MIZZI Computer Software

VME

Utilities/Libraries

Documentation

Version

```
Version: 1.5.1
Date:    97/01/07
```

Copyright

VME Utilities/Libraries

General

The module supports the following features:

VME Bus access

The library allows to a access the VME bus in a system independent way, whenever the VME bus is mappable by the user program.

VME Interrupts

VME interrupts are supported and allow a user to specify an interrupt service routine which is called if a VME interrupt occurs.

VME Bus Traps

If the system supports bus trap error handling, macros will be defined to handles VME bus errors via a standard C setjmp longjmp interface.

Compatibility

VMElib 1.3 is compatible with Version 1.0, 1.1, 1.2. However some of the macro names has been changed for better readability. The old macro names are still supported.

For future unix support you should include the header file vme.h with lowercase characters instead of VME.h.

New functions must be called to initialize the VME bus (for unix and memory protected systems). Systems for which VMElib 1.0 used to run on, this initialization is actually not necessary. However for future compatibility and portability you should implement these calls in your program anyway.

Version 1.2 has changed returned values from vme a16/24 init functions and VMEinterrupt.

Version 1.3 has changed functions for initialization and bus access. These functions are necessary for future RISC processor support.

VME Utilities/Libraries

Include

Using the VME library you must include the following file:

```
#include "vme.h"
```

If this header file is included you should NOT define any symbol beginning with VME or vme because the VME library will define some global data, functions and constants which will cause an error if this data is redefined or accessed in an unauthorized manner.

Linking

Linking your program with the VME library will be defined in the system specific implementation documentation.

For Most systems VMElib consists of a library with the name

```
vme.r or vme.l or vme.a or vme.o
```

which must be linked with your code.

Online Help

For online help a file vme.d exists which contains the information shown in this manual as an ASCII readable file.

VME Utilities/Libraries

Typedefs for VME Bus Access

```
vmebyte      unsigned type to store a D8  data
vmeword      unsigned type to store a D16 data
vmelong      unsigned type to store a D32 data
vmedata      unsigned type to store a D32 data
vmeaddress   unsigned type to store a VME address
```

Initializing the VME bus

Nine functions are used to initialize the VME bus:

These function must be called first before
any VME access can be done by the VME library.

The functions return a virtual address of the VME bus
which must be used when accessing the bus.

Access of the bus is done via macros.

Note

Some systems do not require to initialize the VME
bus because they have mapped the VME bus statically.
However if you decide to write system independent code
you must use these functions to initialize the VME
library.

Functions

```
vmebyte *vmea16d8init(adr,len);
vmeword *vmea16d16init(adr,len);
vmelong *vmea16d32init(adr,len);
vmebyte *vmea24d8init(adr,len);
vmeword *vmea24d16init(adr,len);
vmelong *vmea24d32init(adr,len);
vmebyte *vmea32d8init(adr,len);
vmeword *vmea32d16init(adr,len);
vmelong *vmea32d32init(adr,len);

vmeaddress adr;
vmeaddress len;
```

Description

Initialize

A16D8,A16D16,A16D32,
A24D8,A24D16,A24D32,
A32D8,A32D16,A32D32

VME address space.

Parameters

```
adr      is the first address to be initialized and
adr+len  is the last address+1.
```

Return

The Function returns the virtual address of the VME bus.
or 0 if the VME bus can not be initialized.

Note

The functions can be called more than once to specify more than one section. Sections may overlap and can be initialized more than once.

Initializing the VME bus does not access any VME memory space rather than it makes the VME access available to the user program.

Access to the VME bus

Access to the VME bus is ONLY possible for short, standard and extended modes and your modules must support supervisor and user access. Some systems will access the modules in supervisor mode only other in user (normal program execution) and in supervisor mode (during interrupt processing). Make sure that your modules are configured for both modes or check what modes your CPU is using when accessing the VME bus.

Nine macros are defined to read and write the VME bus.

The macros are not defined with a functional interface because some systems will produce code which slows down the VME bus access by more than 50% if a functional interface is used for the macro definitions.

