ATLAS Internal Note

ATL-D-ES-0004

**Version 1.8**

18 May 2011

VMEbus Application Program Interface

 Authors : R. Spiwoks, M. Joos, C. Parkman, J. Petersen

* comments and queries to Markus Joos, CERN
+41 22 767 2364
Markus.Joos@cern.ch

*Abstract*

This note defines an application program interface (API) for the use of VMEbus in the Read-Out Driver (ROD) system. The API will be used in the ROD Crate DAQ in order to communicate with the ROD(s) and other equipment in the ROD crate which is also to be controlled. The API contains functions related to the use of the VMEbus master mapping, the VMEbus errors, the VMEbus slave mapping, the VMEbus block transfers and the VMEbus interrupts.

Table of Contents

1 Introduction 3

1.1 Description of the API 3

1.2 Design Issues 3

1.3 Implementation Issues 4

1.4 Organization of this Document 5

2 Application Program Interface 7

2.1 Overview 7

2.2 Type Definitions 9

2.3 Functions for Return Codes 10

2.4 General Functions 13

2.5 VMEbus CR/CSR Access 15

2.6 VMEbus Master Mapping and Single Cycles 17

2.7 VMEbus Error Handler 26

2.8 VMEbus Slave Mapping 29

2.9 VMEbus Block Transfers 34

2.10 VMEbus Interrupts 45

3 Programming Examples 57

3.1 Example 1: Functions for Return Codes 57

3.2 Example 2: CR/CSR Space 58

3.3 Example 3: Master Mapping - Safe Access 59

3.4 Example 4: Master Mapping - Fast Access 61

3.5 Example 5: Master Mapping - Bus Error Handler 63

3.6 Example 6: Slave Mapping 65

3.7 Example 7: Block Transfer - Detailed Functions 66

3.8 Example 8: Block Transfer - Integrated Function 68

3.9 Example 9: Interrupts - Synchronous Method 70

3.10 Example 10: Interrupts - Asynchronous Method 72

3.11 Example 11: Interrupts - Generate Interrupts 74

4 VMEbus Utility Programs 75

4.1 VMEbus Configuration Utility 75

4.2 VMEbus Test and Debug Utility 75

4.3 VMEbus Scanning Facility 75

5 Ideas for a C++ Binding 76

5.1 Types 77

5.2 VMEbus library/driver 78

5.3 VMEbus Master Mapping 80

5.4 VMEbus Slave Mapping 81

5.5 VMEbus Block Transfer 82

5.6 VMEbus Interrupts 83

* Introduction
* Description of the API

This note defines an application program interface (API) for the use of VMEbus in the Read-Out Driver (ROD) system. The API will be used in the ROD Crate DAQ (see EDMS note ATL-DQ-EN-0020) in order to communicate with the ROD(s) and other equipment in the ROD crate which is also to be controlled.

The API contains functions related to the following uses of the VMEbus:

* master mappings and single cycles;
* bus error handling;
* slave mappings;
* block transfers;
* interrupts.

The API further contains type definitions, functions to handle the return codes and general functions for the use of the VMEbus.

The API assumes the presence of an operating system and of high-level language compilers on the ROD Crate Processor.

* Design Issues

A few notes on the design guidelines of the API:

* Simplicity and uniformity
* The API was designed to be as simple as possible and only be as complicated as necessary.
* The API provides general-purpose services to the application program.
* The API hides all differences of different hardware platforms. The functions of the API are the same on all different platforms. The return codes can, however, be different.
* Names
* The API uses readable and meaningful names for all of its functions, as well as the common prefix “VME\_”.
* Identifiers
* The API uses identifiers of type “int” for the following complex entities: master mapping, slave mapping, block transfers and interrupts.
* There is one function for each type of entity which creates the corresponding entity and returns the identifier.
* The identifier is to be used in subsequent function calls for this type of entity.
* Return codes
* All functions of the API return an unambiguous return code.
* The return code is equal to 0 if the function has terminated without error. The return code is different from 0 in case the function terminated with an error.
* The return code is of a type compatible with “unsigned int”. It can be a complex data type, if the VMEbus API is implemented with libraries which use a complex type for the return code.
* The return code can be translated into a flat “int” type (à la UNIX errno.h) for comparison with meaningful symbols.
* A textual representation of the return code can be printed to “stdout” or to a string by the application program.
* Known limitations
* No Read-Modify-Write functions are defined in the API. Those can be added later if needed.
* Each VMEbus interrupt vector can only be used by one process.
* No functions are defined in the API to notify the application program of VMEbus power failures, e.g. signalled by ACFAIL. Those can be added later if needed.
* Implementation Issues

The following issues are related to the implementation of the API:

* Layered implementation

The implementation of the API can use low-level libraries and/or system-level drivers if necessary. The number of different libraries or drivers and the dependencies on other external libraries shall, however, be minimised.

* Utility programs

The implementation of the API can be accompanied by a utility program which is used to configure the VMEbus statically and by a utility program which allows to test and debug the VMEbus, see Section VMEbus Utility Programs.

* System-level services

The implementation of the API shall encapsulate all resource management related to the VMEbus bridge, the DMA engine(s), the VMEbus error handling and the VMEbus interrupt handling. The application program shall not deal with those issues explicitly. All resources shall either be allocated statically using the VMEbus configuration utility or dynamically using the various functions of the API.

* Bus error handling

The implementation of the API shall encapsulate, in particular, the VMEbus error handling. At the level of single-word read and write access, the user shall have the choice to check the bus error status or to ignore it. In the latter case, a signal can be sent to the application program to handle the bus error. Separate functions shall be provided for those cases. At the level of VMEbus CR/CSR access and block transfers the bus error handling shall always be included (see Sections VMEbus CR/CSR Access and VMEbus Block Transfers).

* Blocking functions

The API’s blocking functions, e.g. waiting for the end of a VMEbus block transfer or for a VMEbus interrupt shall be implemented in an efficient way. The response time between the external VMEbus event and the return of the function in the application program shall be minimised.

* Interrupts

Since it is not known if the VMEbus interrupters in a system are of type Release-On-Acknowledge (ROAK) or Release-On-Register-Access (RORA), the implementation of the API shall associate VMEbus interrupt levels exclusively to either of the two types. When a VMEbus interrupt from a level associated to ROAK interrupters is received the implementation does not alter the state of the level. When a VMEbus interrupt from a level associated to RORA interrupters is received, the implementation disables that level. The level must be re-enabled by the application program using a function of the API. It is supposed that the association of levels to interrupter types can statically be modified using the VMEbus configuration utility, see Section VMEbus Configuration Utility. The same utility will also be used to statically activate the interrupt levels.

* Multi-processing and multi-threading

The implementation of the API shall allow for several application programs and multiple threads within the same application program to use all functions of the API concurrently. This might require the implementation of one or more drivers for all or parts of the API.

* Logging

The implementation of the API shall log serious errors with a central logging facility, e.g. a global file or kernel messages. The implementation of the API shall also log events of the VMEbus of general interest with the logging facility.

* Language binding

C was chosen for the language binding of the API as presented in Sections Application Program Interface and Programming Examples. Some ideas on a possible C++ language binding or wrapping are presented in Section Ideas for a C++ Binding.

* Data types

For passing data in and out of a VMEbus master mapping using single cycles, separate functions are proposed for the following types, included from types.h (BSD), see Section VMEbus Master Mapping and Single Cycles:

* “unsigned int” or “u\_int” (32 bit),
* “unsigned short” or “u\_short” (16 bit) and
* “unsigned char” or “u\_char” (8 bit).

The user knows the types of values and defines them in the application program. The compiler shall be used to enforce type safety for the function calls. For the C++ binding or wrapping, polymorphic class methods can be used.

* Organization of this Document

Section Application Program Interface contains the definition of the API. For each function the section gives a detailed description of all input and output parameters, a description of the functionality and the return codes. The section contains sub-section for the type definitions used by the API, functions concerning the return codes, general functions and functions for the CR/CSR access, master mapping, bus error handling, slave mapping, block transfers and interrupts.

Section Programming Examples contains programming examples which show how the API is to be used. The examples cover all important cases for return codes, CR/CSR access, master mapping, bus error handling, slave mapping, block transfers and interrupts.

Section VMEbus Utility Programs contains a description of the utility programs which accompany the API implementation. Some implementations will require a VMEbus configuration utility. For all implementations there shall be a test and debugging, as well as a scanning utility.

Section Ideas for a C++ Binding gives some ideas on a possible C++ language binding or wrapping of the API. The public members of the classes are shown.

* Application Program Interface
* Overview

The following list is an overview of all type and function definitions in the VMEbus API:

Type Definitions

* u\_int, u\_short, u\_char
* VME\_ErrorCode\_t
* VME\_BusErrorInfo\_t
* VME\_MasterMap\_t
* VME\_SlaveMap\_t
* VME\_BlockTransferItem\_t
* VME\_BlockTransferList\_t
* VME\_InterruptItem\_t
* VME\_InterruptList\_t
* VME\_InterruptInfo\_t

Functions for Return Codes

* VME\_ErrorPrint
* VME\_ErrorString
* VME\_ErrorNumber

General Functions

* VME\_Open
* VME\_Close

CR/CSR Access

* VME\_ReadCRCSR
* VME\_WriteCRCSR

Bus Error Handling

* VME\_BusErrorRegisterSignal
* VME\_BusErrorInfoGet

Master Mapping and Single Cycles

* VME\_MasterMap
* VME\_MasterMapVirtualAddress
* VME\_MasterMapVirtualLongAddress
* VME\_ReadSafeUInt, VME\_ReadSafeUShort, VME\_ReadSafeUChar
* VME\_WriteSafeUInt, VME\_WriteSafeUShort, VME\_WriteSafeUChar
* VME\_ReadFastUInt, VME\_ReadFastUShort, VME\_ReadFastUChar
* VME\_WriteFastUInt, VME\_WriteFastUShort, VME\_WriteFastUChar
* VME\_MasterUnmap
* VME\_MasterMapDump

