext/priv=s/space=p	Use the "vme" component to access extended (A32) supervisor-mode program space, with a base address of 0

Parameter names should be as self-explanatory as possible, and use common names when common functionality is involved.

3.2.3. Portability

The main goal in creating the concept of components for this API is application portability: an application can be written in a manner which is totally independent of the access method (bus protocol, etc.) used for communicating with the required device(s). This allows the end-user to customize the application for the particular system in which it is to be used. If the author does not wish to do the extra work required for this, however, the application is still extremely portable, requiring only a change of the "name" argument in its call(s) to the "open" method, to completely change the environment in which it may run.

3.3. Implementation View of Components

3.3.1. Component Implementation Guidelines

The most flexible way to implement components is via dynamic loading. This means that a component's module is loaded when its use is requested for the first time, and its methods/functions are dynamically bound. Dynamic binding, however, imposes overhead via indirection. Some implementations may not be able to afford this overhead, so dynamic loading/binding is not a requirement. An implementation could minimize overhead by "hardwiring" the equivalent of one or more components. In order to conform to this specification, such an implementation MUST accept the same format for the "name" argument (in the Slave API) as a corresponding component would. For example, an implementation may only provide the equivalent of the "vme" component (page 97) , in which case it must accept a name of the form:

```
vme:[/]<parameter-name>[=<value>][/ ..]
```

to allow user's code to run unchanged.

3.4. Create new "component management" object

This method shall attempt to create a new component management object. If successfully created, this object shall provide management of components until the object is destroyed. If a remote server is specified and the implementation supports client/server, the object shall provide management of components on the server. Note that this object does not really represent a component; but rather is a "management object" which allows management of one or more components.

C Binding

int **VISIONcompOpen**(VISION_COMPONENT *component, char *server);

(OUT) component: a pointer to an application-allocated variable of type "VISION_COMPONENT". This type is defined by the inclusion of *VISION.h* (see "Compile-time Conventions" (page 85)). Upon success, the implementation shall fill the variable with a "handle" for the application's subsequent use with the other methods in this API.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_COMPONENT myComponent;
/* we want to manage components on "bears" */
status = VISIONcompOpen( &myComponent, "bears" );
```

C++ Binding

(in class VISIONcomponent):
VISIONcomponent(int *status, char *server);

(OUT) status: status is returned in this way due to the semantics of object construction.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall fill *status with 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONcomponent *myComponent;
int status;
// we want to manage components on "bears"
myComponent = new VISIONcomponent( &status, "bears" );
```

Java Binding

(in public class VISIONcomponent):

public **VISIONcomponent**(String server) throws VISIONexception;

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
VISIONcomponent myComponent = null;
// we want to manage components on "bears"
try {
  myComponent = new VISIONcomponent( "bears" );
} catch( VISIONexception error ) {
System.err.println( "Error creating VISIONcomponent!" );
System.exit(-1);
}
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONcompOpen**(ByRef component As Long, ByVal server As String) As Long

(OUT) component: a reference to an application-allocated variable of type "Long". Upon success, the implementation shall fill the variable with a "handle" for the application's subsequent use with the other methods in this API.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long myComponent
REM we want to manage components on "bears"
status& = VISIONcompOpen( ByRef myComponent, "bears" )
```

3.5. Get component-related information

This method shall return information related to components. The application invokes it, specifying a component, and an "option" which represents some type of information. The method shall return to the application the current value of the option. For object-oriented languages, an implementation may provide this functionality in the form of accessors with the same names as the options.

C Binding

int **VISIONcompInfo**(VISION_COMPONENT component, char *name, char *option, void *value, int value_size);

(IN) component: specifies a handle as filled by VISIONcompOpen().

(IN) name: a string representing the name of a component

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(OUT) value: a pointer to application-allocated space which the method will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char name[32];
/* print the names of all available components */
status = VISIONcompInfo( myComponent, NULL, "firstName",
        name, 32 );
while( 0 == status ) {
  printf( "%s\n", name );
  status = VISIONcompInfo( myComponent, NULL, "nextName",
        name, 32 );
  }
```

C++ Binding

(in class VISIONcomponent):

int **Info**(char *name, char *option, void *value, int value_size);

(IN) name: a string representing the name of a component

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which the method will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char name[32];
// print the names of all available components
status = myComponent->Info( NULL, "firstName",
        name, 32 );
while( 0 == status ) {
  printf( "%s\n", name );
  status = myComponent->Info( NULL, "nextName",
        name, 32 );
  }
```

Java Binding

(in public class VISIONcomponent):

public int **Info**(String name, String option) throws VISIONexception;

public String **StringInfo**(String name, String option) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Info() should be used on options whose values are integral;
- StringInfo() should be used on options whose values are strings.

(IN) name: a string representing the name of a component

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
String name = null;
Boolean goAhead = true;
// print the names of all available components
try {
  name = myComponent.StringInfo( null, "firstName" );
  }
catch( VISIONexception error ) {
  goAhead = false;
  }
while( goAhead ) {
  printf( "%s\n", name );
  try {
    name = myComponent.StringInfo( null, "nextName" );
```

```
    }
  catch( VISIONexception error ) {
    goAhead = false;
    }
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONcompInfo**(ByVal component As Long, ByVal name As String, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) component: specifies a handle as filled by VISIONcompOpen().

(IN) name: a string representing the name of a component

(IN) option:

a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(OUT) value: a reference to application-allocated space which the method will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The method will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
String name
REM print the names of all available components
status& = VISIONcompInfo( myComponent, ByVal "",
        ByVal "firstName", name, 255 )
While status = 0
  print name
  status& = VISIONcompInfo( myComponent, ByVal "",
        ByVal "nextName", name, 255 )
End While
```

3.6. Set component-related configuration

This method allows an application to control the behavior of the implementation by setting new values for "options". The application invokes it, specifying a component, an option and a new value for the option. For object-oriented languages, an implementation may also provide this functionality in the form of modifiers with the same names as the options.

C Binding

int **VISIONcompConfig**(VISION_COMPONENT component, char *name, char *option, void *value, int value_size);

(IN) component: specifies a handle as filled by VISIONcompOpen().

(IN) name: a string which represents the name of a component

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(IN) value: a pointer to the value to be set for the specified option

(IN) value_size: the number of bytes used by "value"

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char ourPath[128];
/* set component path for dynamic loading */
strcpy( ourPath, "/opt/fision/current/SunOS-5.4/sun4u" );
status = VISIONcompConfig( myComponent, NULL, "path",
        ourPath, strlen(path) );
```

C++ Binding

(in class VISIONcomponent):

int **Config**(char *name, char *option, void *value, int value_size);

(IN) name: a string which represents the name of a component

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(IN) value: a pointer to the value to be set for the specified option

(IN) value_size: the number of bytes used by "value"

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char ourPath[128];
// set component path for dynamic loading
strcpy( ourPath, "/opt/fision/current/SunOS-5.4/sun4u" );
status = myComponent->Config( NULL, "path",
        ourPath, strlen(path) );
```

Java Binding

(in public class VISIONcomponent):

public void **Config**(String name, String option, int value) throws VISIONexception;

public void **StringConfig**(String name, String option, String value) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Config() should be used on options whose values are integral;
- StringConfig() should be used on options whose values are strings.

(IN) name: a string which represents the name of a component

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(IN) value: the value to be set for the specified option

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
String ourPath = new String(
        "/opt/fision/current/SunOS-5.4/sun4u" );
// set component path for dynamic loading
try {
  myComponent->StringConfig( null, "path", ourPath );
  }
catch( VISIONexception error ) {
  System.err.println( "Error setting path!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONcompConfig**(ByVal component As Long, ByVal name As String, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) component: specifies a handle as filled by VISIONcompOpen().

(IN) name: a string which represents the name of a component

(IN) option: a string which represents the option to configure. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Component Options Table (page 46) lists the options defined by this revision of the API.

