

# An introduction to VMEbus






Markus Joos, CERN

## Overview

- What you already should know
- VMEbus
  - Introduction
  - Addressing
  - Single cycles
  - Block transfers
  - Interrupts
  - VME64x
- System assembly
- Single Board Computers
- Software
- Tools



# What you already should know

- C(++) programming
  - use of pointers 
  - signals 
  - data types (char, short, int) 
  - (use of C++ methods) 
- Linux
  - Makefiles
  - gdb
  - Shared libraries
    - Usually the environment variable `LD_LIBRARY_PATH` tells the linker where to find shared libraries
  - Handling drivers (insmod, device nodes, files in /proc) 
  - General operation (cd, ls, mkdir, etc.)

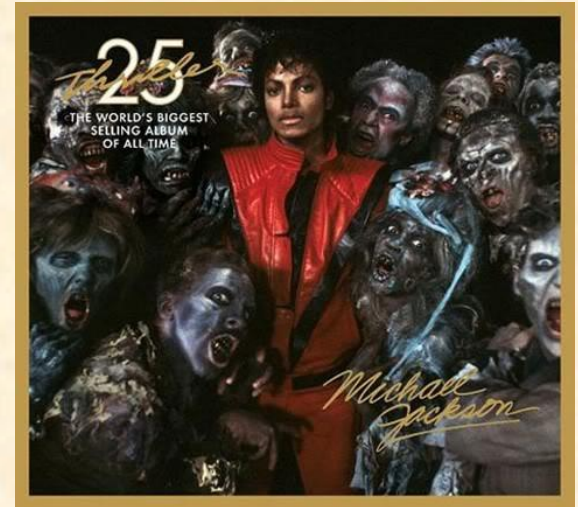
# 1982



Falklands War



First implantation of an artificial heart



Michael Jackson - Thriller



IEEE 1014 - VMEbus

```
1 REPORT ASC "BOOT COUNT: "
2 DFB $0
3 POEM ASC "ELK CLOSER:"
4 DFB $10,$8D
5 ASC ' THE PROGRAM WITH A PERSONALITY'
6 DFB $10,$8D,$10
7 ASC 'IT WILL GET ON ALL YOUR DISKS'
8 DFB $10
9 ASC 'IT WILL INFILTRATE YOUR CHIPS'
10 DFB $10
11 ASC 'YES IT'
12 DFB $A7
13 ASC 'S CLOSER!'
14 DFB $10,$8D
15 ASC 'IT WILL STICK TO YOU LIKE GLUE'
16 DFB $10
17 ASC 'IT WILL MODIFY RAM TOO'
18 DFB $10
19 ASC 'SEND IN THE CLONER!'
20 DFB $10,$10,$10,$10,$0
```