Slave Mapping

* VME\_SlaveMap
* VME\_SlaveMapVmebusAddress
* VME\_SlaveUnmap
* VME\_SlaveMapDump

Block Transfer

* VME\_BlockTransferInit
* VME\_BlockTransferStart
* VME\_BlockTransferWait
* VME\_BlockTransferEnd
* VME\_BlockTransfer
* VME\_BlockTransferStatus
* VME\_BlockTransferRemaining
* VME\_BlockTransferDump

Interrupts

* VME\_InterruptLink
* VME\_InterruptWait
* VME\_InterruptRegisterSignal
* VME\_InterruptInfoGet
* VME\_InteruptReenable
* VME\_InterruptUnlink
* VME\_InterruptGenerate
* VME\_InterruptDump

Miscellaneous functions

* VME\_SendSysreset

The following remarks apply to all functions defined in the API:

* If not stated otherwise, all functions of this API are non-blocking, i.e. they return immediately indicating an error code if necessary. Wherever functions are blocking, i.e. waiting on external events, e.g. end of block transfer or VMEbus interrupt, this is stated explicitly.
* The return values of all function of this API can be used for comparison, after the error code has been converted to an error number. The return value VME\_SUCCESS ( 0) can always be used for comparison.
* The implementation of the API is in the following called the “VMEbus library/driver”.
* Type Definitions

The following types are defined in “vme\_rcc.h” for general use throughout the API:

Data Transfer Types

typedef unsigned int u\_int; (included from types.h; 32 bit)

typedef unsigned short u\_short; (included from types.h; 16 bit)

typedef unsigned char u\_char; (included from types.h; 8 bit)

Return Code Type

typedef unsigned int VME\_ErrorCode\_t;

Other Types

VME\_MasterMap\_t see Section VMEbus Master Mapping and Single Cycles

VME\_BusErrorInfo\_t see Section VMEbus Error Handler

VME\_SlaveMap\_t see Section VMEbus Slave Mapping

VME\_BlockTransferItem\_t see Section VMEbus Block Transfers

VME\_BlockTransferList\_t see Section VMEbus Block Transfers

VME\_InterruptItem\_t see Section VMEbus Interrupts

VME\_InterruptList\_t see Section VMEbus Interrupts

VME\_InterruptInfo\_t see Section VMEbus Interrupts

* Functions for Return Codes

VME\_ErrorPrint()

Synopsis

#include “vme\_rcc.h”

u\_int VME\_ErrorPrint(VME\_ErrorCode\_t error\_code);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_ErrorCode\_t error\_code | in | error code to be printed |

Description

The VME\_ErrorPrint() function prints a textual representation of *error\_code* to “stdout”.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The error code was successfully printed. |
| VME\_NOTKNOWN | The error code is not known. |

Programming Example

For a programming example see Section Example 1: Functions for Return Codes.

Notes

none

VME\_ErrorString()

Synopsis

#include “vme\_rcc.h”

u\_int VME\_ErrorString(VME\_ErrorCode\_t error\_code, char\* error\_string);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_ErrorCode\_t error\_code | in | error code to be converted to a character string |
| char\* error\_string | out | character string containing the textual representation of the error code |

Description

The VME\_ErrorString() function returns a textual representation of *error\_code* in the character string *error\_string*. *error\_string* must contain space for at least *VME\_MAXSTRING* (defined in “vme\_rcc.h”) characters.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The error code was successfully converted to a textual representation. |
| VME\_NOTKNOWN | The error code is not known. |

Programming Example

For a programming example see Section Example 1: Functions for Return Codes.

Notes

none

VME\_ErrorNumber()

Synopsis

#include “vme\_rcc.h”

u\_int VME\_ErrorNumber(VME\_ErrorCode\_t error\_code, int\* error\_number);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_ErrorCode\_t error\_code | in | error code to be converted to an error number |
| int\* error\_number | out | error number corresponding to the error code |

Description

The VME\_ErrorNumber() function converts the possibly complex *error\_code* into a flat error number *error\_number*. The flat error number can then be used for comparison with the return codes defined in this API. The return code VME\_SUCCESS ( 0) can always be used for comparison.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The error code was successfully converted to an error number. |
| VME\_NOTKNOWN | The error code is not known. |

Programming Example

For a programming example see Section Example 1: Functions for Return Codes.

Notes

none

* General Functions

VME\_Open()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_Open(void);

Parameters

none

Description

The VME\_Open() function opens the VMEbus library/driver and allocates the resources required to use the VMEbus. This function must be called prior to any other function of the VMEbus API.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The VMEbus library/driver was successfully opened. |
| *others* | specific to the implementation |

Programming Example

* For a programming example see Section Example 1: Functions for Return Codes

Notes

none

VME\_Close()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_Close(void);

Parameters

none

Description

The VME\_Close() releases all resources which were allocated in a VME\_Open() function call and closes the VMEbus library/driver. This function is the last function of the API to be called by the application program.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The VMEbus library/driver was successfully closed. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| *others* | specific to the implementation |

Programming Example

* For a programming example see Section Example 1: Functions for Return Codes

Notes

none

* VMEbus CR/CSR Access

VME\_ReadCRCSR()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_ReadCRCSR(int slot\_number, u\_int crcsr\_field, u\_int\* value);

Parameters

|  |  |  |
| --- | --- | --- |
| int slot\_number | in | number of VMEbus slot to be addressed (0 to 31) |
| u\_int crcsr\_field | in | field name in the CR/CSR space; see description |
| u\_int\* value | out | value read from CR/CSR space |

Description

The VME\_ReadCRCSR() functions reads a value from the field at *crcsr\_field* in the CR/CSR space of the VMEbus slave at slot *slot\_number.* The symbolic constant *VME\_MYSLOT* (defined in “vme\_rcc.h”) allows access of the CR/CSR space of the VMEbus slave the application program runs on.

Symbolic constants for *crcsr\_field* are provided in the “vme\_rcc.h” file, see also the VME64 and VME64x standard. The VME\_ReadCRCSR() function knows how many bytes, between 1 and 4, must be read for each value.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The value was successfully read from CR/CSR space. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOSLOT | The slot number is invalid. |
| VME\_NOFIELD | The CR/CSR field is invalid. |
| VME\_BUSERROR | A VMEbus error occurred during the read from CR/CSR space. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Example 2: CR/CSR Space.

Notes

The mapping of the CR/CSR space can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility.

VME\_WriteCRCSR()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_WriteCRCSR(int slot\_number, u\_int crcsr\_field, u\_int value);

Parameters

|  |  |  |
| --- | --- | --- |
| int slot\_number | in | number of VMEbus slot to be addressed (0 to 31) |
| u\_int crcsr\_field | in | field name in the CR/CSR space; see description |
| u\_int value | in | value to be written to CR/CSR |

Description

The VME\_WriteCRCSR() functions writes a value to the field at *crcsr\_field* in the CR/CSR space of the VMEbus slave at slot *slot\_number.* The symbolic constant *VME\_MYSLOT* (defined in “vme\_rcc.h”) allows access of the CR/CSR space of the VMEbus slave the application program runs on.

Symbolic constants for *crcsr\_field* are provided in the “vme\_rcc.h” file, see also the VME64 and VME64x standard. The VME\_WriteCRCSR() function knows how many bytes, between 1 and 4, must be written for each value.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The value was successfully written to CR/CSR space. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOSLOT | The slot number is invalid. |
| VME\_NOFIELD | The CR/CSR field is invalid. |
| VME\_BUSERROR | A VMEbus error occurred during the write to CR/CSR space. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Example 2: CR/CSR Space.

Notes

The mapping of the CR/CSR space can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility.

* VMEbus Master Mapping and Single Cycles

VME\_MasterMap\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_int vmebus\_address;

 u\_int window\_size;

 u\_int address\_modifier;

 u\_int options;

} VME\_MasterMap\_t;

Fields

|  |  |
| --- | --- |
| u\_int vmebus\_address | base address of the VMEbus window |
| u\_int window\_size | size of the VMEbus window in number of bytes |
| u\_int address\_modifier | address modifier to be used when accessing the master mapping |
| u\_int options | other options, include read prefetching and write posting |

Description

The VME\_MasterMap\_t type is used to hold input information on a master mapping for use in a VME\_MasterMap() function call. The type definition is provided in the “vme\_rcc.h” file.

*address\_modifier* is one of the following parameters (defined in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_AM09 | address mode 0x09 |
| ... | ... |
| VME\_AM39 | address mode 0x39 |

*options* is a bit-wise combination of the following parameters and possibly some other implementation-specific ones (all defined in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_RP | read prefetching |
| VME\_WP | write posting |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

none

VME\_MasterMap()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_MasterMap(VME\_MasterMap\_t\* master\_map, int\* master\_mapping);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_MasterMap\_t\*master\_map | in | input information on the master mapping |
| int\* master\_mapping | out | identifier of the master mapping;to be used in subsequent function calls |

Description

The VME\_MasterMap() function creates a VMEbus master mapping defined by *master\_map* and returns the identifier *master\_mapping* which is to be used in subsequent function calls.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The master mapping was successfully created. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

Some parameters for the master mapping, e.g. for static mapping or for byte swapping, can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility.

VME\_MasterMapVirtualAddress(), VME\_MasterMapVirtualLongAddress()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_MasterMapVirtualAddress(int master\_mapping, u\_int\* virtual\_address);

VME\_ErrorCode\_t VME\_MasterMapVirtualLongAddress(int master\_mapping, u\_long\* virtual\_address);

Parameters

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mappingobtained in call to VME\_MasterMap() |
| u\_int\* virtual\_addressu\_long\* virtual\_address | out | virtual address associated to the master mapping |

Description

The VME\_MasterMapVirtualAddress() and VME\_MasterMapVirtualLongAddress() functions return the virtual address associated to a *master\_mapping* obtained by a function call to VME\_MasterMap(). This address can be used for fast read and write methods ignoring VMEbus errors, e.g.

value = \*(u\_int \*)(virtual\_address + address\_offset);

\*(u\_int \*)(virtual\_address + address\_offset) = data;

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The virtual address was successfully returned. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The master mapping is not known. |

Programming Example

For a programming example see Section Master Mapping - Fast Access.