(IN) value: a reference to the value to be set for the specified option

(IN) value_size: the number of bytes used by ”value”

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
String ourPath
REM set component path for dynamic loading
ourPath = "/opt/fision/current/SunOS-5.4/sun4u"
status = VISIONcompConfig( myComponent, ByVal "",
        ByVal "path", ourPath, length(ourPath) )
```

3.7. Terminate a ”component management” object

This method shall attempt to terminate the specified object - freeing any resources allocated (by the implementation) on behalf of the application. NOTE: in some environments, if the application terminates without invoking this method for each created object, it may not be possible for the implementation to free the associated resources.

C Binding

int **VISIONcompClose**(VISION_COMPONENT component);

(IN) component: specifies a handle as filled by VISIONcompOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = VISIONcompClose( myComponent );
```

C++ Binding

(in class VISIONcomponent):
~VISIONcomponent();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = myComponent->Close();
```

Java Binding

(garbage collected)

Microsoft Visual BASIC Binding

Public Declare Function **VISIONcompClose**(ByVal component As Long) As Long

(IN) component: specifies a handle as filled by VISIONcompOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status& = VISIONcompClose( myComponent )
```

3.9. Component Options Table

TABLE 3.1: Component Options

Option Name	Data Type	Description
path	STRING	The directory in which to look for component object modules (used for dynamic loading).
load (W)	(value ignored)	Loads the specified component.
unload (W)	(value ignored)	Unloads the specified component.
restart (W)	(value ignored)	Restarts (reinitializes) the specified component.
errorReporter	FUNCTION	The function to invoke when component errors occur.
firstName (R)	STRING	The name of the first available component.
nextName (R)	STRING	The name of the next available component, if any.
firstParam (R)	STRING	The name of the first parameter the component requires.
nextParam (R)	STRING	The name of the next parameter the component requires, if any.
firstParamValues (R)	STRING	A string representing a range of allowed values for the component's first parameter.
nextParamValues (R)	STRING	A string representing a range of allowed values for the component's next parameter, if any.

(R) = read-only / (W) = write-only

Component Options Table Notes

These notes are intended to clarify or give further information about the items in the table:

- **path**:

  The "name" argument is ignored when setting this option.

- **firstName/nextName**:

  The "name" argument is ignored when getting this option. Applications should use "firstName" to obtain the name of the first available component. Subsequent "nextName" requests will return subsequent names of available components until there are no more, in which case a NULL string will be returned. This may be useful in, for example, an interactive application which is meant to allow the use of any available component.

- **firstParam/nextParam**:

- **firstParamValues/nextParamValues**:

  These options allow an application to be completely independent of any component. Such an application can query the user (or a configuration file) for which component to use, and then query that component using these options. The application can use this information to build the correct form of the "name" argument to the "open" method (page 10) . Traversing the "firstParam" and "nextParam" options will return names of all parameters accepted by a component. The names need to be self-explanatory so that a user can enter appropriate values when needed. For example, an application prompts the user for the component name, and he/she enters "vme". The application begins building the string with "vme:". Now, firstParam will return "addrtype". This way, the application will be able to prompt: "addrtype?" completely transparently. Traversing the "firstParamValues" and "nextParamValues" options will return strings representing the allowed values for each parameter. Numeric ranges are to be denoted in this form: "{lower}..{upper}", where {lower} is the lower bound, and {upper} is the upper bound. ".." is used instead of "-" because some string-parsing libraries can interpret the "-" as a minus sign and perform automatic subtraction. For consistency, allowed string values should also be separated by "..". If the user enters ”mem”, the application can append ”/type=mem” to the string. Then, the application uses nextParam to obtain ”number”, and so on. When nextParam has reached the end, it will return a NULL string. Thus, a few lines of code to implement this loop can result in an application which can dynamically handle any hardware environment.

Recommendation 3.1:

The ”firstParam/nextParam” and ”firstParamValues/nextParamValues” options provide a powerful enabler for user-friendly applications. Without them, users will probably be forced to maintain a ”cheat sheet” for their most-used components, and learning to use a new component could be troublesome for them. Therefore, implementation of these features is strongly recommended.

4. Master Access API

The intention of the Master Access API is to allow access to functionality related to the VMEbus master, or of ”general” usefulness. Examples include:

- configuration of bus timers;
- slave map configuration; and
- the ”best” protocol supported by the hardware.

Rule 4.1:

An implementation of the Master Access API for C, C++, Java and/or Microsoft Visual BASIC shall provide the precise class/method names, argument order, argument data types, and return data types given in the sections in this chapter. In other words, the remaining sections of this chapter form rules for implementations wishing to provide bindings to these languages.

Observation 4.1:

In C, C++, Java and Microsoft Visual BASIC, any changes in class/method names, argument order, argument data types, or return data types will render user code non-portable.

Permission 4.1:

Implementations for programming languages not mentioned above may choose other calling conventions.

Recommendation 4.1:

Implementations for programming languages not mentioned above should attempt to adhere to the basic spirit of the calling conventions for the language most similar to it.

4.1. Create new VISION master object

This method shall attempt to create a new ”VISION master object”. If successfully created, this object shall provide access to features of the bus master until the object is destroyed. If a remote server is specified and the implementation supports client/server, the object shall provide access to the bus master on the server.

C Binding

int **VISIONmasterOpen**(VISION_MASTER *master, char *server);

(OUT) master: a pointer to an application-allocated variable of type ”VISION_MASTER”. This type is defined by the inclusion of *VISION.h* (see ”Compile-time Conventions” (page 85)). Upon success, the implementation shall fill the variable with a ”handle” for the application's subsequent use with the other methods in this API.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the ”real” object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_MASTER myMaster;
/* we want to control the master on "bears" */
status = VISIONmasterOpen( &myMaster, "bears" );
```

C++ Binding

(in class VISIONmaster):
VISIONmaster(int *status, char *server);

(OUT) status: status is returned in this way due to the semantics of object construction.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the ”real” object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall fill *status with 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONmaster *myMaster;
int status;
// we want to control the master on "bears"
myMaster = new VISIONmaster( &status, "bears" );
```

Java Binding

(in public class VISIONmaster):

public **VISIONmaster**(String server) throws VISIONexception;

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
VISIONmaster myMaster = null;
// we want to control the master on "bears"
try {
  myMaster = new VISIONmaster( "bears" );
} catch( VISIONexception e ) {
System.err.println( "Error creating VISIONmaster!" );
System.exit(-1);
}
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONmasterOpen**(ByRef master As Long, ByVal server As String) As Long

(OUT) master: a reference to an application-declared variable of type "Long". Upon success, the implementation shall fill the variable with a "handle" for the application's subsequent use with the other methods in this API.

(IN) server: if the implementation supports client/server, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long myMaster
REM we want to control the master on "bears"
status& = VISIONmasterOpen( ByRef myMaster, "bears" )
```

4.2. Get master-related information

This method shall attempt to return information related to the master. The application invokes it, specifying an ”option” which represents some type of information. The method shall return to the application the current value of the option. For object-oriented languages, an implementation may provide this functionality in the form of accessors with the same names as the options.

C Binding

int **VISIONmasterInfo**(VISION_MASTER master, char *option, void *value, int value_size);

(IN) master: specifies a handle as filled by VISIONmasterOpen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a pointer to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char proto_string[32];
/* we want to see the "best" protocol the master can do */
status = VISIONmasterInfo( myMaster, "bestProtocol",
        proto_string, 32 );
```

C++ Binding

(in class VISIONmaster):

int **Info**(char *option, void *value, int value_size);

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a pointer to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
char proto_string[32];
// we want to see the "best" protocol the master can do
status = myMaster->Info( "bestProtocol",
        proto_string, 32 );
```

Java Binding

(in public class VISIONmaster):

public int **Info**(String option) throws VISIONexception;

public String **StringInfo**(String option) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Info() should be used on options whose values are integral;
- StringInfo() should be used on options whose values are strings.

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

Execution Status The methods shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
String best_protocol = null;
// find out the best protocol the master can do
try {
  best_protocol = myMaster.StringInfo( "bestProtocol" );
  }
catch( VISIONexception error ) {
  System.err.println( "Couldn't get bestProtocol!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONmasterInfo**(ByVal master As Long, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) master: specifies a handle as filled by VISIONmasterOpen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a reference to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
String best_protocol
REM find out the best protocol the master can do
status& = VISIONmasterInfo( myMaster,
        ByVal "bestProtocol", best_protocol, 255 )
```

4.3. Set master-related configuration

This method allows an application to control the behavior of the implementation by setting new values for ”options”. The application invokes it, specifying an option and a new value for the option. For object-oriented languages, an implementation may also provide this functionality in the form of modifiers with the same names as the options.

