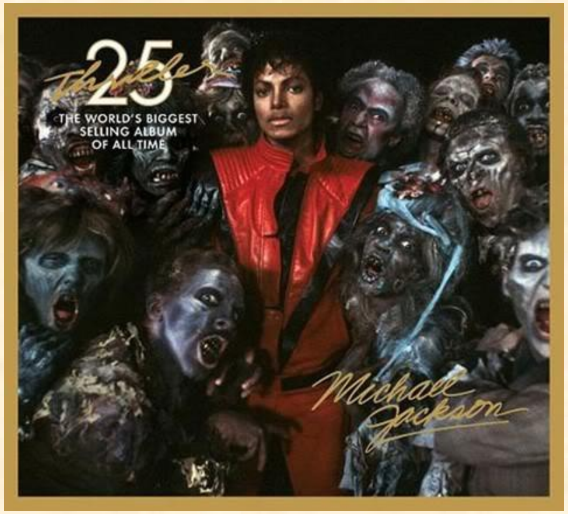
An introduction to VMEbus (in ATLAS)

Markus Joos, CERN, EP-ESE-BE

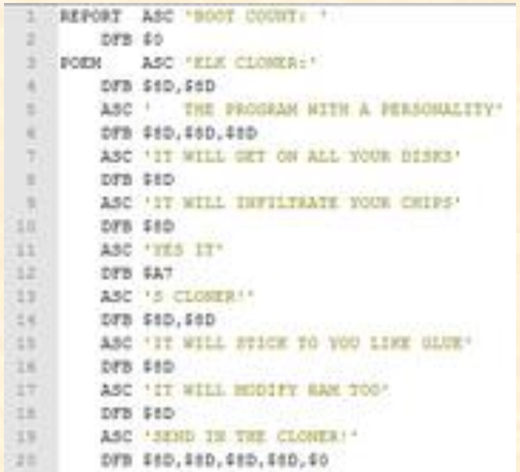
1982

Falklands War

First implantation of an artificial heart



Michael Jackson - Thriller



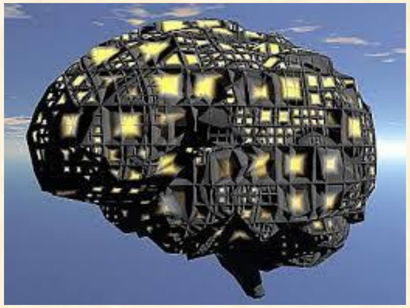
```
REPORT  ASC 'BOOT COUNT: '
    DFB $0
POEM    ASC 'ELK CLONER:'
    DFB $8D,$8D
    ASC '   THE PROGRAM WITH A PERSONALITY'
    DFB $8D,$8D,$8D
    ASC 'IT WILL GET ON ALL YOUR DISKS'
    DFB $8D
    ASC 'IT WILL INFILTRATE YOUR CHIPS'
    DFB $8D
    ASC 'YES IT'
    DFB $A7
    ASC 'S CLONER!'
    DFB $8D,$8D
    ASC 'IT WILL STICK TO YOU LIKE GLUE'
    DFB $8D
    ASC 'IT WILL MODIFY RAM TOO'
    DFB $8D
    ASC 'SEND IN THE CLONER!'
    DFB $8D,$8D,$8D,$8D,$0
```

First computer virus

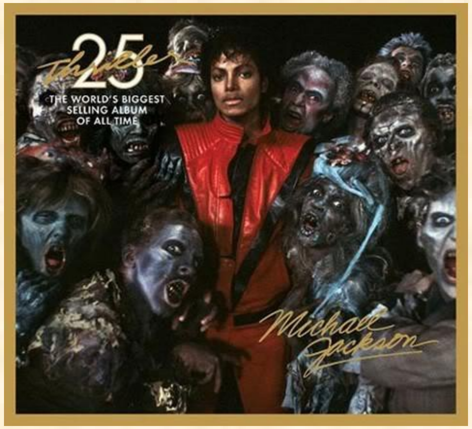
IEEE 1014 - VMEbus

2025

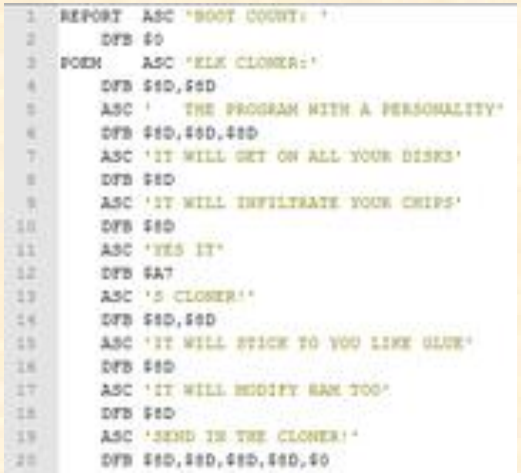
Humanity realizes: War sucks



First implantation of an artificial brain



Michael Jackson successfully cloned

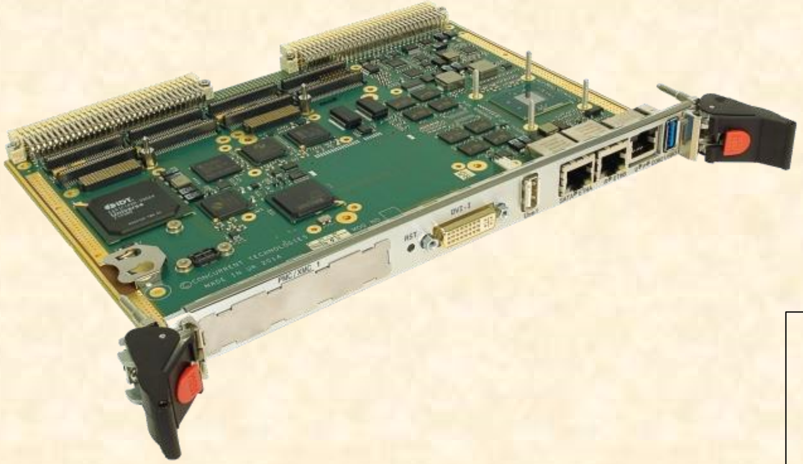


```
REPORT  ASC 'BOOT COUNT: '
        DFB $0
POEM    ASC 'ELK CLONER:'
        DFB $8D,$8D
        ASC '   THE PROGRAM WITH A PERSONALITY'
        DFB $8D,$8D,$8D
        ASC 'IT WILL GET ON ALL YOUR DISKS'
        DFB $8D
        ASC 'IT WILL INFILTRATE YOUR CHIPS'
        DFB $8D
        ASC 'YES IT'
        DFB $A7
        ASC 'S CLONER!'
        DFB $8D,$8D
        ASC 'IT WILL STICK TO YOU LIKE GLUE'
        DFB $8D
        ASC 'IT WILL MODIFY RAM TOO'
        DFB $8D
        ASC 'SEND IN THE CLONER!'
        DFB $8D,$8D,$8D,$8D,$0
```

First computer virus – on a quantum computer

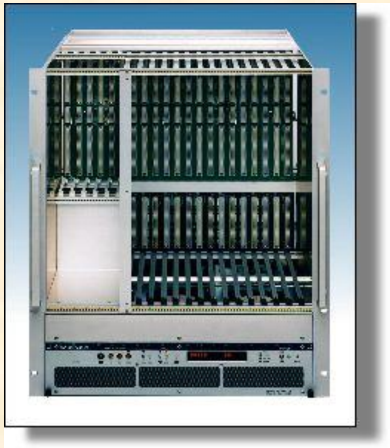
VMEbus still used in ATLAS

The main components



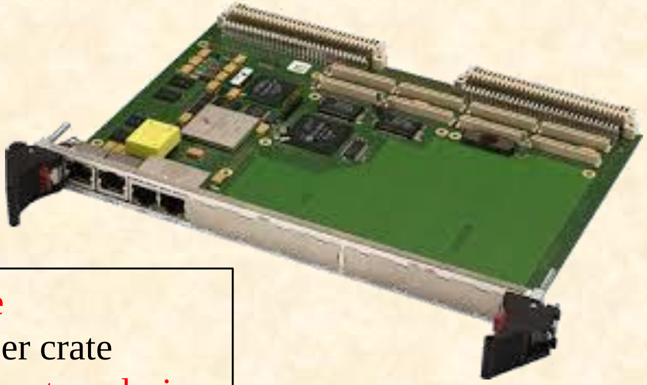
VMEbus master (single board computer, SBC)

- One per crate
- In slot 1 if you want to use interrupts
- Boots Linux from a server maintained by the ATLAS sysadmins



Crates (6U and 9U)

- The fan-tray unit allows to monitor parameters like voltages, currents, fan speeds, temperatures and to remotely power up/down the crate -> DCS
- For support: V. Bobillier, EP/ESE/BE



VMEbus slave

- Up to 20 per crate
- Usually a custom design
- You are the expert!

More about the slaves

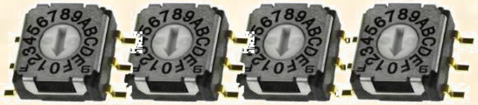
You have to know:

- Base address, AM code and address window size
- Use of interrupts
  - Interrupt level
  - Type of interrupt (ROAK or RORA)
  - Definition of the vector

If you don't know it: Look for documentation or try to find an expert. We cannot tell you.

Selection of the base address of a slave

- Old fashioned: Switches or jumpers on the card



- Modern: VME64x, CR/CSR
  - Access to this window is in A24 mode with AM=0x2f
  - The address of that window is derived from the slot number (geographical addressing) with the formula: address = slot# * 0x80000
  - Not all slaves in ATLAS stick to this rule…

If you are completely lost:

- The scanvme(_tsi) program can help to find slaves but the program has to be run by an expert -> contact us

More about the SBC

Supported types and their “special” features:

Common features

- No local hard disk
- Fragile: handle with care!
- If SBC does not boot:
  - Connect a terminal and check if PXE is enabled in the BIOS
    - VP717/917: You need a special cable and a VGA monitor
    - VP-E24: You need a DVI cable and monitor
    - Dual PMC VP-E24: You need a RJ45 cable, RJ45/DB9 adaptor and the DB9/USB adaptor as well as S/W (putty, minicom, kermit, etc.)
- If you suspect your SBC to be broken
  - Bring it to me (and let me know what you suspect to be wrong with it)

VP-717 and VP-917

- VMEbus interface is called “Tsi148”
- Use tsiconfig, tsiscope, scanvme_tsi

VP-E24

- VMEbus interface is called “Universe”
- Use vmeconfig, cctscope, scanvme

Multi core issues

A typical user request to me: “My S/W has worked with VP110/315. Now that I use a VPx17/VPE24 I have problems.”