Notes

The function VME\_MasterMapVirtualAddress() which was defined originally does not work for 64bit operating systems. Therefore VME\_MasterMapVirtualLongAddress() was added to the API later on. User programs should only use VME\_MasterMapVirtualLongAddress() because this function works for both 32bit and 64bit OS kernels.

VME\_ReadSafe()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_ReadSafeUInt(int master\_mapping, u\_int address\_offset, u\_int\* value);

VME\_ErrorCode\_t VME\_ReadSafeUShort(int master\_mapping, u\_int address\_offset, u\_short\* value);

VME\_ErrorCode\_t VME\_ReadSafeUChar(int master\_mapping, u\_int address\_offset, u\_char\* value);

Parameters

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mapping;obtained in call to VME\_MasterMap() |
| u\_int address\_offset | in | address offset to be read from;must be aligned according to type |
| u\_int, u\_short, u\_char \*value | out | value read from the VMEbus slave |

Description

The VME\_ReadSafeXXX() functions reads safely an “unsigned int”, “unsigned short”, or “unsigned char” value from *master\_mapping* (i.e. from the VMEbus slave) at *address\_offset.* The functions check if a VMEbus error occurred during the read cycle.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The value was read successfully from the master mapping. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The master mapping is not known. |
| VME\_RANGE | The address offset is outside the window for the master mapping. |
| VME\_ALIGN | The address offset is not correctly aligned with respect to the type. |
| VME\_BUSERROR | A VMEbus error occurred during the read cycle. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

none

VME\_WriteSafe()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_WriteSafeUInt(int master\_mapping, u\_int address\_offset, u\_int value);

VME\_ErrorCode\_t VME\_WriteSafeUShort(int master\_mapping, u\_int address\_offset, u\_short value);

VME\_ErrorCode\_t VME\_WriteSafeUChar(int master\_mapping, u\_int address\_offset, u\_char value);

Parameters

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mapping;obtained in call to VME\_MasterMap() |
| u\_int address\_offset | in | address offset to be written to;must be aligned according to type |
| u\_int, u\_short, u\_char value | in | value to be written to the VMEbus slave |

Description

The VME\_WriteSafeXXX() functions write safely an “unsigned int”, “unsigned short”, or “unsigned char” value to *master\_mapping (i.e. to a VMEbus slae)* at *address\_offset.* The functions check if a VMEbus error occurred during the write cycle.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The value was written successfully to the master mapping. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The master mapping is not known. |
| VME\_RANGE | The address offset is outside the window for the master mapping. |
| VME\_ALIGN | The address offset is not correctly aligned with respect to the type. |
| VME\_BUSERROR | A VMEbus error occurred during the write cycle. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

none

VME\_ReadFast()

Synopsis

#include “vme\_rcc.h”

void VME\_ReadFastUInt(int master\_mapping, u\_int address\_offset,
u\_int\* value);

void VME\_ReadFastUShort(int master\_mapping, u\_int address\_offset, u\_short\* value);

void VME\_ReadFastUChar(int master\_mapping, u\_int address\_offset, u\_char\* value);

Parameters

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mapping;obtained in call to VME\_MasterMap() |
| u\_int address\_offset | in | address offset to be read from;must be aligned according to type |
| u\_int, u\_short, u\_char \*value | out | value read from the master mapping |

Description

The VME\_ReadFastXXX() functions reads an “unsigned int”, “unsigned short”, or “unsigned char” value from *master\_mapping* at *address\_offset.* The functions ignore possible VMEbus errors and return immediately. The application program can still receive a signal related to the VMEbus error, see Section VMEbus Error Handler.

The VME\_ReadFastXXX() functions are identical to the following statements:

value\_u\_int = \*(u\_int \*)(virtual\_address + address\_offset);

value\_u\_short = \*(u\_short \*)(virtual\_address + address\_offset);

value\_u\_char = \*(u\_char \*)(virtual\_address + address\_offset);

The virtual address can be obtained using the VME\_MasterMapVirtualAddress() function.

Return Values

none

Programming Example

For a programming example see Section Master Mapping - Fast Access.

Notes

The value read by the VME\_ReadFastXXX() functions can be invalid, if a VMEbus error occurred.

VME\_WriteFast()

Synopsis

#include “vme\_rcc.h”

void VME\_WriteFastUInt(int master\_mapping, u\_int address\_offset, u\_int value);

void VME\_WriteFastUShort(int master\_mapping, u\_int address\_offset, u\_short value);

void VME\_WriteFastUChar(int master\_mapping, u\_int address\_offset, u\_char value);

Parameters

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mapping;obtained in call to VME\_MasterMap() |
| u\_int address\_offset | in | address offset to be written to;must be aligned according to type |
| u\_int, u\_short, u\_char value | out | value to be written to the master mapping |

Description

The VME\_WriteFastXXX() functions write an “unsigned int”, “unsigned short”, or “unsigned char” value to *master\_mapping* at *address\_offset.* The functions ignore possible VMEbus errors and return immediately. The application program can still receive a signal related to the VMEbus error, see Section VMEbus Error Handler.

The VME\_Write FastXXX() functions are identical to the following statements:

\*(u\_int \*)(virtual\_address + address\_offset) = value\_u\_int;

\*(u\_short \*)(virtual\_address + address\_offset) = value\_u\_short;

\*(u\_char \*)(virtual\_address + address\_offset) = value\_u\_char;

The virtual address can be obtained using the VME\_MasterMapVirtualAddress() function.

Return Values

none

Programming Example

For a programming example see Section Master Mapping - Fast Access.

Notes

The value might not be written by the VME\_WriteFastXXX() functions, if a VMEbus error occurred.

VME\_MasterUnmap()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_MasterUnmap(int master\_mapping);

Parameter

|  |  |  |
| --- | --- | --- |
| int master\_mapping | in | identifier of master mapping;obtained in call to VME\_MasterMap() |

Description

The VME\_MasterUnmap() function deletes the VMEbus master mapping associated to *master\_mapping*. The identifier *master\_mapping* shall not be used after this function call.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The master mapping was successfully deleted. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The master mapping is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

none

VME\_MasterMapDump()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_MasterMapDump(void);

Parameters

none

Description

The VME\_MasterMapDump() function dumps system parameters for all VMEbus master mappings to “stdout”.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The master mappings were successfully dumped. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |

Programming Example

For a programming example see Section Master Mapping - Safe Access.

Notes

none

* VMEbus Error Handler

VME\_BusErrorRegisterSignal()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BusErrorRegisterSignal(int signal\_number);

Parameters

|  |  |  |
| --- | --- | --- |
| int signal\_number | in | signal number to be sent in case of VMEbus error |

Description

The VME\_BusErrorRegisterSignal() function registers signal *signal\_number* with the VMEbus library/driver. In case the VMEbus library/driver detects a VMEbus error and the VMEbus error did not occur during one of the following functions:

* VME\_ReadCRCSR() or VME\_WriteCRCSR(),
* VME\_ReadSafeXXX() or VME\_WriteSafeXXX(),
* VME\_BlockTransferXXX(),

a signal with number *signal\_number* will be sent to the process calling this function. If the process wants to handle the signal it must install a signal handler. Installing a signal handler is not part of this API. The value 0 for *signal\_number* is used to “unregister” a signal from the VMEbus library/driver.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The signal was successfully registered. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Bus Error Handler.

Notes

none

VME\_BusErrorInfo\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_int vmebus\_address;

 u\_int address\_modifier;

 u\_int multiple;

} VME\_BusErrorInfo\_t;

Fields

|  |  |
| --- | --- |
| u\_int vmebus\_address | address at which the VMEbus error occurred |
| u\_int address\_modifier | address modifier at which the VMEbus error occurred |
| u\_int multiple | flag indicating if multiple VMEbus errors occurred |

Description

The VME\_BusErrorInfo\_t type is used to retrieve information on a VMEbus error. The type definition is provided in the “vme\_rcc.h” file.

Programming Example

For a programming example see Section Master Mapping - Bus Error Handler.

Notes

none

VME\_BusErrorInfoGet()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BusErrorInfoGet(VME\_BusErrorInfo\_t\* bus\_error\_info);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_BusErrorInfo\_t\* bus\_error\_info | out | information on VMEbus error |

Description

The VME\_BusErrorInfoGet() function returns information on the VMEbus error received. This function can be used in a bus error handler in order to determine where the bus error occurred.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The bus error information was successfully returned. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOBUSERROR | There has not been a bus error; *bus\_error\_info* is empty. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Master Mapping - Bus Error Handler.

Notes

This function is intended for use in the bus error signal handling function, see VME\_BusErrorRegisterSignal().

* VMEbus Slave Mapping

VME\_SlaveMap\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_int system\_iobus\_address;

 u\_int window\_size;

 u\_int address\_width;

 u\_int options;

} VME\_SlaveMap\_t;

Fields

|  |  |
| --- | --- |
| u\_int system\_iobus\_address | (physical) base address of the user space to be mapped |
| u\_int window\_size | size of the user space in number of bytes |
| u\_int address\_width | address width to be used by the slave mapping |
| u\_int options | other options, include read prefetching and write posting |

Description

The VME\_SlaveMap\_t type is used to input information on a slave mapping for use in a VME\_SlaveMap() function call. The type definition is provided in the “vme\_rcc.h” file.

*system\_iobus\_address* must point at contiguous, locked and properly aligned user space*.* The user space can also be a physical resource, e.g. FIFO of the VMEbus master. Obtaining *system\_iobus\_address* for user space is not part of this API.

*address\_width* is one of the following parameters (defined in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_A32 | 32-bit addressing |
| VME\_A24 | 24-bit addressing |

*options* is a bit-wise combination of the following parameters and possibly some other implementation-specific ones (all defined in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_RP | read prefetching |
| VME\_WP | write posting |

Programming Example

For a programming example see Section Slave Mapping.

