VM-USB

User Manual

General Remarks

The only purpose of this manual is a description of the product. It must not be interpreted a declaration of conformity for this product including the product and software.
W-Ie-Ne-R revises this product and manual without notice. Differences of the description in manual and product are possible.
W-Ie-Ne-R excludes completely any liability for loss of profits, loss of business, loss of use or data, interrupt of business, or for indirect, special incidental, or consequential damages of any kind, even if **W-Ie-Ne-R** has been advises of the possibility of such damages arising from any defect or error in this manual or product.
Any use of the product which may influence health of human beings requires the express written permission of **W-Ie-Ne-R**.
Products mentioned in this manual are mentioned for identification purposes only. Product names appearing in this manual may or may not be registered trademarks or copyrights of their respective companies.
No part of this product, including the product and the software may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means with the express written permission of **W-Ie-Ne-R**.

VM-USB and CC-USB are designed by JTEC Instruments.

Table of contents:

1 GENERAL DESCRIPTION

The VM-USB is an intelligent VME master with high speed USB2 interface. Enhanced functionality is given by the programmable internal FPGA logic which provides a VME command sequencer with 4kB stack, 4kB event buffer, and 26kB data buffer. Combined with the 4 front panel I/O ports, this allows VME operation and data acquisition / buffering without any PC or USB activity, other than reading out suitably formatted data buffers.

The VM-USB can be also programmed to act as a VME slave with respect to a master crate controller, while performing master operations on other data acquisition modules. For example, it can be programmed via the VME bus to perform the readout of multiple VME modules, with data buffering in a 24-kByte FIFO. The master module can then retrieve the data from the VM-USB alone, at block transfer rates.

All VM-USB logic is controlled by a XC3S400 XILINX Spartan 3 family FPGA. Upon power-up the FPGA boots from a selected segment (one of four) of flash memory. The configuration flash memory can be reprogrammed via the USB port, allowing convenient updates of the firmware. Following an open platform approach, the user can develop his own FPGA configuration / firmware. The boot sector is selected by setting of a front-panel rotary switch.

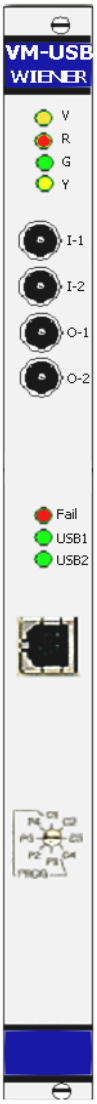
1.1 VM-USB Features

- Low-cost 6U single wide VME master with high speed USB2 interface, auto-selecting USB2 / USB1, LED's for speed and FPGA failure/reset..
- While operated as a slot-one system controller, performs round-robin and fair bus arbitration, generates the 16 MHz system clock, and generates BERR response, when no DTACK appears within 16us from the assertion of data strobes by any controller.
- May be programmed to service any or all of the 7 interrupt requests IRQ1-7.
- Can generate any of the 7 interrupts when configured with a suitable firmware
- 2 multiplexed NIM inputs (with LEMO connectors), with a selection of input signal functionality including 2 32-bit scalers and 2 delay and gate generators.
- 2 multiplexed NIM outputs (with LEMO connectors), with a selection of source signals, including the outputs of the 2 delay and gate generators.
- 4 multiplexed LED's, with a large selection of diagnostic signals.
- Spartan 3 FPGA, XC3S400 based, firmware upgradeable via the USB port from a host PC.
- Built in VME sequencer, 1k x 16 bit VME command stack memory for use in an autonomous data acquisition process. Programmable via USB and/or VME, depending on the active FPGA firmware.
- Open architecture, allowing the user to develop his own FPGA configuration.
- Readout triggered either via USB link, by VME interrupt (IRQ1-7) or by a start signal applied to a (programmable) NIM input.
- 26-kByte of pipelined data buffer (FIFO) with programmable level of transfer trigger
- Low power consumption, only +5V used.

1.2 Data Transfer Modes

- Single word transfer D8, D16, D24, D32 with the full selection of bus placement, including unaligned transfers.
- Addressing modes A16, A24, A32.
- BLT16, and BLT32 block transfers..
- Multi-BLT transfers of up to 8 Mbytes, both write and read, with or without auto-increment of individual BLT addresses.
- Autonomous (intelligent) operation pursuant to user-programmed stack. May include conditional execution of VME commands controlled by the content of a hit register. May include multiple, conditional command stacks, action triggered by either USB, VME IRQ, or external signal.
- Highly effective triply-pipelined stack execution, virtually with no band-width penalty for single (as opposed to block transfer) operations.
- Long USB2 bulk transfers of up to 4 MByte, both write and read.
- Highly customizable.

1.3 VM-USB Front panel



4 user LED's

2 user inputs Lemo / NIM

2 user outputs Lemo / NIM

Failure LED / USB 1 or 2 indicator

USB port

Firmware selector (1 – 4) :
P1 – P4 for programming
C1 – C4 for use / operation

1.4 Technical Data

Packaging	single wide 6U VME module
Interface	USB2 / USB1 auto-detecting / ranging, Connector: USB type B
Inputs	2 user inputs, NIM level , LEMO Multiplexed functionality (firmware 8504):
Outputs	2 multiplexed outputs for VME, USB and DAQ signals, NIM level, LEMO, function firmware dependent
Display	4 programmable User LED's (green, red, green, yellow) 3 USB status LED's (USB1, USB2, Failure)
VME master modes	A16, A24, A32, D8, D16, D24, D32, BLT32, BLT16
System Controller	bus arbiter and / or interrupt handler
Firmware	Software upgradeable, 4 firmware locations Selection via 8 position switch (P=program, C=use)
Performance	D32 via USB (EASY-VME): 128 kB/s D32 with data buffering: 32 MB/s BLT: 32 MB/s

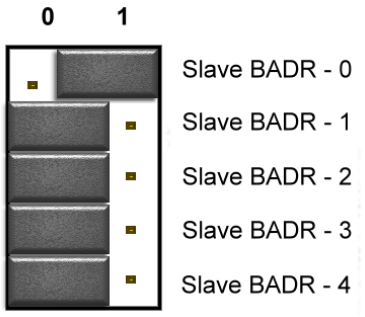
1.5 Power Consumption

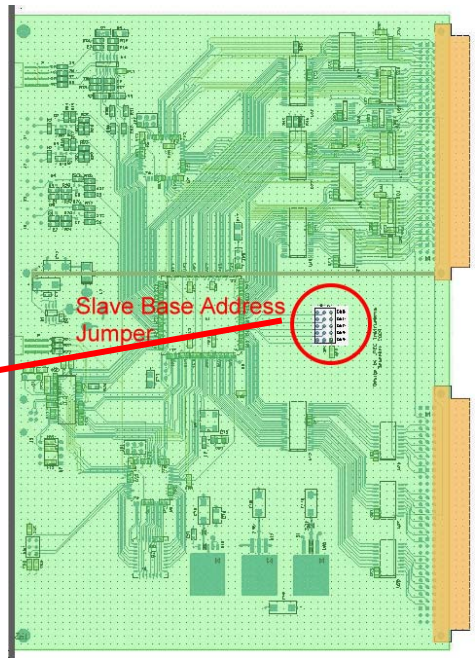
Voltage	Max. current	Power
+5 V	1.2 A	about 8 W

1.6 Slave Module Base Address Jumper Settings

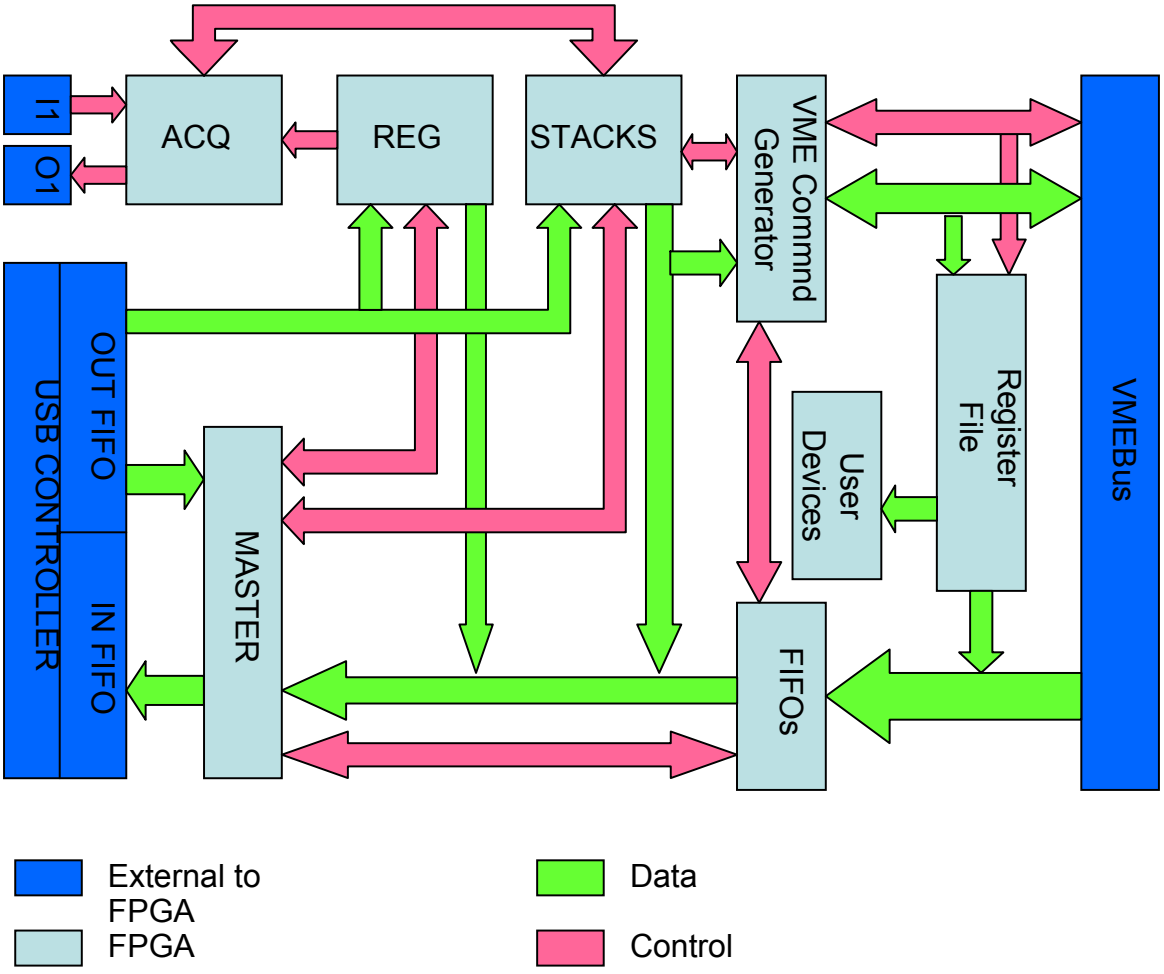
The VM-USB module can be used as a VME slave in A32, A24 or A16 mode. For this purpose the base address BADR can be selected via 5 jumpers, which define bits 27-31(A32), 20-24(A24) or 12-16 (A16) (always 5 most significant bits of the VME address). By default the SLAVE Base Address is:

BADR = 0x08 0000 (A24).
BADR = 0x0800 0000 (A32).





1.7 Block diagram



I1	- User NIM input
O1	- User NIM "Busy" output
ACQ	- Data Acquisition Control
REG	- Register Block
STACKS	- VME Command Stacks (2 kBytes)
VME command Gen.	- VME command Generator
VME	- VME Bus, Including Arbitration
FIFOs	- Three-Stage Pipelined FIFO array(22 kBytes)
Master	- Control Unit
USB Controller	- FX2 CY7C68013 IC
OUT FIFO	- USB Out FIFO (Relative to Host)
IN FIFO	- USB In FIFO (Relative to Host)

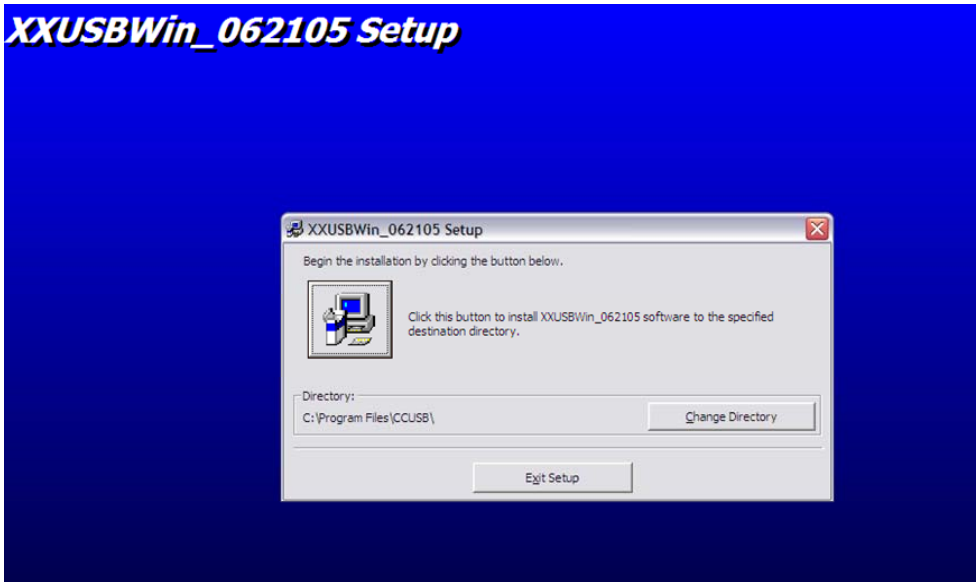
2 VM-USB AND USB DRIVER INSTALLATION

ATTENTION!!! Observe precautions for handling:

- **Electrostatic device!** Handle only at static safe work stations. Do not touch electronic components or wiring
- The VME crate as well as the used PC have to be on the same electric potential. Different potentials can result in unexpected currents between the VM-USB and connected computer which can destroy the units.
- Do not plug the VC-USB into a VME crate under power. **Switch off the VME crate first before inserting or removing any VME module!** For safety reasons the crate should be disconnected from AC mains.

