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 VM-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.

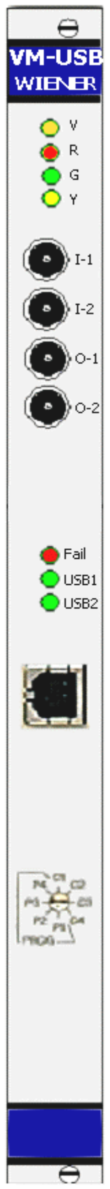
1.1 VM-USB Features

- 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 respond to all 7 interrupt requests IRQ1-7, can generate any of the 7 interrupts
- 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 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 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 Read-out Modes

- Single word transfer D8, D16, D24, D32 with the full selection of bus placement.
- Addressing modes A16, A24, A32
- BLT (with a suitable firmware).
- Autonomous (intelligent) readout pursuant to user-programmed stack. May include conditional readout controlled by the content of a hit register. May include multiple, conditional command stacks, action triggered by either USB, VME or external signal
- Highly effective triply-pipelined stack execution, virtually with no band-width penalty for single (as opposed to block transfer) operations.
- 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

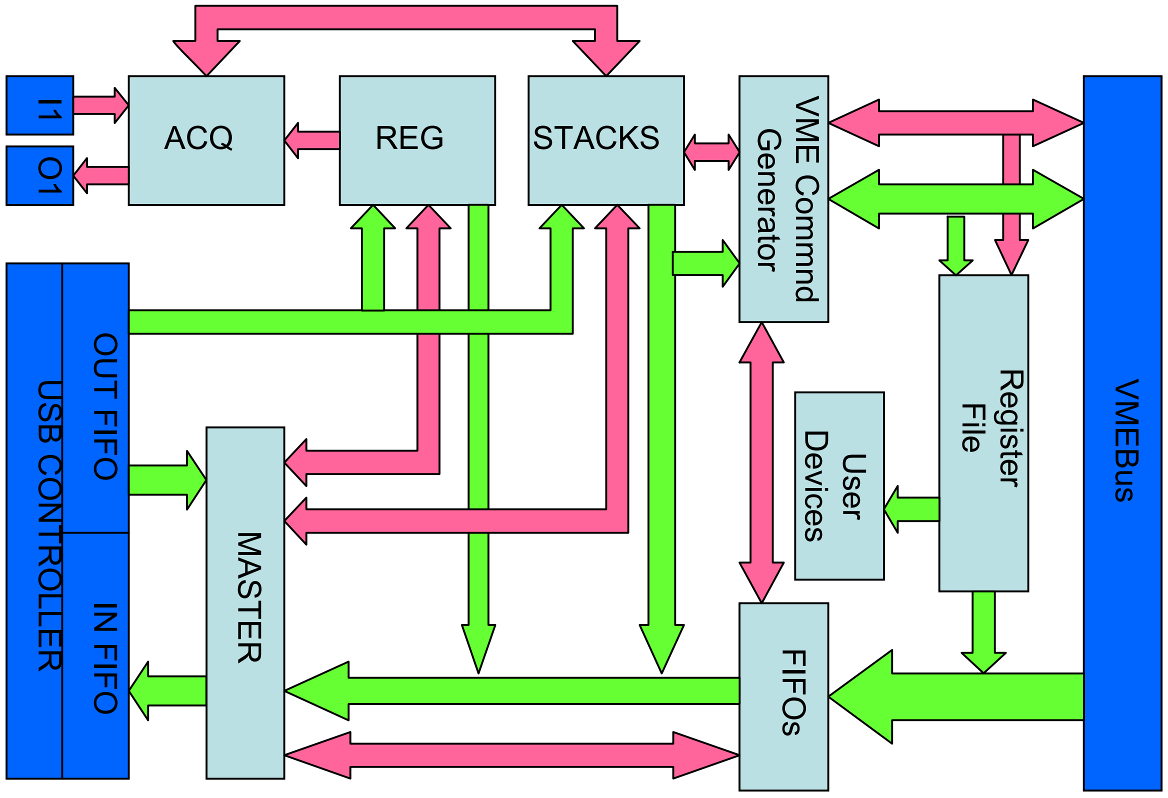
1.4 Technical Data

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 default setting (firmware 7504): O-1: busy O-2: internal event trigger
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
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 Block diagram