- The new generation SBCs are based un multi core CPUs (have a look at /proc/cpuinfo)
- VMEbus transactions are “atomic”
  - A transfer (single cycle) started by thread A cannot get interrupted by a transfer started by thread B
- BUT:
  - If two threads access that same slave at the same time they may interfere. Example:

```
Thread 1:
main{
   vme_write(board_A, reset);
   vme_write(board_A, enable_ch_1);
}
```

```
Thread 2:
main{
   vme_write(board_A, configure_ch_1);
   vme_read(board_A, monitor_ch_1);
}
```

- There may still be some race conditions in the driver
  - One issue with respect to BERR detection has been fixed this week
- You should add synchronization (mutex, semaphore, etc.) at the application level

- The Linux scheduler knows little about I/O transactions
  - If you have many threads doing DMAs some of them may ~never get a time slice
    - We are in discussions with IT………
    - If you can afford it in terms of performance: Run all VMEbus I/O from the same core
    - Implementing a better DMA sharing in the vme_rcc S/W will take time…

OS and TDAQ drivers

Before an SBC can be used you have to:
1) Contact the sysadmins
They have to know where the SBCs will be used and will take care of preparing the boot server.
You have to provide them the MAC address (it can be found on a sticker on the SBC and starts with 00:40:9E)

2) Make a request for the drivers
Fill in the form at: https://atlasop.cern.ch/tdaq/drivers_req/request.php

Each SBC needs 3 TDAQ drivers:

- vme_rcc / vme_rcc_tsi
  - Controls the access to the PCI<->VME interface chip
  - https://edms.cern.ch/document/325729/4
- cmem_rcc
  - Provides buffers of contiguous memory for DMA operations
  - ATTENTION: On a VP-E24 load it with “gfpbpa_zone=1”
  - https://edms5.cern.ch/document/336290/3
- io_rcc
  - Used by some debug tools
  - https://edms5.cern.ch/document/349680/2

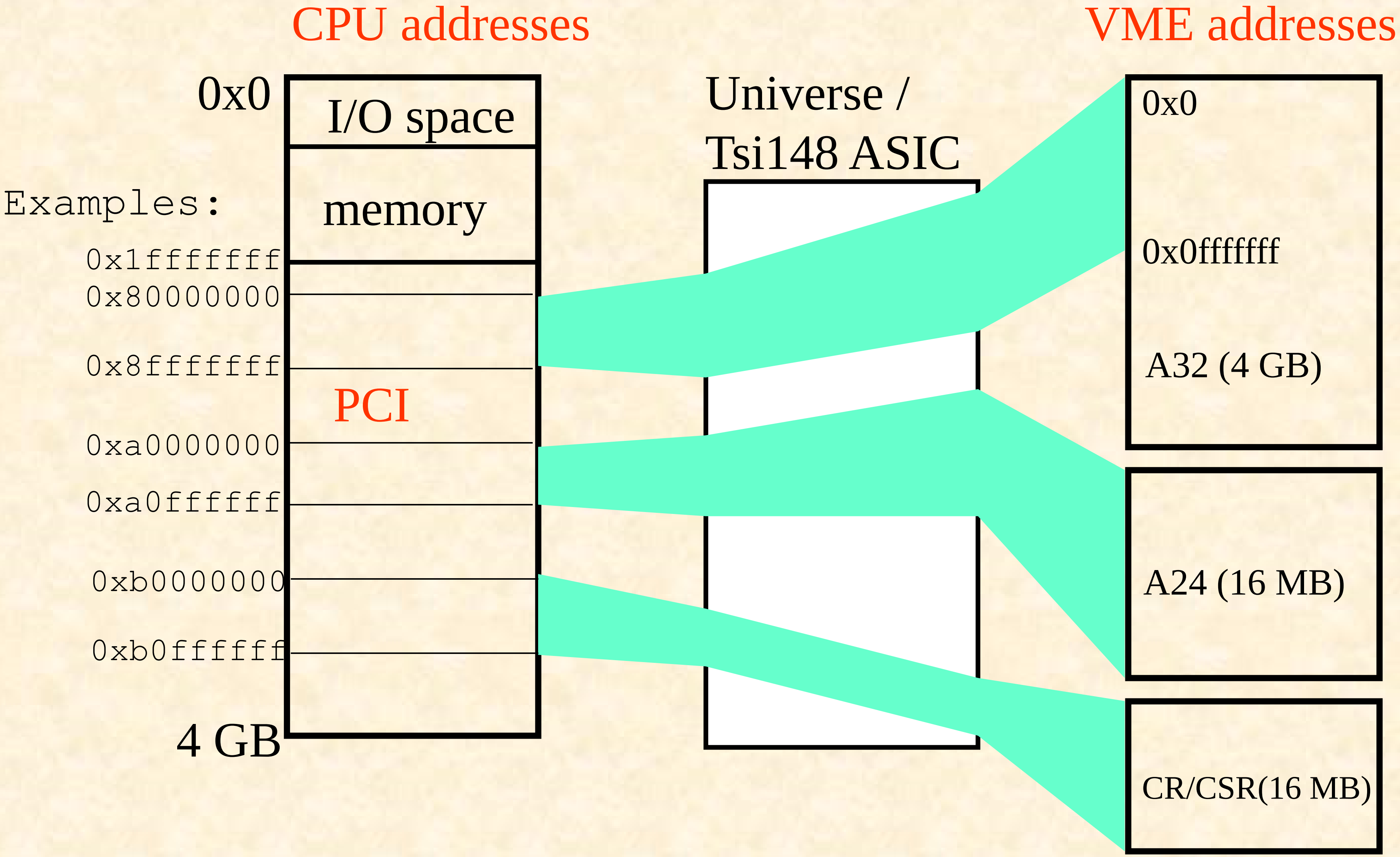
ATTENTION: Always use drivers and libraries from the same TDAQ release
The status of a driver can be checked with: “more /proc/<driver name>”

/proc/vme_rcc(_tsi)

```
sbc-tbed-717:~ % more /proc/vme_rcc_tsi

VMEbus driver for TDAQ release tdaq-06-01-01 (based on SVN tag trunk, vme_rcc_tsi.c revision )
Board type: VP_717

The Tsi148 chip has been initialized with tsiconfig

 VMEbus interrupts

  level|    state| enabled| masked| pending| # interrupts|
  =====|=========|========|=======|========|=============|
      1| disabled|      no|    -- |     -- |          -- |
      2| disabled|      no|    -- |     -- |          -- |
      3|     ROAK|     yes|     no|      no|            0|
      4|     ROAK|     yes|     no|      no|            0|
      5| disabled|      no|    -- |     -- |          -- |
      6| disabled|      no|    -- |     -- |          -- |
      7| disabled|      no|    -- |     -- |          -- |
SYSFAIL| disabled|      no|    -- |     -- |          -- |

# of Bus Error related interrupts: 0
# of DMA related interrupts:       0

(....)

 BERR Descriptor:
 VMEbus address    = 0x00000000
 address modifier = 0x00000000
 IACK             = 0
 LWORD*           = 0
 DS0*             = 0
 DS1*             = 0
 WRITE*           = 0
 multiple bit     = 0x00000000
 bus error flag   = 0x00000000
```

See also: /var/log/tdaq_vme

Address spaces



The famous “vmetab” file

- The “vmetab” file tells the Universe / Tsi148 how to map PCI addresses onto VME addresses and how interrupts are to be configured
  - The file can be generated on the SBC with:
  - Universe: vmeconfig –i <vmetab>
  - Tsi148: tsiconfig –i <vmetab>
  - If called in the form “tsiconfig / vmeconfig –a vmetab” it loads a user set-up into H/W
    - This typically happens automatically at boot time

Ideally the file should be generated by YOU! (If you are nice we help…)

- Documentation is available at: https://twiki.cern.ch/twiki/pub/Atlas/SBCDriverMaintenance/VMEConfig.docx
- This document must be excellent as we have never received and change requests.

Keep in mind:

- As the vmetab file depends on the PCI address lay out of the SBC each type of SBCs (VP917, VP717, single / dual PMC VP-E24) needs a different vmetab
- The vmetab file is a binary. It cannot be modified with a text editor

On the SBC the vmetab file has to be located in:

/daq_area/detectors/<detector name>/vmetab/<vmetab>

This file system is mounted read only on the SBC.