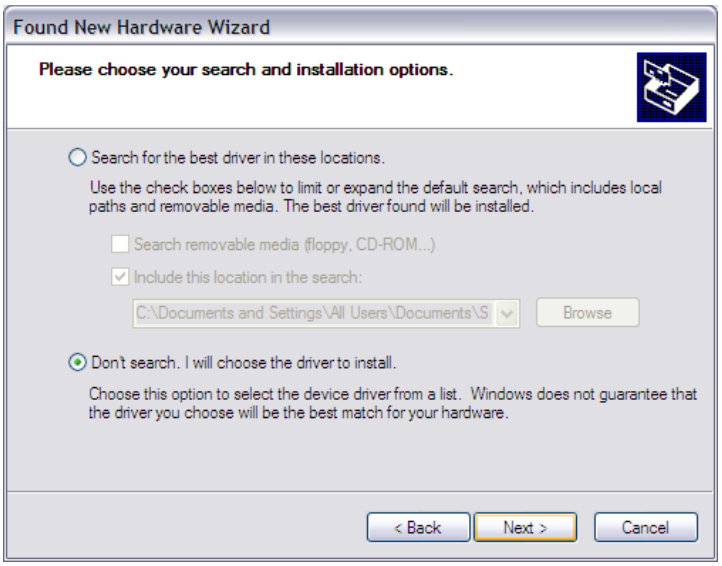
2.1 Installation for Windows Operating Systems

1. Switch off the VME crate and remove the power cord. Plug in the VM-USB on the first left slot 1 if needed as system controller and secure it with the front panel screw. Switch on the VME crate.
2. Insert the driver and software CD-ROM into the CD-ROM drive of the computer and run the setup program in the XXUSBWin_Install folder. Define directory for installation and click the installation button.

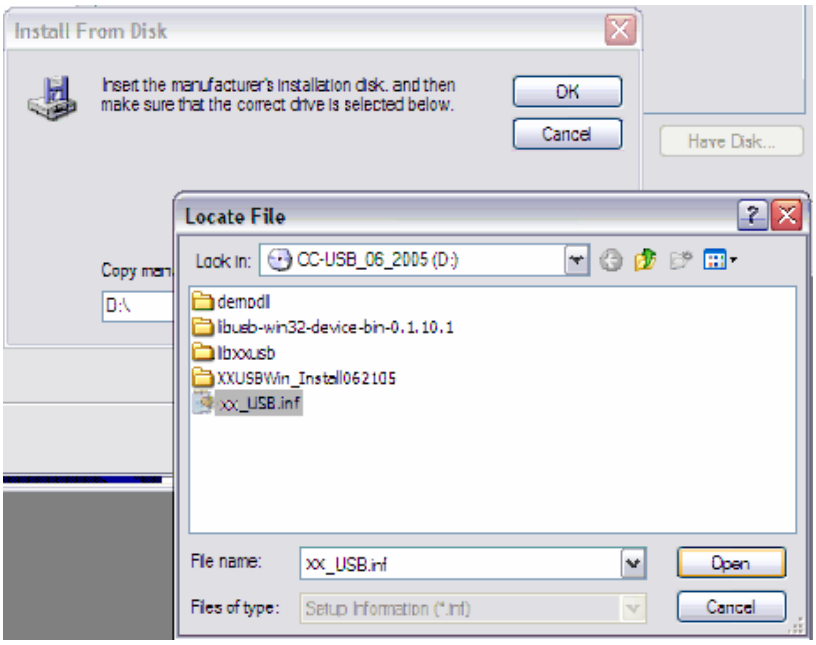


3. Connect the VM-USB via the provided USB cable to a USB port of the computer. Running Windows 2000 or XP the hardware change should be detected and the “New Hardware Wizard” Window should open and show the VME USB controller.
4. Do not use the automatic software installation but chose “installation from specific location”.

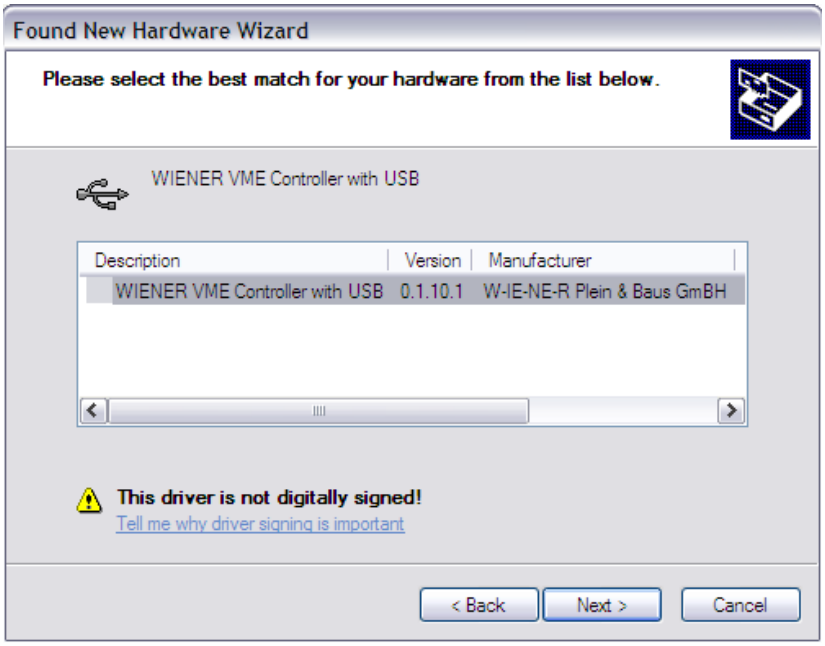
5. Select manual search for the driver
6. Type in the drive letter for the CD-ROM (e.g. D:, F:, …) and locate the file CC-USB.inf. Press Enter to select this driver and to close the window.



7. The WIENER VM-USB driver should be listed and highlighted in the driver list. The driver is not digitally signed which however does not have any effect on it's functionality. Press Next to finish the installation.



8. The "New Hardware Wizard" should copy all driver files into the Windows System32 folders and report a successful installation.



9. In order to use the XXUSBWin.exe program please install it by running the setup.exe program in the XXUSBWin---_Install directory (--- select the right Windows version for your computer).

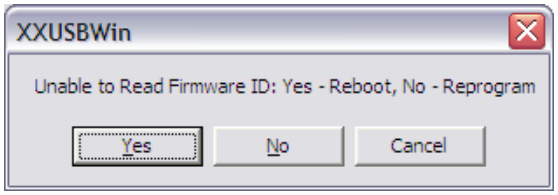
2.2 Installation for Linux Operating Systems

Linux support for the VM_USB is provided through a shared library and header file. To use these file simply copy them to an appropriate location, such as /usr/lib for the library and /usr/include for the header file. All files are open source.

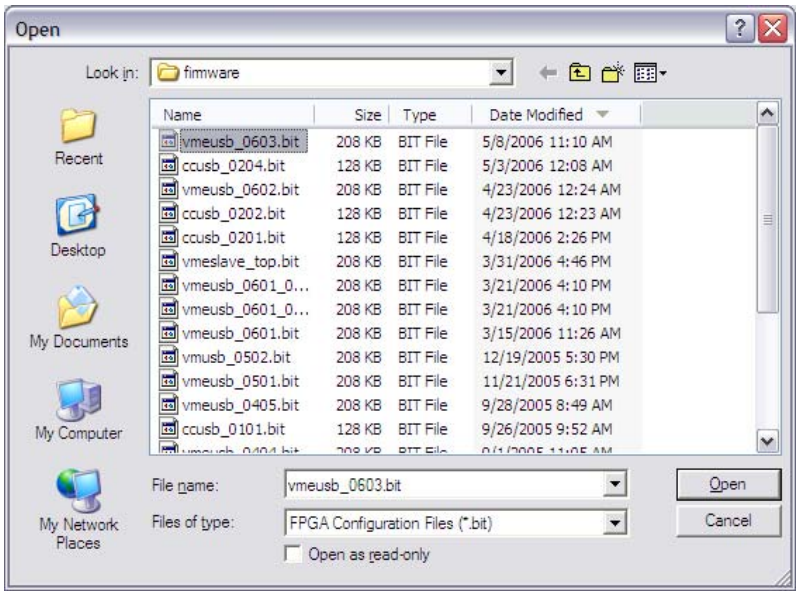
The functions available in the library are exactly the same as those available at for Windows and are described later in the manual. Linux specific details are located in the readme file on the software CD that you received with your module.

2.3 Firmware upgrades

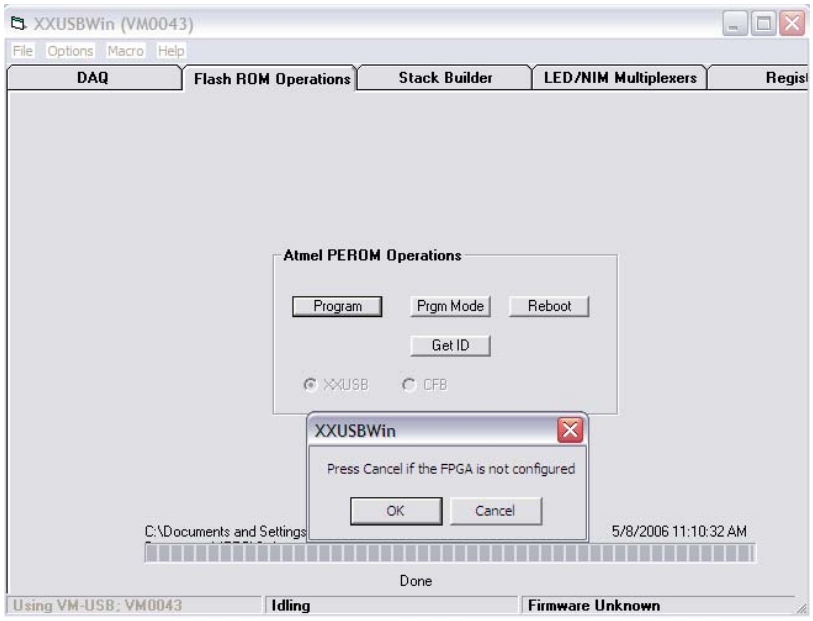
The VM-USB is shipped with the latest firmware for the FPGA loaded however, new versions of it may be available on the web. Please occasionally check at www.wiener-d.com -> support -> downloads if newer versions of firmware, documentation and / or software are available.
The firmware upgrade is done via USB and can be performed by the help of the XXUSBWin program.
To upgrade, switch the firmware selector to one of the 4 firmware programming positions (P1-P4). The red Failure LED will be on. Start the XXUSBWin program which will show the following error message:



Select “Yes” or go to the Flash ROM Operations page and click program. Open the file of the latest firmware (xxx.bit)



When done one has to reset the controller or switch the selector switch to the corresponding run location (C1-C4) and power cycle the crate.



3 GENERAL ARCHITECTURE OF VM-USB AND ITS USER INTERFACE

Note: The information is this section is relevant only for VM_USB controllers with firmware 950000405 or later. If you have an earlier version consider updating or see Appendix C.

Most of the VM_USB registers take the form of a VME address within the VM_USB and can be communicated with by using VME calls to the appropriate VME address. These registers are part of the Internal register file. The registers that cannot be communicated with in this way are in Table 1.

Table 1. Internal VM)USB registers that aren't seen as VME space

PA	Device
1	Action Register (AR)
4	VME Command Generator (VCG)
-	VME Commmand Stacks
-	Common Output Buffer

3.1 Action Register (Address = 1 / 0x1)

12-15	8-11	5-7	4	3	2	1	0
-	-	-	scaler dump	SYSRES	clear	USB trigger	Start/stop

The action register is a special-purpose write-only register controlling the mode of operation of VM-USB and the generation of internal trigger/reset signals. By its design, it can often be accessed when other VM-USB actions are under way and, most notably, when VM-USB is placed in autonomous data acquisition mode.