🟦	External to FPGA	🟩	Data
⬜	FPGA	🟥	Control

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 Piplined 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)

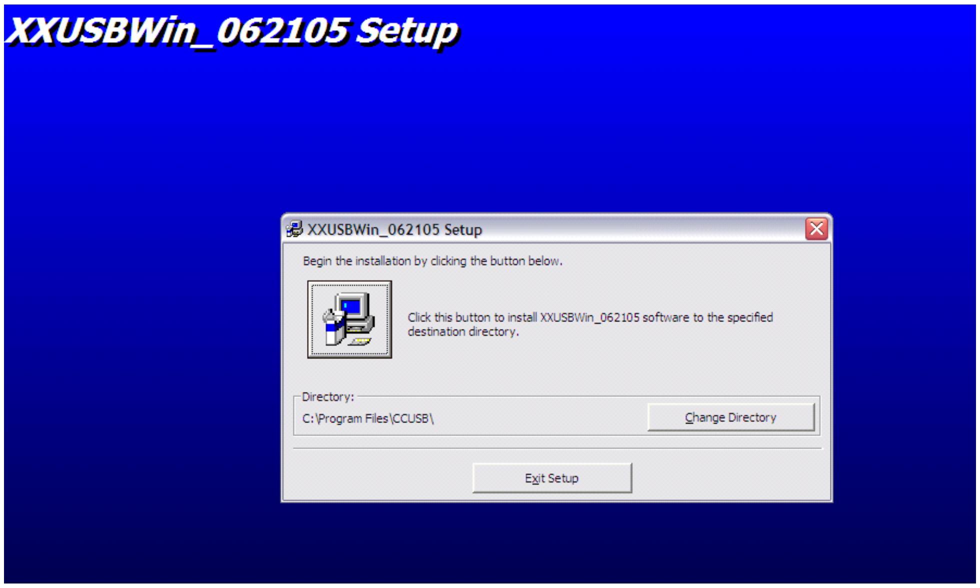
2 VM-USB AND USB DRIVER INSTALLATION

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.


XXUSBWin_062105 Setup

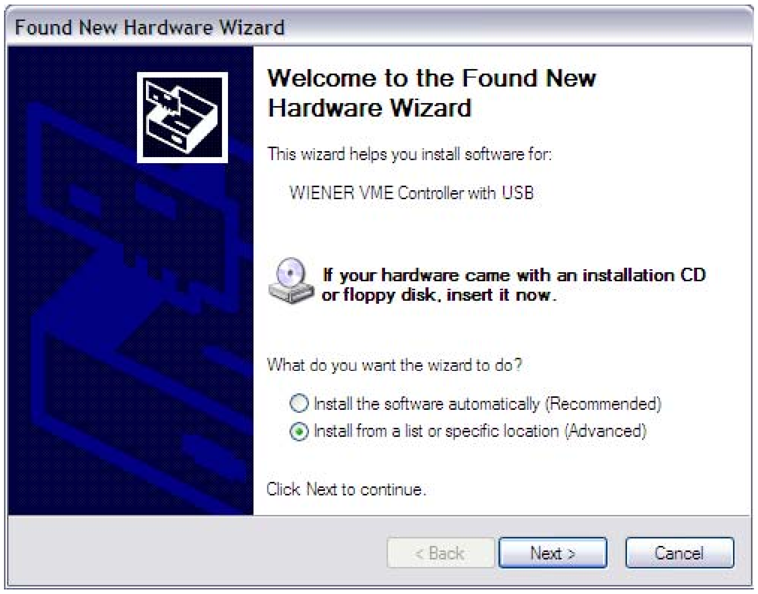
XXUSBWin_062105 Setup

Begin the installation by clicking the button below.

Click this button to install XXUSBWin_062105 software to the specified destination directory.

Directory:
C:\Program Files\CCUSB\

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”.


Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

This wizard helps you install software for:

WIENER VME Controller with USB

If your hardware came with an installation CD or floppy disk, insert it now.