In order to install a vmetab you have to do it from:

- GPN: pc-tbed-grape
- P1: pc-tdq-grape

The path is:

/mnt/daq_area_rw/detectors/<detector name>/vmetab/<vmetab>

The famous “vmetab” file - 2

Once you have loaded the vmetab into the H/W you can check the result with option 2/2 of cctscope / tsiscope:

```
LSI|        VME address range|        PCI address range|   EN| MRPFD|      PFS|       2eSST|      TM|     DBW|   SUP|   PGM|   AM|     OTBS
===|========================|========================|====|======|========|==========|=======|=======|=====|=====|====|========
  0| 0x00000000 - 0x0fffffff| 0x80000000 - 0x8fffffff| Yes|     No| 2 lines|  160 MB/s| single| 32 bit| User| Data| A32| 0x00000
  1| 0x00000000 - 0x00ffffff| 0x90000000 - 0x90ffffff| Yes|     No| 2 lines|  160 MB/s| single| 32 bit| User| Data| A24| 0x00000
  2| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
  3| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
  4| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
  5| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
  6| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
  7| 0x00000000 - 0x0000ffff| 0x00000000 - 0x0000ffff|  No|     No| 2 lines|  160 MB/s| single| 16 bit| User| Data| A16| 0x00000
```

Which PCI addresses can be used:

Have a look at: /proc/iomem

sbc-tbed-717:~ % more /proc/iomem

```
00000000-00000fff : reserved
00001000-0009e7ff : System RAM
…
ffe00000-ffffffff : reserved
100000000-177ffffff : System RAM
```

VP717/917: Use the addresses beyond the last line (`100000000-177ffffff : System RAM`)

VP-E24: Look for:

```
50500000-d05fffff : PCI Bus 0000:07
d0500000-d0500fff : 0000:07:04.0
  or
40300000-c03fffff : PCI Bus 0000:07
c0300000-c0300fff : 0000:07:04.0
```

We have ideas for how to make this semi-automatic….stay tuned

The Universe / Tsi148 chips

- The Universe / Tsi148 chips have a number of limitations:
  - There are only 8 map decoders for master mappings
    - In systems with more than 8 slaves one has to use one decoder for several slaves. I.e. slaves have to be grouped -> slave base address
  - It is not possible to execute block transfers with a constant VMEbus address
    - If you are designing VMEbus slaves: Do not implement single address FIFOs for the read-out of internal memory
  - Data can be lost if a block transfer is terminated with a BERR
  - The Universe is a 32bit PCI device. It can only "see" memory below the 4 GB limit (-> start the cmem_rcc driver with gfpbpa_zone=1)
  - You cannot use more than 2 GB of the PCI address space for mappings to VMEbus on a VP-E24

The Grape tool

- Grape is a Java toot that I have developed years ago for ROS administration
- We also use it in order to configure the drivers for SBCs
- You are welcome to use Grape:

```
alias grape='/usr/lib/jvm/jre-1.7.0-openjdk.x86_64/bin/java -jar /daq_area/tools/ros/Grape/Grape.jar'
```

WARNING: Grape is very powerful. If you make a mistake it can have sever consequences

THEREFORE: Wake up, watch the demo and take notes!

And what about the retired SBCs?

Do not throw away your old VP-110 and VP315 SBCs

- You may still be able to use them in your lab
- Recommendation: Convert them to a local boot (install a HD or USB memory) and configure them with SLC6-32bit
- There is a market for used cards
  - Commercial companies are occasionally looking for VP110/VP315 cards to keep their systems running
  - Let me know if you have cards to give away

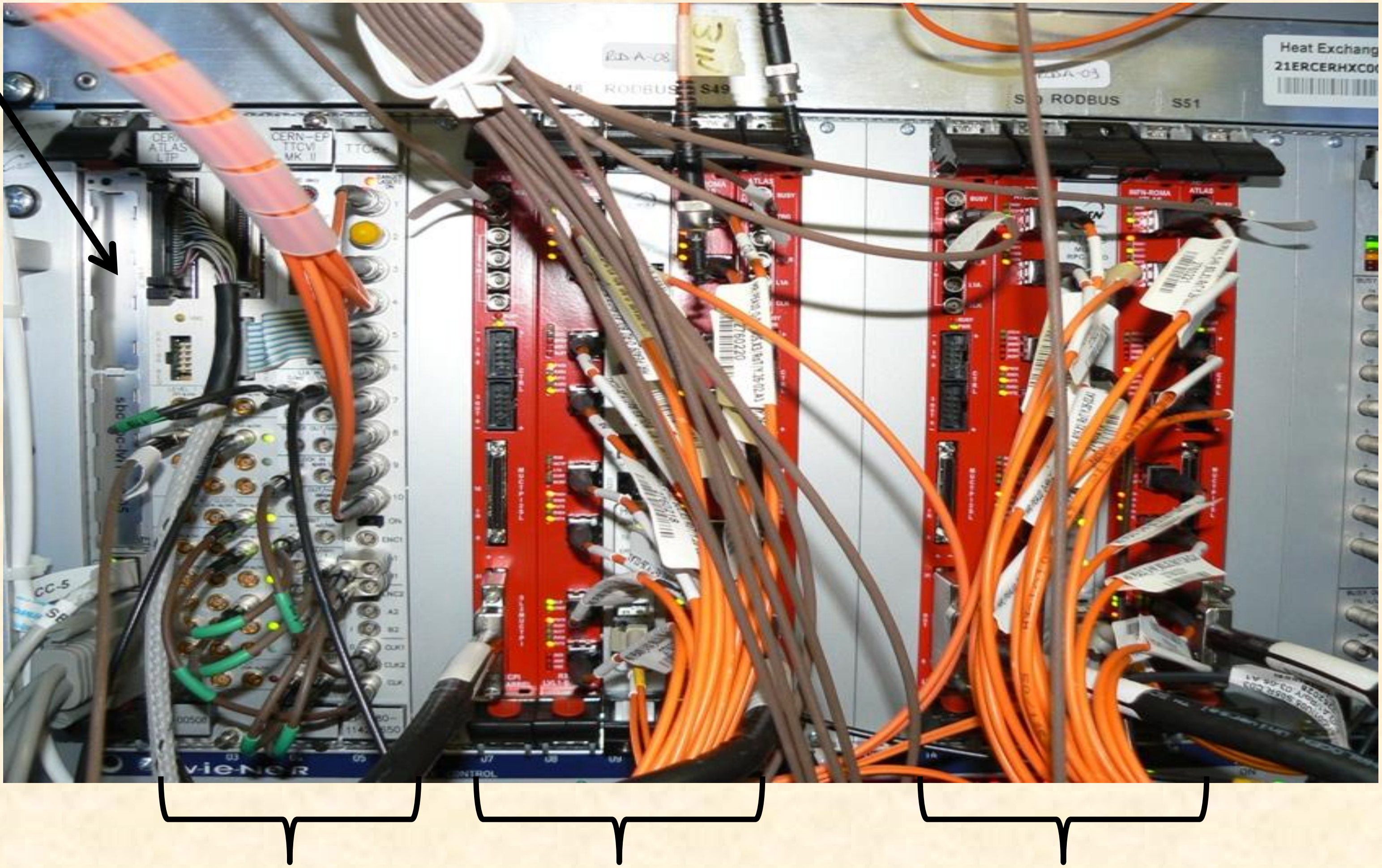
VMEbus in action

The VMEbus crates of the ATLAS RPC detector

VMEbus in action

A single ROD crate

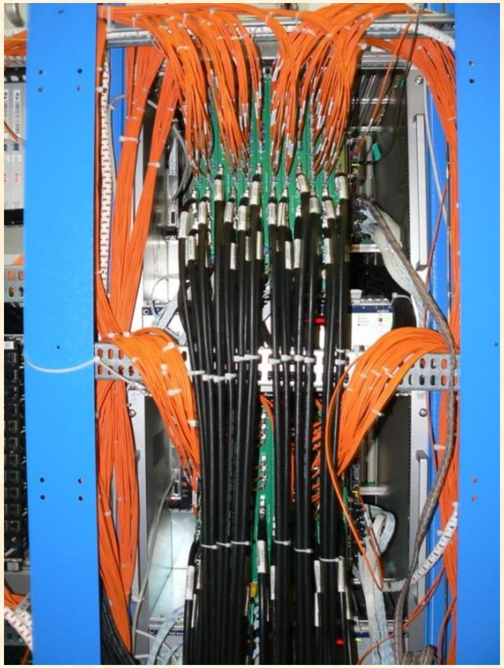
Processor board



TTC interface

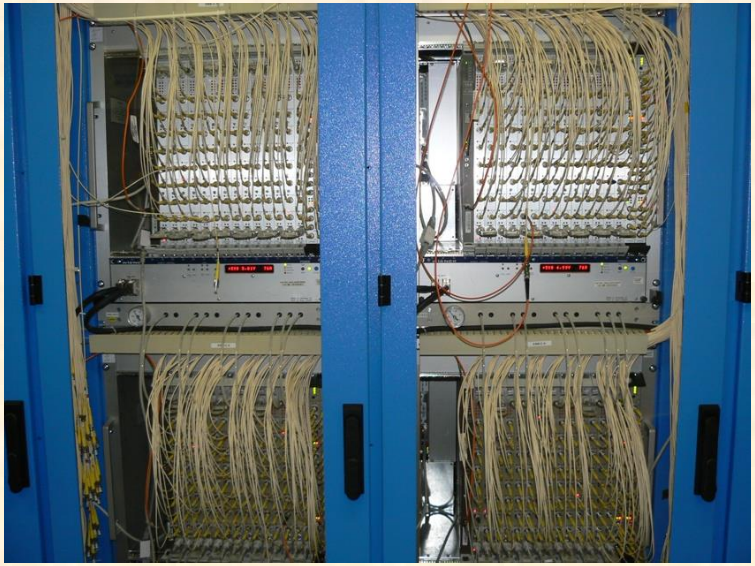
Detector interface

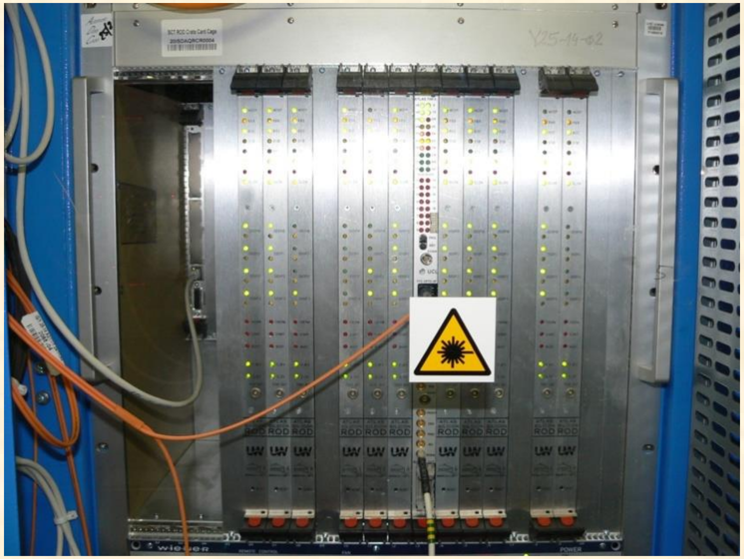
VMEbus in action



More examples of VMEbus systems in ATLAS

In total there are about ~150 crates in ATLAS and they will stay for another 5-10 years





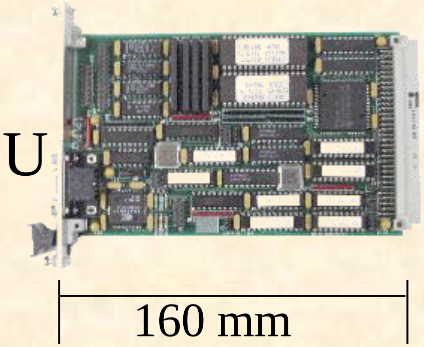
Why VMEbus?