Bit 0 of the Action Register controls the operating mode of VM-USB and distinguishes between the interactive and autonomous data acquisition modes. In the interactive mode, VME commands (individual or a sequence) are performed directly in response to a USB packet received from the host, while in the data acquisition mode, sequences of VME commands are executed, pursuant to lists stored in a dedicated VM-USB memory block. Any of the configurable 8 lists (stacks) is executed upon the receipt of the associated trigger signal, which may be a signal at the user NIM input I1, detection of a valid VME IRQ, or a USB trigger of scaler readout. When bit 0 of the Action Register is set, VM-USB is in data acquisition mode; otherwise it is in interactive mode.

Bit 1 of the Action Register is a write-only bit, such that writing "1" to it generates an internal signal of 150ns duration, called USB Trigger. This signal can be routed to user NIM outputs O1 or O2, used as an input to multiplexed internal user devices of VM-USB (such as delay and gate generators and scalers), and/or displayed on user LEDs.

Bit 2 of the Action Register is a write-only bit, such that writing "1" to it, clears a number of internal registers. It is intended primarily for use during firmware debugging.

Bit 3 of the Action Register is a write-only bit linked to the SYSRES line, active when VM-USB is a slot one controller. When set to “1” VM-USB generates SYSRES. When set to “0”, the SYSRES is cancelled.

Bit 4 (write-only) of the action register is used to trigger scaler readout when VM-USB is in data acquisition mode. This interactive mode of the scaler readout combines with the two automatic readout modes (event-based and timer-based) on “whichever-comes-first” basis.

3.2 VME Command Generator / EASY-VME (Address = 4 / 0x4)

The VME Command Generator decodes lists of coded VME commands, submits the commands for execution in VME cycles, and causes the received data, if any, to be stored in a 4kB event memory (FIFO). At the end of the list (end of an event), the content of the event FIFO is compiled into the main data buffer (up to 26 kB), for a subsequent transfer to the USB controller. The VME Command Generator receives coded data either directly from the host (in interactive or Easy-VME mode), or reads it from the VME Command Stacks (in autonomous data acquisition mode). The structure of the list is identical in both modes of operation and is discussed in detail further below. An integral part of the VME command is the VME address of a target register/memory location. In the context of the discussed VM-USB architecture, this VME address is a secondary address and its space includes the internal VME address space of VM-USB, associated with the internal register file of VM-USB.

3.3 VME Command Stacks

VME Command Stacks (VCS) are used to store suitably formatted lists or sequences of VME Commands to be performed in response to event trigger signals while VM-USB is operating in autonomous data acquisition mode of operation (DAQ mode). In this mode of operation, VM-USB issues VME commands, reads the data received in response to them, and buffers the data in a data buffer. When the buffer (up to 26kB) is full, VM-USB dumps it to the FIFO of the USB controller IC for the retrieval by the host. It is this data buffering that allows one to take advantage of the superior band width of the USB2 interface in bulk transfer mode and to achieve throughputs in excess of 30 Mbytes/s.

Starting with firmware 16000503 the number of stacks that could be defined was increased from 2 to 8. The new firmware allows one to define up to 8 stacks, with Stack ID=0-7, within the allocated memory of 2kBytes = 1kWords. The starting addresses of stacks can be set arbitrarily. While stacks with ID=2-7 are dedicated to interrupt handling, stacks with ID=0 and 1 are of dual use. Unless the latter stacks are used for interrupt handling, they default to a regular stack with ID=0, executed upon trigger pulse (NIM I1 or USBStart) and the periodic stack with ID=1 (scaler stack).

When not working in DAQ mode, the VME command stacks can also be executed by the VME Command Generator (VCG). Chapter 4.4 provides details about the VCS and VCG stack structure and how to write the stack to the VM-USB. Stack instructions and examples are shown in chapter 4.5.
While any arbitrary block of data can be stored in the stack memory and then read back, all stacks are expected to contain properly encoded sequences of VME operations and their associated options, such that they can be meaningfully decoded by the VME Command Generator module and submitted for execution, while VM-USB operates in autonomous data acquisition mode.

3.4 Internal Register File

Writing/reading to/from the internal register file is done by executing a 32 bit VME write/read command with the VME offset assigned to a given register, while bit 12 (value 4096) of the VME command word is set to 1. Details about the VME command word are found in section 4.5

Table 2. Register VME address offsets and their functionality

Offset		Register	Note
HEX	Dec		
0x0	**0**	Firmware ID	32 bits, Read-only
0x4	**4**	Global Mode	16 bits, Read/Write
0x8	**8**	Data Acquisition Settings	32 bits, Read/Write
0xC	**12**	User LED Source Selector	32 bits, Read/Write
0x10	**16**	User Devices Source Selector	32 bits, Read/Write
0x14	**20**	DGG_A Delay/Gate Settings	32 bits, Read/Write
0x18	**24**	DGG_B Delay/Gate Settings	32 bits, Read/Write
0x1C	**28**	Scaler_A Data	32 bits, Read/Clear/Enable
0x20	**32**	Scaler_B Data	32 bits, Read/Clear/Enable
0x24	**36**	Number Extract Mask	32 bits, Read/Write
0x28	**40**	Interrupt Service Vectors 1 + 2	32 bits, Read/Write
0x2C	**44**	Interrupt Service Vectors 3 + 4	32 bits, Read/Write
0x30	**48**	Interrupt Service Vectors 5 + 6	32 bits, Read/Write
0x34	**52**	Interrupt Service Vectors 7 + 8	32 bits, Read/Write
0x38	**56**	Extended DGG_A/B Delay Settings	32 bits, Read/Write
0x3C	**60**	USB Bulk Transfer Setup	32 bits, Read/Write

3.4.1 Firmware ID Register - Read only
Offset = 0 / 0x0

29-31	24-28	16-23	8-15	0-7
Month	Year	Beta Version	Major Revision	Minor Revision

This Firmware ID register identifies the acting FPGA firmware in eight hexadecimal digits MYBBFFRR, where M and Y represent the month and year of creation, B represents beta version, and F and R represent the firmware main and revision numbers, respectively. Please see the notes about the different firmware features in the Appendix section of this manual.

3.4.2 Global Mode Register – Read/Write
Offset = 4 / 0x4

The global mode register has the following 16-bit structure:

Bits	15	14	13	12	11 - 9	8	7	6	5	4	3 - 0
Value	-	BusReq			-	HeaderOpt	-	EvtSepOpt	-	BuffOpt	

Firmware 95000405 and later, offers additionally an option to buffer regular data and scaler data in common data buffers. This option is selected by setting bit 5 of Global Mode Register to "1" and it allows one to avoid short buffers, which tend to waste the USB bandwidth. In mixed buffers, scaler events are identified by bit 15 set in the event header word. The global mode register for firmware 95000405 and later has the following 16-bit structure.

Bits	**15**	**12-14**	**9-11**	**8**	**7**	**6**	**5**	**4**	**3 - 1**
Value	-	BusReq	-	HeaderOpt	Align32	EvtSepOpt	MixtBuff	BuffOpt	

The BuffOpt bits (0-3) define the output buffer length. Bit 4 controls the mode of buffer filling, such that 0 closes buffers at event boundaries and 1 allows spreading events across the adjacent buffers:

BuffOpt Value	**Buffer Length (words)**
0	13k (default)
1	8k
2	4k
3	2k
4	1k = 1024
5	512
6	256
7	128
8	64
9	Single Event

The MixtBuff=1 selects mixed buffer option which allows one to pack data associated with different triggers (NIM, scaler, IRQs) into the same data buffers. In mixed buffers, events associated with different triggers are identified by an Event Type ID encoded in bits 13-15 of the event header word. Note that the bits 0 to 11 of this word list the event length and bit 12 indicates event length exceeding the length of the 2k Word event assembly buffer (i.e., that the event is to be continued in next event buffer).

The EvtSepOpt sets the number of event terminator words (hexadecimal 5555 and AAAA), such that EvtSepOpt=0/1 cause one/two terminator word/s written at the end of each event.

The Align32 bit controls the alignment of data in the data buffer. The default alignment (Align32=0) is on 16-bit boundaries. Setting Align32=1 causes all header (buffer and event) and terminator words to be converted to 32-bit words, by adding blank 16-bit words. Also 16-bit data words that may be returned by some VME read commands are converted to 32-bit numbers.

The HeaderOpt bit controls the structure of the buffer header, such that HeaderOpt=0 writes out one header word identifying the buffer type (bit 15=1 – watchdog buffer, bit 14=0 – data buffer, bit 14=1 – scaler buffer) and the number of events in buffer. When HeaderOpt = 1, the second header word is written out listing the number of words in the buffer.
The BusReq bits identify the VME Bus Request level (0 to 4) to be used by VM-USB, when not operated as a slot 1 controller (bus arbiter). BusReq=1,2,3, and 4 cause BR0, BR1, BR2, and BR3 lines to be used, respectively.

3.4.3 Data Acquisition Settings Register – Read/Write
Offset = 8 / 0x8

The Data Acquisition Settings register stores the desired readout trigger delay and the scaler readout mode and frequency. The readout trigger delay represents time in microseconds that is allowed to lapse, counting from the start signal applied to the NIM I1 input or IRQ received, before the stack execution is started, and is stored in bits 0-7 of the register. The scaler readout frequency defines the frequency at which scaler stack is to be executed during the data acquisition. The stored value is equal to the number of data events separating the scaler readout events. When the value is zero, scaler readout is suppressed. The number is stored in bits 16-31 of the register.. The event-based and timer-based triggers are set to cooperate on "whichever-comes-first" policy.

The composition of the Data Acquisition Settings Register is as follows shown in the table below

16-31	8-15	0-7
Scaler Readout Frequency	**Scaler Readout Period**	**Readout Trigger Delay**

3.4.4 User LED Source Selectors – Read /Write
Offset = 0x10 / 0xC

The Data stored in this register identifies the sources of the four user LED's. The actual selection of sources is firmware specific and subject to customization. The general bit composition of the selector word is shown in the table below

LED	Unused Bits	Latch Bit	Invert Bit	Code Bits
Top Yellow	5-7	4	3	0-2
Red	13-15	12	11	8-10
Green	21-23	20	19	16-18
Bottom Yellow	29-32	28	27	24-26

The 3-bit code identifies the (one-of-eight) source of the signal. The four first sources may differ for different LED's, but the last four are shared among all four LED's. The source Codes are as follows:

Code	Top Yellow	Red	Green	Bottom Yellow
0	USB OutFIFO Not Empty	Event Trigg.	Acquire	Not Slot One
1	USB InFIFO Not Empty	NIM I1	Stack Not Empty	USB Trigger
2	Scaler Event	NIM I2	Event Ready	USB Reset
3	USB InFIFO Full	Busy	Event Trigger	VME BERR
4	VME DTACK	VME DTACK	VME DTACK	VME DTACK
5	VME BERR	VME BERR	VME BERR	VME BERR
6	VME Bus Request	VME Bus Request	VME Bus Request	VME Bus Request
7	VME Bus Granted	VME Bus Granted	VME Bus Granted	VME Bus Granted

3.4.5 User Devices Source Selector - Read/Write
Offset = 16 0x10

There are six user devices set up within the FPGA resources of the VM-USB – two NIM outputs, O1 and O2, two delay and gate generators or pulsers, DGG_1/P_1 and DGG_B/P_1, and two 32-bit scalers, SCLR_A and SCLR_B. All of these devices may use various signals as input/trigger signals, with the selection identified by respective code bits stored in the User Devices Source Selector register. Additionally, this register accommodates bits that enable and clear the two scalers. The bit composition of the User Devices source selector register is shown in the table below.

Device	Reset	Enable	Latch Bit	Invert Bit	Code
NIM O1	-	-	4	3	0-2
NIM O2	-	-	12	11	8-10
SCLR_A	19	18	-	-	16-17
SCLR_B	23	22	-	-	20-21
DGG_A	-	-	-		24-26
DGG_B	-	-	-	-	28-30

For the NIM outputs a 3-bit code identifies the (1 of 8) source of the signal. Like for LEDs, the first four sources may differ for different NIM outputs, but the last four are shared., The source codes are as follows

Code	NIM O1	NIM O2
0	Busy	USB Trigger
1	Event Trigger	Executing VME Command
2	Bus Request	VME Address Strobe AS
3	Xfer Event to Data Buffer	Xfer Data Buffer to USB FIFO
4	DGG_A	DGG_A
5	DGG_B	DGG_B
6	End of Event	End of Event
7	USB Trigger	USB Trigger

Note 1. “Busy” signal indicates that stack processing is in progress, with VME operations not being completed. “Busy” is asserted when event readout is triggered and removed as soon as VME operations are completed.