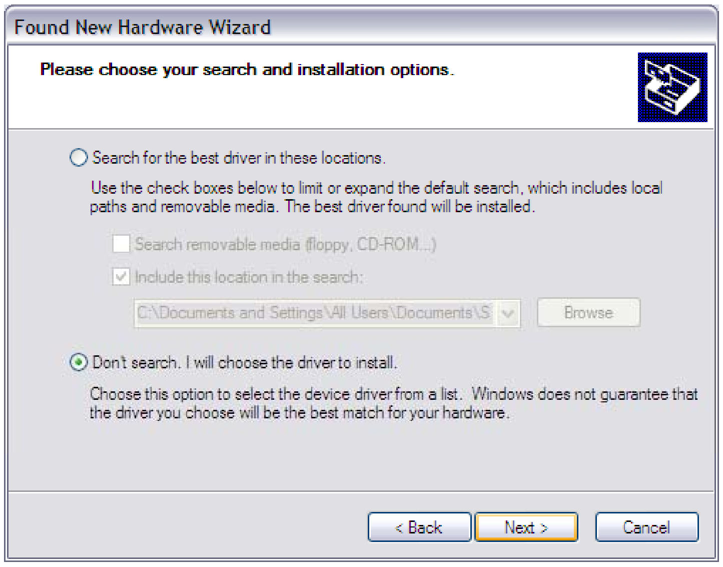
What do you want the wizard to do?

○ Install the software automatically (Recommended)

[✓ Install from a list or specific location (Advanced)]

Click Next to continue.

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.


Found New Hardware Wizard

Please choose your search and installation options.

○ Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

☐ Search removable media (floppy, CD-ROM...)

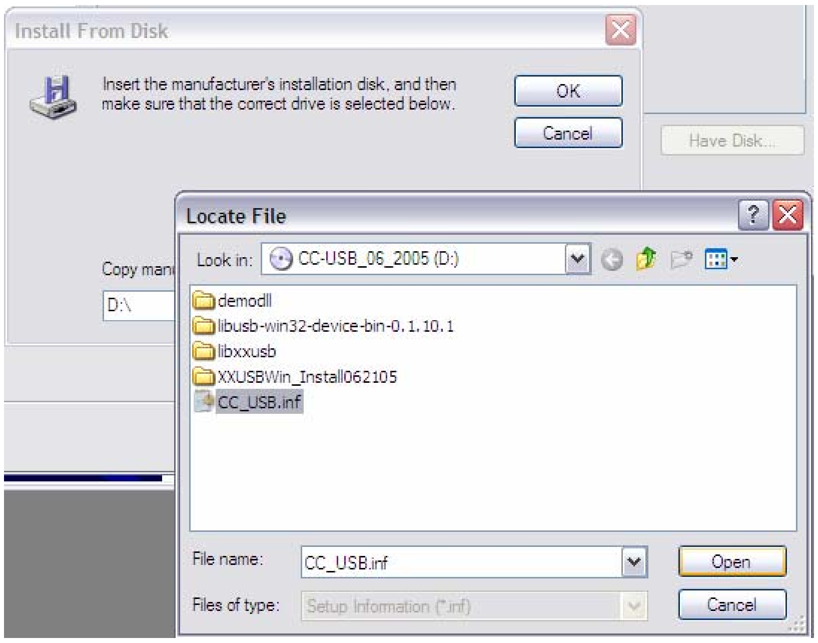
☑ Include this location in the search:

C:\Documents and Settings\All Users\Documents\S

[✓ Don't search. I will choose the driver to install.]

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

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.