C Binding

int **VISIONmasterConfig**(VISION_MASTER master, char *option, void *value, int value_size);

(IN) master: specifies a handle as filled by VISIONmasterOpen().

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
int level;
...
/* set our bus request level to 2 */
level = 2;
status = VISIONmasterConfig( myMaster, "requestLevel",
        (void *)&level, sizeof(level) );
```

C++ Binding

(in class VISIONmaster):

int **Config**(char *option, void *value, int value_size);

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// set our bus request level to 2
int level = 2;
status = myMaster->Config( "requestLevel",
        (void *)&level, sizeof(level) );
```

Java Binding

(in public class VISIONmaster):

public int **Config**(String option, int value) throws VISIONexception;

public String **StringConfig**(String option, String value) throws VISIONexception;

Since Java does not support true polymorphism, there are two methods:

- Config() should be used on options whose values are integral;
- StringConfig() should be used on options whose values are strings.

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: the new value for the option.

Execution Status The methods shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// set our bus request level to 2
try {
  myMaster.Config( "requestLevel", 2 );
  }
catch( VISIONexception error ) {
  System.err.println( "Couldn't set requestLevel!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONmasterConfig**(ByVal master As Long, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) master: specifies a handle as filled by VISIONmasterOpen().

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by "value".

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
REM set our bus request level to 2
level& = 2
status& = VISIONmasterConfig( myMaster,
        ByVal "requestLevel", ByRef level, 4 )
```

4.4. Terminate VISION master object

This method shall attempt to terminate the specified object - freeing any resources allocated (by the implementation) on behalf of the application. NOTE: in some environments, if the application terminates without invoking this method for each created object, it may not be possible for the implementation to free the associated resources.

C Binding

int **VISIONmasterClose**(VISION_MASTER master);

(IN) master: specifies a handle as filled by VISIONmasterOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = VISIONmasterClose( myMaster );
```

C++ Binding

(in class VISIONmaster):
~VISIONmaster();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
delete myMaster;
```

Java Binding

(garbage collected)

Microsoft Visual BASIC Binding

Public Declare Function **VISIONmasterClose**(ByVal master As Long) As Long

(IN) master: specifies a handle as filled by VISIONmasterOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status& = VISIONmasterClose( myMaster )
```

4.5. Master Options Table

TABLE 4.1: Master Options

Option Name	Data Type	Description
bestType (R)	STRING	The name of the ”best” (fastest) transfer type the master supports (using the same names as in the slave option ”transferType”).
apiRev (R)	INTEGER	The revision level of the API the implementation understands.
softwareName (R)	STRING	The name of the implementation.
softwareRev (R)	STRING	The revision level of the implementation.
hardwareName (R)	STRING	The name of the type of bus controller (chip) of the master.
hardwareRev (R)	STRING	The revision ID (if any) of the controller.
errorReporter	FUNCTION	The function to invoke when master errors occur.
arbiterMode	STRING	The arbiter mode (if an arbiter is implemented on the master): • ”PRI” (DEFAULT): priority arbitration (honors request levels) • ”RRS”: round-robin bus grants • ”SGL”: single-level arbitration (level 3 only)
requestMode	STRING	The request mode (if a requester is implemented on the master): • ”FAIR” (DEFAULT): request the bus if no other requests on same level • ”UNFAIR”: request the bus regardless of other same-level requests
releaseMode	STRING	The release mode (if a requester is implemented on the master): • ”RWD” (DEFAULT): Release-When-Done • ”ROR”: Release-On-Request (release only when another master requests bus)
requestLevel	INTEGER	The bus request level (0-3).
firstSlaveMap	STRING	The first (existing or potential) slave map.
nextSlaveMap	STRING	The next (if any) slave map.
interrupterLevel	INTEGER	The level (1-7) at which the master may generate a bus interrupt.
interrupterVector	INTEGER	The vector number (0-255) the master may use for a bus interrupt.
interruptBus (W)	(value ignored)	Generate a bus interrupt at the configured level using the configured vector number.
arbitrationTimer	INTEGER	The arbitration timeout in microseconds.
transactionTimer	INTEGER	The transaction timeout in microseconds.

(R) = read-only / (W) = write-only

Master Options Table Notes

The following notes are intended to clarify or provide further information about the items in the table:

- **requestMode**:

  ”FAIR” mode is an enhancement made to the VMEbus arbitration scheme in the early days to make it more deterministic. In the old (or ”UNFAIR”) mode, a master may request the bus even if another request on the same level is incoming on the daisy chain. "FAIR" mode means to request only if no other requests of the same level are incoming.

- **firstSlaveMap/nextSlaveMap**:

  these may be used to traverse a list of (potential and/or existing) slave maps (that is, maps between local and VMEbus address spaces as seen by OTHER masters on the bus). When a potential map is unconfigured, it will appear as: "unconf:{...}", while a configured map will appear (and should be configured as) "vme:{...}". The "{...}" should be a specification of addressing properties as used by the "vme" component, with the following additions:

  - for unconfigured maps, parameter values will appear as possible ranges, as in: "addr=0..0xff00" (the use of ".." instead of "-" is to aid in parsing using standard C libraries);
  - when configuring a map, the application must add a local address, as in: "/local={address}".

- **arbitrationTimer/transactionTimer**:

  Although VME does not strictly specify the implementation of timers, the choices here are based on common practice. The "arbitrationTimer" value specifies how long to attempt to acquire the bus before timing out. The "transactionTimer" value specifies how long to wait for any other type of acknowledgement during data transfer. If hardware implements a different set of timers (perhaps more or less granularity of bus operations), an implementation should attempt a logical division or combination of the values requested.

5. List Management API

The intention of the List Management API is to allow management of ”lists” of I/O requests to be executed as a single operation. The underlying hardware may support this directly, or software may emulate it. An application uses this API by following these steps:

1. it ”marks” the start of a list using the ”begin” method;
2. makes several I/O requests via the Slave API (which will NOT result in actual I/O at that time);
3. marks the end of the list using the ”end” method;
4. executes the list using the ”exec” method;
5. verifies completion and status of the I/O operations in the list using the ”sync” method.

The last two steps may be repeated as many times as needed. Note that there is no explicit client/server support (i.e. no ”server” argument) in this API. This is because lists may cross servers; the implementation must use the information in the ”enclosed” slave objects to determine what to do.

Rule 5.1:

An implementation of the List Management API for C, C++, Java and/or Microsoft Visual BASIC shall provide the precise class/method names, argument order, argument data types, and return data types given in the sections in this chapter. In other words, the remaining sections of this chapter form rules for implementations wishing to provide bindings to these languages.

Observation 5.1:

In C, C++, Java and Microsoft Visual BASIC, any changes in class/method names, argument order, argument data types, or return data types will render user code non-portable.

Permission 5.1:

Implementations for programming languages not mentioned above may choose other calling conventions.

Recommendation 5.1:

Implementations for programming languages not mentioned above should attempt to adhere to the basic spirit of the calling conventions for the language most similar to it.

5.1. Create new list object

This method shall attempt to create a new list object. If successfully created, this object shall provide access to a mechanism for building and executing lists of I/O requests, until it is destroyed.