Note 2. “Acquire” indicates that the data acquisition mode is active.

Note 3. “USB Trigger” is generated in response to writing to bit 1 of Action Register.

Note 4. “Event Trigger” indicates that event readout has been triggered.

Note 5. Invert bit causes the signal to be inverted

Note 6. Latch bit causes the signal to be latched. To release the latch one must toggle the bit.

The meaning of the input selector codes for scalers and delay and gate generators is shown in the table below

Code	SCLR_A	SCLR_B	DGG_A/P_A	DGG_B/P_B
0	Disabled	Disabled	Disabled	Disabled
1	NIM I1	NIM I1	NIM I1	NIM I1
2	NIM I2	NIM I2	NIM I2	NIM I2
3	Event	Event	Event Trigger	Event Trigger
4	-	-	End of Event	End of Event
5	-	-	USB Trigger	USB Trigger
6			Pulser	Pulser

3.4.6 Delay and Gate Generator / Pulser Registers – Read/Write

DGG _A Offset = 20 / 0x14
DGG _B Offset = 24 / 0x18
DGG _ Ext Offset =56/0x38

The two Delay and Gate Generator / Pulser Registers DGG_A and DGG_B as well as the DGG_ Extend register store data defining the length of the delay and the length of the gate in units of 12.5 ns (80 MHz clock) for either the gate and delay generator or for the pulser. These values can be set for channel A and B independently. The pulser is re-triggering after the defined delay time, i.e. the delay time + gate length defines the pulser repetition rate. The value of the delay is a composite of a high resolution value (12.5ns) and a coarse range value which was added with firmware 6.0 to increase the possible time range up to 53.5s . Earlier firmware versions use only the fine (12.5ns) value.

Gate length ***= 12.5ns * Gate***

Delay ***= 12.5ns * Delay_fine + 819.2µs* Delay_coarse***

Pulser repetition period= Gate + Delay

DGG_A 16-31	DGG_A 0-15
Gate	Delay_fine

DGG_B Offset = 24 / 0x18

DGG_B 16-31	DGG_ B 0-15
Gate	Delay_fine

DGG_ Ext Offset =56/0x38

DGG_B Ext (16-31)	DGG_A Ext 0-15
Delay coarse	Delay coarse

3.4.7 Scaler Registers SLR_A and SCLR_B – Read
Scaler_A Offset = 28 / 0x1C
Scaler_B Offset = 32 / 0x20

Scaler registers SLR_A and SCLR_B store 32-bit data from the two scalers SCLR_A and SCLR_B respectively. The scaler input (event, NIM1 or NIM2) are defined within the User Device Selector (see 3.4.5), which also allows resetting or disabling them.

3.4.8 Number Extract Mask Register - Read/Write
Offset = 36 / 0x24

The Number Extract Mask Register stores a logical AND mask that is used to extract a number from 32-bit VME data. This number is then used to override the block transfer length, set in bits 24-31 of the first line of the subsequent VME command.

3.4.9 Interrupt Service Vectors - Read/Write
Offset = 40 / 0x28
Offset = 44 / 0x2C
Offset = 48 / 0x30
Offset = 52 / 0x34

Firmware 16000503 and later increased the number of stack from 2 to 8 and allows stacks execution to be triggered by VME interrupts. If your VM_USB shipped before 3/1/06 you will need to upgrade the CPLD firmware before taking advantage of the VME interrupts. For more information on this contact WIENER support.

Any Stack ID can be linked to any combination of IRQ and the IRQ-ID returned by the requestor module, such that one Stack ID can be shared by many interrupts. Upon detection of a valid interrupt, VM-USB executes the stack linked to this particular interrupt. The Interrupt Service Vectors are stored in four registers at VM-USB register addresses 0x28 to 0x34 in arbitrary order and contain two vectors per register. Each vector identifies in its 8 least significant bits the IRQ-ID (to be received from the requestor upon interrupt acknowledge IACK), in bits 8-10 (starting from bit 0) the IRQ level (1-7), and in bits 12-14 the ID (0-7) of the associated stack. The vector is disregarded whenever the IRQ-ID is set to zero, e.g., allowing one to use Stacks 0 and 1 as regular and periodic (scaler) stacks, respectively.

Vector 1:

Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	**-**	**Stack ID**			**-**	**IRQ**			**IRQ-ID**							

Vector 2:

Bits	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Value	**-**	**Stack ID**			**-**	**IRQ**			**IRQ-ID**							

3.4.10 USB Bulk Transfer Setup Register – Read/Write Offset=60/0x3C

In order to benefit from the high bandwidth of the USB2 interface, overheads associated with single transfer operations should be avoided. Therefore, one must strive to reduce the number of transfers by extending the length of bulk transfers. VM-USB, by default closes USB buffer (generates a "packet end") either at the end of the data buffer or at the end of event. This guarantees bulk transfer lengths of only 26 kBytes for short events and lengths equal to event lengths, in the case of long events. Such default setting does not allow one to utilize the USB2 bandwidth when short events are acquired and, therefore, VM-USB offers an option to "bundle" multiple data buffers together for a single bulk transfer.

Since in the case of short events the time of filling multiple buffers is variable, the option includes setting of a watchdog timer, which will guarantee that a "packet end" signal is generated at timeout, should the data buffers fail to fill sufficiently fast. This watchdog timeout should be made shorter than the software timeout set for bulk read. The relevant numbers for multi-buffer bulk transfer are stored in the USB Bulk Transfer Setup Register at offset = 0x3C (decimal 60) such that the number of buffers is specified in bits 0 – 7 of this register and the timeout is specified in bits 16-18. The 3-bit timeout represents the number of seconds in excess of 1s, after which the packet end signal is issued, should the specified number of buffers not be completed by that time. Note that the default (minimum) is 1s.

Bits	**12-31**	**8-11**	**4-7**	**0-3**
Value	-	Time out	Number of buffers	

3.5 IRQ Mask

Firmware 86000801 introduced improved bus arbitration and IRQ handling. If your VM_USB shipped before 8/16/06 you will need to upgrade the CPLD firmware before taking advantage of the IRQ mask. For more information on this contact WIENER support.

The VM_USB has the option to mask IRQs via the IRQ mask register. This allows the user to adjust which IRQs the module responds to without removing the VM_USB from the crate.

A 7-bit IRQ mask can be setup, which defines which IRQs will be hidden from the VM_USB. Setting a bit to 1 causes the corresponding IRQ to be ignored.

Bit	6	5	4	3	2	1	0
IRQ	7	6	5	4	3	2	1

Because of the architecture of the VM-USB, writing of the IRQ mask and reading it back is not as easy as writing to other registers. This is because the mask resides in the Interface CPLD (top XC95144XL CPLD, named U7) and there are no dedicated connections for this purpose.

To write the IRQ mask:

1) Write the IRQ mask to bits 0-6 of the global register and set bit 15 to 1
2) Set bit 1 of the action register to 1.
3) While bit 15 of the global register is set, bits 24-30 of the Firmware ID register will return the IRQ mask
4) Set bit 15 of the Global register to 0
5) Reset the proper bits to the global register to define the operational parameters desired.

4 COMMUNICATING WITH VM-USB

Communication with the VM-USB consists of writing and reading data buffers to/from the USB2 port of the VM-USB using bulk-transfer mode. Borrowing from the USB language, the buffers to be written to the VM-USB will be called Out Packets, and they are sent to pipe 0 of the USB port. The buffers to be read will be called In Packets, and they are read from pipe 2 of the USB port.

The USB controller IC, when connected to a USB2 port configures packet lengths to 512 bytes. For USB1 (full speed), the packet length is set to 64 bytes. The Out Packets must be properly formatted to be understood by the internal devices of VM-USB and, by the same token, the format of the In Packets retrieved from the VM-USB must be understood by the user in order to be useful.

User may send Out Packets to four devices – the Register Block (RB), VME Readout Stacks (VDS and VSS), and the VME Generator (VGen). User may read In Packets only from the Common Output Buffer. Reading back data from the RB, VDS, and VSS is achieved by, first sending a data request Out Packet to these devices and then by reading the In Packet containing the requested data from the Common Output Buffer.

Writing to the VME Generator constitutes implicitly a request for data, such that in response to such a writing, VM-USB performs the requested VME operation (including the ones addressed to internal registers of VM-USB) and returns the VME data in the Common Output Buffer. Both, In and Out Packets are of a variable length, depending on which internal address is involved and what the content of the message is.

Important Note:

With some drivers (EZUSB in conjunction with Windows API), read operations from the USB port are blocking operations, such that the host program will stop executing until the data are available at the port. Therefore, the host program must make sure (by first requesting data) that VM-USB has placed data in the Common Output Buffer (physically this is the FIFO of the USB controller IC), before the read command is issued. VM-USB provides a mechanism for supplying data, even when the host program is "frozen" in a state of waiting for data. The mechanism consists in starting a second copy of the program and issuing a bare request for data command from this second copy, not followed by the read IN Packet command.

The libxxusb package of VM-USB access functions makes overlapped USB calls that have preset timeout periods. When no data is available until the end of this period, the I/O is canceled and the respective function returns error code. The user is then expected to take proper actions, which may include resubmitting the call.

It is important to specify a sufficiently long In Packet size to be at least of the size of the actual data buffer available at the Common Output Buffer. This is especially important in the case of reading VME data buffers which differ in size substantially depending on the structure of the VME Readout Stack.

4.1 General structure of Out Packets

Since internally, the USB controller of the VM-USB is set up as a 16-bit wide FIFO (First-In-First-Out Memory), the In and Out Packets are organized as collections of 16-bit words.

For the purpose of the software, and more specifically, of the Windows Application Programming Interface (API) routines, the data are packed in byte-wide buffers, a process that may remain transparent to the user when proper set of routines (DLLs) is used. Also, much of the technical information on writing and reading back data from the internal devices of the VM-USB may be considered redundant, when a set of routines is available to perform the task. This information is, however, necessary for writing such routines.

First (16-bit) word in an Out Packet identifies the internal device/address for which the packet is intended and whether the packet represents a request for data or it contains the data to be stored/interpreted to/by the target device. The latter information is coded in bit 2 (value=4) of the header word, with bit 2 set for write operations and reset for read operations (request for data). The meaning of the second word in the Out Packet depends on the address and represents the sub-address in the case of the Register Block and the number of words to follow, in the case of VME Stacks (VDS and VSS) and the VME Generator (VCMD). The subsequent words in the buffer, if any, represent the data to be stored in the target device or the data to be interpreted and acted upon by the target device (in the case of the VCMD). A detailed description of Out Packets for the four target devices is given below.

4.2 Writing Data to the Register Block

The Out Packet for writing data to the Action Register (the single register) of the Register block is composed of the following words:

1. Target Address = 5 (1 + 4) the target address identifying the register block + the "write" bit.

2. Register Sub-Address = 10 secondary address of the Action Register

3. Data To be Written an 8-bit Action Register data word.

4.3 Reading Back Data from the Register Block

To read back data from the Register block, one must first send a request Out Packet to the Register Block consisting of two words:

1. Target Address = 1 the target address of the Register Block

2. Register secondary address =10 of the Action Register.

4.4 Writing Data to the VME Command Stacks / VME Command Generator

Given the width of the VME address and data buses, the Stacks and the VME commands are organized as a sequence of 32-bit words but coded in lines of 16-bit words.

The Out Packets targeting the 8 VME Command Stacks (VCS) for the DAQ mode and the VME Command Generator (VCG) have identical structure, differing only in the Target Address and in the allowed length. Writing a stack to the VM-USB the first line of the USB out packet contains the Target Address (see table below). The following stack data define first the length (number of following lines) as well as the starting address (0 – 1023, with wrap-around) of the particular VCS stack within the 1kWord stack memory space. For the starting address only the 9 least significant bits are relevant. The starting address is

disregarded in earlier firmware versions (<6.1). The stack data are continued by 2 lines with 16-bit words for each 32 bit instruction.

The VME Command Generator (VCG) is an internal module that interprets the information found either in the VME Stacks (when VM-USB is in data acquisition mode) or in the Out Packet received from the USB port (when VM-USB is in interactive mode / EASY-VME mode). In the latter case the VCG performs a VME command stack, which can be longer than the stack memory itself in order to perform more complex functions. The content of the second line depends on whether the stack data are to be stored in the stack memory or whether the stack is to be executed interactively. In the former case, it contains the starting address of the stack within the stack memory space and in the latter case it represents the high 16 bits of the number of lines in the long VCG stacks.

Please see the example and comments about stack generation at the end of chapter 4.5.