Notes

none

VME\_SlaveMap()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_SlaveMap(VME\_SlaveMap\_t\* slave\_map, int\* slave\_mapping);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_SlaveMap\_t\* slave\_map | in | information on the slave mapping |
| int\* slave\_mapping | out | identifier of the slave mapping;to be used in subsequent function calls |

Description

The VME\_SlaveMap() function creates a slave mapping defined by *slave\_map* and returns the identifier *slave\_mapping* which is to be used in subsequent function calls.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The slave mapping was successfully created. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Slave Mapping.

Notes

Some parameters for the slave mapping, e.g. for static mapping or for byte swapping, can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility.

The window size of the created slave mapping will be at least as large as the size requested; it might be larger.

VME\_SlaveMapVmebusAddress()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_SlaveMapVmebusAddress(int slave\_mapping, u\_int\* vmebus\_address);

Parameters

|  |  |  |
| --- | --- | --- |
| int slave\_mapping | in | identifier of the slave mappingobtained in call to VME\_SlaveMap() |
| u\_int\* vmebus\_address | out | VMEbus address associated to the slave mapping |

Description

The VME\_SlaveMapVmebusAddress() function returns the VMEbus address associated to *slave\_mapping*. This address can be used by other VMEbus masters.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The VMEbus address was successfully returned. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The slave mapping is not known. |

Programming Example

For a programming example see Section Slave Mapping.

Notes

none

VME\_SlaveUnmap()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_SlaveUnmap(int slave\_mapping);

Parameters

|  |  |  |
| --- | --- | --- |
| int slave\_mapping | in | identifier of the slave mappingobtained in call to VME\_SlaveMap() |

Description

The VME\_SlaveUnmap() function deletes the VMEbus slave mapping associated to *slave\_mapping*. The identifier *slave\_mapping* shall not be used after this function call.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The slave mapping was successfully deleted. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The slave mapping is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Slave Mapping.

Notes

none

VME\_SlaveMapDump()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_SlaveMapDump(void);

Parameters

none

Description

The VME\_SlaveMapDump() function dumps system parameters for all VMEbus slave mappings to “stdout”*.*

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The slave mappings were successfully dumped. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |

Programming Example

For a programming example see Section Slave Mapping.

Notes

none

* VMEbus Block Transfers

VME\_BlockTransferItem\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_int vmebus\_address;

 u\_int system\_iobus\_address;

 u\_int size\_requested;

 u\_int control\_word;

 u\_int size\_remaining;

 u\_int status\_word;

} VME\_BlockTransferItem\_t;

Fields

|  |  |
| --- | --- |
| u\_int vmebus\_address | VMEbus address |
| u\_int system\_iobus\_address | system I/O bus address |
| u\_int size\_requested | size of requested transfer in number of bytes |
| u\_int control\_word | direction and type of block transfer; the type includes address mode and specifies possibly enhanced transfer protocols |
| u\_int size\_remaining | size of remaining transfer in number of bytes |
| u\_int status\_word | status of the block transfer |

Description

The VME\_BlockTransferItem\_t type is used to describe a single block transfer in a block transfer list. This is a requested block transfer which might be split by the subsequent function calls into one or more actual VMEbus block transfers, e.g. for alignment and size reasons.

*system\_iobus\_address* must point to contiguous, locked and properly aligned memory. The memory management is not part of this API.

*control\_word* specifies the direction and type of block transfer. The type includes the address mode and specifies possibly enhanced transfer protocols. *control\_word* contains one of the following parameters (defined in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_DMA\_D32W | transfer data from system I/O bus to VMEbus using 32-bit words |
| VME\_DMA\_D32R | transfer data from VMEbus to system I/O bus using 32-bit words |
| VME\_DMA\_D64W | transfer data from system I/O bus to VMEbus using 64-bit words |
| VME\_DMA\_D64R | transfer data from VMEbus to system I/O bus using 64bit words |
| VME\_DMA\_2EVMER | transfer data from system I/O bus to VMEbus using 2eVME mode |
| VME\_DMA\_2EVMEW | transfer data from VMEbus to system I/O bus using 2eVME mode |
| VME\_DMA\_2ESSTR | transfer data from system I/O bus to VMEbus using 2eSST mode |
| VME\_DMA\_2ESSTW | transfer data from VMEbus to system I/O bus using 2eSST mode |

*control\_word* must be ORed bit-wise with one of the following address modes:

|  |  |
| --- | --- |
| VME\_A32 | transfer data using 32-bit addressing |
| VME\_A24 | transfer data using 64-bit addressing |

*status\_word* and *size\_remaining* are filled by the function VME\_BlockTransferWait(). They indicate the status of each block in the block transfer list. The fields can be interpreted with the help of the VME\_BlockTransferStatus() and VME\_BlockTransferRemaining()functions.

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

The block transfer list used by the application program can be independent of another block transfer list used by the VMEbus library/driver internally. This is because the actual block transfers carried out by the VMEbus library/driver might differ from the requested ones due to boundary and alignment restrictions.

On the Tundra Universe II VMEbus bridge chip, the PCI and VMEbus addresses must be aligned on a 4-byte boundary. In addition, the difference between the PCI and the VMEbus addresses must be a multiple of 8 byte.

VME\_BlockTransferList\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 int number\_of\_items;

 VME\_BlockTransferItem\_t list\_of\_items [VME\_MAXBLOCK];

} VME\_BlockTransferList\_t;

Fields

|  |  |
| --- | --- |
| int number\_of\_items | number of items used in the block transfer list |
| VME\_BlockTransferItem\_tlist\_of\_items [VME\_MAXBLOCK] | list of block transfer items |

Description

The VME\_BlockTransferList\_t type is used to define VMEbus block transfers. The type definition and the maximum number of blocks *VME\_MAXBLOCK* are provided in the “vme\_rcc.h” file.

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

A single block transfer must use a block transfer list with only one VME\_Block-TransferItem\_t at *list\_of\_items*[0] and *number\_of\_items* = 1.

VME\_BlockTransferInit()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferInit(VME\_BlockTransferList\_t\* block\_transfer\_list, int\* block\_transfer);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_BlockTransferList\_t\* block\_transfer\_list | in | list of block transfers |
| int\* block\_transfer | out | identifier of the block transfer;to be used in subsequent function calls |

Description

The VME\_BlockTransferInit() function allocates resources for the VME\_BlockTransferList\_t *block\_transfer\_list* and returns the identifier *block\_transfer* which is to be used in subsequent function calls. It might actually break up the block transfers into an internal list of actual VMEbus block transfers, e.g. for alignment and size reasons.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The block transfer was successfully initialised. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOMEM | There is not enough memory to allocate the resources for the required block transfer list. |
| VME\_TOOLONG | The internally generated block transfer list is too long. |
| VME\_NOSIZE | The requested size is invalid. |
| VME\_ALIGN | The addresses are not correctly aligned. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

none

VME\_BlockTransferStart()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferStart(int block\_transfer);

Parameters

|  |  |  |
| --- | --- | --- |
| int block\_transfer | in | identifier of the block transferobtained in call to VME\_BlockTransferInit() |

Description

The VME\_BlockTransferStart() function starts the block transfer associated to *block\_transfer* obtained by a call to VME\_BlockTransferInit().

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The block transfer was successfully started. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The block transfer is not known. |
| VME\_DMABUSY | The DMA engine(s) are busy. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

This function shall not be blocking. The implementation of this function shall return immediately, either indicating that the resources of the DMA engine(s) are not available at the moment (*VME\_DMABUSY*), or by using internal queuing of tasks.

VME\_BlockTransferWait()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferWait(int block\_transfer, int time\_out, VME\_BlockTransferList\_t\* block\_transfer\_list);

Parameters

|  |  |  |
| --- | --- | --- |
| int block\_transfer | in | identifier of the block transferobtained in call to VME\_BlockTransferInit() |
| int time\_out | in | time-out parameter, see description |
| VME\_BlockTransferList\_t\* block\_transfer\_list | out | list of block transfers |

Description

The VME\_BlockTransferWait() function waits until the block transfer associated to *block\_transfer* is finished or until the time-out has elapsed, whichever occurs first.

*time\_out* is an estimate for the time-out period in milliseconds. The value 0 is used to bypass the time-out mechanism and to return immediately, indicating the status of the block transfer. The value -1 is used to bypass the time-out mechanism and to wait until the end of the block transfer.

The return code contains general status information of the whole block transfer; the individual status of a single block transfer can be checked using *block\_transfer\_list* and the VME\_BlockTransferStatus() and the VME\_Block-TransferRemaining() functions.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The block transfer was successfully started |
| VME\_NOTOPEN | The VMEbus library/driver was not opened |
| VME\_NOTKNOWN | The block transfer is not known |
| VME\_INVALIDTO | The time-out is invalid |
| VME\_TIMEOUT | A time-out occurred (if *time\_out* > 0) |
| VME\_DMABUSY | The block transfer is busy (if *time\_out* = 0) |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

This function is generally blocking, except for *time\_out*  0.

VME\_BlockTransferEnd()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferEnd(int block\_transfer);

Parameters

|  |  |  |
| --- | --- | --- |
| int block\_transfer | in | identifier of the block transferobtained in call to VME\_BlockTransferInit() |

Description

The VME\_BlockTransferEnd() function releases the resources allocated for the block transfer associated to *block\_transfer*. It must be called at the end of a block transfer. The identifier *block\_transfer* shall not be used after this function call.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The block transfer was successfully ended. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The block transfer is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

none

VME\_BlockTransfer()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransfer(VME\_BlockTransferList\_t\* block\_transfer\_list, int time\_out);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_BlockTransferList\_t\* block\_transfer\_list | in/out | list of block transfers |
| int time\_out | in | time-out parameter, see description |

Description

The VME\_BlockTransfer() function uses the VME\_BlockTransferList\_t *block\_transfer\_list* and calls the following functions in the order shown:

* VME\_BlockTransferInit(),
* VME\_Block-TransferStart(),
* VME\_BlockTransferWait() and
* VME\_BlockTransferEnd().