- (Complex) DAQ systems usually require custom built electronics modules which have to be:
  - Housed
  - Powered
  - Configured
  - Read out
- VMEbus has traditionally been the technology of choice in many HEP experiments and accelerator control systems because it offers features such as:
  - A well proven open standard that includes mechanical, electrical and protocol sections
  - Suitable card sizes
  - A data transfer protocol that is relatively easy to implement
  - An "ecosystem" of third party products (crates, processors, I/O modules, etc.) which are supported by the manufacturers for long durations
- Currently there are more than 1000 VMEbus systems at CERN (accelerator & experiments)
  - Learn to use VMEbus (you will become an indispensable expert)

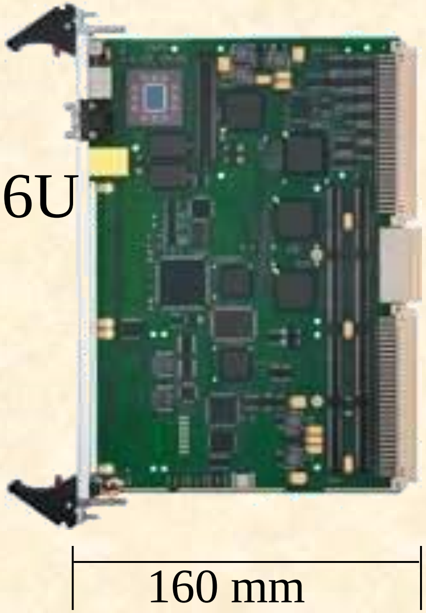
VMEbus mechanics

VMEbus cards exist in 3 standard heights: 3U, 6U and 9U (1U = 1.75 inch) and 2 depths: 160 mm (3U, 6U) and 340 mm (9U)

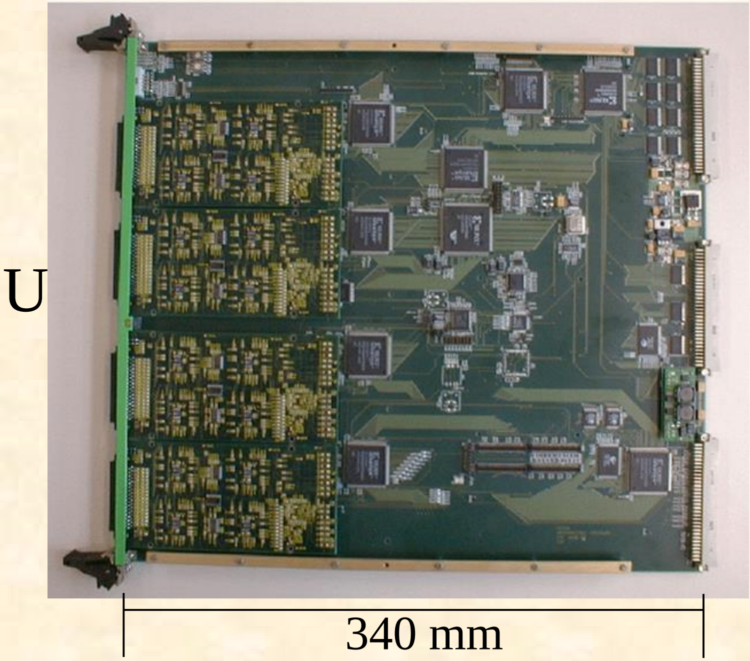
3U



6U

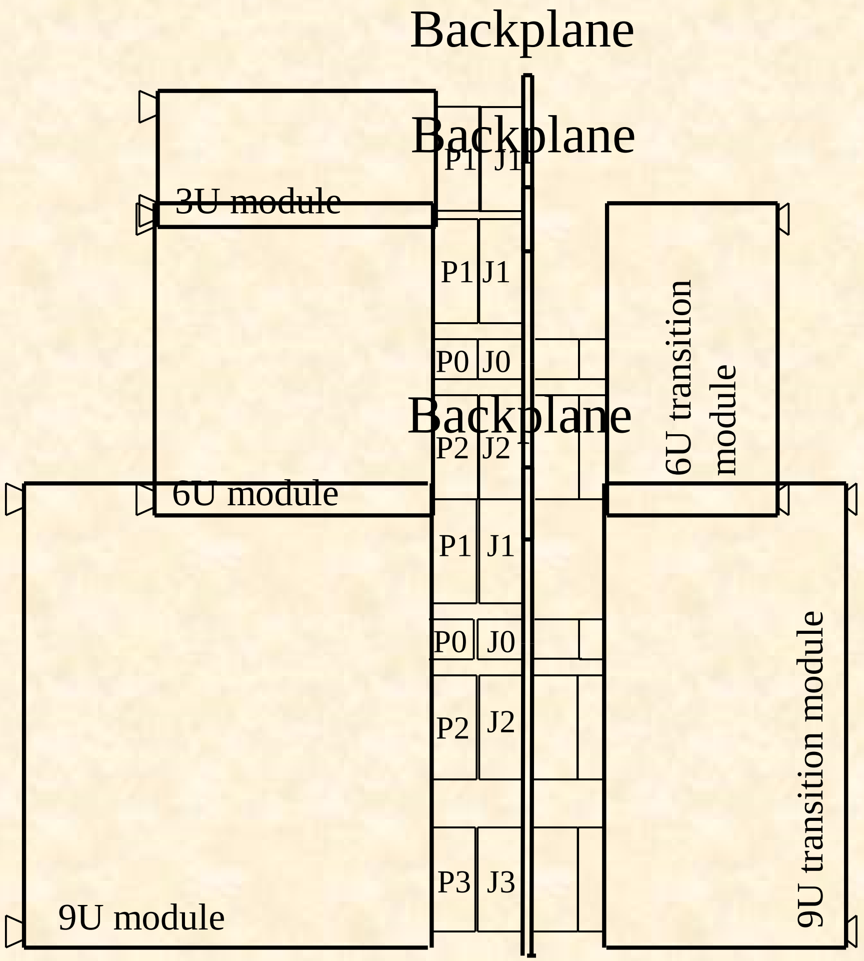


9U



All cards are 0.8 inch (20.3 mm) wide

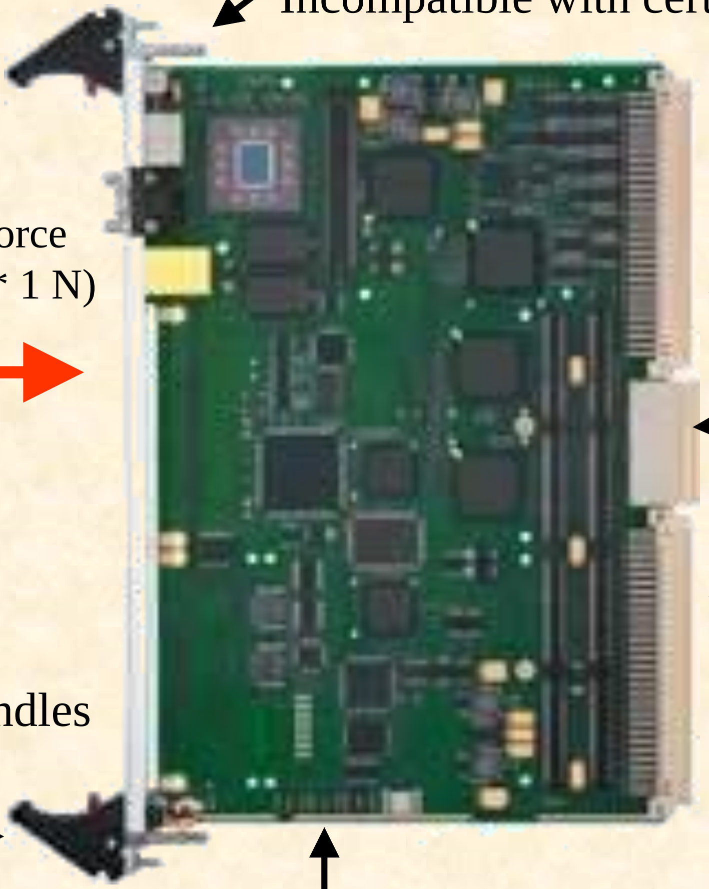
VMEbus mechanics (2)



In 6U and 9U systems there can be transition modules installed on the rear side of the backplane. Transition modules do not connect to VMEbus but just to the VMEbus module on the opposite side of the backplane via the user defined pins of the J0, J2 and J3 connectors

VMEbus mechanics (3)

Example: 6U VME64x module



Alignment pin
Incompatible with certain old crates

40 kg

Insertion force
(415 pins * 1 N)

5 row P1 connector
160 pins used for VMEbus

P0 connector
Used for PMC I/O
Incompatible with certain crates (Jaux, VME64xP)

Injector / extractor handles
Push button

5 row P2 connector
32 pins used for VMEbus
Other pins user defined (e.g. for transition modules)

Discharge strip

VMEbus basics

- Classes of modules (logical)
  - Master
    - A module that can initiate data transfers
  - Slave
    - A module that responds to a master
  - Interrupter
    - A module that can send an interrupt (usually a slave)
  - Interrupt handler
    - A module that can receive (and handle) interrupts (usually a Single Board Computer)
  - Arbiter
    - A piece of electronics (usually included in the SBC) that arbitrates bus access and monitors the status of the bus. It should always be installed in slot 1 of the VMEbus crate if interrupts are used

VMEbus basics (2)

- Electrical properties
  - All lines use TTL levels
  - Low = 0 ... 0.6 V
  - High = 2.4 ... 5 V
  - Address, address modifier and data lines are active high
  - Protocol lines (e.g. AM, LWORD, DS0/1) are active low
- Protocol
  - Asynchronous with 4-edge handshaking
  - The duration of a VMEbus cycle depends on the speed of the master and the slave
- Byte ordering
  - VMEbus is big endian. It stores the most significant byte of a 32-bit word at the lowest byte address (0x0)
  - PCI and Intel CPUs are little endian. They store the most significant byte of a 32-bit word at the highest byte address (0x3)
  - All ATLAS SBCs have automatic byte swapping logic

VMEbus basics (3)

- Main types of data transfers
  - Single cycles
    - Transfer 8, 16 or 32 bits of data under the control of the CPU on the master
    - Mnemonic: D8, D16 and D32
    - Typical duration: 1 µs + S/W overhead
  - Block transfers (DMA = Direct Memory Access)
    - Transfer any amount of data (usually 32 or 64 bit at a time) under the control of a DMA controller (CPU independent)
    - Mnemonic: D32BLT and D64MBLT
    - Data is transferred in bursts of up to 256 (D32) or 2048 (D64) bytes
    - Typical duration: 150 ns per data word
  - Interrupts
    - Used by slaves to signal a condition (e.g. data available, internal error, etc.)
- VMEbus addresses
  - Either 24 or 32 valid bits. Mnemonic: A24 or A32

VMEbus protocol

Why do I have to understand how the protocol works if I am not designing cards?