First computer virus



# VMEbus in action



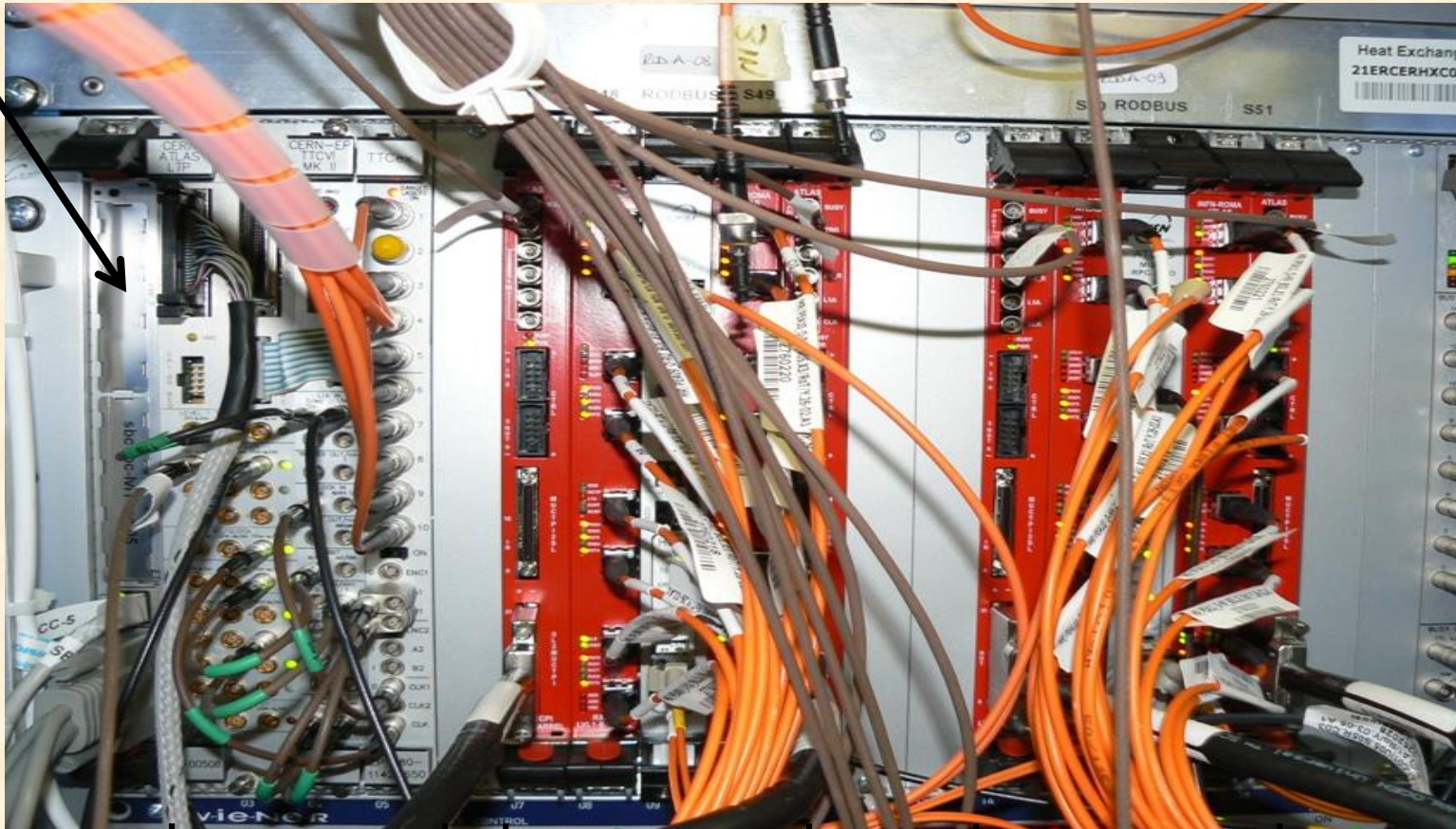
The VMEbus crates of the ATLAS RPC detector