*time\_out* is the parameter for the VME\_BlockTransferWait() function call.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The block transfer was successfully started. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_INVALIDTO | The time-out is invalid. |
| VME\_TIMEOUT | A time-out occurred (if *time\_out* > 0). |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Integrated Function.

Notes

This function is generally blocking. The value 0 for *time\_out* in this function shall not be used because the VME\_BlockTransferWait() function will return immediately and the VME\_Block-TransferEnd() function will be called regardless of the state of the transfer.

VME\_BlockTransferStatus()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferStatus(VME\_BlockTransferList\_t\* block\_transfer\_list, int position\_of\_block, VME\_ErrorCode\_t\* status);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_BlockTransferList\_t\* block\_transfer\_list | in | list of block transfers |
| int position\_of\_block | in | position of block in block transfer list |
| VME\_ErrorCode\_t\* status | out | status of block transfer at position *position\_of\_block* |

Description

The VME\_BlockTransferStatus() function returns the status code for the block transfer at *position\_of\_block* in *block\_transfer\_list*. This function is added for convenience, the function is equivalent to the following statement:

status = block\_transfer\_list.list\_of\_items[position\_of\_block].status\_word;

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The status was successfully returned. |
| VME\_RANGE | The position is outside the range of the block transfer list. |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

none

VME\_BlockTransferRemaining()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferRemaining(VME\_BlockTransferList\_t block\_transfer\_list, int position\_of\_block, u\_int\* remaining);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_BlockTransferList\_t\* block\_transfer\_list | in | list of block transfers |
| int position\_of\_block | in | position of block in block transfer list |
| u\_int\* remaining | out | number of bytes remaining to be transferred at position *position\_of\_block* |

Description

The VME\_BlockTransferRemaining () function returns the number of bytes remaining to be transferred for the block transfer at *position\_of\_block* in *block\_transfer\_list*. After successful transfer this value shall be equal to 0. This function is added for convenience, it is equivalent to the following statement:

remaining = block\_transfer\_list.list\_of\_items[position\_of\_block].size\_remaining;

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The byte number remaining was successfully returned. |
| VME\_RANGE | The position is outside the range of the block transfer list. |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

none

VME\_BlockTransferDump()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_BlockTransferDump(void);

Parameters

none

Description

The VME\_BlockTransferDump() function dumps the status of the DMA engine(s) to “stdout”.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The status of the DMA engine(s) was successfully dumped. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

none

* VMEbus Interrupts

VME\_InterruptItem\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_char vector;

 u\_int level;

 u\_int type;

} VME\_InterruptItem\_t;

Fields

|  |  |
| --- | --- |
| u\_char vector | VMEbus interrupt vector |
| u\_int level | VMEbus interrupt level |
| u\_int type | flag indicating the type of interrupt handling to be used(see description) |

Description

The VME\_InterruptItem\_t type is used to describe a single interrupt in a list of interrupts. Each interrupt is defined by the vector, the level and the type of the VMEbus interrupt that the application program requests to be linked to.

*type* specifies the interrupt handling to be used for the interrupt. The following types are defined (in “vme\_rcc.h”):

|  |  |
| --- | --- |
| VME\_INT\_ROAK | “Release-On-Acknowledge” |
| VME\_INT\_RORA | “Release-On-Register-Access” |

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

The interrupt handling type is required in order to distinguish VMEbus interrupters of RORA and ROAK type. The interrupt handling type can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility. Usually, the type will be allowed to be configured individually for each VMEbus interrupt level. A given level must therefore only be used by VMEbus interrupts of the associated type.

VME\_InterruptList\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 int number\_of\_items;

 VME\_InterruptItem\_t list\_of\_items [VME\_MAXINTERRUPT];

} VME\_InterruptList\_t;

Fields

|  |  |
| --- | --- |
| int number\_of\_items | number of items used in the interrupt list |
| VME\_InterruptItem\_t list\_of\_items[VME\_MAXINTERRUPT] | list of interrupt items |

Description

The VME\_InterruptList\_t type is used to define a list of interrupts. The type definition and the maximum number of interrupts *VME\_MAXINTERRUPT* are provided in the “vme\_rcc.h” file.

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

A single interrupt must use an interrupt list with only one VME\_InterruptItem\_t at *list\_of\_items*[0] with *number\_of\_items* = 1.

VME\_InterruptLink()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptLink(VME\_InterruptList\* vmebus\_interrupt\_list, int\* interrupt);

Parameters

|  |  |  |
| --- | --- | --- |
| VME\_InterruptList\* vmebus\_interrupt\_list | in | list of VMEbus interrupts |
| int\* interrupt  | out | identifier of the interrupt;to be used in subsequent function calls |

Description

The VME\_InterruptLink() function creates a link between a list of VMEbus interrupts and the application program. It returns the identifier *interrupt* which is to be used in subsequent function calls.

By default, after creation of the interrupt link, the application program applies a synchronous method waiting for interrupts using the VME\_InterruptWait() function. If the application program wants to apply an asynchronous method, the VME\_InterruptRegisterSignal() function must be used.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The link to the VMEbus interrupt was successfully created. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_TOOMANYINT | The list of interrupts requested is too long. |
| VME\_ILLINTLEVEL | The interrupt level is illegal. |
| VME\_ILLINTTYPE | The interrupt type is illegal. |
| VME\_INTCONF | The list of interrupts was not linked to the application programbecause an interrupt is in conflict with the static configuration. |
| VME\_INTUSED | The list of interrupts cannot be linked to the application programbecause an interrupt is already being used. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

A ROAK (“Release-On-AcKnowledge”) type of VMEbus interrupter releases the interrupt after the VMEbus Acknowledge cycle; the VMEbus driver therefore does not disable reception of subsequent interrupts. A RORA (“Release-On-Register-Access”) type of VMEbus interrupter releases the interrupt only after access to a register of the VMEbus module; the VMEbus driver therefore disables reception of subsequent interrupts on the same VMEbus interrupt level.

Some parameters for the VMEbus interrupts, e.g. for interrupt levels and the interrupt handling types, can be configured statically using the VMEbus configuration utility, see Section 4.1. The VME\_InterruptLink() function checks if the requested VMEbus interrupt level has been enabled and configured for the requested type. A given VMEbus vector can only be used by one process.

VME\_InterruptInfo\_t

Synopsis

in vme\_rcc.h:

typedef struct {

 u\_char vector;

 u\_int level;

 u\_int type;

 u\_int multiple;

} VME\_InterruptInfo\_t;

Fields

|  |  |
| --- | --- |
| u\_char vector | VMEbus interrupt vector |
| u\_int level | VMEbus interrupt level |
| u\_int type | type of VMEbus interrupter (ROAK or RORA) |
| u\_int multiple | flag indicating if the VMEbus interrupt occurred multiple times |

Description

The VME\_InterruptInfo\_t type is used to retrieve information on a VMEbus interrupt. The type definition is provided in the “vme\_rcc.h” file.

Programming Example

For a programming example see Section Interrupts - Asynchronous Method.

Notes

none

VME\_InterruptWait()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptWait(int interrupt, int time\_out, VME\_InterruptInfo\_t\* interrupt\_info);

Parameters

|  |  |  |
| --- | --- | --- |
| int interrupt  | in | identifier of the interruptobtained in call to VME\_InterruptLink() |
| int time\_out | in  | time-out parameter, see description |
| VME\_InterruptInfo\_t\* interrupt\_info | out | information on VMEbus interrupts |

Description

The VME\_InterruptWait() function waits until an interrupt associated to *interrupt* is received or until the time-out has elapsed, whichever occurs first.

*time\_out* is an estimate for the time-out period in milliseconds. The value 0 is used to bypass the time-out mechanism and to return immediately, indicating the status of the interrupt. The value -1 is used to bypass the time-out mechanism and to wait until an interrupt is received.

After a VMEbus interrupt has been received *interrupt\_info* contains the information on the VMEbus interrupt actually received. Depending on *time\_out* and on the return code, *interrupt\_info* might be empty.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | A VMEbus interrupt was successfully received. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The interrupt is not known. |
| VME\_TIMEOUT | A time-out occurred (if time\_out > 0). |
| VME\_NOINTERRUPT | No interrupt has been received (if time\_out = 0). |
| VME\_INTBYSIGNAL | The function returned because a signal was received. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

This function is generally blocking, except for *time\_out*  0.

VME\_InterruptRegisterSignal()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptRegisterSignal(int interrupt, int signal\_number);

Parameters

|  |  |  |
| --- | --- | --- |
| int interrupt  | in | identifier of the interruptobtained in call to VME\_InterruptLink() |
| int signal\_number | in | signal number to be sent in case of VMEbus interrupt |

Description

The VME\_InterruptRegisterSignal() function registers signal *signal\_number* with the VMEbus library/driver. In case the VMEbus library/driver receives a VMEbus interrupt of type *interrupt*, a signal with number *signal\_number* will be sent to the process calling this function. If the process wants to handle the signal it must install a signal handler. Installing a signal handler is not part of this API. The value 0 for *signal\_number* is used to “unregister” a signal from the VMEbus library/driver.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | A signal was successfully registered. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The interrupt is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Interrupts - Asynchronous Method.

Notes

none

VME\_InterruptInfoGet()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptInfoGet(int interrupt, VME\_InterruptInfo\_t\* interrupt\_info);

Parameters

|  |  |  |
| --- | --- | --- |
| int interrupt | in | identifier of the interruptobtained in call to VME\_InterruptLink() |
| VME\_InterruptInfo\_t\* interrupt\_info | out | information on VMEbus interrupts |

Description

The VME\_InterrupInfotGet() function returns information on the VMEbus interrupt received. The function can be called at any time. It returns *VME\_NOINTERRUPT* if no interrupt has been received.

The VME\_InterruptInfoGet() function must be called for each interrupt, either after a VME\_InterruptWait() function or in a signal handler associated to that interrupt using the VME\_InterruptRegisterSignal() function.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The interrupt information was successfully returned. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The interrupt is not known. |
| VME\_NOINTERRUPT | There has not been an interrupt; *interrupt\_info* is empty. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Interrupts - Asynchronous Method.