C Binding

int **VISIONlistOpen**(VISION_LIST *list);

(OUT) list: a pointer to a variable of type "VISION_LIST" which will be used as a "handle" to reference the list in subsequent calls to this API.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_LIST myList;
/* create a list */
status = VISIONlistOpen( &myList );
```

C++ Binding

(in class VISIONlist):
VISIONlist(int *status);

(OUT) status: status is returned in this way due to the semantics of object construction.

Execution Status The method shall fill *status with 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONlist *myList;
int status;
// create a new list
myList = new VISIONlist( &status );
```

Java Binding

(in public class VISIONlist):

public **VISIONlist**() throws VISIONexception;

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
VISIONlist myList = null;
// create a new list
try {
  myList = new VISIONlist();
} catch( VISIONexception e ) {
System.err.println( "Error creating VISIONlist!" );
System.exit(-1);
}
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistOpen**(ByRef list As Long) As Long

(OUT) list: a reference to a variable of type "Long" which will be used as a "handle" to reference the list in subsequent calls to this API.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long myList
REM create a new list
status& = VISIONlistOpen( ByRef myList )
```

5.2. Mark start of a list of I/O to be performed later

This method shall cause subsequent I/O requests (Slave API calls) to be queued into a ”list” rather than executed.

C Binding

int **VISIONlistBegin**(VISION_LIST list);

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
/* we want the following slave I/O in our list */
status = VISIONlistBegin( myList );
```

C++ Binding

(in class VISIONlist):

int **Begin**();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// we want the following slave I/O in our list
status = myList->Begin();
```

Java Binding

(in public class VISIONlist):

public void **Begin**() throws VISIONexception;

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// we want the following slave I/O in our list
try {
  myList.Begin();
  }
catch( VISIONexception error ) {
  System.err.println( "Error starting list!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistBegin**(ByVal list As Long) As Long

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
REM we want the following slave I/O in our list
status = VISIONlistBegin( myList )
```

5.3. Mark end of list

This method shall denote the end of a list, so that subsequent I/O requests (Slave API calls) are executed rather than stored in a list.

C Binding

int **VISIONlistEnd**(VISION_LIST list);

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
{Slave API calls}
/* end of list */
status = VISIONlistEnd( myList );
```

C++ Binding

(in class VISIONlist):

int **End**();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
{Slave API calls}
// end of list
status = myList->End();
```

Java Binding

(in public class VISIONlist):

public void **End**() throws VISIONexception;

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
{Slave API calls}
// end of list
try {
  myList.End();
  }
catch( VISIONexception error ) {
  System.err.println( "Error ending list!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistEnd**(ByVal list As Long) As Long

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
{Slave API calls}
REM end of list
status = VISIONlistEnd( myList )
```

5.4. Execute list of I/O

This method shall attempt to execute a list of I/O requests (Slave API calls), assuming a list was previously built using this API's "begin" and "end" methods to surround the I/O requests. This is a non-blocking call. That is, this method may return BEFORE the I/O is complete. Use the "sync" method (page 68) to ensure that the I/O is complete before issuing another I/O request. Provision shall be made to use the latest data in any referenced buffers for writes, and update any referenced buffers with read data. List execution shall terminate if ANY error is detected. Applications can determine where the list stopped by zeroing their "real_bytes" storage before invoking this method.

C Binding

int **VISIONlistExec**(VISION_LIST list);

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
/* execute our list */
status = VISIONlistExec( myList );
```

C++ Binding

(in class VISIONlist):

int **Exec**();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// execute our list
status = myList->Exec();
```

Java Binding

(in public class VISIONlist):

public void **Exec**() throws VISIONexception;

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// execute our list
try {
  myList.Exec();
  }
catch( VISIONexception error ) {
```

```
System.err.println( "Error executing list!" );
}
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistExec**(ByVal list As Long) As Long

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
REM execute our list
status = VISIONlistExec( myList )
```

5.5. Ensure completion of a list

This method shall block the caller until the specified list is complete (successfully or with error). An application must invoke this method to ensure completion of any outstanding list I/O, and obtain the overall execution status of a list.

C Binding

int **VISIONlistSync**(VISION_LIST list);

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
/* ensure our list is complete */
status = VISIONlistSync( myList );
```

C++ Binding

(in class VISIONlist):

int **Sync**();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// ensure our list is complete
status = myList->Sync();
```

Java Binding

(in public class VISIONlist):

public void **Sync**() throws VISIONexception;

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// ensure our list is complete
try {
  myList.Sync();
  }
catch( VISIONexception error ) {
  System.err.println( "Error syncing with list!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistSync**(ByVal list As Long) As Long

(IN) list: a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
REM ensure our list is complete
status = VISIONlistSync( myList )
```

5.6. Terminate a list object

This method shall attempt to terminate the specified object - freeing any resources allocated (by the implementation) on behalf of the application. NOTE: in some environments, if the application terminates without invoking this method for each created object, it may not be possible for the implementation to free the associated resources.

C Binding

int **VISIONlistClose**(VISION_LIST list);

(IN) list: specifies a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = VISIONlistClose( myList );
```

C++ Binding

(in class VISIONlist):
ṼISIONlist();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = myList->Close();
```

Java Binding

(garbage collected)

Microsoft Visual BASIC Binding

Public Declare Function **VISIONlistClose**(ByVal list As Long) As Long

(IN) list: specifies a handle as filled by VISIONlistOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status& = VISIONlistClose( myList )
```

6. Interrupt Access API

The intention of the Interrupt Access API is to allow applications to customize the master's response to bus interrupts (which are generated by slaves, from the master's point of view). When used in conjunction with an "intelligent" Slave API component (page 35) , applications can be written to be reasonably independent of factors like the number, locations, and types of interrupters in a system.

The basic idea is that an application creates an "interrupt object" by specifying a bus interrupt level and possibly a vector number (note that on CPU architectures without a hardware implementation of vectors, a software implementation must exist). Then, the application can wait or test for the arrival of the interrupt, specify a custom function to invoke upon its arrival, get information about, or change the configuration of the object.

Although this API supports client/server, it is unlikely a reasonable network implementation can be done with present technology. It exists in the API because some clever person will undoubtedly think of a way to do it someday.

Rule 6.1:

An implementation of the Interrupt Access API for C, C++, Java and/or Microsoft Visual BASIC shall provide the precise class/method names, argument order, argument data types, and return data types given in the sections in this chapter. In other words, the remaining sections of this chapter form rules for implementations wishing to provide bindings to these languages.

Observation 6.1:

In C, C++, Java and Microsoft Visual BASIC, any changes in class/method names, argument order, argument data types, or return data types will render user code non-portable.

Permission 6.1:

Implementations for programming languages not mentioned above may choose other calling conventions.

Recommendation 6.1:

Implementations for programming languages not mentioned above should attempt to adhere to the basic spirit of the calling conventions for the language most similar to it.

Observation 6.2:

The order of the arguments obeys a convention; the first argument is a software identification of the interrupt object (in object-oriented languages like C++, this argument is implicit). The next arguments follow in order from the most compulsory to the most optional. The arguments contain only large and/or varying numbers, such as addresses, lengths, and pointers. The choice of interrupt level and vector are specified at the time of creation of the interrupt object.

6.1. Create ”interrupt object”

This method shall attempt to create a new ”interrupt object” - a software construct to allow access to interrupts. The new object shall be initialized with default values for interrupt options (page 84) . Note that the default value for its ”state” is disabled, to give the application time to program the interrupt source.

C Binding

int **VISIONintOpen**(VISION_INTERRUPT *intr, int busLevel, int *vecNum, char *server);

(OUT) intr: a pointer to a ”handle” which the function will fill, for the application's future use (with the other functions in the API).

(IN) busLevel: specifies the bus interrupt level. Valid values for VMEbus are 1-7.

(IN/OUT) vecNum: If *vecNum is negative, a vector number will be assigned by the system and copied there upon successful return. It is the application's responsibility to program the appropriate register in the slave. If *vecNum is 0 or greater, the application is dictating the interrupt vector number associated with the desired interrupt. Application programmers must exercise caution when using it in the second way, because the implementation may not have a way to determine the legitimacy of the request. Some CPU architectures do not have hardware interrupt implementations; in this case, a software implementation must be provided (as is common practice).