1. Target Address: 8 VCS (ID 0-7) and VCG

5	4	3	2	1	0
VCS ID Bits1-2		VCG Bit	Write Bit	VCS Bit	VCS ID Bit 0

2. Stack size: Number of subsequent words (low word of Number for VCG)

3. Stack starting address (high word of Number for VCG)

4. Stack word 1

5. Stack word 2

…

N*2. Last stack word (N=Number of 32-bit words in stack + 2)

4.5 Structure of the VME Stack

The VME stack consists of a sequence VME commands encoded in 32-bit words. Every command is encoded in two or more words, with the first one always specifying the VME address modifier along with a number of options (Mode Word)). With the exception of multi-block transfer, the second word specifies the VME address (Address Word).

The first Mode Word stores the 6-bit VME address modifier, along with other data defining the write/read mode and the format of the VME data, and some additional options:

15	14	13	12	11	10	9	8	7-6	5-0
*	DLY	MRK	SLF	MB	NA	*	NW	DS	AM

31-24	23-22	21-20	19	18	17	16
BLT	NT	*	HM	ND	HD	BE

Where the individual bits have the following meaning:

AM 6-bit VME Address Modifier

DS VME data strobes DS0 and DS1. Zero indicates that a particular strobe is generated, while 1 suppresses the strobe.

NW Write mode – NW=0 indicates "Write" operation and NW=1, "Read" operation.

MB Multi Block transfer, with firmware 16000503 32-bit multi-block transfer can be performed by repeating standard 32-bit block transfers for a predetermined number of times, with the starting address being incremented every time by 0x100 (i.e. 256 bytes). The number of repetitions is practically unlimited (32-bit value). To use this option, bit 11 (value 0x8000) of the Mode Word must be set and the word itself is to be followed by a 32-bit word specifying the desired number of (auto-incrementing) repetitions. The length of the blocks is defined by bits 24-31 of the Mode Word.

NA **NA=1 suppresses address auto-increment in multi-BLT transfers.**

SLF SLF=1 indicates access to Internal Register File in firmware 95000405 and later.

MRK **MRK=1 indicates writing of a marker word directly into the output data stream.**

DLY **DLY=1 indcicates that the VM-USB should pause for a specified amount of time before executing the rest of the out packet. The delay time is specfied in microseconds in the BLT bits (24-31). (Before Firmware 805, this bit is ignored.)**

***** Don't care

BE Endianess – BE=1 sets data mode to big endian (VME default) and BE=0 indicates little endian.

HD Hit Data - identifies the data as a 16-bit hit register data (coincidence register data), to be used for the conditional readout of subsequent VME modules. There may be one or more hit registers in a stack, with the most recent being active.

ND Number Data - identifies the data as a 32-bit word containing in certain field the desired length of the subsequent transfer. The actual number is extracted by taking a logical AND with a 32-bit mask word stored in Number Extract Mask Register and it overrides the number specified in the dedicated bit field of the subsequent block transfer command (first command line, bits 24-31).

HM Hit Mode - instructs the command Generator to condition the readout with the content of the latest hit pattern read in the event. The Number of Product Terms used to condition the readout must be specified as well.

NT Number of Product Terms – specifies the number of 32-bit words in the stack that follow and that constitute bit masks for constructing a logical equation used in deciding whether the given operation is to be performed for the particular hit register data.

BLT Number of transfers in block transfer mode (defined in address modifier word, AM)

The second stack line contains VME address, which can be 16-bit, 24-bit, or 32-bit wide, depending on the address modifier word AM. Bit 0 represents the VME LWORD.

The following rules apply:

(i) When NW=0 (write mode), the second stack line must be followed by one (single "write") or BLT (block transfer) data lines.

(ii) When either SLF or MRK bit is set, the VME address modifier code AM is disregarded.

The "write" data must be properly formatted, according to their endianess and their placement on the bus. The latter is defined by values of the data strobes, A(0) (LWORD), and A(1).

Various "write" data formatting patterns and the corresponding Mode / Address bits are illustrated in the table below.

BE	DS1	DS0	A(1)	A(0) LWORD	D31-D24	D23-D16	D15-D8	D7-0
1	0	0	0	0	Byte(0)	Byte(1)	Byte(2)	Byte(3)
1	0	1	0	0	Byte(0)	Byte(1)	Byte(2)	-
1	1	0	0	0	-	Byte(1)	Byte(2)	Byte(3)
1	0	0	1	0	-	Byte(1)	Byte(2)	-
1	0	0	1	1	-	-	Byte(2)	Byte(3)
1	0	0	0	1	-	-	Byte(0)	Byte(1)
1	1	0	1	1	-	-	-	Byte(3)
1	0	1	1	1	-	-	Byte(2)	-
1	1	0	0	1	-	-	-	Byte(1)
1	0	1	0	1	-	-	Byte(0)	-
0	0	0	0	0	Byte(3)	Byte(2)	Byte(1)	Byte(0)
0	0	1	0	0	Byte(2)	Byte(1)	Byte(0)	-
0	1	0	0	0	-	Byte(3)	Byte(2)	Byte(1)
0	0	0	1	0	-	Byte(2)	Byte(1)	-
0	0	0	1	1	-	-	Byte(3)	Byte(2)
0	0	0	0	1	-	-	Byte(1)	Byte(0)
0	1	0	1	1	-	-	-	Byte(0)
0	0	1	1	1	-	-	Byte(1)	-
0	1	0	0	1	-	-	-	Byte(2)
0	0	1	0	1	-	-	Byte(3)	-

Byte(0) is the least significant byte. The A(1) and A(0)/LWORD bits have to be defined accordingly in the Address Word.

The data read from the VMEBus are either 16-bit or 32-bit wide. They are filled always from Byte(0) in a contiguous manner. Single-byte and two-byte data are stored in 16-bit words, while longer data are stored in 32-bit words.

Since the stack can be quite complex, it is advisable to write a proper routine or macro to set it up. As a convenient option, one may utilize the XXUSBWin Windows application to build the stack and save it to disk.

4.5.11 Single Transfer Commands

A single transfer commands consists of a proper Mode Word followed by an Address Word and, if applicable (a *Write* command), by the data word.

4.5.12 Block Transfer Commands

A block transfer command consists of a proper Mode Word followed by an Address Word specifying the starting address of the block. A block cannot cross the 256-byte boundary and, thus, its maximum length (default) is 256 bytes. If blocks of length shorter than 256 bytes are to be transfered, the desired length is to be specified in bits 24-31, otherwise this field is to be left 0x00. In the case of block *Write* command, the above two words are to be followed by the data words.

Block transfer terminates upon receipt of bus error BERR and, in the case of *Read* commands, 0xFF..FF is written to the output data stream to signal the fact of such a termination.

4.5.13 Multiblock Transfer

A multiblock transfer feature allows one to perform a sequence of block transfers with or without starting address increment for the consecutive blocks and, thus, to read/write large amounts of data from/to addressable memories and FIFOs. The increment of the starting addresses for the subsequent block, if not suppressed, is 256 bytes, i.e., 0x100. The starting address must be on the 256-byte boundary, i.e., to have the least significant byte equal zero. The block length may be set to less than 256 bytes, in which case noncontiguous memory locations will be accessed.

To suppress incrementing of the starting addresses of consecutive blocks, such as might be needed to access long FIFOs, on must set bit 10 (value 0x400) of the Mode word.

In the case when the starting or end address of the desired transfer does not fall on the 256-byte boundary, single block transfers must be performed to handle the partial blocks, with the balance of the transfer executed by one multiblock command.

Multiblock transfer terminates upon receipt of bus error BERR and, in the case of *Read* commands, 0xFF..FF is written to the output data stream to signal the fact of such a termination.

4.5.14 Writing Marker Words into the Output Data Stream

Firmware 66000701 and newer allows one to insert marker words into the output data stream to mark desired locations of data within an event to facilitate viewing and unpacking of the data buffers. A typical use is to mark the ends of events and ends of long blocks of data. To write a marker, a two-word *Write Marker* command must be inserted into the command stack, such that the Mode Word identifies in its bit 13 (value 0x2000) the command as a *Write Marker* command. This Mode Word may be simply 0x2000 (the AM code is disregarded) and must be followed by a 32-bit marker data data specifying the desired marker

in its 16 least significant bits. When a 32-bit alignment of data is requested, two marker words are inserted with the second one composed of the 16 most significant bits of the specified data. No address word is to be inserted.

4.5.15 Use of Hit Registers for Conditional Execution of Stack Commands

VM-USB allows one to use hit registers (coincidence registers) to perform conditional execution of any stack commands. To perform such a conditional execution, one must first define a module to be read as a hit register module and then identify any of the subsequent commands as conditional ones, while specifying the conditions.

A module is identified in the command stack as a hit register by setting bit 17 (value 0x20000) in the Mode Word. The data read from such a module are stored in a register and used in subsequent conditional operations to verify if a desired condition is met. One may have any number of hit registers defined, each overriding the latest hit word and providing a new hit pattern for subsequent conditional readouts.

A command to be executed conditionally is identified in the command stack by setting bit 19 (value 0x80000) of its Mode Word. Additionally, one must specify in bits 22-23 (values 0x400000 and 0x800000) of the Mode Word the number of terms less one (i.e., zero indicates one mask word) in the logical equation (condition) and supply the specified number of 32-bit mask words. The mask words must follow the address word.

The command is executed when the following logical condition composed of up to fourfold OR of 32-fold ANDs is met:

(BMask(1) AND HD = BMask(1)) OR

(BMask(2) AND HD = BMask(2)) OR

(BMask(3) AND HD = BMask(3)) OR

(BMask(4) AND HD = BMask(4)),

where HD represents the most recent hit data retrieved from a hit register and BMask(*) are the (up to four) 32-bit mask words. Note that the command will be executed whenever in the most recent hit register data all bits specified in any of the used Bit Masks are set. Note that one can readily use more than four terms by repeating the same conditional command with a set of different mask words.

4.5.16 Using Dynamical Block Sizing for Block Transfers

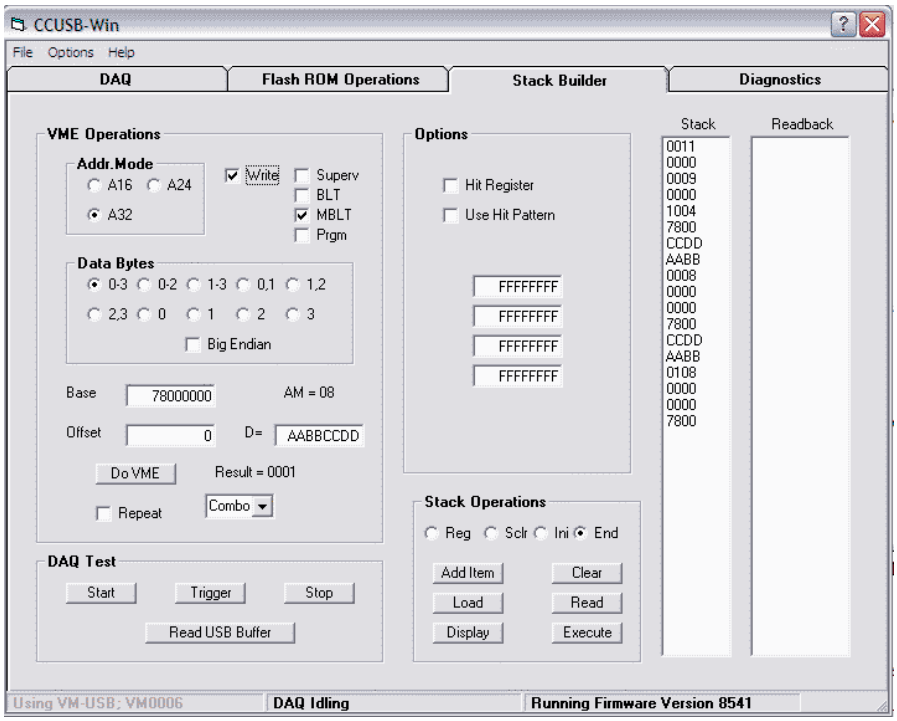
VM-USB allows one to perform block transfers with block lengths retrieved dynamically from any of the VME devices. To use this feature, one must first specify which portion of the retrieved data (field) contains the desired number of transfers to be performed. This is done

by storing the suitable mask word in Number Extract Mask Register. Then, in the command stack, one must identify a command that is to retrieve the desired number, by setting bit 18 (value 0x40000) in its Mode Register. With this done, the subsequent block transfer will use the retrieved number as the number of transfers, overriding any data stored in bits 24-31 of the Mode Word.

The number extract mask word must have contiguous set of bits set to identify the position of the desired number within the data word retrieved. Valid numbers are 0 – 256.

4.5.17 Using XXUSBWin Application to Handle Stacks

The MS Windows application XXUSBWin allows one to create, save, and read, as well as to upload the VME command stack list in an easy and convenient way.



Further, it is possible to create the VME command stack with either a text editor or user program. All required programming details are given in chapter 4.5.