Notes

none

VME\_InterruptReenable()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptReenable(int interrupt);

Parameters

|  |  |  |
| --- | --- | --- |
| int interrupt  | in | identifier of the interruptobtained in call to VME\_InterruptLink() |

Description

The VME\_InterruptRenable() function re-enables the interrupt associated to *interrupt*, if the interrupt received came from a “Release-On-Register-Access” (RORA) interrupter.

The VME\_InterruptReenable() function must be called in case the interrupt received came from a RORA interrupter. If it came from a ROAK interrupter the interrupt will be automatically be re-enabled by the VMEbus library/driver.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The VMEbus interrupt was successfully enabled. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The interrupt is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Block Transfer - Detailed Functions.

Notes

The VMEbus configuration utility is used to associate VMEbus interrupt levels to either of the two different types of VMEbus interrupters. When the VMEbus library/driver receives an interrupt from a level which has been associated to RORA interrupters it disables that level. The application program will receive the interrupt after a call to the VME\_InterruptWait() function or using a signal handler previously installed with the VME\_Interrupt-Signal-Register() function. After handling the interrupt, the application program must call the VME\_InterruptReenable() function in order to re-enable the associated VMEbus level.

Enabling VMEbus interrupts generated by a VMEbus interrupter is independent of this function and not part of this API.

VME\_InterruptUnlink()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptUnlink(int interrupt);

Parameters

|  |  |  |
| --- | --- | --- |
| int interrupt  | in | identifier of the interruptobtained in call to VME\_InterruptLink() |

Description

The VME\_InterruptUnlink() function deletes the link between the VMEbus interrupts associated to *interrupt* and the application program. The identifier *interrupt* shall not be used after this function call.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The link to the interrupt was successfully deleted. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_NOTKNOWN | The interrupt is not known. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

none

VME\_InterruptGenerate()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptGenerate(u\_char vector, u\_int level);

Parameters

|  |  |  |
| --- | --- | --- |
| u\_char vector | in | VMEbus interrupt vector |
| u\_int level | in | VMEbus interrupt level |

Description

The VME\_InterruptGenerate() function generates a VMEbus interrupt at level *level* with vector *vector*. This function can be used in order to send an interrupt to another VMEbus interrupt handler.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The interrupt was successfully generated. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |
| VME\_IRGBUSY | The VMEbus interrupter is busy. |
| *others* | specific to the implementation |

Programming Example

For a programming example see Section Example 11: Interrupts - Generate Interrupts.

Notes

Some parameters for VMEbus interrupt generation, e.g. for the interrupt level, can be configured statically using the VMEbus configuration utility, see Section VMEbus Configuration Utility.

VME\_InterruptDump()

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_InterruptDump(void);

Parameters

none

Description

The VME\_InterruptDump() function dumps system parameters associated to interrupt handling and generation to “stdout”.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The status of the interrupt handling was successfully dumped. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |

Programming Example

For a programming example see Section Interrupts - Synchronous Method.

Notes

none

* Miscellaneous functions

VME\_SendSysreset

Synopsis

#include “vme\_rcc.h”

VME\_ErrorCode\_t VME\_SendSysreset(void);

Parameters

none

Description

The VME\_SendSysreset() function asserts the SYSRESET\* line on the VMEbus backplane for 200ms. Keep in mind that SYSRESET is seen by all VMEbus modules in the crate (including the module that sends it). It may therefore be necessary to mask SYSRESET on certain modules to avoid loss of configuration data or a reboot of a SBC.

Return Values

|  |  |
| --- | --- |
| VME\_SUCCESS | The status of the interrupt handling was successfully dumped. |
| VME\_NOTOPEN | The VMEbus library/driver was not opened. |

Programming Example

none

Notes

none

* Programming Examples
* Example 1: Functions for Return Codes

#include “vme\_rcc.h”

...

VME\_ErrorCode\_t error\_code;

char error\_string[VME\_MAXSTRING];

u\_int error\_number;

...

error\_code = VME\_Open();

if(error\_code != VME\_SUCCESS) {

 */\* compare error code to VME\_SUCCESS \*/*

 VME\_ErrorPrint(error\_code);

 */\* print error code to stdout \*/*

 return(error\_code);

}

...

error\_code = VME\_Close();

if(error\_code != VME\_SUCCESS) {

 */\* compare error code to VME\_SUCCESS \*/*

 VME\_ErrorString(error\_code,error\_string);

 */\* print error code to char string \*/*

 printf(“ERROR in example program: %s\n”,error\_string);

 return(error\_code);

}

...

error\_code = VME\_Close();

VME\_ErrorNumber(error\_code,error\_number);

 */\* convert error code to error number \*/*

if(error\_number == VME\_NOTOPEN) {

 */\* compare error number to return value \*/*

 printf(“ERROR in example program: already closed\n”);

 return(error\_code);

}

* Example 2: CR/CSR Space

#include “vme\_rcc.h”

...

int slot\_number = 5;

u\_int module\_identifier;

u\_int vmebus\_address = 0x22000000;

VME\_ErrorCode\_t error\_code;

...

if(error\_code = VME\_ReadCRCSR(slot\_number, VME\_CR\_MODULEID, &module\_identifier)) {

 */\* read from the CR/CSR space: e.g. module identifier \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

if(error\_code = VME\_WriteCRCSR(slot\_number, VME\_CSR\_ADER0, vmebus\_address)) {

 */\* write to CR/CSR space:*

 *e.g. base address to address decode comparator \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 3: Master Mapping - Safe Access

#include “vme\_rcc.h”

...

VME\_MasterMap\_t master\_map;

int master\_mapping;

u\_int value\_u\_int;

u\_int address\_offset = 0x200;

VME\_ErrorCode\_t error\_code;

u\_int error\_number;

...

master\_map.vmebus\_address = 0x22000000;

master\_map.window\_size = 0x00800000;

master\_map.address\_modifier = VME\_AM09;

master\_map.options = 0;

 */\* fill master mapping input information \*/*

if(error\_code = VME\_MasterMap(&master\_map, &master\_mapping)) {

 */\* create a new master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

if(error\_code = VME\_ReadSafeUInt(master\_mapping, address\_offest, &value\_u\_int)) {

 */\* read safely from the master mapping \*/*

 VME\_ErrorNumber(error\_code, &erorr\_number);

 if(error\_number != VME\_BUSERROR) {

 printf(“ERROR in example program: bus error\n”);

 }

 return(error\_code);

}

...

 */\* continued on next page \*/*

value\_u\_int = 0xFFFFFFFF;

if(error\_code = VME\_WriteSafeUInt(master\_mapping, address\_offset, value\_u\_int)) {

 */\* write safely to the master mapping \*/*

 VME\_ErrorNumber(error\_code, &erorr\_number);

 if(error\_number != VME\_BUSERROR) {

 printf(“ERROR in example program: bus error\n”);

 }

 return(error\_code);

}

...

VME\_MasterMapDump();

 */\* dump system parameters for all master mappings \*/*

...

if(error\_code = VME\_MasterUnmap(master\_mapping)) {

 */\* delete the master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 4: Master Mapping - Fast Access

#include “vme\_rcc.h”

...

VME\_MasterMap\_t master\_map;

int master\_mapping;

u\_int virtual\_address;

u\_int value\_u\_int;

u\_int address\_offset = 0x200;

VME\_ErrorCode\_t error\_code;

...

master\_map.vmebus\_address = 0x22000000;

master\_map.window\_size = 0x00800000;

master\_map.address\_modifier = VME\_AM09;

master\_map.options = 0;

 */\* fill master mapping input information \*/*

if(error\_code = VME\_MasterMap(&master\_map, &master\_mapping)) {

 */\* create a new master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

VME\_MasterMapVirtualAddress(master\_mapping, &virtual\_address);

 */\* get virtual address for the master mapping \*/*

...

VME\_ReadFastUInt(master\_mapping, address\_offset, &value\_u\_int);

 */\* read fast from the master mapping, ignore bus error \*/*

 */\* alternatively use \*/*

value\_u\_int = \*(u\_int \*)(virtual\_address + address\_offset);

...

value\_u\_int = 0xFFFFFFFF;

VME\_WriteSafeUInt(master\_mapping, address\_offset, value\_u\_int);

 */\* write fast to the master mapping, ignore bus error \*/*

 */\* alternatively use \*/*

\*(u\_int \*)(virtual\_address + address\_offset) = 0xFFFFFFFF;

...

 */\* continued on next page \*/*

if(error\_code = VME\_MasterUnmap(master\_mapping)) {

 */\* delete the master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 5: Master Mapping - Bus Error Handler

#include “vme\_rcc.h”

#include <signal.h>

...

void my\_bus\_error\_handler(int sig) {

  */\* bus error handler function \*/*

 static VME\_BusErrorInfo\_t bus\_error\_info;

 static u\_int error\_code;

 if(error\_code = VME\_BusErrorInfoGet(&bus\_error\_info) {

 */\* get information on bus error \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

 }

 printf(“ERROR in example program: bus error at address = %08x,

 am = %02x\n”,bus\_error\_info.vmebus\_address,

 bus\_error\_info.address\_modifier);

}

...

VME\_MasterMap\_t master\_map;

int master\_mapping;

u\_int value\_u\_int;

u\_int address\_offset = 0x200;

VME\_ErrorCode\_t error\_code;

...

master\_map.vmebus\_address = 0x22000000;

master\_map.window\_size = 0x00800000;

master\_map.address\_offset = VME\_AM09;

master\_map.options = 0;

 */\* fill master mapping input information \*/*

if(error\_code = VME\_MasterMap(&master\_map, &master\_mapping)) {

 */\* create a new master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

 */\* continued on next page \*/*

 */\* install bus error handler for signal,*

 *not part of this API, see function sigaction() \*/*

...

if(error\_code = VME\_BusErrorRegisterSignal(SIGBUS)) {

 */\* register signal for bus error handling \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

VME\_ReadFastUInt(master\_mapping, address\_offset, &value\_u\_int);

 */\* read fast from the master mapping,*

 *bus error will be caught by example bus error handler \*/*

...

if(error\_code = VME\_BusErrorRegisterSignal(0)) {

 */\* un-register signal for bus error handling \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

if(error\_code = VME\_MasterUnmap(master\_mapping)) {

 */\* delete the master mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 6: Slave Mapping

#include “vme\_rcc.h”

...