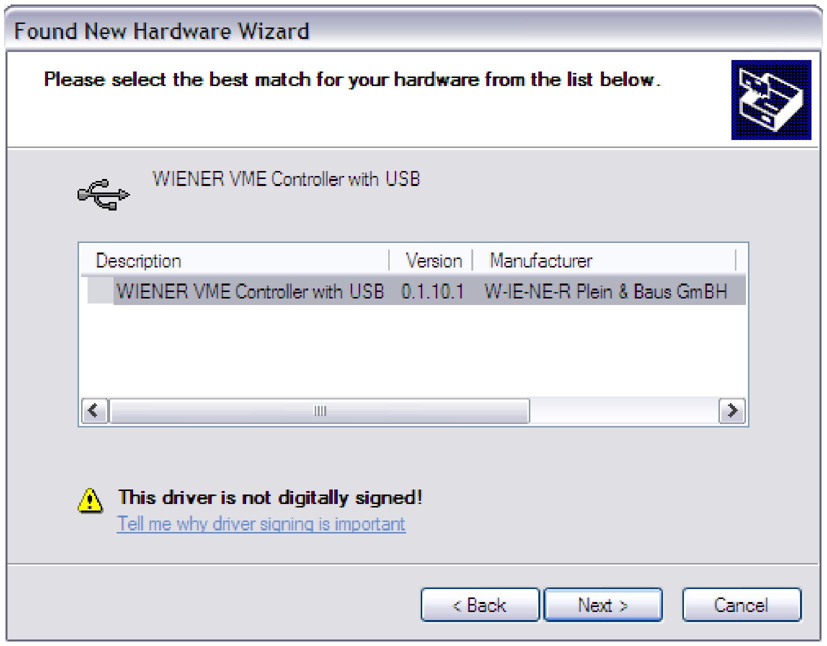

Install From Disk

Insert the manufacturer's installation disk, and then make sure that the correct drive is selected below.

Locate File

Look in: CC-USB_06_2005 (D:)
demodll
libusb-win32-device-bin-0.1.10.1
libxxusb
XXUSBWin_Install062105
CC_USB.inf
File name: CC_USB.inf
Files of type: Setup Information (*.inf)
Open
Cancel

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

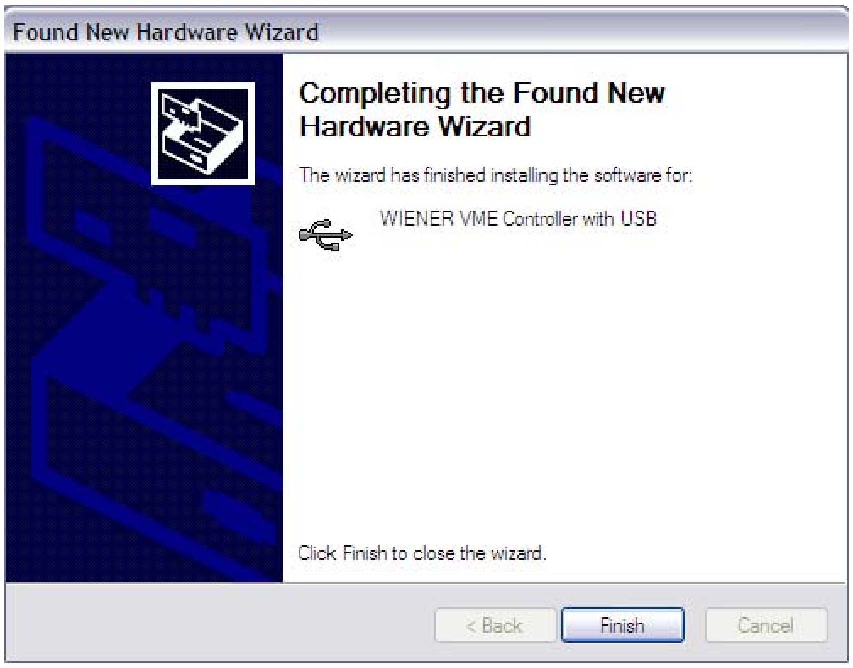

Found New Hardware Wizard

Please select the best match for your hardware from the list below.

WIENER VME Controller with USB

Description	Version	Manufacturer
WIENER VME Controller with USB	0.1.10.1	W-IE-NE-R Plein & Baus GmBH

This driver is not digitally signed!
Tell me why driver signing is important


Found New Hardware Wizard

Completing the Found New Hardware Wizard

The wizard has finished installing the software for:

WIENER VME Controller with USB

Click Finish to close the wizard.

9. Run the XXUSBWin.exe program from the program directory or use one of the sample programming packages to communicate with the VM-USB.