The following example shows the VME command stack (as saved to file) for a VME 32-bit write to address 0x78000020 / data 0xAAAAFFFF as well as a VME 16-bit read from address 0x78000120, both with AM=0x09. Please note that the VME command stack is based on 32-bit words but arrange in 16-bit lines, i.e. 2 consequent lines belong to one 32-bit word. Explanations are added in blue color:

VM-USB Command Stack Generated on 8/29/2005 at 5:56:57 PM

```
        A               // number of lines (decimal 11)
        0000            // the starting address to store the stack within the stack buffer
        0009            // AM / write mode (bits 0-15)
        0000            // BLT / special modes (bits 16-31)
        0020            // VME address (bits 0-15)
        7800            // VME address (bits 16-31)
        FFFF            // data (bits 0-15)
        AAAA            // data (bits 16-31)
        0109            // AM / write mode (bits 0-15), bit 8=1 for read
        0000            // BLT / special modes (bits 16-31)
        0121            // VME address (bits 0-15), A0=1 for 16bit
7800            // VME address (bits 16-31)
```

4.6 Structure of the IN Packets

The General Output Buffer is associated with Endpoint 6 of the USB2 controller IC, which is configured as a 512 byte deep FIFO. This endpoint is configured for bulk transfer and one can specify lengths of buffers to be read of any length (up to 26 kBytes) compatible with the VM-USB functionality. All data supplied by the VM-USB is to be read from the Endpoint 6. While reading, it is important to specify the length of the buffer not shorter than the length of the actual data buffer written by the VM-USB into this endpoint.

The structure of data retrieved in conjunction with direct requests for data addressed to the Register Block and to the VME Stacks is straightforward, such that the buffer consists only of the requested data.

The data buffers read during the data acquisition process have a structure depending on the buffer filling mode selected by bit 4 of BuffOpt code specified in the Global Register. The default filling is such that the buffer contains only complete events (bit 4 of BuffOpt=0). On the other hand, when bit 4 of BuffOpt is set, continuous filling is selected allowing single events to span two or more buffers. Whenever the size of a single event exceeds the declared size of the data buffer and the filling mode is set for complete events, the filling mode switches to continuous mode, with this fact tagged by setting of the bit 13 of the buffer header word. For the Complete Event Mode, the data buffer has the following structure:

1. Header word

15	14	13	12	11-0
LB	Scaler	Cont	MB	NE

LB= Last Buffer, is 1 if the buffer is last buffer of a run

Scaler=Scaler Buffer, is 1 if the data in the buffer is from the scaler stack

Cont= Indicates the module switched to a continuous mode

MB= Indicates that the event data spans several buffers

NE=Number of Events, indicates how many events are in the buffer

2. Optional 2nd Header Word Bits 0-11 represent the number of words in the buffer.

3. Event Header

15-13	12	11-0
Stack Id	Continuation bit	Event length

Event length (in 16 or 32-bit words depending on read type) including terminator words.

4. Event Data

5. Optional Event Terminator (user definable, 0xAAAA in firmware before 660007010

…

… ***Subsequent Events***

…

Buffer Terminator 0xFFFF

Second Buffer Terminator 0xFFFF – firmware 66000701 and newer

Firmware 95000405 and later allows one to mix regular and scaler events in a common data buffer. In a mixed buffer mode, the scaler data are identified by bit 15=1 of the Event Length word.

Note that write operations performed in autonomous mode (data acquisition) do not return data into the data stream.

The unpacking of the events must be done in accordance with the VME Stack that is involved in generating the buffer.

In the Continuous (split-event) mode, when events span two or more buffers, no buffer terminator is written.

For the direct access of the VME Command Generator (Interactive VME operations), no header words are written and the In Packet contains only one event.

VM-USB has dedicated 2kWords-long event FIFO to compile events. To handle longer events, VM-USB splits the long event into parts, each of which appears as a separate event in the output buffer. The partial events are tagged by setting bit 12 of the Event Length word, except for the last part. Also, only the last "installment" is terminated by the Event Terminator word (s).

VM-USB has a provision to automatically switch the output buffer packing mode to Split-Event mode, whenever the Event Length exceeds the length of the Integer-Event buffer. Setting of bit 13 in the buffer header word indicates the fact of such a change.

With firmware 16000503 and the introduction of up to 8 executable stacks the structure of the event header (specifying the number of words in the event) has been modified to identify the stack ID associated with this event. The 3-bit stack ID is stored in bits 13-15 (counting from 0) of the event header word. Unless a mixed-event option is selected for data buffering (recommended for increased band width), the buffers will contain events with identical stack IDs.

In interactive mode, VME Write command returns a single word, but only when this command is the last command in the interactive stack. Note that this includes the case when the write command is the only command in the stack. The returned word is "1" when the

operation was successful (DTACK was received) and is “0” when bus error, BERR was received instead of DTACK.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	S

VME 8 and 16-bit Read returns a single word with data in bits 0-8 and 0-15, respectively:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1.	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

VME 24-bit Read: returns 2 words, with data bits 0-15 and 16-23, respectively:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1.	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
2.	0	0	0	0	0	0	0	0	D23	D22	D21	D20	D19	D18	D17	D16

VME 32-bit Read: returns 2 words, with data bits 0-15 and 16-31, respectively:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1.	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
2.	D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16

5 GUIDE TO LIST MODE DATA ACQUISITION WITH VM-USB

VM-USB is intended for use in list mode data acquisition, where it performs sequences of desired VME operations pursuant to stack(s) stored in it, upon receipt of event trigger. VM-USB then formats the data read form the VME bus and buffers them in a data buffer. When the buffer is full, VM-USB transfers its content to the In FIFO of the USB controller IC for readout by host software.

To set up VM-USB for data acquisition in list mode the following steps are advised:

1. Build the regular VME command stack by adding all the desired simple and complex commands to it. One must make sure that the stack sequence will clear all VME modules. It is recommended to first execute the stack from the host software to verify that it performs as intended. For this purpose the libxxusb library function xxusb_stack_execute can be used. For firmware 66000701 and newer, insert one or more marker words at the end of the stack to mark the end of an event in the data stream.
2. Load the stack into the VM-USB memory, e.g., by calling the libxxusb library function xxusb_stack_write. It is recommended to read back the stack (function xxusb_stack_read), to verify that the stack is correctly stored.
3. When the setup calls for it, build and load the auxiliary (scaler) stack and define the readout mode and frequency.
4. Set the trigger delay (time from the receipt of an event to the commencement of the stack execution).
5. If VM-USB is not the slot 1 controller, set up the bus request level, by writing the bus request level code into bits 12-14 of the Global Register.
6. Set up event termination mode. By default, VM-USB terminates every event by one terminator word 0x5555, with the second word 0xAAAA being optional – applicable to old firmware only. For firmware 0x66000701 and newer, insert a suitable marker word into the command stack to mark the end of an event.
7. Set up buffering mode and data buffer length by writing a suitable 5-bit code into bits 0-4 of the Global Mode Register. The default is buffer length of 13k words and events fitting into one buffer.
8. Set buffer header option. By default, VM-USB writes one buffer header word containing information on the number of events in the buffer, buffer type (regular, or periodic auxiliary), and the buffer termination mode (regular or watchdog).
9. Start acquisition by setting bit 0 of the Action Register to 1. End acquisition by resetting this bit to “0”. While in acquisition mode, the host software is expected to read the USB port In FIFO in a loop, to empty it and make space for subsequent events.

6 LIBXXUSB LIBRARY FOR WINDOWS AND LINUX

A dedicated library of functions was developed to facilitate the utilization of VM-USB and its CAMAC counterpart, CC-USB. This library is called libxxusb and requires the libusb0.sys driver to be installed. It is in fact a wrapper library for the general-use libusb-win32 library available via www.sourceforge.net at no charge. All functions are part of the libxxusb.dll dynamically loadable library.

For linux the library is called libxx_usb. All the functions are identical to the ones used in Windows.

All xxusb functions for both 32-bit MS Windows (Win98SE, WinME, Win2k, WinXP) as well as for Linux rely on the USB library "libusb-win32"(Windows) or "libusb" (Linux). For further details about these libraries please see www.sourceforge.net or http://sf.net/projects/libusb/ .
The following functions are used for both the VM_USB and its' counterpart the CC_USB.

6.1 xxusb_devices_find

The xxusb_devices_find function retrieves relevant parameters of USB ports of all XX-USB devices attached to the host and returns these in an array of proper structures. This is the first command to be issued when attempting to establish communication with an XX-USB.

```
WORD xxusb_devices_find{
    XXUSB_DEVICE_TYPE lpXXUSBDevice,
};
```

Parameters

lpXXUSBDevice

[out] Pointer to an array of structures storing parameters of all XX-USB devices identified.

Return Values

On success, the function returns the number of XX-USB devices found, including 0.
A negative return value indicates that the handle to a valid device could not be opened as a result of insufficient privileges. It is recommended to retry in Superuser mode.

6.2 xxusb_device_open

The xxusb_device_open function obtains handle to the desired XX-USB device, identified by xxusb_devices_find command. This is the second command to be issued when attempting to establish communication with an XX-USB. The obtained handle is then to be used while calling various xxusb_*_* functions, that require the handle. Upon termination of a XX-USB session, the handle is to be released by calling xxusb_handle_close.

```
WORD xxusb_device_open{
    USB_DEVICE_TYPE lpUSBDevice,
};
```

Parameters
lpUSBDevice
[in] Pointer to a structure storing parameters of the target XX-USB devices.

Return Values
On success, the function returns the handle to the target XX-USB device. A negative return value indicates that the handle to a valid device could not be opened as a result of insufficient privileges. It is recommended to retry in Superuser mode.

Remarks
While all xxusb functions rely on the libusb (www.sourceforge.net) functions while communicating with XX-USB, xxusb_device_open and xxusb_handle_close are simply macros creating aliases to usb_open and usb_close functions of the libusb library.

6.3 xxusb_serial_open

Opens a xxusb device (CC-USB or VM-USB) whose serial number is given

```
USB_DEV_HANDLE* xxusb_serial_open{
        char *SerialString
};
```

Parameters:
SerialString
a char string that gives the serial number of the device you wish to open. It takes the form:
VM0009 - for a VM_USB with serial number 9 or
CC0009 - for a CC_USB with serial number 9

Returns:
LpUSBDevice
[out] Pointer to a variable containing the handle to the controller

6.4 xxusb_device_close

The xxusb_device_close function closes the handle to the desired XX-USB device, obtained by a xxusb_device_open call. This function is to be called upon termination of an XX-USB session.

```
WORD xxusb_device_close{
        USB_DEV_HANDLE lpUSBDevice,
};
```

Parameters
lpUSBDevice

[in] Pointer to a variable containing the handle to be closed.

Return Values
Returns negative upon failure.

Remarks
While all xxusb functions rely on the libusb (www.sourceforge.net) functions while communicating with XX-USB, xxusb_device_open and xxusb_handle_close are simply macros creating aliases to usb_open and usb_close functions of the libusb library.

6.5 xxusb_reset_toggle

The xxusb_reset_toggle function toggles the reset state of the FPGA while XX-USB is in programming mode – rotary selector set in one of four P* positions.

```
WORD xxusb_reset_toggle{
    HANDLE hDevice,
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.

Return Values
Returns negative upon failure.

6.6 xxusb_register_write

The xxusb_register_write sends a data buffer to XX-USB, causing the latter to store the desired data in the target register.

```
WORD xxusb_register_write{
    HANDLE hDevice,
    WORD wRegisterAddress,
    DWORD dwRegisterData
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.

wRegisterAddress
[in] Address of the XX-USB register.

For a list of XX-USB addresses, see Remarks

dwRegisterData

[in] Data to be stored in the register.

Return Values
On success, the function returns the number of bytes sent to XX-USB.
Function returns 0 on attempted writes to read-only registers and negative numbers on failures.

6.7 xxusb_register_read

The xxusb_register_read function first, sends a buffer to XX-USB, causing the latter to write the content of a desired register to its USB port FIFO and then, obtains the value by reading the buffer from the XX-USB.

```
WORD xxusb_register_read{
    HANDLE hDevice,
    WORD wRegisterAddress,
    LPDWORD lpRegisterData
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.

wRegisterAddress
[in] Address of the XX-USB register.

For a list of XX-USB addresses, see Remarks

lpRegisterData
[out] Pointer to a variable that receives the data returned by the operation, i.e., the value stored at wRegisterAddress of XX-USB.

Return Values
On success, the function returns the number of bytes read XX-USB. Valid values are 2 and 4, with the latter only for LAM Mask and LAM registers.
Function returns a negative number on a failure.

6.8 xxusb_stack_write

The xxusb_stack_write function sends a buffer to XX-USB, causing the latter to store this content in a dedicated block RAM, for use when data acquisition mode is active. This content can be read back using *xxusb_stack_read* function.

```
WORD xxusb_stack_write{
    HANDLE hDevice,
    WORD wStackType,
    LPDWORD lpStackData
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

wStackAddress

[in] Type of the XX-USB stack, the content of which is to be read. Valid types are 2, for the regular stack and 3, for the periodic (scaler) readout stack.