VME\_SlaveMap\_t slave\_map;

int slave\_mapping;

u\_int vmebus\_address;

VME\_ErrorCode\_t error\_code;

...

slave\_map.window\_size = 0x00800000;

slave\_map.address\_modifier = VME\_AM09;

slave\_map.options = 0;

 */\* fill master mapping input information \*/*

 */\* obtain contiguous, memory-locked and aligned user\_space,*

 *not part of this API \*/*

slave\_map.system\_iobus\_address = *my\_pci\_allocate*();

if(error\_code = VME\_SlaveMap(&slave\_map, &slave\_mapping)) {

 */\* create a new slave mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

VME\_SlaveMapVmebusAddress(slave\_mapping, &vmebus\_address);

 */\* get VMEbus address for the slave mapping,*

 *to be used by a VMEbus master \*/*

...

 */\* read from and write to user space,*

 *not part of this API \*/*

...

VME\_SlaveMapDump();

 */\* dump system parameters for all slave mappings \*/*

...

if(error\_code = VME\_SlaveUnmap(slave\_mapping)) {

 */\* delete the slave mapping \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 7: Block Transfer - Detailed Functions

#include “vme\_rcc.h”

...

u\_int pci\_address;

VME\_BlockTransferList\_t block\_transfer\_list;

int block\_transfer;

VME\_ErrorCode\_t status;

int remaining;

int time\_out = 10;

 */\* time-out about 10 msec \*/*

VME\_ErrorCode\_t error\_code;

char error\_string[VME\_MAXSTRING];

...

 */\* get contiguous, memory-locked and aligned user space,*

 *not part of this API \*/*

sys\_address = *my\_pci\_allocate*();

...

block\_transfer\_list.list\_of\_items[0].vmebus\_address = 0x22000200;

block\_transfer\_list.list\_of\_items[0].system\_iobus\_address = pci\_address;

block\_transfer\_list.list\_of\_items[0].size\_requested = 0x100;

block\_transfer\_list.list\_of\_items[0].control\_word = VME\_DMA\_D32R;

 */\* fill parameters for first block transfer \*/*

block\_transfer\_list.list\_of\_items[1].vmebus\_address = 0x23000200;

block\_transfer\_list.list\_of\_items[1].system\_iobus\_address = pci\_address + 0x100;

block\_transfer\_list.list\_of\_items[1].size\_requested = 0x100;

block\_transfer\_list.list\_of\_items[1].control\_word = VME\_DMA\_D32R;

 */\* fill parameters for second block transfer \*/*

block\_transfer\_list.number\_of\_items = 2;

 */\* total number of block transfers \*/*

if(error\_code = VME\_BlockTransferInit(&block\_transfer\_list, &block\_transfer) {

 */\* initialise block transfer \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

 */\* continued on next page \*/*

if(error\_code = VME\_BlockTransferStart(block\_transfer) {

 */\* start block transfer \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

if(error\_code = VME\_BlockTransferWait(block\_transfer, time\_out, &block\_transfer\_list) {

 */\* wait for block transfer \*/*

 for(i=0; i< block\_transfer\_list.number\_of\_items; i++) {

 if(!VME\_BlockTransferStatus(block\_transfer\_list,i,&status)) {

 */\* check status of each block transfer \*/*

 VME\_ErrorString(status,error\_string);

 printf(“ERROR in example program: block = %d, status = %s\n”,

 i,error\_string);

 }

 if(!VME\_BlockTransferRemaining(block\_transfer\_list,i,&remaining)) {

 */\* check remaining words of each block transfer \*/*

 printf(“ERROR in example program: block = %d, remaining = %d\n”,

 i,remaining);

 }

 }

 return(error\_code);

}

...

VME\_BlockTransferDump();

 */\* dump system parameters for all DMA engines \*/*

...

if(error\_code = VME\_BlockTransferEnd(block\_transfer) {

 */\* end block transfer \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 8: Block Transfer - Integrated Function

#include “vme\_rcc.h”

...

u\_int pci\_address;

VME\_BlockTransferList\_t block\_transfer\_list;

int block\_transfer;

VME\_ErrorCode\_t status;

int remaining;

int time\_out = 10;

 */\* time-out about 10 msec \*/*

VME\_ErrorCode\_t error\_code;

char error\_string[VME\_MAXSTRING];

...

 */\* get contiguous, memory-locked and aligned user space,*

 *not part of this API \*/*

sys\_address = *my\_pci\_allocate*();

...

block\_transfer\_list.list\_of\_items[0].vmebus\_address = 0x22000200;

block\_transfer\_list.list\_of\_items[0].system\_iobus\_address = pci\_address;

block\_transfer\_list.list\_of\_items[0].size\_requested = 0x100;

block\_transfer\_list.list\_of\_items[0].control\_word = VME\_DMA\_D32R;

 */\* fill parameters for first block transfer \*/*

block\_transfer\_list.list\_of\_items[1].vmebus\_address = 0x23000200;

block\_transfer\_list.list\_of\_items[1].system\_iobus\_address = pci\_address + 0x100;

block\_transfer\_list.list\_of\_items[1].size\_requested = 0x100;

block\_transfer\_list.list\_of\_items[1].control\_word = VME\_DMA\_D32R;

 */\* fill parameters for second block transfer \*/*

block\_transfer\_list.number\_of\_items = 2;

 */\* total number of block transfers \*/*

 */\* continued on next page \*/*

if(error\_code = VME\_BlockTransfer(&block\_transfer\_list, time\_out) {

 */\* integrated function for block transfer \*/*

 for(i=0; i< block\_transfer\_list.number\_of\_items; i++) {

 if(!VME\_BlockTransferStatus(block\_transfer\_list,i,&status)) {

 */\* check status of each block transfer \*/*

 VME\_ErrorString(status,error\_string);

 printf(“ERROR in example program: block = %d, status = %s\n”,

 i,error\_string);

 }

 if(!VME\_BlockTransferRemaining(block\_transfer\_list,i,&remaining)) {

 */\* check remaining words of each block transfer \*/*

 printf(“ERROR in example program: block = %d, remaining = %d\n”,

 i,remaining);

 }

 }

 return(error\_code);

}

* Example 9: Interrupts - Synchronous Method

#include “vme\_rcc.h”

...

VME\_InterruptList\_t interrupt\_list;

int interrupt;

VME\_InterruptInfo\_t interrupt\_info;

int time\_out = 100000;

 */\* time-out about 100 sec \*/*

VME\_ErrorCode\_t error\_code;

u\_int error\_number;

...

interrupt\_list.list\_of\_items[0].vector = 0x11;

interrupt\_list.list\_of\_items[0].level = 1;

interrupt\_list.list\_of\_items[0].type = VME\_INT\_RORA;

 */\* fill parameters for first interrupt \*/*

interrupt\_list.list\_of\_items[1].vector = 0x22;

interrupt\_list.list\_of\_items[1].level = 2;

interrupt\_list.list\_of\_items[1].type = VME\_INT\_ROAK;

 */\* fill parameters for second interrupt \*/*

interrupt\_list.number\_of\_items = 2;

 */\* total number of interrupts \*/*

if(error\_code = VME\_InterruptLink(&interrupt\_list, &interrupt) {

 */\* link VMEbus interrupt list to application program \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

 */\* continued on next page \*/*

if(error\_code = VME\_InterruptWait(interrupt, time\_out, &interrupt\_info){

 */\* wait for interrupt \*/*

 VME\_ErrorNumber(error\_code,error\_number);

 */\* convert error code to error number \*/*

 if(error\_number == VME\_TIMEOUT) {

 */\* compare error number to return value \*/*

 printf(“ERROR in example program: no interrupt in 100 sec\n”);

 }

 else {

 VME\_ErrorPrint(error\_code);

 }

 return(error\_code);

}

if(error\_code = VME\_InterruptInfoGet(&interrupt\_info) {

 */\* get information on interrupt \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

if(interrupt\_info.level == 1) {

 */\* interrupt from a level assigned to RORA interrupters? \*/*

 if(error\_code = VME\_InterruptRenable(interrupt) {

 */\* re-enable interrupt => can wait again on interrupt \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

 }

}

...

VME\_InterruptDump();

 */\* dump system parameters for all VMEbus interrupts \*/*

...

if(error\_code = VME\_InterruptUnlink(interrupt) {

 */\* unlink VMEbus interrupt list from application program \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 10: Interrupts - Asynchronous Method

#include “vme\_rcc.h”

#include <signal.h>

...

int global\_interrupt;

...

void my\_interrupt\_handler(int sig) {

  */\* interrupt handler function \*/*

 static VME\_InterruptInfo\_t interrupt\_info;

 static VME\_ErrorCode\_t errod\_code;

 if(error\_code = VME\_InterruptInfoGet(global\_interrupt, &interrupt\_info) {

 */\* get information on interrupt \*/*

 VME\_ErrorPrint(error\_code);

 return;

 }

 printf(“INTERRUPT in example program: vector =%02x, multiple =

 %d\n”, interrupt\_info.vector,

 interrupt.multiple);

 if(interrupt\_info.level == 1) {

 */\* interrupt from a level assigned to RORA interrupters? \*/*

 if(error\_code = VME\_InterruptRenable(global\_interrupt) {

 */\* re-enable interrupt => can wait again on interrupt \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

 }

 }

...

}

...

VME\_InterruptList\_t interrupt\_list;

VME\_InterruptInfo\_t interrupt\_info;

int time\_out = 100000;

 */\* time-out about 100 sec \*/*

VME\_ErrorCode\_t error\_code;

u\_int error\_number;

...