2.2 Installation for Linux Operating Systems

Linux provides a library libusb that allows to perform the bulk transfer to and from the VM-USB USB port. The documentation and the library is available from http://libusb.sourceforge.net. Fortunately, it is also included in modern Linux distributions, as is the USB2 EHCI driver.

The prerequisites are:
(i) EHCI driver loaded - this is part of newer Linux distributions
(ii) libusb installed - installed automatically with newer Linux distributions.

IMPORTANT NOTE: At the time of this writing, root privileges are needed to use libusb, when hot-plugging the device. This is needed for being able to write to the usb file system usbfs. Please execute su before calling the demo program.

3 GENERAL ARCHITECTURE OF VM-USB AND ITS USER INTERFACE

The VM-USB presents to the user six types of internal devices identified by primary (PA) and, if applicable, secondary (SA) addresses, shown in Table 1:

Table 1. Internal devices of VM-USB and their PA and SA addresses

PA	SA	Device
1	10	Action Register (AR)
2	-	Main VME Command Stack (MCS)
3	-	Alternate VME Command Stack (ACS)
4	-	VME Command Generator (CG)
4	VME	Internal Register File
-	-	Common Output Buffer

3.1 Action Register

The action register is a special-purpose 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.

Bit 0 of the Action Register is a read/write bit controlling the operating mode of VM-USB and distinguishing between the interactive and autonomous data acquisition modes. In the interactive mode, VME commands 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 a list stored in a dedicated VM-USB memory block, upon the receipt of a trigger signal at the user NIM input I1. 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 writ-only bit, such that writing "1" to it, clears a number of internal registers. It is intended primarily for use during firmware debugging.

3.2 VME Command Stacks

VME command stacks are used to store suitably coded lists or sequences of VME Commands to be performed in response to event trigger signals while VM-USB is set to operate in (autonomous) data acquisition mode of operation. 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.

VM-USB has two dedicated blocks of 16-bit wide memory to accommodate two stacks of coded VME commands, one for a regular (in response to every event trigger) event readout / processing and one for an optional periodic execution at a desired frequency (after a predetermined number of event triggers). The regular or main stack is 768 words deep and the alternate stack is 256 words deep.

While any arbitrary block of data can be stored in the stack memory and then read back, both 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.

3.3 VME Command Generator / EASY-VME

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.

3.4 Internal Register File

The internal register file of VM-USB shares the primary address (8) with the VME Command Generator. The secondary addresses (A32 VME addresses) identify various internal registers of VM-USB. The secondary address identifies in bits 27-31 (A32 mode) the VM-USB self as a target and must coincide with the settings of the five ID jumpers.

Table 2. Register VME address offsets and their functionality

Offset	Register	Note
0	**Firmware ID**	**32 bits, Read-only**
4	**Global Mode**	**16 bits, Read/Write**
8	**Data Acquisition Settings**	**32 bits, Read/Write**
12	**User LED Source Selector**	**32 bits, Read/Write**
16	**User Devices Source Selector**	**32 bits, Read/Write**
20	**DGG_A Delay/Gate Settings**	**32 bits, Read/Write**
24	**DGG_B Delay/Gate**	**32 bits, Read/Write**
28	**Scaler_A**	**32 bits, Read/Clear/Enable**
32	**Scaler_B**	**32 bits, Read/Clear/Enable**
36	**Number Extract Mask**	**32 bits, Read/Write**

The individual registers are then identified by bits A2-A5 (VME D32 data mode) of the VME address. Note that the access to the internal register file requires A32 addressing and D32 data modes of VME. The addresses (offsets) of various internal registers are shown in Table below and the functionality of the registers is discussed further below.

3.4.1 Firmware ID Register

This Firmware ID register identifies the acting FPGA firmware in eight hexadecimal digits MY000F0R, where M and Y represent the month and year of creation, and F and R represent the firmware main and revision numbers, respectively.

3.4.2 Global Mode Register

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

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	-	BusReq			-			HeaderOpt	-	EvtSepOpt	-	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
1	8k
2	4k
3	2k
4	1k = 1024
5	512
6	256
7	128
8	64
9	Single Event

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 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

The Data Acquisition Settings register stores the desired readout trigger delay and the scaler readout 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, 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.