(IN) server: if the implementation supports client/server interrupts, the specified server shall be contacted and the ”real” object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISION_INTERRUPT myInterrupt;
int myVec;
/* get vector number from system */
myVec = -1;
status = VISIONintOpen( &myInterrupt, 4, &myVec, "" );
```

C++ Binding

(in class VISIONinterrupt):
VISIONinterrupt(int *status, int busLevel, int *vecNum, char *server);

(OUT) status: status is returned in this way due to the semantics of object construction.

(IN) busLevel: specifies the bus interrupt level. Valid values for VMEbus are 1-7.

(IN/OUT) vecNum: If *vecNum is negative, a vector number will be assigned by the system and copied there upon successful return. It is the application's responsibility to program the appropriate register in the slave. If *vecNum is 0 or greater, the application is dictating the interrupt vector number associated with the desired interrupt. Application programmers must exercise caution when using it in the second way, because the implementation may not have a way to determine the legitimacy of the request. Some CPU architectures do not have hardware interrupt implementations; in this case, a software implementation must be provided (as is common practice).

(IN) server: if the implementation supports client/server interrupts, the specified server shall be contacted and the ”real” object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall fill *status with 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
#include "VISION.h"
VISIONinterrupt *myInterrupt;
// get vector number from system
int myVec = -1;
myInterrupt = new VISIONinterrupt( &status, 4, &myVec, "" );
```

Java Binding

(in public class VISIONinterrupt):

public **VISIONinterrupt**(int busLevel, int vecNum[], String server) throws VISIONexception;

(IN) busLevel: specifies the bus interrupt level. Valid values for VMEbus are 1-7.

(IN/OUT) vecNum: If vecNum[0] is negative, a vector number will be assigned by the system and copied there upon successful return. It is the application's responsibility to program the appropriate register in the slave. If vecNum[0] is 0 or greater, the application is dictating the interrupt vector number associated with the desired interrupt. Application programmers must exercise caution when using it in the second way, because the implementation may not have a way to determine the legitimacy of the request. Some CPU architectures do not have hardware interrupt implementations; in this case, a software implementation must be provided (as is common practice).

(IN) server: if the implementation supports client/server interrupts, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
VISIONinterrupt myInterrupt = null;
// get vector number from system
vecNum = new int[1];
vecNum[0] = -1;
try {
  myInterrupt = new VISIONinterrupt( 4, vecNum, "" );
} catch( VISIONexception e ) {
System.err.println( "Error creating VISIONinterrupt!" );
System.exit(-1);
}
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONintOpen**(ByRef intr As Long, ByVal server As String) As Long

(OUT) intr: a reference to a "handle" which the function will fill, for the application's future use (with the other functions in the API).

(IN) busLevel: specifies the bus interrupt level. Valid values for VMEbus are 1-7.

(IN/OUT) vecNum: If vecNum is negative, a vector number will be assigned by the system and copied there upon successful return. It is the application's responsibility to program the appropriate register in the slave. If vecNum is 0 or greater, the application is dictating the interrupt vector number associated with the desired interrupt. Application programmers must exercise caution when using it in the second way, because the implementation may not have a way to determine the legitimacy of the request. Some CPU architectures do not have hardware interrupt implementations; in this case, a software implementation must be provided (as is common practice).

(IN) server: if the implementation supports client/server interrupts, the specified server shall be contacted and the "real" object shall be created there, so that subsequent uses of the object will result in access to the server. To run locally, this argument must be either NULL, the null string, or the name of the local host.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long myInterrupt
Long myVec
REM get vector number from system
myVec = -1
status = VISIONintOpen( ByRef myInterrupt, 4,
        ByRef myVec, ByVal "" )
```

6.2. Block caller while awaiting interrupt

This method allows an application to ensure that an interrupt arrives within a specified time period. If the timeout is specified as 0 (zero), the method will check the status of the interrupt and return immediately to the caller, returning EINPROGRESS if the interrupt has not yet arrived. Otherwise, the caller will be blocked for the specified time period, or until the interrupt arrives. If the interrupt has not arrived when the timeout expires, the method will return the error ETIMEDOUT.

C Binding

int **VISIONIntWait**(VISION_INTERRUPT intr, unsigned int *count, unsigned int timeout);

(IN) intr: specifies a handle as filled by VISIONintOpen().

(OUT) count: a pointer to space allocated by the application to contain the number of times this interrupt has occurred since the last zeroing of its counter; if the reference is NULL, the implementation will not attempt to store this information.

(IN) timeout: specifies a timeout, in microseconds, after which the function will return.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
unsigned int count;
/* check arrival of interrupt within 10 ms */
status = VISIONintWait( myInterrupt, &count, 10 );
```

C++ Binding

(in class VISIONinterrupt):

int **Wait**(unsigned int *count, unsigned int timeout);

(OUT) count: a pointer to space allocated by the application to contain the number of times this interrupt has occurred since the last zeroing of its counter; if the reference is NULL, the implementation will not attempt to store this information.

(IN) timeout: specifies a timeout, in microseconds, after which the function will return.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// check arrival of interrupt within 10 ms
unsigned int count;
status = myInterrupt->Wait( &count, 10 );
```

Java Binding

(in public class VISIONinterrupt):

public int **Wait**(int timeout) throws VISIONexception;

(IN) timeout: specifies a timeout, in microseconds, after which the function will return.

Execution Status The method shall complete to indicate success (returning the current count of this interrupt), or throw a VISIONexception (page 96) to indicate failure.

Example

```
// check arrival of interrupt within 10 ms
int count;
try {
  count = myInterrupt.Wait( 10 );
  }
catch( VISIONexception e ) {
  System.err.println( "Error waiting for interrupt!" );
  System.exit(-1);
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONintWait**(ByVal interrupt As Long, ByRef count As Long, ByVal timeout As Long) As Long

(IN) intr: specifies a handle as returned by VISIONintOpen().

(OUT) count: a reference to space allocated by the application to contain the number of times this interrupt has occurred since the last zeroing of its counter; if the reference is NULL, the implementation will not attempt to store this information.

(IN) timeout: specifies a timeout, in microseconds, after which the function will return.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long count
REM check arrival of interrupt within 10 ms
status = VISIONintWait( myInterrupt, ByRef count, 10 )
```

6.3. Install application-specific interrupt handler

This method allows an application to install its own handler for the interrupt. The user-defined function shall execute at a normal user level, NOT at interrupt or supervisor level. On some platforms, this may be slower. If the user needs better performance, he/she will have to write platform-specific code anyway, and therefore should bypass the use of this API for this purpose.

C Binding

int **VISIONintAction**(VISION_INTERRUPT intr, void (*function)(void), void (**oldFunction)(void));

(IN) intr: a VISION_INTERRUPT, as filled by VISIONintOpen().

(IN) function: A pointer to the void-valued function to connect. The function should be declared as having no arguments.

(OUT) oldFunction: This allows the application to restore things to their previous state by reconnecting the previous function; *oldFunction will be filled with a pointer to the function that was previously installed as the interrupt handler.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
void myHandler(void);
.
.
void (*oldHandler)(void);
status = VISIONintAction( myInterrupt, myHandler,
        &oldHandler );
```

C++ Binding

(in class VISIONinterrupt):

int **Action**(void (*function)(void), void (**oldFunction)(void));

(IN) function: A pointer to the void-valued function to connect. The function should be declared as having no arguments.