- Some designers make mistakes and their VMEbus cards do not work at all or fail in combination with certain other cards
- Debugging VMEbus traffic by S/W (printf(), gdb, etc.) is difficult or even impossible
- A great help for fixing such problems are VMEbus analyzers
- A VMEbus analyzer also tells you if you are really executing the desired types of cycles (e.g. D8, D16, D32, etc.)
- In order to understand the output of such an analyzer you have to have some knowledge of the protocol

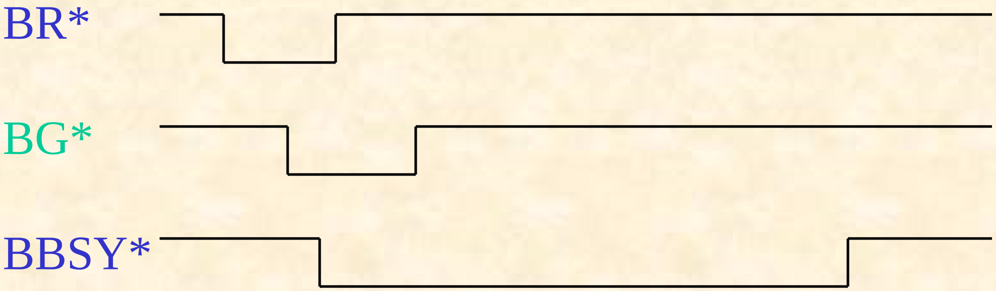
Important signals

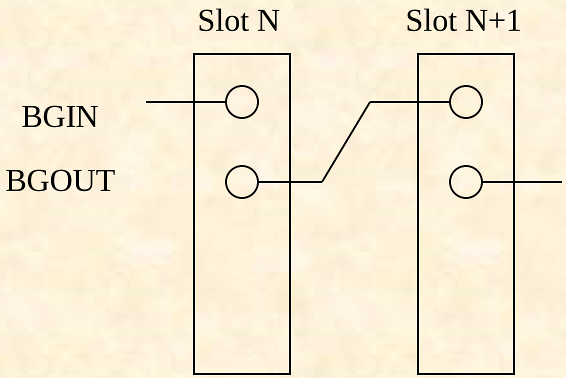
Name	Description
BBSY*	Bus Busy. Once a master has been granted the bus it drives BBSY*. As long as BBSY* is asserted no other master can get the bus
A[31..1]	Address lines (can carry data in D64 multiplexed transfers). A00 does not exist
D[31..0]	Data lines
AM[5..0]	Address modifier. Defines the number of valid address bits and the cycle type
DS0* and DS1*	Data strobes. Tell the slave when the master is ready. Also encode the number of bytes to be transferred
LWORD*	Contributes to the definition of the transfer size and carries data in multiplexed block transfers
AS*	Address Strobe. Tells the slaves when the address on the bus is valid
WRITE*	Defines the direction of the data transfer
DTACK*	Data acknowledge. Used by a slave to tell the master that it has read / written the data
BERR*	Bus error. Used by slaves or arbiters to signal errors
IRQ1* .. IRQ7*	Interrupt request lines. Asserted by the interrupter
IACK*	Interrupt acknowledge. Used by the interrupt handler to retrieve an interrupt vector from the interrupter

Note: “*” denotes active low signals

Arbitration

- Before a master can transfer data it has to request the bus. It does this by asserting one of the four bus request lines
  - These lines (BR0, BR1, BR2 and BR3) can be used to prioritize requests in multi-master systems
- The arbiter (usually in slot 1) knows (by looking at the BBSY line) if the bus is busy or idle. Once it is idle it asserts one of the four Bus Grant out lines (BGOUT 0..3)
- If a master detects a "1" on the BGIN line corresponding to its BR it claims the bus by asserting BBSY (otherwise it passes BGIN on to BGOUT to close the daisy chain)





BG daisy chain

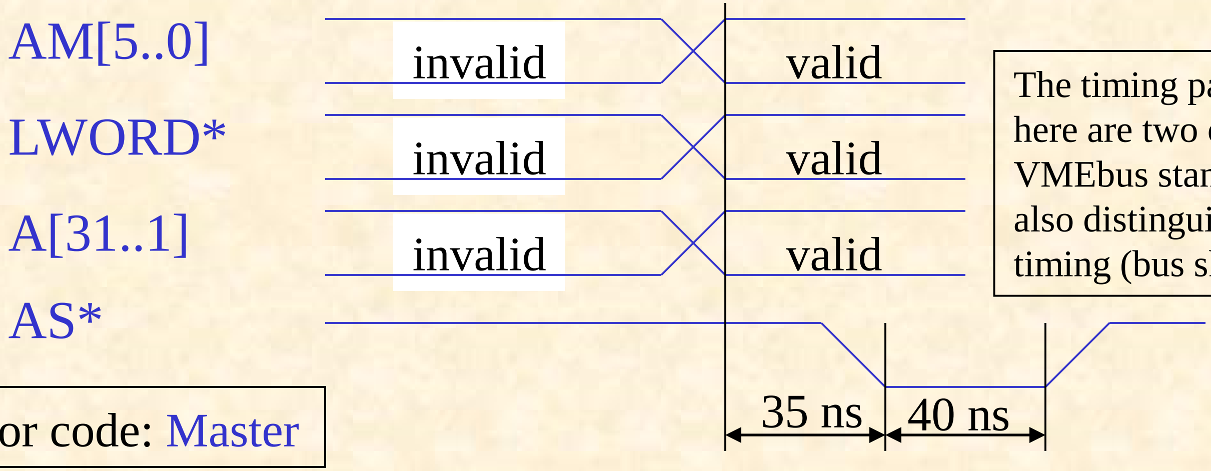
Color code: Arbiter - Master

Addressing

- The VMEbus backplane has 31 address lines: A01..A31
- There is no A00 address line on the backplane. This information is encoded in the DS0/1 protocol lines
- A slave is selected by two criteria:
  - Address (usually 24 or 32 valid bits)
  - Address modifier (6 bits). It defines:
    - The number of valid address bits
    - The access mode (user/supervisor, program/data, CR/CSR)
    - The transfer type (single cycle or block transfer)
- Typically slaves respond to only one address width (A24 or A32; read the manual of the slave) but may allow both single cycles and block transfers
- The base address of a slave can be set:
  - Mechanically: on-board Jumpers, DIP switches
  - By S/W: VME64x geographical addressing, CR/CSR

Addressing protocol

- First the master drives AM, Address and LWORD*. Then it waits 35 ns and finally drives AS* to validate the information
- The slave has to decode the address information within 40 ns (even though most masters keep AS* asserted much longer)
- The master does not know if a slave has accepted the address information. It continues with the data transfer until it either receives a DTACK* or a BERR*
- If two or more slaves believe to be addressed you have a problem…

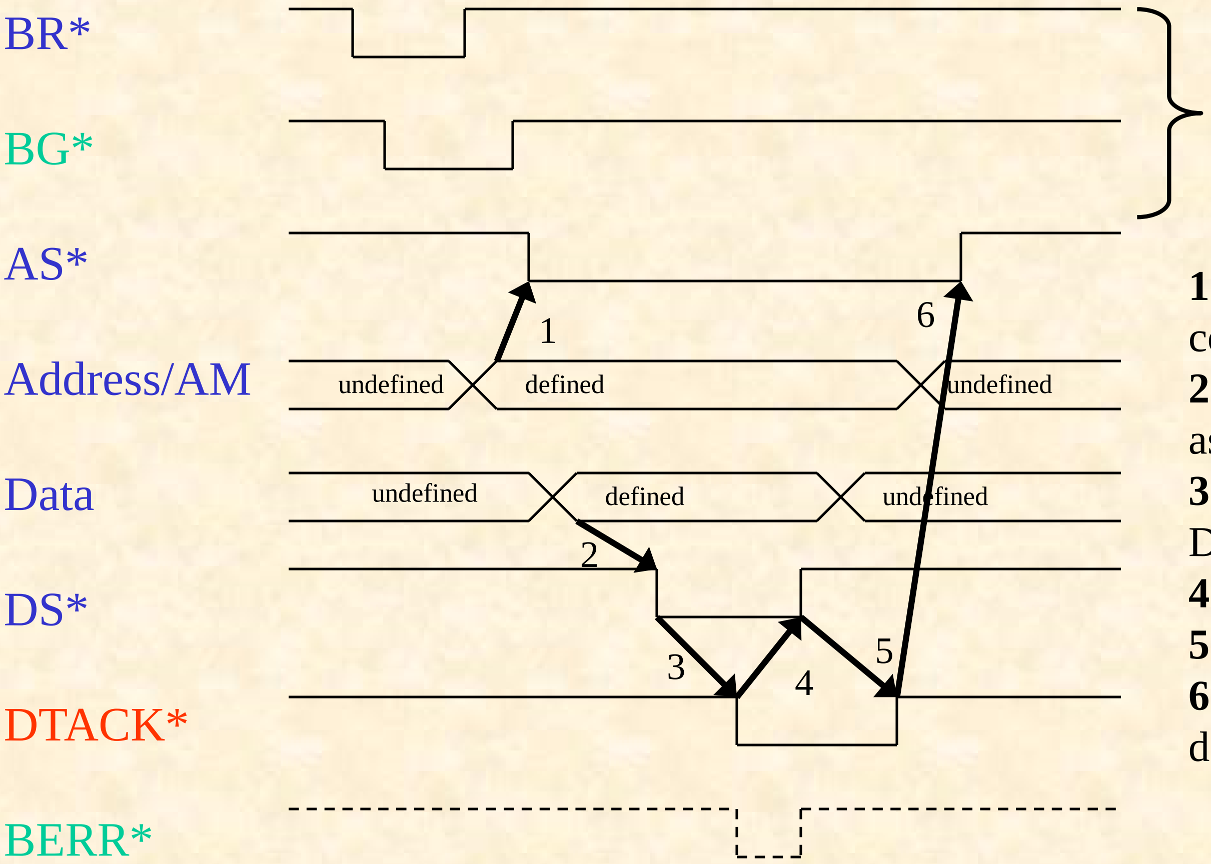


The timing parameters mentioned here are two of about 50 in the VMEbus standard. The standard also distinguishes master and slave timing (bus skew)