3.4.4 User LED Source Selectors

Number stored in this register identifies 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 LEDs, but the last four are shared among all four LEDs. For firmware 85000402, the sources 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

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, DGG_1 and DGG_B, 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. For firmware 85000402, the sources 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 deasserted 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	DGG_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	-	-	-	-
7	-	-	-	-

3.4.6 Delay and Gate Generator Registers DGG_A and DGG_B

The two Delay and Gate Generator Registers DGG_A and DGG_B store data defining the length of the delay and the length of the gate in units of 12.5 ns (80 MHz clock). The value of the delay is coded in bits 0-16 and the value of the gate, in bits 16-31 of the register word.

16-31	0-15
Gate	Delay

3.4.7 Scaler Registers SLR_A and SCLR_B

Scaler registers store 32-bit scaler data in a straightforward manner. The use of the scalers is described further below.

3.4.8 Number Extract Mask Register

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.

4 COMMUNICATING WITH VM-USB

Communication with the VM-USB consists in writing and reading of buffers of data 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 represents the data to be stored/interpreted to/by the target device. The latter information is coded in bit 3 (value=4) of the header word, with bit 3 set for requests 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 = 1	the target address identifying the register block
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 + 4 = 5	the target address of the Register Block + the data request bit (bit 3)
2. Register secondary address=10	of the Action Register.

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

The Out Packets targeting the two VME Command Stacks, main (MCS) and auxiliary (ACS) and the VME Command Generator (VCG) have identical structure, differing only in the Target Address and in the allowed length. 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.

1. Target Address	2, 3, or 8, for MCS, ACS, and VCG
2. Number of subsequent words in the stack	
3. Zero word	
4-N*2. Sequence of stack words	where N=Number of 32-bit words in stack + 2

The VME Main Stack (Address=2) is 384 (32-bit) words deep and is intended for storing information on the sequence of the VME commands to be performed when an event readout is initiated, while VM-USB is operating in data acquision (as opposed to interactive) mode. The VME Auxiliary Stack (Address=3) is 128 (32-bit) words deep and is intended for storing information on the sequence of the periodic readout commands, when a periodic readout of select devices (e.g., scalers) is desired. The VME Command Generator (Address 8) 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).

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 and the second one always specifying the VME address modifier with options (Mode word)) and the VME address (Address Word), respectively.

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 options:

31-24	23-22	21-20	19	18	17	16	15-9	8	7-6	5-0
BLT	NT	*	HM	ND	HD	BE	*	NW	DS	AM

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.
*	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 block 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 the Hit Mode (HM) bit is set, the Number of Terms bits must be declared. The second stack line must be followed by the specified number of 32-bit data lines representing bit masks BMask(1 to NT), to be used in constructing the logical condition for performing the command. The logical equation is:

(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)),

i.e., the command will be performed whenever all bits in any of the specified Bit Maks are set in the most recent hit register data.

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.