Macros

```
vmea16d8(adr)     A16D8   VME bus access
vmea16d16(adr)    A16D16  VME bus access
vmea16d32(adr)    A16D32  VME bus access
vmea24d8(adr)     A24D8   VME bus access
vmea24d16(adr)    A24D16  VME bus access
vmea24d32(adr)    A24D32  VME bus access
vmea32d8(adr)     A32D8   VME bus access
vmea32d16(adr)    A32D16  VME bus access
vmea32d32(adr)    A32D32  VME bus access
```

Description

Once you have initialized the VME bus, you can access the bus with the macros shown above.

The macros can be used in expresions and left values. If used in an expression the VME bus is accessed with a read. If used as a left value the macros performs a write to the VME bus.

E.g.:

```
vmeword i=123;
vmea16d16(a)=i    /* vme write */
i=vmea16d16(a)    /* vme read */
```

Parameter

adr is the address returned by the corresponding VME initialization function.

adr can be incremented and you can add values to adr to get the next words from the VME bus. See Examples for more information.

Expression Values

All macros produce an expression which read or write
the bus as an unsigned integer.

E.g.:

```
vmeword *mymodule=vmea24d16init(0xff,10); /* offset */
vmea24d16(mymodule)=3.12;
```

converts 3.12 to an unsigned integer (3) and writes
it to address 0xff with a A24D16 VME access.

Example

The following example checks if a csr register
is set to 1 and writes to a data register the value 255.

```
vmeword *mymodule=vmea24d16init(0xff,10); /* offset */
vmeword *mycsr=(mymodule+0);       /* get csr adr */
vmeword *myrreg=(mymodule+1);      /* get other registers */
vmeword *mywreg=(mymodule+2);      /* at base+2 and base+4 */

/* note that mymodule+1 will access the VME bus at 0xff+2 */
/* because mymodule is declared as a word pointer */

if(vmea24d16(mycsr)==0x1) vmea24d16(myrreg)=255;
```

Note

Some C-Compilers (e.g. gcc) produce strongly optimized
code and will remove VME bus access instructions
if polling on registers is used in your program.

When using the macros above these optimizations are
suppressed and all accesses to the VME bus are performed.

However you should carefully write programs as well as
expressions which access the VME bus twice because you
never know in which sequence the instructions are
performed. You must use standard C techniques to avoid
this problems.

Restrictions

A32 access is not supported on all CPUs and sometimes
restricted to a limited range of addresses. See
implementation specific information for more information.
If your system does not have an VME P2 connector (lower
connector of the VME bus) the upper 16 Bits in a D32 access
are undefined due to the non existing D16-D31 data lines on
the VME bus.

Dirty tricks

The returned address of the vme...init functions is a pointer to a variable representing the type of data access. Sometimes you may wish to increment these addresses in bytes rather than in words as shown above. In such cases you should use the following type casts:

```
/* get offset */

vmeaddress mymodule=(vmeaddress)vmea24d16init(0xff,10);

/* access the bus at base+4 */

i=vmea24d16(mymodule+4);
```

However mymodule++ do not increment the vmeaddress to the next D16 word rather than it increments to the next byte which can not be accessed by a D16 access.

High Speed Data Transfers

If you want to read or write the VME bus at high speeds you should use the following code:

```
/* read 10 words at address a to data array d with A24D16 */

highspeedfunction(a,d)
vmeword *a;
vmeword *d;
{
  register vmeword *adr=a;
  register vmeword *data=d;
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
  *data++ = vmea24d16(adr++);
}
```

Bus Traps

```
If the system supports bus trap error handling,
macros are defined to handle VME bus errors via
a standard C setjmp longjmp interface.
```

Macro

```
vmebustrap(i)
```

Description

```
Enable, disable bus trap error handling.
```

Parameters

```
vmebustrap if called with i=1 returns zero and enables
bus trap handling. vmebustrap(0) resets the action to
system defaults which normally terminates the program
with an error if a bus error occurs.
```

Return

```
If the bus trap is enabled and a bus error occurs execution
continues as if the vmebustrap had just returned the value 1.
See examples for more information.
```

Note

```
On most system the code continues as described above
if a normal memory access violation is detected (e.g.
accessing a memory area not defined in the user address
space).
```

Restrictions

```
1.) vmebustrap is very critical to use because it
can not be implemented on all systems. If the macros
can not be implemented they are ignored and programs
will terminate with an error if a bus trap occurs.
2.) You must reset bus traps before you return from a
function which has set up a bus trap. Otherwise the
program will crash if a bus trap occurs.
3.) You can not use bus traps inside interrupt service
routines.
```

See Also

```
Standard C setjmp, longjmp functions.
```

Interrupt Processing

Interrupts are handled via asynchronous interrupt service routines. These routines are called if an interrupt is generated on the VME bus.