Color code: Master

Single cycles

Example: (Simplified) write cycle



Arbitration

1: Master drives address and AM code. Then it asserts AS

2: Master puts data on the bus. Then it asserts DS

3: Slave latches data and drives DTACK

4: Master removes DS

5: Slave removes DTACK

6: Master releases Address, AM and data lines. Then it releases AS

Color code: Master - Slave - Arbiter

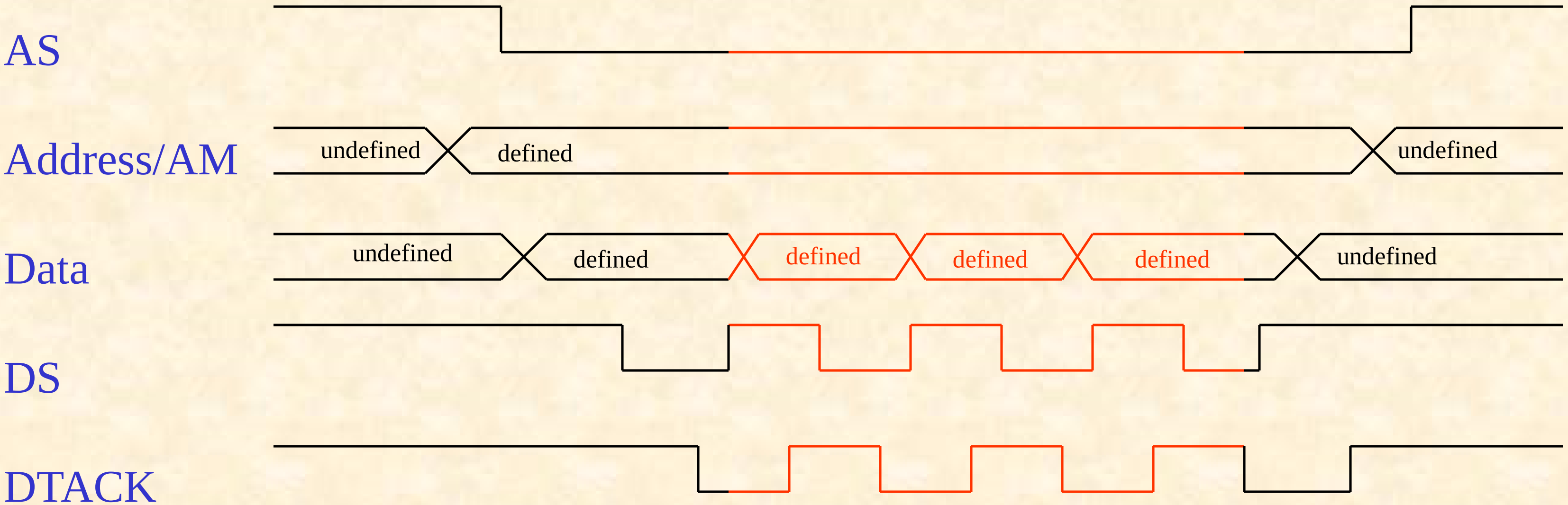
Single cycles (2)

- The number of bytes to be transferred (1, 2 or 4) is encoded in the DS0, DS1 and LWORD protocol lines
- Remember that some slaves support only certain data widths (e.g. D8 and D16 but not D32)
- The VMEbus address should be aligned to the data size
  - Reading a D32 word e.g. from address 0x000003 may not be a good idea
- Remember that VMEbus is big endian. Example:

Address	Action	Result
0x00000000	D32 write 0x11223344	--
0x00000000	D32 read	0x11223344
0x00000000	D8 read	0x11
0x00000003	D8 read	0x44

Block transfers

Example: D32 write



- The Block transfer protocol is based on the single cycle protocol
- The address lines on the backplane do not change state during the transfer. Both master and slave use internal counters to keep track of the address
- As the address lines are not used they can carry data: 64-bit multiplexed DMA In this case the slave uses DTACK for two purposes:
  - Directly after the assertion of AS to acknowledge the address
  - After each assertion of DS to acknowledge the data

Color code: Single cycle protocol – block transfer

Block transfers (2)

- Reading out single address FIFOs is not foreseen by the standard and requires special masters
- VMEbus interface chips may require a relative alignment of the remote (VMEbus) and local (PCI) addresses
  - In case of the Universe the VMEbus and PCI addresses must be 8-byte aligned with respect to each other
- Contiguous buffers
  - Memory obtained with `malloc()` may be fragmented. Most DMA controllers, however, need contiguous buffers
  - Contiguous buffers can be provided by special drivers (e.g. ATLAS: cmem_rcc) based on kernel functions (e.g. get_free_pages) or new features of the Linux kernel (CMA)

VMEbus typical performance

- Being a handshaked, asynchronous protocol there is no fixed transfer rate. The timing parameters (see VMEbus standard) however set an upper limit.
- Single cycles: Typical performance = 1 µs per transfer
  - D8 = 1 MB/s
  - D16 = 2 MB/s
  - D32 = 4 MB/s
- Write posting decouples PCI and VMEbus cycle. This increases the performance to ~ 10 MB/s for D32
- Block transfers
  - D32 = 20..25 MB/s (theoretical: 40 MB/s)
  - D64 = 40..50 MB/s (theoretical: 80 MB/s)

Bus errors

- In VMEbus errors can occur under two conditions
  - A slave has been addressed but is incapable of performing the requested transfer. In this case the BERR signal is issued by the slave and reaches the master within a few µs.
  - The master has issued an address that no slave recognizes. Such cycles get terminated by the bus monitor (arbiter) by asserting BERR after a programmable delay (typical values are 16 or 256 µs)
- There is no standard way for the delivery of a BERR from the VMEbus interface to the CPU of the SBC
  - If you use the safe cycles or DMA: Evaluate the error code of the function
    - ecode = VME_ReadSafeUInt();
  - If you are using fast single cycles: Install a signal handler
    - VME_BusErrorRegisterSignal();

Interrupts

- VMEbus provides 7 interrupt levels (= bus lines) to prioritize interrupts
  - You are most likely fine with one level
- The interrupt handler uses (under H/W control) a special type of single cycle (IACK cycle) to obtain an 8-bit vector from the interrupter. This vector (set by jumpers or S/W) must be unique (within the crate) and identifies the source of the interrupt
- There are two types of interrupters:
  - ROAK (preferred)
    - The IACK cycle clears the interrupt
  - RORA
    - The interrupt is cleared by an additional register access (single read or write cycle)
- Typically an interrupt gets handled by the H/W in a few µs (once the VMEbus is free). However there can be additional (possibly large) S/W overheads depending on the operating system used and the state of the CPU

VME64x

- VME64x is a set of extensions to the VMEbus standard made in 1997
- Most features are optional and fall into one of four categories:
  - Mechanics
    - 5-row P1/J1 and P2/J2 connectors
    - J0/P0 connector
    - Alignment pin
    - EMC gaskets
    - Injector / extractor handles
    - Discharge strips
    - Card keys
    - Solder side covers
  - Plug-and-play
    - Geographical addressing (access a module by its slot number)
    - CR/CSR space: Standardised registers for the automatic configuration of a module (base address(es), interrupt vector(s), etc.)
  - Power
    - 3.3 V and 48 V
    - Additional 5 V
  - 2eVME Protocol: A rarely used way of speeding up block transfers (theoretical bandwidth: 160 MB/s)

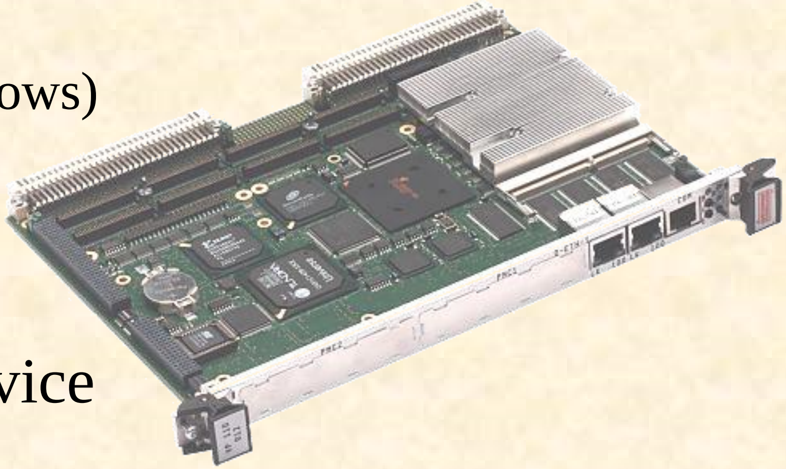
CR/CSR space access and geographical addressing

- “Classic” VMEbus slaves use on-board jumpers or switches for the initialization of the base address and the interrupt vectors
- The VME64(x) standard proposes a S/W based mechanism (plug-and-play) The basic principles are:
  - Each slave has a special window of 512 kB consisting of a Configuration ROM (CR) and a Control and Status Register (CSR) section
  - Access to this window is in A24 mode with AM=0x2f
  - The address of that window is either set by jumpers (VME64) or derived from the slot number (geographical addressing, VME64x) with the formula:

    address = slot# * 0x80000
  - The CR/CSR space contains many (mostly optional) features to specify and control the functions of a slave board
  - Slave boards are identified by a manufacturer + board ID stored in the CR. These IDs have to be unique
  - The most important CSR space registers are the eight ADER registers. They are used to define the base address(es) of the main function(s) of the slave.