ID	wStackAddress		Device	Trigger
0	2	0x02	Regular Stack	NIM I1, USBstart
1	3	0x03	Scaler Stack	Periodic
2	18	0x12	IRQ stack	VME IRQ 1-7
3	19	0x13	IRQ stack	VME IRQ 1-7
4	34	0x22	IRQ stack	VME IRQ 1-7
5	35	0x23	IRQ stack	VME IRQ 1-7
6	50	0x32	IRQ stack	VME IRQ 1-7
7	51	0x33	IRQ stack	VME IRQ 1-7

lpStackData

[in] Pointer to a variable array that contains the data to be stored in the target stack.

Return Values

On success, the function returns the number of bytes sent to XX-USB. The latter value is twice the length of the stack plus 2 (for a header word identifying a stack as a target).
Function returns a negative number on a failure.

Remarks

The physical length of the regular stack is 768 16-bit words for CC-USB and 768 32-bit words for VM-USB.
The physical length of the periodic (scaler) stack is 256 16-bit words for CC-USB and 256 32-bit words for VM-USB.
While the stack is expected to contain properly encoded sequence of VME (VM-USB) or VME (VM-USB) commands to be performed by XX-USB, it can store any sequence of numbers.

6.9 xxusb_stack_read

The xxusb_stack_read function first, sends a buffer to XX-USB, causing the latter to write the content of a desired stack to its USB port FIFO and then, obtains this content by reading a buffer from the XX-USB.

```
WORD xxusb_stack_read{
    HANDLE hDevice,
    WORD wStackType,
    LPDWORD lpStackData
};
```

Parameters

hDevice
[in] Handle to the XX-USB device.

wStackAddress
[in] Type of the XX-USB stack, the content of which is to be read. For valid stack addresses see table above at 6.8.

lpStackData
[out] Pointer to a variable array that receives the data returned by the operation, i.e., the content of a XX-USB stack.

Return Values
On success, the function returns the number of bytes read from XX-USB. The valid value is twice the length of the stack, as the latter stores 2-byte words.
Function returns a negative number on a failure.

6.10 xxusb_stack_execute

The xxusb_stack_execute function first, sends a buffer to XX-USB, causing the latter to interprete its content as a series of simple and complex VME commands and to actually execute these commands and to write the returned VME data to the USB port FIFO. Then, xxusb_stack_execute reads a buffer from XX-USB, containing the desired VME data.

```
WORD xxusb_stack_execute{
    HANDLE hDevice,
    LPDWORD lpData,
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.

lpData
[in] Pointer to a dual-use variable array. When calling the function, the array contains the data encoding the sequence of desired commands (VME commands for VM-USB and VME commands for VM-USB) to be performed by XX-USB. The first element of the array is the number of bytes. The following command has to be defined similar to the VME / VME command stack (see paragraph 4.5). Upon return, the array contains the VME (VM-USB) or VME (VM-USB) data, respectively.

Return Values
On success, the function returns the number of bytes read from XX-USB. The valid value is twice the number of 16-bit data words returned plus 2 (CC-USB) or 4(VM-USB). The latter "overhead" bytes contain event terminator word (0xFF for VM-USB, and 0xFFFF for VM-USB).
Function returns a negative number on a failure.

6.11 xxusb_ longstack_execute

Executes stack array passed to the function and returns the data read from the VME bus

```
int  xxusb_longstack_execute{
        HANDLE hDevice,
        void *DataBuffer,
        int lDataLen,
        int timeout
};
```

Paramters:

hDevice

[in] Handle to the XX-USB device.

DataBuffer

pointer to the dual use buffer; when calling DataBuffer contains (unsigned short) stack data, with first word serving as a placeholder, upon successful return, DataBuffer contains (unsigned short) VME data

lDataLen

The number of bytes to be fetched from VME bus - not less than the actual number expected, or the function will return -5 code. For stack consisting only of write operations, lDataLen may be set to 1.

Timeout

The time in ms that should be spent tryimg to write data.

Return Values

When Successful, the number of bytes read from xxusb.
Upon failure, a negative number

Remarks

The function must pass a pointer to an array of unsigned integer stack data, in which the first word is left empty to serve as a placeholder.
The function is intended for executing long stacks, up to 4 MBytes long, both "write" and "read" oriented, such as using multi-block transfer operations.

Structure upon call:

DataBuffer(0) = 0(don't care place holder)
DataBuffer(1) = (unsigned short)StackLength bits 0-15
DataBuffer(2) = (unsigned short)StackLength bits 16-20
DataBuffer(3 - StackLength +2) (unsigned short) stack data

StackLength represents the number of words following DataBuffer(1) word, thus the total number of words is StackLength+2

Structure upon return:

DataBuffer(0 - (ReturnValue/2-1)) - (unsigned short)array of returned data when ReturnValue>0

6.12 xxusb_usbfifo_read

The xxusb_usbfifo_read function reads the content of the USB port FIFO of XX-USB or times out whenever this FIFO has not set the “FIFO Full” flag.

```
WORD xxusb_usbfifo_read{
    HANDLE hDevice,
    LPDWORD lpData,
    WORD wDataLen,
    WORD wTimeout
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

lpData

[out] Pointer to a variable array that receives the data returned by the operation, i.e., the content of the USB port output FIFO of the XX-USB.

wDataLen

[in] Number of words to read. This number must be not less than the number of bytes stored in the output FIFO.

wTimeout

[in] Time in milliseconds, after which the I/O operation is canceled, should there be no data available for the readout.

Return Values

On success, the function returns the number of bytes read from XX-USB.
Function returns a negative number on a failure, which in most cases signifies a timeout condition.

Remarks

The xxusb_usbfifo_read is intended for use while XX-USB is in data acquisition mode. Upon timeouts, the host application receives the control and may reissue the command or terminate the acquisition

6.13 xxusb_bulk_read

The xxusb_bulk_read function reads the content of the USB port FIFO of XX-USB or times out whenever this FIFO has not set the “FIFO Full” flag.

```
WORD xxusb_bulk_read{
```

```
        HANDLE hDevice,
        xxxx *pData,
        WORD wDataLen,
        WORD wTimeout
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

pData

[out] pointer to an array to store data that is read from the VME bus; the array may be declared as byte, unsigned short, or unsigned long.

wDataLen

[in] Number of bytes to read. This number must be not less than the number of bytes stored in the output FIFO.

wTimeout

[in] Time in milliseconds, after which the I/O operation is canceled, should there be no data available for the readout.

Return Values

On success, the function returns the number of bytes read from XX-USB. Function returns a negative number on a failure, which in most cases signifies a timeout condition.

Remarks

Depending upon the actual need, the function may be used to return the data in a form of an array of bytes, unsigned short integers (16 bits), or unsigned long integers (32 bits). The latter option of passing a pointer to an array of unsigned long integers is meaningful when xxusb data buffering option is used (bit 7=128 of the global register) that requires data 32-bit data alignment.

6.14 xxusb_bulk_write

The xxusb_bulk_write function writes a character array to the USB port FIFO of XX-USB.

```
WORD xxusb_bulk_write{
        HANDLE hDevice,
        xxxx *pData,
        WORD wDataLen,
        WORD wTimeout
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

pData
[out] pointer to an array storing the data to be sent; the array may be declared as byte, unsigned short, or unsigned long.

wDataLen
[in] Number of bytes to read. This number must be not less than the number of bytes stored in the output FIFO.

wTimeout
[in] Time in milliseconds, after which the I/O operation is canceled, should there be no data available for the readout.

Return Values
On success, the function returns the number of bytes read from XX-USB. Function returns a negative number on a failure, which in most cases signifies a timeout condition.

Remarks
Depending upon the actual need, the function may be used to pass to xxusb the data in a form of an array of bytes, unsigned short integers (16 bits), or unsigned long integers (32 bits).

6.15 xxusb_flashblock_program

The xxusb_flashblock_program function programs one sector of 256 bytes of the flash memory (FPGA configuration memory)

```
WORD xxusb_usbfifo_read{
    HANDLE hDevice,
    UCHAR *pData,
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.
pData
[out] Pointer to the configuration (byte) data array.

Return Values
On success, the function returns the number of bytes written to XX-USB – the correct number is 518.
Function returns a negative number on a failure, which in most cases signifies a timeout condition.

Remarks
To program the flash memory, one must call repeatedly xxusb_flashblock_program, while pausing for at least 30ms between consecutive calls and incrementing the pointer to the data array by 256 on each consecutive call. The device must be in programming mode with the rotary selector in one of the 4 “P” positions and with the red “Fail” LED on.

The configuration file of a XC3S200 FPGA of VM-USB will occupy 512 sectors of flash memory (512 calls to the xxusb_flashblock_program). The XC3S400 FPGA of VM-USB will occupy 830 sectors of flash memory.

7 VM_USB SPECIFIC FUNCTIONS

The following functions are specific to the VM_USB. They are built on top of the general purpose functions described in section 6 and provide users with an easier and more transparent way of communicating with the controller.

7.1 VME_register_write

The VME_register_write function writes to the internal registers of the VM_USB as described in section 3.4.

```
short VME_register_write{
        HANDLE hDevice,
        USHORT Address,
        LONG Data
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

Address

[in] VME offset of the register you wish to write to

Data

[in] Data to be written to the specified register

Return Values

On success, the function returns the number of bytes written to VM-USB
Function returns a negative number on a failure

7.2 VME_register_read

The VME_register_read function reads from the internal registers of the VM_USB as described in section 3.4.

```
short VME_register_read{
        HANDLE hDevice,
        USHORT Address,
        LONG Data
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

Address

[in] VME offset to read from, as specified in Section 3.4

Data

[out] Data read from the specified register

Return Values

On success, the function returns the number of bytes read from the VM-USB
Function returns a negative number on a failure

7.3 VME_DGG

The VME_DGG function allows the user to setup the characteristics of the Delay and Generator channels of the VM_USB.

```
short VME_DGG{
        HANDLE hDevice,
        USHORT channel,
        USHORT trigger,
        USHORT output,
        LONG delay,
        USHORT gate,
        USHORT invert,
        USHORT latch
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

channel

[in] The DGG channel you wish to modify. Valid values are:
- 0 – For DGG channel A
- 1 – For DGG channel B

trigger

[in] Determines the start of the DGG. Valid Values are:
- 0 – Channel Disabled
- 1 – NIM input 1
- 2 – NIM input 2
- 3 – Event Trigger
- 4 - End of Event
- 5 - USB Trigger
- 6 - Pulser

output

[in] Determines the NIM ouput used for the DGG channel. Valid values are:
- 0 – NIM O1
- 1 – NIM O2

delay

[in] 32 bit word in steps of 12.5ns between trigger and start of gate consisting of lower 16 bits: Delay_fine

upper 16 bits: Delay_coarse

gate

[in] Sets the length of the gate in units of 12.5ns

invert

[in] Determines whether or not the DGG is inverted. Valid values are:

0 – Not inverted

1 – Is inverted

latch

[in] Determines whether or not the DGG is latched. Valid values are:

0 – Not latched

1 – Is latched

Return Values

On success, the function returns 1

Function returns a negative number on a failure

7.4 VME_LED_settings

The VME_LED_settings function allows the user to setup the LEDs on the front panel of the VM_USB. Details about the LED settings are found in section 3.4.4.

```
short VME_LED_settings{
    HANDLE hDevice,
    USHORT LED,
    USHORT code,
    USHORT invert,
    USHORT latch
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

LED

[in] The LED you wish to modify. Valid values are:

0 – Top YELLOW

1 – RED

2 – GREEN

3 – Bottom Yellow

code

[in] Determines what event the LED is linked to. Valid valuesare 0-7 and are described in section 3.4.3

invert

[in] Determines whether or not the LED is inverted. Valid values are:

0 – Not inverted

1 – Is inverted

latch

[in] Determines whether or not the LED is latched. Valid values are:

0 – Not latched
1 – Is latched

Return Values
On success, the function returns the number of bytes from from the VM_USB.
Function returns a negative number on a failure

7.5 VME_Output_settings

The VME_Output_settings function allows the user to setup the NIM outputs on the front panel of the VME_USB. Details about the output settings are found in section 3.4.5

```
short VME_Output_settings{
        HANDLE hDevice,
        USHORT Channel,
        USHORT code,
        USHORT invert,
        USHORT latch
};
```

Parameters
hDevice
[in] Handle to the XX-USB device.
Channel
[in] The NIIM output channel you wish to modify. Valid values are:
1 – O1
2 – O2
code
[in] Determines what event the output is linked to. Valid valuesare 0-7 and are described in section 2.4.5
invert
[in] Determines whether or not the output is inverted. Valid values are:
0 – Not inverted
1 – Is inverted
latch
[in] Determines whether or not the output is latched. Valid values are:
0 – Not latched
1 – Is latched

Return Values
On success, the function returns the number of bytes from from the VM_USB.
Function returns a negative number on a failure

7.6 VME_scaler _settings

Configures the internal VM-USB scaler (SelSource register)

```
short  VME_scaler_settings{
        usb_dev_handle *hdev,
        short channel,
        short trigger,
        int enable,
        int reset}
```

Parameters:

hDevice
[in] Handle to the XX-USB device

channel
[in] The number which corresponds to the scaler channel:
0 - for Scaler A
1 - for Scaler B

code
[in] The Output selector code, valid values are listed in the manual

enable
[in] =1 enables scaler

latch
[in] =1 resets the scaler at time of call

Return Values
On success, the function returns the number of bytes from from the VM_USB.
Function returns a negative number on a failure.

7.7 VME_write_32

The VME_write_32 function writes a 32 bit word to a VME address.