Function

```
int vmeinterrupt(vector,routine,parameter)

int vector;
void (*routine)();
int parameter;
```

Description

Connect the given routine to an interrupt vector.

Parameters

vector: Vector number to be connected 1..255

routine: Routine to be called for an interrupt service

parameter: Parameter to be passed to the routine as first argument if an interrupt occurs.

Return

The function returns the parameter vector if the interrupt vector is installed otherwise 0 will be returned.

Note

If routine is a NULL pointer the old interrupt service is restored. You must restore the old vector before the program terminates. Otherwise the system may crash on some systems if the interrupt service routine is removed from memory before interrupts are disabled.

Restrictions and Side Effects

In some system environments (e.g. OS9) interrupts are connected in system state and some restrictions are applied to this systems:

1.) If an other process has already used the vector it will be overwritten by this routine. At the moment no plan exists to overcome this restriction.

2.) If the interrupt service is delivered in system state only data manipulation is allowed in the interrupt service routine. Functions like printf can NOT be used for debugging!

Systems which scheduled interrupts in user state (e.g LynxOS) does not have such restrictions. You can use printf and other functions for debuging but you must take care of the restrictions applied to threads as described in the System Specific Information.

Interrupt vectors are sometimes used for system specific devices like disks and tapes and terminal lines. If you overwrite such vectors with your own interrupt services your system will crash or hang. Specially some processors like the 680xx will use interrupt vectors to signal memory faults, floating point faults and traps to handle system calls (most often vectors below 64 are used for such vectors). If you use such a vector the system will crash immediatly (most often because the timer interrupt does not work). A good idea is not to use vectors below 64 and to check vectors above 64 carefully. When ever possible you should use vectors above 200 which are most often unused by the operating system.

Function

```
int vmepriority(vector,prio)

int vector;
int prio;
```

Description

Sets the priority for a interrupt service routine.

Parameter

vector: VME interrupt vector.

prio: Priority relative to the caller.

Return

The function returns the absolute priority of the VME interrupt service routine which is in any case a positive number in the range 1...1000.

Note

vmepriority uses only relative priorities when computing absolute priorities of the interrupt service routines. The scheduling policy is: higher priorities are running before lower priorities.

Example:

If your program is running at priority 17 and you call

```
vmepriority(240, 1);
vmepriority(241, 2);
vmepriority(242,-1);
```

interrupt vector 240 is running with priority 18 and vector 241 with priority 19. Vector 242 is set up with priority 16 and runs only if your main program is not computing.

Due to the limited priority range you must check with the returned value of vmepriority whether or not the interrupts are scheduled with different priorities. To get the priority of the caller you may call vmepriority(vector,0). After checking the returned value you call vmepriority again. You can change the VME priority as often as you want but you must set the priority before you set up the interrupt with vmeinterrupt. Note that vmepriority does not change the priority of an interrupt which was previously connected with vmeinterrupt. vmepriority only sets the priority for further vmeinterrupt calls.

Side Effects

The function is only supported for systems which schedule an interrupt service not in system state. Systems which runs interrupts in system state take the priority from the VME bus interrupt priority directly. Higher VME priorities are running before lower priorities. Interrupts with not different priorities are acknowledged first as closer they are to the CPU module.

If the system is using the VME bus priority the function returns 0 in any case.