The VMEbus single board computer

- Usually this is the only master and interrupt handler in the crate
- It also provides the arbiter functionality (and should therefore be installed in slot 1, despite what will be said about cooling)
- It behaves like a normal PC
  - Operating system: Linux, (RT OS, Windows)
  - Development tools: gcc, g++, gdb
  - Environment: Shell, Xterm, vi, emacs
  - Accessed via: RS232, Ethernet, VGA



- It interfaces to VMEbus via a PCI device
  - Universe / Tsi148
- SBCs can be equipped with mezzanines (PMC) but this is another story

System integration

- Find the right crate for your modules
  - J0 / Jaux incompatibility
  - VME64x (alignment pin, geographical addressing)
- Find out if your crate still has BG/IACK jumpers
  - Rule: Each slot must be equipped with 1 card or 5 jumpers
  - Attention: Jumpers may be on either side of the J1 connector depending on backplane type
- Card handling and insertion
  - VMEbus cards can be sensitive to electrostatic discharge. Take precautions
  - Never add or remove a card if the crate is switched on
  - Depending on the type of module the insertion force is between 20 and 50 kg. Check twice that the card really has been inserted properly!!
  - Do not trust LEDs on the front panel. On certain (VME64x) cards the power pins are longer than the protocol pins.
- Cooling
  - Avoid installing CPUs in the leftmost or rightmost slot (there are special arbiter modules)
  - Leave one or two slots empty between cards, if possible
  - Close the front of the crate with blind panels
  - Check the fan speed
  - Check if your VMEbus cards have temperature sensors
- Address lay-out
  - Check that the address windows of the slave modules do not overlap
  - Try to map similar slaves (e.g. A32, A24) to consecutive address ranges

VMEbus S/W

True “Real time” S/W is rarely required (→ buffer chains)

(Linux) Drivers

- In (almost) all cases access to the VMEbus is via a device driver
- The driver allows to use the VMEbus in multi-processing environments
- Interrupts are handled by the driver and signalled to the user application e.g. by means of signals or semaphores
- Drivers may also provide DMA request lists. A block transfer may therefore not take place immediately but be delayed by other DMA requests
- Accessing the bus via a driver has disadvantages too
  - Additional overhead due to context switching (S/W overhead can be 10 * H/W latency)
  - Drivers are difficult to debug
  - Sometimes commercial drivers lack desirable features and performance
- The “ATLAS approach”: We have developed our own driver and library
  - Reduces dependency on commercial companies
  - Allows for the implementation of performance optimized code
    - Driver can be bypassed for fast H/W access
    - Use of contiguous memory optimizes transfer of large blocks of data
  - May not be justifiable for smaller projects

VMEbus S/W (2)

Libraries

- The driver is not used directly by the application but via a user library
- There is no standard API for such libraries
  - Switching from one type of master to another imposes issues for S/W portability

Performance optimization

- Avoid memcopy(); just pass pointers
- Avoid context switching (bypass drivers)
- Avoid single cycles (this may require special features at the H/W level)
- Use contiguous buffers (for efficient DMA)
- Don’t be too generous with interrupts (less context switching)
- Only implement the features you need (e.g. multi-processing support)
- Understand the latencies in your S/W -> profiling

The library of the vme_rcc package

- There are four major groups of functions:
  - Single cycles
  - Block transfers
  - Interrupts
  - Service functions (including bus errors)
- Presenting all functions here would take too long. The program vme_rcc_test.cpp shows how the different functions are to be used
- There exists a C++ wrapper in a separate package (RCDVme)
- All functions return error codes in the format defined in the rcc_error package

Review the code of your colleagues!

Debugging tools

- H/W (Examples) available at the El. Pool
  - VMEtro VBT325 bus analyzer
    - Stores up to 16000 VMEbus cycles
    - Powerful trigger and sequencer
    - Supports protocol analysis
    - To operate it you need a VT100 (Falco) terminal or a PC with a terminal program (e.g. HypeTerm, minicom, kermit, putty)
  - CES VMDIS8004
    - Low cost bus monitor. Displays the most recent cycle
    - Can latch the first cycle with a bus error or an interrupt
    - Has a built in arbiter (useful if SBC runs hot in slot 1)
- S/W
  - Standard tools for code debugging (gdb, printf(), etc.)
  - Special tools depend on the S/W package

Debugging tools - 2

- Look at the /proc/vme_rcc(_tsi) file of the driver
- In the vme_rcc package you find special applications
  - Scanvme(_tsi): Scan VMEbus for modules
  - vme_rcc_test: Use the functions of the library interactively. This program is also a good programming example
  - cctscope / tsiscope: Decode and dump the configuration of the Universe / Tsi148 chip (and some other VMEbus related resources) in human readable form

cctscope example output of function 2/2:

```
=========================================================================================
LSI  VME address range  PCI address range   EN   WP  VDW  VAS      AM     Type  PCI space
  0  00000000-10000000  90000000-a0000000  Yes   No  D32  A32      UD       SC    PCI MEM
  1  00000000-01000000  a0000000-a1000000  Yes   No  D32  A24      UD       SC    PCI MEM
  2  00000000-00010000  a1000000-a1010000  Yes   No  D32  A16      UD       SC    PCI MEM
  3  00000000-01000000  a2000000-a3000000  Yes   No  D32  CR/CSR   UD       SC    PCI MEM
  4  00000000-ffffffff  00000000-ffffffff   No   No  D32  A32      UD       SC    PCI MEM
  5  00000000-00000000  00000000-00000000   No   No  D32  A32      UD       SC    PCI MEM
  6  00000000-00000000  00000000-00000000   No   No  D32  A32      UD       SC    PCI MEM
  7  00000000-00000000  00000000-00000000   No   No  D32  A32      UD       SC    PCI MEM
=========================================================================================
```

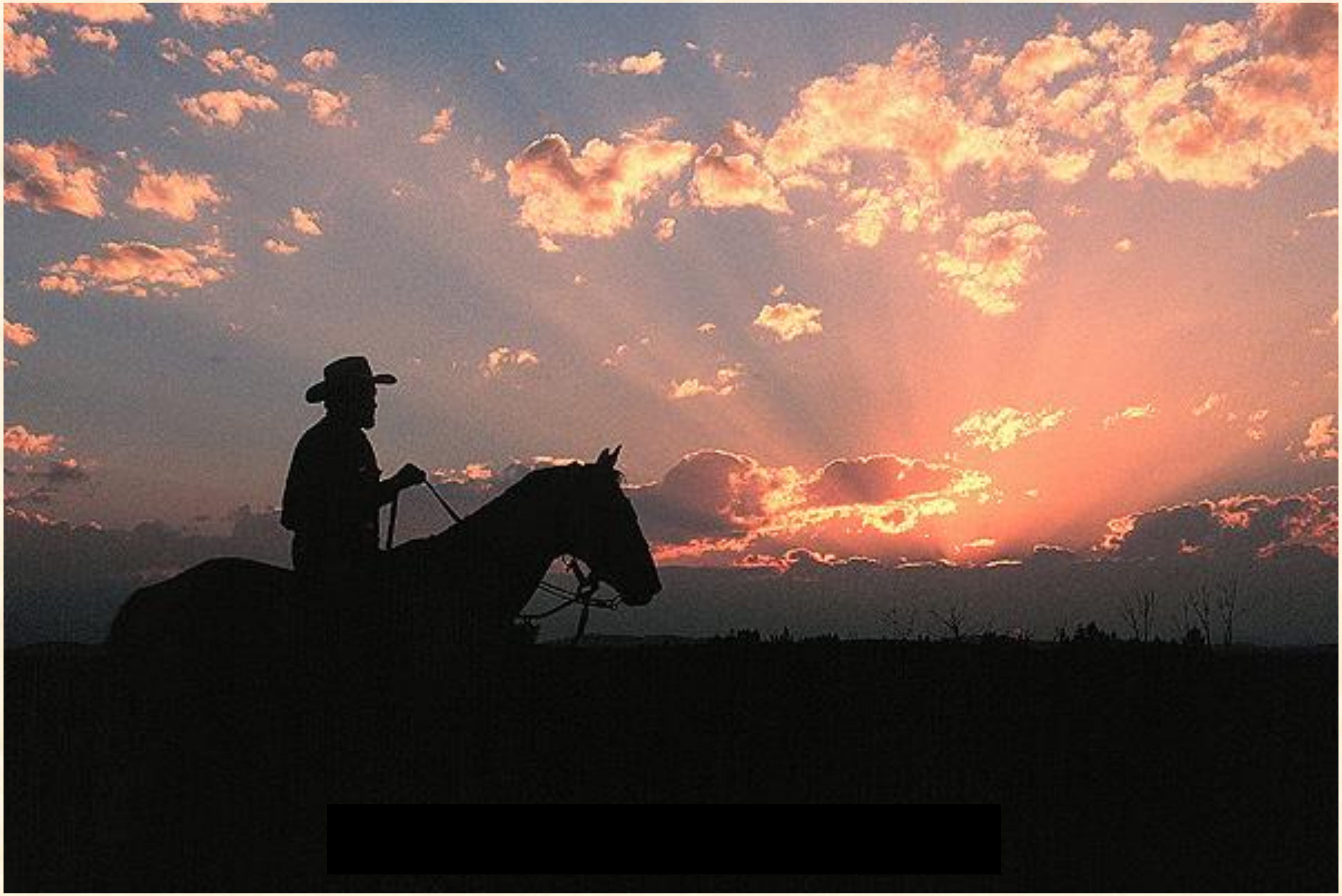
The “odd” stuff

- vme_rcc offers support for
  - CR/CSR space read / write (only D8)
  - User defined AM codes
  - SYSFAIL interrupts
  - Interrupt generation
  - Supervisor / program AM codes
  - Constant address DMA in single cycle mode
  - Full bus error detection
- vme_rcc does not support
  - Read-Modify-Write cycles
  - Address only cycles
  - ACFAIL interrupt
  - A number of other exotic features of the Universe /Tsi148 chip for which nobody has requested support so far