(OUT) oldFunction: This allows the application to restore things to their previous state by reconnecting the previous function; *oldFunction will be filled with a pointer to the function that was previously installed as the interrupt handler.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
void myHandler(void);
.
.
void (*oldHandler)(void);
status = myInterrupt->Action( myHandler, &oldHandler );
```

Java Binding

A direct Java binding is not possible for this function.

Microsoft Visual BASIC Binding

A direct BASIC binding is not possible for this function.

6.4. Get interrupt-related information

This method will return information related to interrupts. The application invokes it, specifying an "option" which represents some type of information. The function returns to the application the current value of the option. For object-oriented languages, an implementation may provide this functionality in the form of accessors with the same names as the options.

C Binding

int **VISIONintInfo**(VISION_INTERRUPT intr, char *option, void *value, int value_size);

(IN) intr: specifies a VISION_INTERRUPT as filled by VISIONintOpen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: a pointer to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
int state;
/* check whether the interrupt is enabled */
status = VISIONintInfo( myInterrupt, "state",
         (void *)state, sizeof(int) );
```

C++ Binding

(in class VISIONinterrupt):

int **Info**(char *option, void *value, int value_size);

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: a pointer to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// check whether the interrupt is enabled
int state;
status = myInterrupt->Info( "state",
         (void *)state, sizeof(int) );
```

Java Binding

(in public class VISIONinterrupt):

public int **Info**(String option) throws VISIONexception;

public String **StringInfo**(String option) throws VISIONexception;

Execution Status The methods shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// check whether the interrupt is enabled
int state;
try {
  state = myInterrupt.Info( "state" );
  }
catch( VISIONexception error ) {
  System.err.println( "Couldn't get state of interrupt!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONintInfo**(ByVal intr As Long, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) intr: specifies a handle as filled by VISIONintOpen().

(IN) option: a string which represents the type of information requested. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which the function will fill with the current value of the option.

(IN) value_size: the number of bytes the application allocated to hold the value. The function will not write more than this many bytes into the value.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long state
REM check whether the interrupt is enabled
status = VISIONintInfo( myInterrupt, ByVal "state",
        ByRef state, 4 )
```

6.5. Set interrupt-related configuration

This method allows an application to configure an interrupt object. The application invokes it, specifying an ”option” and a new value for the option. For object-oriented languages, an implementation may also provide this functionality in the form of modifiers with the same names as the options.

C Binding

int **VISIONintConfig**(VISION_INTERRUPT intr, char *option, void *value, int value_size);

(IN) intr: specifies a VISION_INTERRUPT as filled by VISIONintOpen().

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
int state;
...
/* enable the interrupt (set its "state" to 1) */
state = 1;
status = VISIONintConfig( myInterrupt, "state",
        (void *)&state, sizeof(int) );
```

C++ Binding

(in class VISIONinterrupt):

int **Config**(char *option, void *value, int value_size);

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
// enable the interrupt (set its "state" to 1)
int state = 1;
status = myInterrupt->Config( "state",
        (void *)&state, sizeof(int) );
```

Java Binding

(in public class VISIONinterrupt):

public int **Config**(String option, int value) throws VISIONexception;

public String **StringConfig**(String option, String value) throws VISIONexception;

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Interrupt Options Table (page 84) lists the options defined by this revision of the API.

(OUT) value: the new value for the option.

Execution Status The methods shall complete to indicate success, or throw a VISIONexception (page 96) to indicate failure.

Example

```
// enable the interrupt (set its "state" to 1)
int state = 1;
try {
  myInterrupt.Config( "state", state );
  }
catch( VISIONexception error ) {
  System.err.println( "Couldn't enable interrupt!" );
  }
```

Microsoft Visual BASIC Binding

Public Declare Function **VISIONintConfig**(ByVal intr As Long, ByVal optname As String, value As Any, ByVal valSize As Long) As Long

(IN) intr: specifies a handle as filled by VISIONintOpen().

(IN) option: a string which represents the option to set. All options whose names begin with a lowercase letter are reserved for the API; components adding custom options should begin their names with an uppercase letter or other symbol. The Master Options Table (page 57) lists the options defined by this revision of the API.

(OUT) value: a reference (pointer) to application-allocated space which contains the new value for the option.

(IN) value_size: the number of bytes used by ”value”.

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
Long state
...
REM enable the interrupt (set its "state" to 1)
state = 1
status = VISIONintConfig( myInterrupt, "state",
        (void *)&state, sizeof(int) )
```

6.6. Terminate "interrupt object"

This method shall attempt to terminate the specified object - freeing any resources allocated (by the implementation) on behalf of the application. NOTE: in some environments, if the application terminates without invoking this method for each created object, it may not be possible for the implementation to free the associated resources.

C Binding

int **VISIONintClose**(VISION_INTERRUPT intr);

(IN) intr: specifies a VISION_INTERRUPT as filled by VISIONintOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = VISIONintClose( myInterrupt );
```

C++ Binding

(in class VISIONinterrupt):
~VISIONinterrupt();

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status = myInterrupt->Close();
```

Java Binding

(garbage collected)

Microsoft Visual BASIC Binding

Public Declare Function **VISIONintClose**(ByVal intr As Long) As Long

(IN) intr: specifies a handle as filled by VISIONintOpen().

Execution Status The method shall return 0 (zero) to indicate successful execution, or a non-zero error code from the error code list (page 95) to indicate failure.

Example