Function

```
int vmeirq(vector,irqlevel)

int vector;
int irqlevel;
```

Description

vmeirq is used to specify a VME interrupt level for a given interrupt vector. This is nessesary if your VME module supports ROR (reset on register) interrupts only. In case that the VME interrupts will run on IRQ level 0 (this is the default) the system will hang because the interrupt is not turned off during the interrupt processing. You must specify an IRQ level at least with the level your module is requesting to hold off the interrupts from the CPU. If your module is releasing interrupts with the ROA (reset on acknowledge) method you must not take care of this behavier because in such cases the module must be programmed to allow further interrupts at the end of the interrupt service routine. But however you can use vmeirq to disable VME interrupts for lower levels during processing of an interrupt. In any case you should use ROA interrupts if possible.

Parameters

vector: VME interrupt vector.

irqlevel: VME IRQ level the interrupt processing should run.

Return

The function returns the VME IRQ level used for for the given interrupt vector.

Note

If you abort your program with interrupts running at IRQ level other than zero and the interrupts are not disabled (by programming the VME modules) the system will disable all interrupts with lower levels until you set up a new interrupt service for the blocked vectors. This is nessesary because otherwise the system would block.

The function must be called before vmeinterrupt is called to set up an interrupt services routine.

Side Effects

The function is only supported for systems which schedule an interrupt service not in system state. Systems which runs interrupts in system state are using the IRQ level of the requested interrupt. In this case the function returns 0.

Function

```
int vmestacksize(vector,stack)

int vector;
int stack;
```

Description

Setup stack size for interrupt services routines.

Parameters

vector: VME interrupt vector.

stack: Stack size in bytes to use for interrupt vectors given by vector.

Return

The function returns the stack size in bytes used for for the given interrupt vector.

Note

If stack is set to 0 the stack is set to a default value.

Note that vmestacksize does not change the stacksize of an interrupt service routine which was previously connected with vmeinterrupt. vmestacksize only sets the stacksize for further vmeinterrupt calls.

Side Effects

The function is only supported for systems which schedule an interrupt service not in system state. Systems which runs interrupts in system state are using the stack of the system. In this case the function returns 0.

Debugging

There is one function to show the internal state of the VME library.

Function

```
int vmedebug(what)

int what;
```

Description

Shows connected interrupt vectors, the current interrupt handler, the addresses of service routines, old services, and number of interrupts delivered.

This function does not show the same information on all systems. It is used to show information about the state of the library and can not be called from interrupt service routines.

Parameters

The parameter what is currently ignored and should be specified as 0 for future use.

Return

The Function returns 0.

Examples

Two examples are shown below:

See files vmeinterrupt.c and vmebustrap.c for similar programs.

Bus Trap Error Handling

This program enables bus trap handling and is stopped if a bus trap occurs.

Code

```
#include "vme.h"

main()
{
long i,i1;

vmea24init(0,0x1000000);

if(vmebustrap(1))
{
  printf("buserr detected\n");
  vmebustrap(0);
  exit(0);
}

while( printf("enter addr: "),scanf("%ld",&i) )
{
  i1=vmea24d32(i);
  printf("no buserr detected %ld\n",i1);
  continue;
}

}
```

Wait for interrupt

```
Wait for interrupt 248.

This example runs only if you can create a VME interrupt
with vector 248 which is reset automatically by the
interrupt acknowledge signal.
```

Code

```
#include <stdio.h>
#include "vme.h"

static int interr=0;
static int iparam=0;

int service(ivec)
{
interr+=1;
iparam=ivec;
return 0;
}

main()
{
int count=0;

vmea24init(0,1024);
vmea16init(0,0x10000);
printf("Installing interrupts ...\n");
vmestacksize(248,10000);
vmepriority(248,10);
vmestacksize(249,0);
vmepriority(249,-1);
vmeinterrupt(248,service,1000);
vmeinterrupt(249,service,1001);
vmeinterrupt(250,service,1002);
vmeinterrupt(251,service,1003);
vmeinterrupt(252,service,1004);

vmedebug(0);

for(count=1;(!interr) && count <= 10; count ++)
{
  sleep(1);
  printf("waiting for interrupt %10d\n",count);
}

sleep(1);
if(interr) printf("\nInterrupt detected!\n");
printf("\nInterrupt calls = %d  Parameter = %d\n",interr,iparam);

printf("Removing interrupts ...\n\n");
vmeinterrupt(248,0,0);
vmeinterrupt(249,0,0);
vmeinterrupt(250,0,0);
vmeinterrupt(251,0,0);
vmeinterrupt(252,0,0);
vmedebug(0);

}
```

MIZZI Computer Software

VME

Utilities/Libraries

OS9 Implementation

Version

```
Version: 1.3
Date:    Jan 1994
```

Copyright

Implementation specific Information

The OS9 Implementation of the VMElib supports all features defined in the VME library.

