ATLAS Internal Note
ATL-D-ES-0004
Version 1.5
17 June 2002

VMEbus Application Program Interface

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

comments and queries to Ralf Spiwoks, CERN
+41 22 767 3871
Ralf.Spiwoks@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

1.1 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-D-ES-0007) 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.

1.2 Design Issues

A few notes on the design guidelines of the API:

1. 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.
2. Names
   - The API uses readable and meaningful names for all of its functions, as well as the common prefix “VME_”.
3. 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.
4. 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.

5. Known limitations
   - No Read-Modify-Write functions are defined in the API. Those can be added later if needed.
   - Each VMEbus vector can only be used by one process.
   - No functions are defined in the API to notify the application program of VMEbus failures, e.g. signalled by SYSFAIL or ACFAIL. Those can be added later if needed.

1.3 Implementation Issues

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

1. 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.

2. 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 4.

3. 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.

4. 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 2.5 and 2.9).

5. 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.

6. 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 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 4.1. The same utility will also be used to statically activate the interrupt levels.

7. 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.

8. 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.

9. Language binding

   C was chosen for the language binding of the API as presented in Sections 2 and 3. Some ideas on a possible C++ language binding or wrapping are presented in Section 5.

10. 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 2.6:

   - "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.

1.4 Organization of this Document

Section 2 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 3 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 4 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 5 gives some ideas on a possible C++ language binding or wrapping of the API. The public members of the classes are shown.

2 Application Program Interface

2.1 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_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

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”.

2.2 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 2.6, page 17;

VME_BusErrorInfo_t see Section 2.7, page 27;

VME_SlaveMap_t see Section 2.8, page 29;

VME_BlockTransferItem_t see Section 2.9, page 34;
VME_BlockTransferList_t see Section 2.9, page 36;

VME_InterruptItem_t see Section 2.10, page 45;
VME_InterruptList_t see Section 2.10, page 46;
VME_InterruptInfo_t see Section 2.10, page 49;

2.3 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 3.1.

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 3.1.

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 3.1.

Notes

none

2.4 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 3.2.

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 3.2.

Notes

none

2.5 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 3.2.

Notes

The mapping of the CR/CSR space can be configured statically using the VMEbus configuration utility, see Section 4.1.

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 3.2.

Notes

The mapping of the CR/CSR space can be configured statically using the VMEbus configuration utility, see Section 4.1.

2.6 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 3.3.

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 3.3.

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 4.1.

VME_MasterMapVirtualAddress()

Synopsis

```
#include "vme_rcc.h"
VME_ErrorCode_t VME_MasterMapVirtualAddress(int master_mapping, u_int*
virtual_address);
```

Parameters

int master_mapping	in	identifier of master mapping obtained in call to VME_MasterMap()
u_int* virtual_address	out	virtual address associated to the master mapping

Description

The VME_MasterMapVirtualAddress() function returns the virtual address associated to *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 3.4.

Notes

none

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 master mapping

Description

The VME_ReadSafeXXX() functions reads safely an "unsigned int", "unsigned short", or "unsigned char" value from *master_mapping* at *address_offset.* The functions check if a VME-bus 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 3.3.

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_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 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_WriteSafeXXX() functions write safely an "unsigned int", "unsigned short", or "unsigned char" value to *master_mapping* 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 3.3.

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 2.7.

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 3.4.

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 2.7.

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 3.4.

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 3.3.

Notes

none

VME_MasterMapDump()

Synopsis

```
#include "vme_rcc.h"
VME_ErrorCode_t VME_MasterMapDump(void);
```

Parameter

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 3.3.

Notes

none

2.7 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 VME-bus library/driver. In case the VMEbus library/driver detects a VMEbus error and the VMEbus error did <u>not</u> 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 3.5.

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 3.5.

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 3.5.

Notes

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

2.8 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 3.6.

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 3.6.

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 4.1.

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 mapping obtained 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 3.6.

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 mapping obtained 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 3.6.

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 3.6.

Notes

none

2.9 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 3.7.

Notes

The block transfer list used by the application program can be independent of an 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_t list_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 3.7.

Notes

A single block transfer must use a block transfer list with only one VME_BlockTransferItem_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 VME-bus 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 3.7.

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 transfer obtained 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 3.7.

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 transfer obtained 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_BlockTransferRemaining() 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 3.7.

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 transfer obtained 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 3.7.

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:

1. VME_BlockTransferInit(),
2. VME_BlockTransferStart(),
3. VME_BlockTransferWait() and
4. 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 3.8.

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_BlockTransferEnd() 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 3.7.

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 3.7.

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 3.7.

Notes

none

2.10 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 3.9.

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 4.1. 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 3.9.

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 program because an interrupt is in conflict with the static configuration.
VME_INTUSED	The list of interrupts cannot be linked to the application program because an interrupt is already being used.
others	specific to the implementation

Programming Example

For a programming example see Section 3.9.

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 3.10.

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 interrupt obtained 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 3.9.

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 interrupt obtained 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 3.10.

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 interrupt obtained 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 3.10.

Notes

none

VME_InterruptReenable()

Synopsis

```
#include "vme_rcc.h"
VME_ErrorCode_t VME_InterruptReenable(int interrupt);
```

Parameters

int interrupt	in	identifier of the interrupt obtained 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 3.7.

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_InterruptSignalRegister() 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 interrupt obtained 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 3.9.

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 3.11.

Notes

Some parameters for VMEbus interrupt generation, e.g. for the interrupt level, can be configured statically using the VMEbus configuration utility, see Section 4.1.

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 3.9.

Notes

none

3 Programming Examples

3.1 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);
}
```

3.2 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);
}
```

3.3 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);
}
```

3.4 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);
}
```

3.5 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);
}
```

3.6 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);
}
```

3.7 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);
}
```

3.8 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);
}
```

3.9 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);
}
```

3.10 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 u_int                   errod_code;

    if(error_code = VME_InterruptInfoGet(&interrupt_info) {
             /* get information on interrupt */

        VME_ErrorPrint(error_code);
        return(error_code);
    }

    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);
}
```

3.11 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(level, vector) {
        /* generate VMEbus interrupt */

    VME_ErrorPrint(error_code);
    return(error_code);
}
```

4 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.

4.1 VMEbus Configuration Utility

The VMEbus configuration utility (“*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.

4.2 VMEbus Test and Debug Utility

The VMEbus test and debug facility (“*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.

4.3 VMEbus Scanning Facility

The VMEbus scanning facility (“*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.

5 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:

1. **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.

2. **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.

5.1 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;
```

5.2 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
    ...
};
```

5.3 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
        ...
};
```

5.4 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
        ...
};
```

5.5 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
        ...
};
```

5.6 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
        ...
};
```