```
status& = VISIONintClose( myInterrupt )
```

6.7. Interrupt Options Table

TABLE 6.1: Interrupt Options

Option Name	**Data Type**	**Description**
state	INTEGER	The ”state” of the interrupt: • 0 (DEFAULT): disabled (from the CPU’s viewpoint) • 1: enabled (from the CPU’s viewpoint)
counter	INTEGER	A count of the number of times the interrupt has arrived.
errorReporter	FUNCTION	The function to invoke when interrupt errors occur.
busLevel (R)	INTEGER	The bus interrupt level specified when the object was created.
vecNum (R)	INTEGER	The vector number associated with the interrupt object.

(R) = read-only / (W) = write-only

7. Compile-time Conventions

Rule 7.1:

Implementations providing bindings for C and/or C++ shall provide the file *VISION.h*, which users must include. This file shall provide, either directly or by inclusion of other files, all definitions necessary to compile user applications under ANSI C and/or C++. For example, it must define VISION_SLAVE for C users, and/or the class VISIONslave for C++ users.

Permission 7.1:

The file *VISION.h* may also include support for non-ANSI compilers.

Rule 7.2:

Implementations providing Java bindings must provide a package for each implemented API, and one for the exceptions. The packages should have suitable prefixes (like the vendor's internet domain name), with the basenames of: slave, master, component, list, interrupt, and exception.

Rule 7.3:

Microsoft Visual BASIC implementations shall provide the file *VISION.BAS*, which users must include. This file shall provide all definitions necessary to run user applications under Microsoft Visual BASIC.

8. Link-time Conventions

Recommendation 8.1:

A server library should be called libVISIONserver.a (and/or libVISIONserver.so) under UNIX-like systems, or VISNxxxx.LIB (and/or VISNxxxx.DLL) under Microsoft Windows, where xxxx somehow designates the bus hardware name.

Recommendation 8.2:

A client-only library should be called libVISIONclient.a (and/or libVISIONclient.so) under UNIX-like systems, or VISIONCL.LIB (and/or VISIONCL.DLL) under Microsoft Windows.

Recommendation 8.3:

A single library providing both server and client functions should be called libVISION.a (and/or libVISION.so) under UNIX-like systems, or VISION.LIB (and/or VISION.DLL) under Microsoft Windows.

9. The VISION Network Protocol

Although this specification does not mandate it, it is desirable that VISION implementations with networking use a common protocol so that they may interoperate. An efficient, robust, field-proven protocol for the TCP/IP transport has been developed as part of the FISION package, and it is specified here. It is based on the ISO **Basic Encoding Rules** (ISO 8825), which defines a machine-independent mechanism for data transport over a network. The BER is widely used today in applications such as MIME (Multimedia Internet Mail Extensions) and SNMP (Simple Network Management Protocol). Perhaps most importantly, it provides a robust data ”packaging” scheme which ensures compatibility by providing reliable data type and size information. The following describes the rules which a host should follow to conform to this protocol.

Requests are issued by a client and processed by a server, irrespective of whether the request is a read or a write. The term ”message” is used herein to denote a stream of data, regardless of how many packets the transport layer segments it into. At the socket layer, a message is created by a call to send() or write(). Some connection-based semantics are implicit in the protocol, so minor modifications would be necessary to map it to connectionless semantics (and presumably a connectionless transport like UDP/IP).

Recommendation 9.1:

To ensure compatibility, the Internet suite of protocols should be used for communication. In particular, the TCP/IP transport should be used.

Recommendation 9.2:

To ensure compatibility, the Basic Encoding Rules (ISO 8825) should be used for data encapsulation.

Recommendation 9.3:

To ensure compatibility, a connection-based scheme should be used. This implies the following typical sequence of events:

1. a client requests creation of a new object on a server using a well-known communication channel (”port” in Internet protocol - see Rec. 9.10);
2. the server establishes a ”session” for the client on a new communication channel, whose sole purpose is servicing this object;
3. the client uses the new channel for all subsequent requests on the object;
4. the client eventually requests termination of the object;
5. the server terminates the object and its ”session”, and both sides shut down the connection.

This scheme minimizes overheads when performing the ”important” operations (like data transfer). A connectionless (or ”stateless”) scheme imposes the same (potentially large) overheads on all operations.

Recommendation 9.4:

To ensure compatibility, the following messages should be sent by clients on the well-known port when requesting new objects on servers:

TABLE 9.1: Object Creation Protocol

To create a:	Send the BER type(s):	With the value(s):
slave	INTEGER	1
..	CharacterString	”name” argument
component	INTEGER	2
master	INTEGER	3
interrupt	INTEGER	4
..	INTEGER	interrupt level
..	INTEGER	interrupt vector
list	INTEGER	5

A server should respond to these messages with a BER SEQUENCE message consisting of an INTEGER return status (0 indicating success), and upon failure, a CharacterString message explaining the error. For interrupt objects, the vector number should also be returned.

Recommendation 9.5:

To ensure compatibility, the message formats shown in the tables below should be used for operations on objects. The messages are formatted as BER SEQUENCEs, with the following exceptions:

1. On the slave write request (from client) or read reply (from server), if the ”data” portion would cause the SEQUENCE to be too large to fit into the recommended 512-byte buffer, it is sent in a separate, subsequent OCTET STRING, NOT part of any SEQUENCE.

2. On the list sync reply (from server), the message is divided into an arbitrary number of SEQUENCEs, making a best attempt to keep each one within the recommended 512-byte buffer.

3. A server’s response to a dump request is simply a CharacterString.

The reason for this 512-byte ”buffering” is as per the IETF recommendation to ensure compatibility with older networking hardware.

TABLE 9.2: Slave Protocol

Object type	Function	Sent by	SEQUENCE OF:	Value
slave	close	client	INTEGER	0x11
slave	close	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
slave	read	client	INTEGER	0x12
..	..	..	INTEGER	”offset” argument
..	..	..	INTEGER	”req_bytes” argument
slave	read	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	INTEGER	Number of bytes read
..	..	..	OCTET STRING	Data read (see Rec. 9.5)
slave	write	client	INTEGER	0x13
..	..	..	INTEGER	”offset” argument
..	..	..	INTEGER	”req_bytes” argument
..	..	..	OCTET STRING	Data to write (see Rec. 9.5)
slave	write	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	INTEGER	Number of bytes written
slave	sync	client	INTEGER	0x14
..	..	..	INTEGER	”timeout” argument
slave	sync	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
slave	info	client	INTEGER	0x15
..	..	..	CharacterString	Option name
..	..	..	INTEGER	”value_size” argument
slave	info	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	OCTET STRING	Current value
slave	config	client	INTEGER	0x16
..	..	..	CharacterString	Option name
..	..	..	OCTET STRING	New value
slave	config	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)

TABLE 9.3: Component Protocol

Object type	Function	Sent by	SEQUENCE OF:	Value
component	close	client	INTEGER	0x21
component	close	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
component	info	client	INTEGER	0x22
..	..	..	CharacterString	Component name
..	..	..	CharacterString	Option name
..	..	..	INTEGER	”value_size” argument
component	info	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	OCTET STRING	Current value
component	config	client	INTEGER	0x23
..	..	..	CharacterString	Component name
..	..	..	CharacterString	Option name
..	..	..	OCTET STRING	New value
component	config	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)

TABLE 9.4: Master Protocol

Object type	Function	Sent by	SEQUENCE OF:	Value
master	close	client	INTEGER	0x31
master	close	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
master	info	client	INTEGER	0x32
..	..	..	CharacterString	Option name
..	..	..	INTEGER	”value_size” argument
master	info	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	OCTET STRING	Current value
master	config	client	INTEGER	0x33
..	..	..	CharacterString	Option name
..	..	..	OCTET STRING	New value
master	config	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)

TABLE 9.5: Interrupt Protocol

Object type	Function	Sent by	SEQUENCE OF:	Value
interrupt	close	client	INTEGER	0x41
interrupt	close	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
interrupt	wait	client	INTEGER	0x42
..	..	..	INTEGER	”timeout” argument
interrupt	wait	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	INTEGER	Count of interrupts
interrupt	action	client	INTEGER	0x43
interrupt	action	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
interrupt	info	client	INTEGER	0x44
..	..	..	CharacterString	Option name
..	..	..	INTEGER	”value_size” argument
interrupt	info	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
..	..	..	OCTET STRING	Current value
interrupt	config	client	INTEGER	0x45
..	..	..	CharacterString	Option name
..	..	..	OCTET STRING	New value
interrupt	config	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)

TABLE 9.6: List Protocol

Object type	Function	Sent by	SEQUENCE OF:	Value
list	close	client	INTEGER	0x51
list	close	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
list	begin	client	INTEGER	0x52
list	begin	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
list	end	client	INTEGER	0x53
..	..	..	(see Rec. 9.6)	..
..	..	..	(1st slave request)	..
..	..	..	..	..
..	..	..	(Nth slave request)	..
list	end	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
list	exec	client	INTEGER	0x54
..	..	..	OCTET STRING	1st data (if write or config)
..	..	..	(See Rec. 9.5, 9.7)	..
..	..	..	OCTET STRING	Nth data (if write or config)
list	exec	server	INTEGER	Status code
..	..	..	CharacterString	Error message (if error)
list	sync	client	INTEGER	0x55
list	sync	server	(see Rec. 9.5, 9.8)	..
..	..	..	INTEGER	Status code
..	..	..	INTEGER	1st real_bytes
..	..	..	OCTET STRING	1st data (if read or info)