Notes

Interrupts are handled in OS9 System state. Therefore only data manipulation is recommended in interrupt service routines which resides in the global data module or with automatic defined data. printf and other functions which are not fully reentrant can not be called by the interrupt service routine.

VME interrupt service routines must be restored before the program terminates. Otherwise the system will crash if an interrupt occurs.

vmestacksize and vmepriority are ignored and return 0.

The library works currently only with OS9 systems without memory management support.

Linking

There is one library file called vme.l which must be linked together with your program.

MIZZI Computer Software

VME

Utilities/Libraries

LynxOS Implementation

Version

```
Version: 1.5
Date:    Sep 1996
```

Copyright

Availability

The LynxOS VMElib is currently available on:

```
Cetia VMTV2d 68040 Board
Cetia VMTR2/VGPW2 PowerPC Board
CES   FIC 8234 68040 Board
```

Implementation specific Information

All library functions supported.

Files

```
vme.a        --- library
vme.o        --- library object file
vme.h        --- header file
vme.d        --- header file
vmedriver    --- program to load the vme driver
vmedriver.o --- lynxos driver for the vme lib.
```

Note

Interrupts are handled via system threads.

When vmeinterrupt is called for the first time the function extents the main thread stacksize to 64KBytes and the stack of the interrupt thread is located after the last thread currently running. If you need more stacksize in the main thread you must extend the main thread stackzize to the value you need for your main thread. If you had already started an other thread you must be sure that your main thread has 64K free stack space. Otherwise vmeinterrupt will fail with with a segmentation violation.

VME interrupts can only be handled if the VME interrupt is reset on acknowledge (ROA). Reset on register (ROR) can only be handled by a special driver.

See lynxos documentation for more information.

Linking

Programs using the vmelib must be linked and compiled with the -m flag on LynxOS versions <= 2.2 (cc -m vmetest.c vme.a). LynxOS version >=2.3 requires the -mthreads flag instead of the incompatible -m flag of previous version.
(cc -mthreads vmetest.c vme.a)

Restrictions

The A32 VME bus access is by default available in the A32 region 0x10000000(256MB) - 0x13000000(304MB) (total amount 48MB). You can change this section when loading the vmedriver. See installing the VME driver.

VME Utilities/Libraries LynxOS

Installing the VME driver

The vme driver must be loaded at system startup (or later) with the command (root privileges required):

```
vmedriver load
```

and can be unloaded with the command:

```
vmedriver unload
```

The file vmedriver.o must be located in the current working directory, when vmedriver load is called.

The default A32 region can be overwritten with two additional parameters.

```
vmedriver load <startaddress in hex> <lenght in hex>
```

For example:

```
vmedriver load 2000000 2000000
```

makes a A32 region from address 32MB with a lenght of 32MB available to the user programs. All other addresses will return an error when initializing the corresponding memory section!

Try to use the default section which should be available on all processors.

Hardware Specific Informations

Restrictions for the CES FIC 8234

1.) Due to the nature of LynxOS 2.1 there is no way to install ROR (reset on register) interrupts. Use ROA (reset on acknoledge) interrupts instead of ROR interrupts if your module supports this. Otherwise you have no chance to get the interrupt running (the CPU will crash immediatly).

2.) There is only one A32 page with a size of 16 Mbyte available. This restriction applies to the very strange VME bus mapping of CES.

3.) VME bus traps are not supported because CES does not support this feature. What happens is rather unpredictable. Sometimes i have seen that you get a random value after 15 sec, sometimes endless loops on the VME bus are executed repeating the buserr for ever. In special cases the CPU crashes.

4.) The driver is only a few times unloadable! If no other program is mapping the VME bus you can load and unload the driver 5 times. Further loads and unloads will crash the CPU.