```
short VME_write_32{
        HANDLE hDevice,
        USHORT Address_Modifier,
        LONG VME_Address,
        LONG Data
};
```

Parameters

hDevice
[in] Handle to the XX-USB device.

Address_Modifier
[in] VME Address modifier

VME_Address
[in] VME address to write Data to

Data
[in] data written to the VM_USB

Return Values

On success, the function returns the number of bytes written to VM-USB
Function returns a negative number on a failure.

7.8 VME_read_32

The VME_read_32 function reads a 32 bit word from a VME address.

```
short VME_read_32{
        HANDLE hDevice,
        USHORT Address_Modifier,
        LONG VME_Address,
        LONG Data
};
```

Parameters

hDevice
[in] Handle to the XX-USB device.

Address_Modifier
[in] VME Address modifier

VME_Address
[in] VME address to read Data from

Data
[in] data read from the VM_USB

Return Values

On success, the function returns the number of bytes read from VM-USB
Function returns a negative number on a failure

7.9 VME_write_16

The VME_write_16 function writes a 16 bit word to a VME address.

```
short VME_write_32{
        HANDLE hDevice,
        USHORT Address_Modifier,
        LONG VME_Address,
        LONG Data
};
```

Parameters

hDevice
[in] Handle to the XX-USB device.

Address_Modifier
[in] VME Address modifier

VME_Address

[in] VME address to write Data to

Data

[in] data written to the VM_USB

Return Values

On success, the function returns the number of bytes written to VM-USB
Function returns a negative number on a failure

7.10 VME_read_16

The VME_read_16 function reads a 16 bit word from a VME address.

```
short VME_read_32{
        HANDLE hDevice,
        USHORT Address_Modifier,
        LONG VME_Address,
        LONG Data
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

Address_Modifier

[in] VME Address modifier

VME_Address

[in] VME address to read Data from

Data

[in] data read from the VM_USB

Return Values

On success, the function returns the number of bytes read from VM-USB
Function returns a negative number on a failure

7.11 VME_BLT_read_32

The VME_read_32 function performs a 32 bit block transfer from a VME address. The number of times the read is repeated depends on the parameters passed

```
short VME_read_32{
        HANDLE hDevice,
        USHORT Address_Modifier,
        USHORT count,
        LONG VME_Address,
        LONG Data
```

};

Parameters

hDevice

[in] Handle to the XX-USB device.

Address_Modifier

[in] VME Address modifier

count

[in] The number of times to repeat the read

VME_Address

[in] VME address to read Data from

Data

[out] pointer to an array where Data can be stored read from the VM_USB

Return Values

On success, the function returns the number of bytes read from VM-USB
Function returns a negative number on a failure

APPENDIX A: Programming and the Use of Flash Memory

The VM-USB stores the FPGA configuration files in a 1-MByte flash memory (organized as two 512-kByte ICs). This size of the memory allows one to accommodate up to 4 configuration files of an XC3S400 FPGA, such as used in VM-USB. The individual address spaces of the four possible configuration files are selected by the state of two bits a front-panel rotary selector switch. This rotary switch provides, in fact, for a 3-bit (1 out of 8) selection, with third bit being used to select one of the two possible modes of operation – "Configuring" (C) and "Programming" (P) modes, respectively. Normally, VM-USB is operated in the C (C1-C4 labels) mode, with the FPGA cold-booting itself upon power up from a selected segment of the memory, identified by the digit of the label.

The design of the VM-USB allows one to reprogram any of the four segments of the flash memory via the USB interface. The programming and reprogramming is possible only in the Programming mode, selected by any of the P* settings (P1-P4 labels) of the rotary selector and when the FPGA is kept in a reset mode by the boot manager CPLD, which is a default after cold-booting in the P mode (red "Fail" LED on). While in the P mode, the manager CPLD can be instructed, via the USB interface, to release the FPGA from reset. When released from the reset state, the FPGA will attempt to boot itself from the selected segment of flash memory, with a successful boot indicated by the red "Fail" LED turning off. Subsequently, but only upon a successful boot, the CPLD can be instructed to assert again the reset signal for the FPGA, allowing one to undertake another programming sequence.

While the VM-USB can be operated in the P mode, with the FPGA reset released, it is recommended to switch to a respective C setting for a regular operation.

In fact, there are only two types of operations specific to P mode – programming of a 256-byte sector of the flash memory (with automatic address increment) and toggling of the reset status of the FPGA. Accordingly, there are two types of data buffers that must be sent to the USB port by the host to perform the desired operation.

The sector programming buffer is 518 bytes long and contains three "sector unlock" (AT29LV040 software protection override) data bytes and the 256 configuration file bytes in odd bytes (counting from 0), with all even bytes being disregarded:

0	0'**	don't care
1	0'AA	first "sector unlock" code
2	0'**	
3	0'55	second "sector unlock" code
4	0'**	
5	0'A0	third "sector unlock" code
2*n+6	0'**	where n = 0 - 255
2*n+7	byte(k)	consecutive byte of the configuration file

The FPGA "reset toggle" buffer is only two bytes long with the first byte being disregarded and the second one being equal to 0'FF or decimal 255.

In accordance with the above, to program/reprogram a segment of the flash memory one needs to execute the following sequence:

With the FPGA booted ("Fail" LED off):

1. Set the rotary switch to a P position pointing to the desired segment => the red "Fail" LED should turn on.

With VM-USB off or the FPGA failure to boot:

1a. Cold-boot VM-USB with the rotary switch in the desired P position.

2. Send successively the 830 sector programming buffers containing a complete configuration file of the XC3S400 FPGA to the USB port in bulk transfer mode, while specifying the data length as 518 bytes and allowing at least 30ms wait time between the consecutive buffers.

3. Send an FPGA "reset toggle" buffer, or cold-boot VM-USB in the corresponding C mode. A successful boot will be indicated by the "Fail" LED turned off.

The use of the flash memory programming capability is largely facilitated by the availability of dedicated libxxusb functions xxusb_flashblock_program and xxusb_reset_toggle, as illustrated by the following sample codes written in Visual Basic:

```
'Flash memory programming
For i = 0 To 829
   k = i * 256 + 1
   ll = xxusb_flashblock_program(EZHandle, bytes(k))
   DLY (30)   '30ms delay generator
Next i

'Releasing or setting the FPGA reset:
ll=xxusb_reset_toggle(EZHandle)
```

In the above two samples, EZHandle represents the USB handle of the VM-USB and bytes() is an array storing the desired FPGA configuration file (220 kB).

8 APPENDIX B: USE OF MULTIPLEXED USER DEVICES

The FPGA configuration of the VM-USB may set up optionally various user devices, that are beyond the scope of a VME controller, but which are intended to facilitate and reduce the cost of a data acquisition setup. The "release" firmware of the VM-USB, (Firmware Id = 85000402) sets up two delay and gate generators, DGG_A and DGG_B and two 32-bit scalers, SCLR_A and SCLR_B.

8.1 Characteristics and the Use of Delay and Gate Generators

The two user gate and delay generators allow one to generate delays and gates in the range of 12.5 ns – approx. 800 us, with the 12.5 ns granularity.

To make use of an DGG_A or DGG_B, one simply needs to select the desired trigger signal by properly setting the respective selector code bits in the User Devices Register and set write the desired delay and gate data (in units of 12.5 ns) into the respective DGG register, as described in Sections 3.4.5 and 3.4.6.

8.2 Characteristics and the Use of Scalers

The two user scalers allow one to count various signals and read out the resulting numbers in VME-like commands. The latter commands address the VME address space allocated to the VM-USB and do not generate any activity on the VME bus. Both scalers are asynchronous with respect to the VM-USB clock, each using a dedicated fast clock network driven by the selected clock signal.

The use of the scalers is straightforward and entails selecting their respective input sources and enabling their operation by setting the respective "enable" bits. Optionally, one may wish to disable them by resetting the respective "enable" bits or clearing them by writing "1" to the respective "reset" bits.

9 APPENDIX C: ENHANCMENTS IMPLEMENTED IN FIRMWARE 95000405

The following enhancements are implemented in firmware 95000405, as compared with the earlier versions:

1. The internal Register File is accessed in an "invariant" way by setting bit 12 =1 of the first VME command word (the AM/Options word).
2. Mixing of regular and scaler events in a common data buffer is possible, by setting bit 5 =1 in Global Register. In this mode of mixed buffering, scaler events are identified by bit 15 =1 of the event header word (otherwise specifying the number of words in the event).
3. Two new modes of triggering the scaler readout while in data acquisition mode are implemented – interactive scaler readout by writing 1 to bit 4 of the Action Register (not that one must, in fact write "17" to this register, to preserve the acquisition mode), and timed readout. The latter is selected by writing a non-zero readout period in units of ½ seconds into bits 8-15 of the Data Acquisition Settings register. All three

scaler readout triggers operate on “whichever-comes-first” basis, with any trigger resetting the control counters (event and timer counters).

10 APPENDIX D: ENHANCMENTS IMPLEMENTED IN FIRMWARE 16000503

The following enhancements are implemented in firmware 16000503, as compared with the earlier versions:

1. A 32-bit multi-block transfer can be performed by the firmware such that a standard 32-bit block transfer is performed repeatedly, for a predetermined number of times, with the starting address being incremented every time by 0x100 (i.e. 256 bytes). The number of repetitions is limited to a 32-bit number, i.e., is practically unlimited. To use this option, bit 11 (value 0x8000) of the Mode Word (containing the AM code and options bits) must be set and the word itself is to be followed by a 32-bit word specifying the desired number of (auto-incrementing) repetitions. The length of the block (bits 24-31 of the Mode Word) is in this case disregarded as it defaults to 64 (i.e., 256 bytes), to provide for a contiguous readout.
2. VME command stack structure has been enhanced so as to allow one to define up to 8 stacks, with Stack ID=0-7, within the allocated memory of 2kBytes = 1kWords. The starting addresses of stacks can be set arbitrarily. While stacks with ID=2-7 are dedicated to interrupt handling, stacks with ID=0 and 1 are of dual use. Unless the latter stacks are used for interrupt handling, they default to a regular stack with ID=0, executed upon trigger pulse (NIM I1 or USBStart) and the periodic stack with ID=1 (scaler stack). To allow writing to the stacks and reading back their content, the internal device address space has been expanded to include the additional 2 bits required to identify the stack (in addition to bit 0). As in earlier versions, the stack ID is sent to VM-USB in the first word (2 bytes) of the USB out packet, with bit 0 of the ID mapped to bit 0 of the address word and bits 1 and 2 to bits 4 (value 16) and 5 (value 32) of the address word, respectively. The 9 least significant bits of the second word of the out packet (disregarded in earlier versions of the firmware) define the starting address (0 – 1023, with wrap-around) of the particular stack within the 1kWord stack memory space.
3. Interrupt handling is provided, with all 7 IRQs (1 – 7) being monitored, when so desired. Any Stack ID can be linked to any combination of IRQ and the IRQ-ID returned by the requestor module, such that one Stack ID can be shared by many interrupts. Upon detection of a valid interrupt, VM-USB executes the stack linked to this particular interrupt. The interrupt service vectors are stored in four registers, two vectors per register. Each vector identifies in its 8 least significant bits the IRQ-ID (to be received from the requestor upon interrupt acknowledge IACK), in bits 8-10 (starting from bit 0) the IRQ level (1-7), and in bits 12-14 the ID (0-7) of the associated stack. The vector is disregarded whenever the IRQ-ID is set to zero, e.g., allowing one to use Stacks 0 and 1 as regular and periodic (scaler) stacks, respectively. The vectors are stored at addresses (offsets) 40, 44, 48, and 52 in arbitrary order.
4. The structure of the event header (specifying the number of words in the event) has been modified to identify the stack ID associated with this event. The 3-bit stack ID is stored in bits 13-15 (counting from 0) of the event header word. Unless a mixed-event

option is selected for data buffering (recommended for increased band width), the buffers will contain events with identical stack IDs.

5. A pulser option has been added to the DGG input options, such that the DGG is retriggered with the trailing edge of the gate pulse. With this option selected, the DGG will produce a train of pulses of width equal to the set gate width and the period equal to the sum of set delay and gate widths.