..	..	..	..	..
..	..	..	INTEGER	Nth real_bytes
..	..	..	OCTET STRING	Nth data (if read or info)

Recommendation 9.6:

To ensure compatibility, the following sequence of events should be implemented for list downloading, at list "end" time:

1. If a given server has not yet been contacted, a list should be opened on it using the list open message, and the list begun using the list begin message.
2. On the communication channel associated with the server list, all the stored operations for this server should be sent, using the exact message formats that would be used for sending individual slave requests, with one exception: the server must return the slave handle (upon success) as the last element in the reply message. It is not required that the client make use of this.
3. After the list is downloaded to a given server, the client should send it a list end message.

Recommendation 9.7:

To ensure compatibility, the following sequence of events should be implemented at list "exec" time:

1. Each server in the list must be sent the "exec" message, which should include an ordered list of data for all writes and configs on the server.

Observation 9.1:

The above implies that a client must build a list internally from list "begin" time, keeping at least the following information for each list element:

- the associated server
- operation (write or read, recommend also info and config)
- offset (for write or read)
- req_bytes (or value_size for info or config)
- a place to store real_bytes (for write or read)
- data (write or config) or a place to store data (read or info)

Recommendation 9.8:

To ensure compatibility, servers should implement the following sequence of messages upon successful completion of a list "sync":

1. The server should first send the overall list status.
2. For each element in the list, the server should send the number of bytes transferred and, if the operation was a read or info, the data. It should break this data into messages (SEQUENCEs) at appropriate intervals.

Recommendation 9.9:

To ensure compatibility, numeric data should be sent over the network in Internet's "network byte order" ("Big Endian"). Appropriate conversions should be done at the client and/or server side. Most internet implementations provide the functions htons(), htonl(), ntohs(), and ntohl() for this purpose.

Recommendation 9.10:

VISION hosts adhering to the above recommendations should use TCP/IP port number 50019, since it is officially registered for this protocol.

Rule 9.1:

VISION hosts not adhering to the above recommendations shall NOT use TCP/IP port number 50019.

10. Error Code Summary

Rule 10.1:

Conforming implementations supporting C, C++, or BASIC must use only the error codes listed below. They are listed by their symbolic name and their integer decimal value, and their rough interpretation.

Error Codes

Symbolic Name	Decimal Value	Meaning
ECOMM	85	An error occurred communicating with the server.
ENOMEM	12	Memory allocation failed.
EINVAL	22	The value of one or more arguments is illegal.
EIO	5	An I/O error (bus error, DMA controller error, etc.) occurred.
ENOENT	2	There is no entry (in an internal list) corresponding to the application's request.
ETIMEDOUT	145	The specified timeout expired and the operation was still not complete.
EINPROGRESS	36	The operation is not complete, but a timeout of zero was specified, denoting that the application simply wanted to check the status of the operation.
E2BIG	7	A value to be returned to the application was too large for the application's buffer, and so was truncated.
EPERM	1	Permission to use a resource is denied.

11. Java Exception Summary

Rule 11.1:

Conforming implementations supporting Java must use only the exceptions listed below, which must all inherit from VISIONexception.

Java Exceptions

Exception	Meaning
VISIONnetworkException	An error occurred communicating with the server.
VISIONoutOfMemoryException	Memory allocation failed.
VISIONillegalArgumentException	The value of one or more arguments is illegal.
VISIONbusException	An I/O error (bus error, DMA controller error, etc.) occurred.
VISIONentryNotFoundException	There is no entry (in an internal list) corresponding to the application's request.
VISIONtimeoutException	The specified timeout expired and the operation was still not complete.
VISIONarrayTooSmallException	A value to be returned to the application was too large for the application's buffer, and so was truncated.
VISIONpermissionException	Permission to use a resource is denied.

Annex A: The ”vme” Component

The ”vme” component is designed to support standard VMEbus addressing.

Parameters

Table A.1 shows the parameters a ”vme” component shall accept. They can specify an AM code directly or indirectly via ”friendly” terms, a VMEbus address range, and which bus to use (in case the hardware has more than one bus interface).

TABLE A.1: ”vme” Parameters

Name	Mandatory?	Values	Description
addrtype	Yes, unless using ”am” parameter	”short”, ”std”, ”ext”, ”long” or ”A40”	With ”priv” and ”space”, defines AM code
priv	No (defaults to ”n”)	anything starting with ”s”, ”S”, ”n”, ”N”, ”u”, or ”U”	With ”addrtype” and ”space”, defines AM code (”s” or ”S” for supervisory, ”n”, ”N”, ”u” or ”U” for user mode)
space	No (defaults to ”d”)	anything starting with ”p”, ”P”, ”d”, ”D”, or ”blt” or ”blt64”	With ”addrtype” and ”priv”, defines AM code (”p” or ”P” for single-word program space, ”d” or ”D” for single-word data space, ”blt” for BLock Transfer, ”blt64” for BLT64).
am	Yes, unless using ”addrtype”	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	the AM (Address Modifier) code for use on the VMEbus
base	No (defaults to 0)	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	Starting address on the VMEbus (offsets are relative to this)
size	No	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	Size (in bytes) of the address range; allows PIO hardware to be preprogrammed for faster access, also allows range-checking
bus	No (defaults to 0)	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	Denotes which VMEbus interface to use, starting at 0

Examples

Table A.2 shows example uses of the ”vme” component and their interpretations.

TABLE A.2: ”vme” Examples

Name	Description
vme:addrtype=ext/space=blt/base=0x20000	A32 user-mode BLT access starting at 0x20000
vme:am=9/base=0x80000000/size=0x100000	A32 user-mode data space access starting at 0x80000000 for 1MB
vme:/addrtype=short/priv=s/base=0x8000	A16 supervisory access starting at 0x8000
vme:/addrtype=std/space=blt/bus=1	A24 user-mode BLT starting at 0, on the second VMEbus interface

Annex B: The ”vmePnP” Component

The ”vmePnP” component is designed to allow access to slaves which support the ”Address Space Relocation” features as defined in VME64x (VITA 1.1-199x) using a geographically-addressed CR/CSR space.

Parameters

Table B.1 shows the parameters a ”vmePnP” component shall accept. They can specify the slot number, function number, desired AM code, and which bus to use (in case the hardware has more than one bus interface). This allows software to effect the programming of a function’s AM code and address range, or take advantage of what has already been programmed.

TABLE B.1: ”vmePnP” Parameters

Name	Mandatory?	Values	Description
slot	Yes	An integer from 1 to 31	Specifies a slot number
function	No (defaults to 0)	An integer from 0 to 7	Specifies the function within the module
am	No (may already be set up)	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	The AM (Address Modifier) code for use on the VMEbus
bus	No (defaults to 0)	Any integer in decimal, hex (use leading 0x), or octal (use leading 0)	Denotes which VMEbus interface to use, starting at 0

Examples

Table B.2 shows example uses of the ”vmePnP” component and their interpretations.

TABLE B.2: ”vmePnP” Examples

Name	Description
vmePnP:slot=4/am=9	A32 user-mode data space access to the first function in slot 4
vmePnP:slot=10/function=1/am=0xb	AM code 0xB (A32 user-mode BLT) access to the second function in slot 10
vmePnP:/slot=6	access to the first function in slot 6, using whatever AM code is already programmed there (will fail if nothing programmed)
vmePnP:/slot=2/function=2/bus=1	access to the third function in slot 2 on the second VMEbus interface, using whatever AM code is already programmed there (will fail if nothing programmed)

Annex C: The ”pmc” Component

The pmc component is designed to support the PMC (PCI Mezzanine Card) standard for daughter modules.

Parameters

Table C.1 shows the parameters a ”pmc” component shall accept. They can specify which PMC, which PCI space (I/O, MEM, or CONFIG), and which space number (since a device can provide more than one).

TABLE C.1: ”pmc” Parameters

Name	Mandatory?	Values	Description
position	No (defaults to ”a”)	”a”, ”b”, ”c” or ”d”	Specifies which PMC
space	Yes	”io”, ”mem”, or ”config” (”config” is intended only for ”advanced” uses)	Specifies the PCI address space
spaceNum	No (defaults to 0)	An integer from 0 to 5	Specifies which of the {space} spaces to access, since a device may provide multiple I/O and/or memory spaces

Examples

Table C.2 shows example uses of the ”pmc” component and their interpretations.

TABLE C.2: ”pmc” Examples

Name	Description
pmc:/space=mem/spaceNum=1	Access the second memory space in the first PMC
pmc:/position=b/space=io	Access the first I/O space in the second (”b” position) PMC
pmc:position=d/space=config	Access the configuration space of the fourth PMC

Annex D: The ”ip” Component

The ip component is designed to support the IndustryPack standard for daughter modules (see ”IndustryPack Logic Interface Specification, Revision 1.0”; GreenSpring Computers, Inc., 1204 O'Brien Drive, Menlo Park, CA 94025).

Parameters

Table D.1 shows the parameters an ”ip” component shall accept. They can specify which IP module, which space (ID, I/O, or MEM), and a starting address.

TABLE D.1: ”ip” Parameters

Name	**Mandatory?**	**Values**	**Description**
position	No (defaults to ”a”)	”a”, ”b”, ”ab”, ”c”, ”d”, or ”cd”	Specifies which IndustryPack position
space	Yes	”id”, ”io”, or ”mem”	Specifies the IP address space

Examples

Table D.2 shows example uses of the ”ip” component and their interpretations.

TABLE D.2: ”ip” Examples

Name	**Description**
ip:/space=mem	Access memory space in the first IndustryPack
ip:/position=b/space=io	Access I/O space in the second IP module
ip:position=cd/space=id	Access ID space in the ”CD” position (double-width) IP module