Links for further information

- VMEbus standard
  - www.vita.com (unfortunately the standard is open but not freely available)
- Atlas S/W
  - https://edms.cern.ch/document/325729/4
  - https://edms.cern.ch/document/349680/2
  - https://edms.cern.ch/document/336290/3
  - https://edms.cern.ch/file/325729/4/wrapper.pdf
- Alternative open source VMEbus drivers for the Universe chip
  - http://www.vmelinux.org/ (Webpage last changed on May 24, 2003)
- VMEbus market overview
  - http://www.vita.com

The End



[redacted]

Additional slides

The slides below provide additional information at a more detailed level. Some of them are based on the ATLAS VMEbus S/W

Use of pointers to generate VMEbus cycles

```
unsigned int ui_data, *ui_ptr, virtual_address;
unsigned short us_dat, *us_ptr;
unsigned char uc_data, *uc_ptr;

Main()
{
virtual_address = Map_VME_module(physical_address, AMcode , …);  //Hypothetical function

ui_ptr = (unsigned int *) virtual_address;
us_ptr = (unsigned short *) virtual_address;
uc_ptr = (unsigned char *) virtual_address;

ui_data = *ui_ptr; //D32 read
*ui_ptr = ui_data; //D32 write
us_data = *us_ptr; //D16 read
*us_ptr = us_data; //D16 write
uc_data = *uc_ptr; //D8 read
*uc_ptr = uc_data; //D8 write

ui_data = ui_ptr[0];  // equivalent to *ui_ptr;
ui_data = ui_ptr[4];  // Read D32 at offset 0x10 (4 * 4 bytes)
uc_data = uc_ptr[4];  // Read D8 at offset 0x4 (4 * 1 byte)
}
```

Signal handling

```
#include <signal.h>
//Prototypes
void SigBusHandler(int signum);

main
{
  struct sigaction sa2;

  sigemptyset(&sa2.sa_mask);
  sa2.sa_flags = 0;
  sa2.sa_handler = SigBusHandler;
  stat = sigaction(SIGBUS, &sa2, NULL);
  if (stat < 0)
  {
    printf("Cannot install SIGBUS handler (error=%d)\n", stat);
    exit(-1);
  }
}

void SigBusHandler(int signum)
{
  printf(“Bus error received\n”);
}
```

Managing drivers

- Is my driver loaded?
  - `/sbin/lsmod`
- In what state is my driver?
  - `more /proc/<name>` (value of name depends on the driver used. ATLAS: “vme_rcc(_tsi)”, “cmem_rcc” or “io_rcc”)
- Is the driver currently used
  - `/sbin/lsmod`
  - Check the “Used by” number in the third column.
- How to restart a driver
  - “Used by” has to be 0
  - `sudo atlas_tdaq_drivers unload / load`
  - Or reboot the SBC

(Common) VMEbus AM codes

AM code	Description
0x08	A32, user, 64-bit (MBLT) block transfer
0x09	A32, user, data, single cycle
0x0A	A32, user, program, single cycle
0x0B	A32, user, 32-bit (BLT) block transfer
0x0C	A32, supervisor, 64-bit (MBLT) block transfer
0x0D	A32, supervisor, data, single cycle
0x0E	A32, supervisor, program, single cycle
0x0F	A32, supervisor, 32-bit (BLT) block transfer
0x29	A16, user, data, single cycle
0x2C	A16, supervisor, data, single cycle
0x2F	CR/CSR single cycle (geographical addressing)
0x38	A24, user, 64-bit (MBLT) block transfer
0x39	A24, user, data, single cycle
0x3A	A24, user, program, single cycle
0x3B	A24, user, 32-bit (BLT) block transfer
0x3C	A24, supervisor, 64-bit (MBLT) block transfer
0x3D	A24, supervisor, data, single cycle
0x3E	A24, supervisor, program, single cycle
0x3F	A24, supervisor, 32-bit (BLT) block transfer

Glossary

ACFAIL: A line on the VMEbus backplane driven by the power supply. If asserted the +5V power will be available for at least an other 4 ms and then drop below 4.875 V

BBSY: The protocol line that indicates if the bus is being used (Bus Busy)

BERR: The protocol line that signals a bus error

BG0..3: Protocol lines used by the arbiter to grant the bus to a master

BR0..3: Protocol lines used by masters to request the bus

CR/CSR: Configuration ROM / Control and Status Registers, a feature of VME64(x) slave cards for the plug-and-play configuration of the on-board functions

Daisy chain: Some of the signal lines of a VMEbus backplane are not bussed but connect only two adjacent slots. In order to pass a signal from slot N to slot N+m the VMEbus modules in between these slots have to pass the signal from the input side to the output side. In case of an empty slot the connection has to be made mechanically (jumper) or automatic (special backplane)

DMA: Direct Memory Access (block transfers)

EMC: ElectroMagnetic Compatibility:

IACK: Protocol line used for the interrupt handshake

J0, J1, J2, J3: The female jacks (connectors) on the VMEbus backplane

Jaux: A special connector sitting between J1 and J2 on some backplanes used at CERN. Required for certain front-end modules but incompatible with the P0 connector of VME64x

P0, P1, P2, P3: The male plugs on VMEbus cards connecting to the backplane

ROAK: Release On AKnowledge, A type of VMEbus interrupter that clears the interrupt in response to the IACK cycle

RORA: Release On Register Access, A type of VMEbus interrupter that requires a special intervention from the master to clear an interrupt

SBC: Single Board Computer

SYSFAIL: A VMEbus line that indicates a problem with one card. SYSFAIL can be monitored e.g. by an SBC where it would be converted to an interrupt

Write Posting: A way of speeding up single write cycles. The PCI cycle gets acknowledged before the VMEbus cycle completes. This decoupling of the busses increases the speed but can complicate the detection of bus errors

A simple program doing single cycles

Let’s have a look at a very simple program executing a single read cycle. For this purpose I assume that there is a VMEbus D32/A32 slave at address 0x02000000 with a total size of 4 Kb and a readable register at offset 0x80

```
#include "rcc_error/rcc_error.h"
#include "vme_rcc.h"

int main(void)
{
  VME_MasterMap_t master_map;
  volatile u_int *lptr, ldata;
  u_int ret, vbase;
  int handle;

 ret = VME_Open();
  if (ret != VME_SUCCESS)
  {
    VME_ErrorPrint(ret);
    exit(-1);
  }
```

Declare VMEbus pointers volatile to avoid problems with code optimization

Never call a function without checking for errors!!!

A simple program (2)

```
 master_map.vmebus_address = 0x02000000;
 master_map.window_size = 0x1000;
 master_map.address_modifier = VME_A32;
 master_map.options = 0;
 ret = VME_MasterMap(&master_map, &handle);
 if (ret != VME_SUCCESS)
 {
   VME_ErrorPrint(ret);
   exit(-1);
 }
```

Create a master mapping. Remember: Your “vmetab” must support these parameters

```
 ret = VME_MasterMapVirtualAddress(handle, &vbase);
 if (ret != VME_SUCCESS)
 {
   VME_ErrorPrint(ret);
   exit(-1);
 }
```

Get the virtual address for fast access. Alternatively you could use the safe (but slow) functions of the API

```
lptr = (u_int *)(vbase + 0x80);
```

Cast the generic pointer to a 32-bit data type and add the register offset

```
ldata = *lptr;
```

Execute the VMEbus cycle

A simple program (3)

```
ret = VME_MasterUnmap(handle);
  if (ret != VME_SUCCESS)
  {
    VME_ErrorPrint(ret);
    exit(-1);
  }
 ret = VME_Close();
  if (ret != VME_SUCCESS)
  {
    VME_ErrorPrint(ret);
    exit(-1);
  }
}
```

Always clean up when you're done!

Interrupts

- A VMEbus interrupt (identified by its unique 8-bit vector) can be converted by the library to either a signal or a semaphore
- It is possible to link several interrupts (of the same type) to one signal or semaphore
- It is not possible to service both RORA and ROAK interrupts on the same interrupt level
- If you are using RORA interrupts you have to re-enable the respective interrupt level after each interrupt
- Remember: before you can use an interrupt level you have to enable it with vmeconfig / tsiconfig

```
irq_list.list_of_items[i].vector = 0x77;
irq_list.list_of_items[i].level  = 5;
irq_list.list_of_items[i].type   = VME_INT_ROAK;
signum = 42;

ret = VME_InterruptLink(&irq_list, &int_handle);
ret = VME_InterruptWait(int_handle, timeout, &ir_info);
ret = VME_InterruptRegisterSignal(int_handle, signum);
ret = VME_InterruptUnlink(int_handle);
```

Block transfers

- Block transfers can only be made to physically contiguous memory buffers (-> cmem_rcc). Using memory allocated by malloc() would technically be possible but requires additional code in the driver to lock and chain the pages and there would also be a performance penalty
- Supported modes are: A24D32, A32D32, A32D64 and “single cycle DMA”
- The VMEbus and PCI addresses have to be 8-byte aligned with respect to each other (also for D32)
- The library supports chained DMA
- Block transfers are independent of the master map decoders
- The driver can manage multiple DMA requests from several processes. It is therefore possible that a transfer does not start immediately

```
blist.list_of_items[0].vmebus_address       = 0x10000000;
blist.list_of_items[0].system_iobus_address = 0x24000000;  //PCI MEM space ->CMEM_RCC
blist.list_of_items[0].size_requested       = 0x1000;
blist.list_of_items[0].control_word         = VME_DMA_D32W;  //Implies A32
time_out = 100;
ret = VME_BlockTransfer(&blist, time_out);
if (ret == VME_DMAERR) {
  printf("Status:          %d\n", blist.list_of_items[0].status_word);
  printf("Bytes remaining: %d\n", blist.list_of_items[0].size_remaining);
}
```