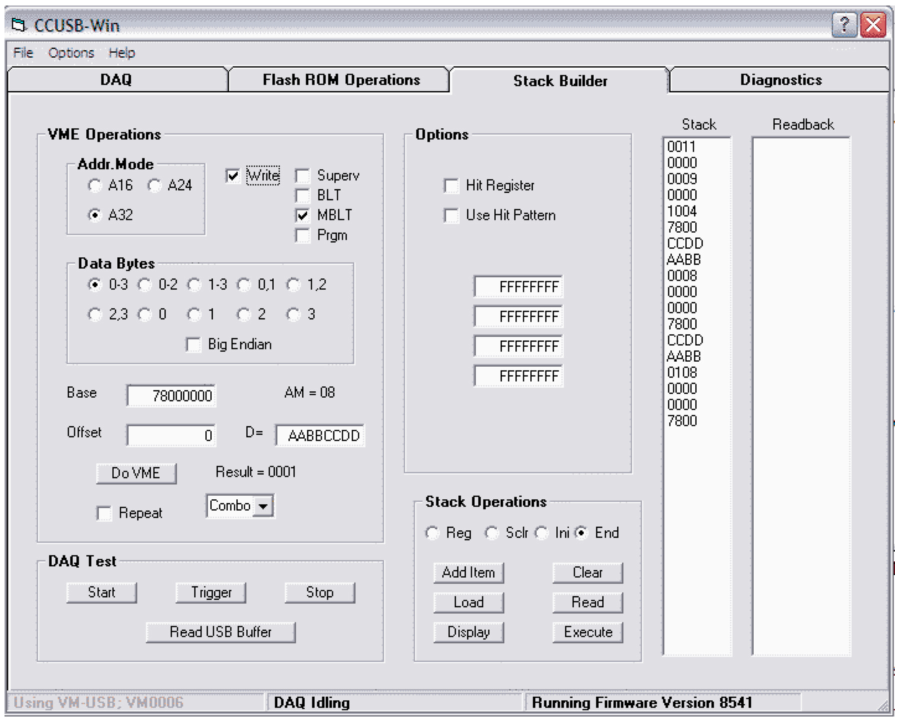
BE	DS1	DS0	A(1)	A(0) LWORD	D31-D24	D23-D16	D15-D8	D7-0
1	0	0	0	0	Byte(0)	Byte(1)	Byte(2)	Byte(3)
1	0	1	0	0	Byte(0)	Byte(1)	Byte(2)	-
1	1	0	0	0	-	Byte(1)	Byte(2)	Byte(3)
1	0	0	1	0	-	Byte(1)	Byte(2)	-
1	0	0	1	1	-	-	Byte(2)	Byte(3)
1	0	0	0	1	-	-	Byte(0)	Byte(1)
1	1	0	1	1	-	-	-	Byte(3)
1	0	1	1	1	-	-	Byte(2)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

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.



Addr.Mode: A16 A24 [✓ A32]

[✓ Write] Superv BLT [✓ MBLT] Prgm

Data Bytes: [✓ 0-3] 0-2 1-3 0,1 1,2 2,3 0 1 2 3 Big Endian

Base 78000000 AM = 08

Offset 0 D= AABBCCDD

Result = 0001 Combo

FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF

Stack Operations: Reg Sclr Ini [✓ End]

Stack:

```
0011
0000
0009
0000
1004
7800
CCDD
AABB
0008
0000
0000
7800
CCDD
AABB
0108
0000
0000
7800
```

Using VM-USB; VM0006 | DAQ Idling | Running Firmware Version 8541

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

11	*// number of following lines*
0000	*// 0 line to match 32-bit structure*
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	Bit 15 set indicates a watchdog buffer, bit 14 set indicates a scaler buffer. Bit 13 indicates an "emergency" switch to a continuous mode, Bits 0 – 9 represent the number of events in the buffer.
2. Optional 2nd Header Word	Bits 0-11 represent the number of words in the buffer.
3. Event Length	Event length including terminator words.
4-N1. Event Data	
N2. Event Terminator	hexadecimal 5555
N3. Optional 2nd Terminator	hexadecimal AAAA

.

. Subsequent Events

.

N5. Buffer Terminator hex FFFF

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. The fact of such a change is indicated by setting of bit 13 in the buffer header word.

VME Write returns a single word, with bit 0 = 1 signaling successful operation (S=1), while bit 0=0 signals bus error condition, BERR (S=0):

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.
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 stack.
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..
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 VME counterpart, VM-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 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/ .

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_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.

6.4 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.

6.5 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.

6.6 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.

6.7 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 0, for the regular stack and 1, for the periodic (scaler) readout stack.

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 VM-USB and 768 32-bit words for VM-USB.
The physical length of the periodic (scaler) stack is 256 16-bit words for VM-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.

6.8 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. Valid types are 0, for the regular stack and 1, for the periodic (scaler) readout stack.

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.

6.9 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 (VM-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.

6.10 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

6.11 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,
	CHAR *pData,
	WORD wDataLen,
	WORD wTimeout
};
```

Parameters

hDevice

[in] Handle to the XX-USB device.

pData

[out] Pointer to a character 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 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

The xxusb_bulk_read is given for the sake of completeness.

6.12 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,
	CHAR *pData,
	WORD wDataLen,
```

WORD wTimeout
};

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

pData
[out] Pointer to a character 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 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
The xxusb_usbfifo_read is given for the sake of completeness.

6.13 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 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 Section ???

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.