 */\* continued on next page \*/*

interrupt\_list.list\_of\_items[0].vector = 0x11;

interrupt\_list.list\_of\_items[0].level = 1;

interrupt\_list.list\_of\_items[0].type = VME\_INT\_RORA;

 */\* fill parameters for first interrupt \*/*

interrupt\_list.list\_of\_items[1].vector = 0x22;

interrupt\_list.list\_of\_items[1].level = 2;

interrupt\_list.list\_of\_items[1].type = VME\_INT\_ROAK;

 */\* fill parameters for second interrupt \*/*

interrupt\_list.number\_of\_items = 2;

 */\* total number of interrupts \*/*

if(error\_code = VME\_InterruptLink(&interrupt\_list, &global\_interrupt) {

 */\* link VMEbus interrupt list to application program \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

if(error\_code = VME\_InterruptRegisterSignal(global\_interrupt,SIGBUS) {

 */\* register SIGBUS signal for VMEbus interrupt list \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

...

 */\* install example interrupt handler for SIGBUS signal,*

  *not part of this API, see function sigaction() \*/*

 */\* VMEbus interrupts will be caught asynchronously by*

 *example interrupt handler \*/*

...

if(error\_code = VME\_InterruptRegisterSignal(global\_interrupt,0) {

 */\* (un-)register signal for VMEbus interrupt list \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

if(error\_code = VME\_InterruptUnlink(global\_interrupt) {

 */\* unlink VMEbus interrupt list from application program \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* Example 11: Interrupts - Generate Interrupts

#include “vme\_rcc.h”

#include <signal.h>

...

u\_int level = 1;

u\_int vector = 0x11;

VME\_ErrorCode\_t error\_code;

...

if(error\_code = VME\_InterruptGenerate(vector, level) {

 */\* generate VMEbus interrupt \*/*

 VME\_ErrorPrint(error\_code);

 return(error\_code);

}

* VMEbus Utility Programs

Two utility programs can accompany the API implementation: a utility program to configure the VMEbus statically must accompany the VMEbus API, if the implementation requires this. A utility program to test and debug the VMEbus (and the API implementation) and a facility to scan the VMEbus address space shall always accompany the VMEbus API.

* VMEbus Configuration Utility

The VMEbus configuration utility (e.g. named “*vmeconfig”*), if required by the API implementation, is used to configure some static parameters necessary to use the VMEbus. It is used, in particular, to configure the following parameters:

* static mapping parameters for VMEbus CR/CSR space;
* static mapping parameters for VMEbus master and slave mappings;
* byte swapping capabilities of the VMEbus bridge;
* enabling and disabling of VMEbus interrupt levels, specifying of the associated VMEbus interrupter types (ROAK or RORA), required for handling of VMEbus interrupts.

The VMEbus configuration utility is intended to be run at boot time when the ROD Crate Processor is started.

* VMEbus Test and Debug Utility

The VMEbus test and debug facility (e.g. called “*vmescope”*) must allow to test and debug the VMEbus (and the API implementation). It has, in particular, to provide means to perform the following functions:

* dump system parameters of the VMEbus bridge;
* create VMEbus master and slave mapping and read and write single values;
* perform VMEbus block transfers;
* receive VMEbus interrupts.
* VMEbus Scanning Facility

The VMEbus scanning facility (e.g. called “*vmescan*”) must allow to scan the VMEbus and to report any found VMEbus modules. It has, in particular, to perform the following functions:

* scan the whole VMEbus address space;
* scan the whole VMEbus CR/CSR space.
* Ideas for a C++ Binding

This section presents some first ideas on a possible C++ binding or wrapping of the VMEbus API. It shows the public members of the classes along with some of the private members.

There are two levels of VMEbus related classes:

* VME class:

The VME class is a singleton which is generated and deleted using static methods. It contains some members for return codes, CR/CSR access, bus error handling and printing of general information. The VME class is used to generate and delete all other VMEbus related classes; in that sense it is a factory of the other VMebus related classes.

* VMEMasterMap, VMESlaveMap, VMEBlockTransfer and VMEInterrupt classes:

Those classes are used for master mappings and single cycles, slave mappings, block transfers and interrupts, respectively. They correspond to the identifiers used in the C binding, i.e. *master\_mapping*, *slave\_mapping*, *block\_transfer* and *interrupt*. Their constructors correspond to the functions returning an identifier, their destructors to those invalidating the identifier.

* Types

// bus error information

typedef VME\_BusErrorInfo\_t VMEBusErrorInfo;

// VMEbus interrupt information

typedef VME\_InterruptList\_t VMEInterruptList;

typedef VME\_InterruptInfo\_t VMEInterruptInfo;

// block transfer

typedef VME\_BlockTransferItem\_t VMEBlockTransferItem;

typedef VME\_BlockTransferList\_t VMEBlockTransferList;

typedef VME\_BlockTransferInfo\_t VMEBlockTransferInfo;

* VMEbus library/driver

class VME {

 public:

 // singleton members

 static VME\* Open();

 static u\_int Close();

 // members for return codes

 static int ErrorPrint(u\_int error\_code);

 static int ErrorString(u\_int error\_code, string\* error\_string);

 static int ErrorNumber(u\_int erro\_code, u\_int\* error\_number);

 // members for CR/CSR access

 u\_int ReadCRCSR(int slot, u\_int crcsr\_field, u\_int\* value);

 u\_int WriteCRCSR(int slot, u\_int crcsr\_field, u\_int data);

 // members for bus error handling

 u\_int BusErrorRegisterSignal(int signal\_number);

 u\_int BusErrorInfoGet(VME\_BusErrorInfo& bus\_error\_info);

 // factory members

 VMEMasterMap\* MasterMap(u\_int vmebus\_address, u\_int window\_size
 u\_int address\_modifier, u\_int options);

 u\_int MasterUnmap(VME\_MasterMap\* master\_map);

 VMESlaveMap\* SlaveMap(u\_int system\_iobus\_address, u\_int window\_size,
 address\_width, u\_int options);

 u\_int SlaveUnmap(VME\_SlaveMap\*slave\_map);

 VMEBlockTransfer\* BlockTransfer(const VMEBlockTransferList&
 block\_transfer\_list);

 u\_int BlockTransferDelete(VME\_BlockTransfer\* block\_transfer);

 VMEInterrupt\* Interrupt(const VMEInterruptList&
 interrupt\_list);

 u\_int InterruptDelete(VMEInterrupt\* interrupt);

 u\_int InterruptGenerate(u\_char vector, u\_int level);

 // status dumps

 u\_int MasterMapDump() const;

 u\_int SlaveMapDump() const;

 u\_int BlockTransferDump() const;

 u\_int InterruptDump() const;

 // continued on next page

 private:

 VME();

 ~VME();

 static VME\* my\_instance;

 static int my\_users;

 // internals

 ...

};

* VMEbus Master Mapping

class VMEMasterMap {

 public:

 // members for safe access

 u\_int ReadSafe(u\_int address\_offset, u\_int\* value);

 u\_int WriteSafe(u\_int address\_offset, u\_int data);

 u\_int ReadSafe(u\_int address\_offset, u\_short\* value);

 u\_int WriteSafe(u\_int address\_offset, u\_short data);

 u\_int ReadSafe(u\_int address\_offset, u\_char\* value);

 u\_int WriteSafe(u\_int address\_offset, u\_char data);

 // members for fast access

 inline void ReadFast(u\_int address\_offset, u\_int\* value);

 inline void WriteFast(u\_int address\_offset, u\_int data);

 inline void ReadFast(u\_int address\_offset, u\_short\* value);

 inline void WriteFast(u\_int address\_offset, u\_short data);

 inline void ReadFast(u\_int address\_offset, u\_char\* value);

 inline void WriteFast(u\_int address\_offset, u\_char data);

 // helpers

 u\_int VirtualAddress(u\_int\* virtual\_address) const;

 u\_int Dump() const;

 // operator to return status of object

 u\_int operator()();

 // friends

 friend class VME;

 private:

 VME\_MasterMap(u\_int vmebus\_address, u\_int window\_size,
 u\_int address\_modifier, u\_int options);

 ~VME\_MasterMap();

 int my\_identifier;

 VME\_MasterMap\_t my\_master\_map;

 int my\_status;

 // internals

 ...

};

* VMEbus Slave Mapping

class VMESlaveMap {

 public:

 // helpers

 u\_int VmebusAddress(u\_int\* vmebus\_address) const ;

 u\_int Dump() const;

 // operator to return status of object

 u\_int operator()();

 // friend

 friend class VME;

 private:

 VME\_SlaveMap(u\_int system\_iobus\_address, u\_int window\_size,
 u\_int address\_width, u\_int options);

 ~VME\_SlaveMap();

 int my\_identifier;

 VME\_SlaveMap\_t my\_slave\_map;

 u\_int my\_status;

 // internals

 ...

};

* VMEbus Block Transfer

class VMEBlockTransfer {

 public:

 // main members

 u\_int Start();

 u\_int Wait(int time\_out);

 // helpers

 u\_int Status(int position\_of\_block, u\_int\* status);

 u\_int Remaining(int position\_of\_block, int\* remaining);

 u\_int Dump() const;

 // operator to return status of object

 u\_int operator()();

 // friend

 friend class VME;

 private:

 VME\_BlockTransfer(const VMEBlockTransferList&
 block\_transfer\_list);

 ~VME\_BlockTransfer();

 int my\_identifier;

 VMEBlockTransferList my\_block\_transfer\_list;

 u\_int my\_status;

 // internals

 ...

};

* VMEbus Interrupts

class VMEInterrupt {

 public:

 // main members

 u\_int Wait(int time\_out, VMEInterruptInfo& interrupt\_info);

 u\_int SignalRegister(int signal\_number);

 u\_int InfoGet(VMEInterruptInfo& interrupt\_info);

 u\_int Reenable();

 // helper

 u\_int Dump() const;

 // operator to return status of object

 u\_int operator()();

 // friend

 friend class VME;

 private:

 VME\_Interrupt(const VMEinterruptList& interrupt\_list);

 ~VME\_Interrupt();

 int my\_identifier;

 VMEInterruptListt my\_interrupt\_list;

 u\_int my\_status;

 // internals

 ...

};