# VMEbus in action

A single ROD crate

Processor board

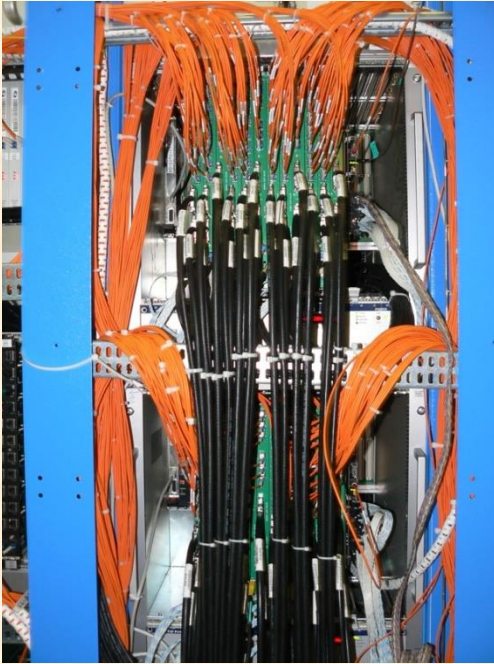


TTC interface

Detector interface

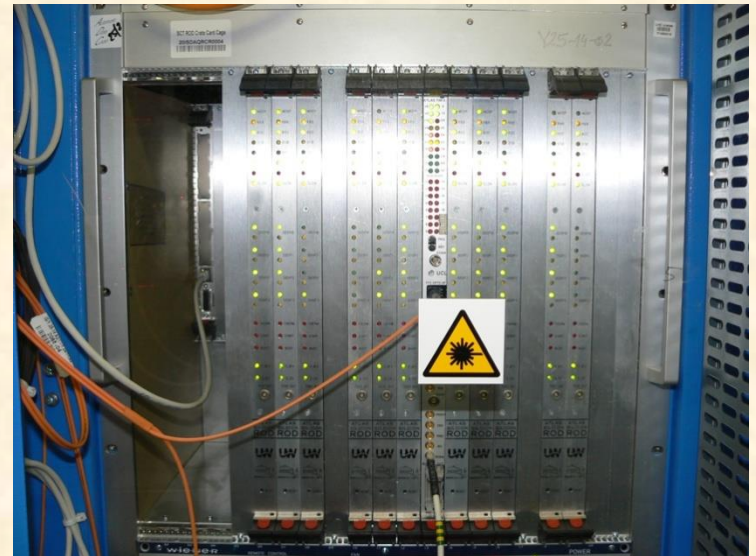
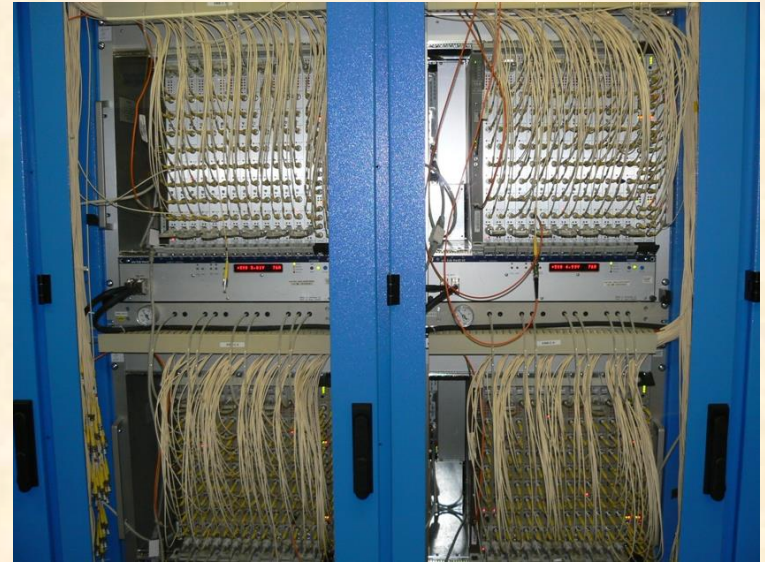


# VMEbus in action



More examples of VMEbus systems in ATLAS

In total there are about ~150 crates in ATLAS and some of them 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)



## Four Decades After Introduction, Linux's VMEbus Support To Return To Demoted State

Written by [Michael Larabel](#) in [Linux Kernel](#) on 12 June 2022 at 06:53 AM EDT. [4 Comments](#)



Introduced in the early 80's for the Motorola 68000 series was Versa Module Eurocard "VMEbus" standard that continues to see industrial uses. While still seeing some reported use today, unfortunately the same cannot be said for the quality of the VMEbus Linux support. After the VME subsystem was promoted out of staging a decade ago, the VME hardware drivers failed to ever leave staging and the code has fallen into disrepair and no maintenance now for the past half-decade. So the VME subsystem support is preparing to depart back to the land of the kernel's staging area.

With a [patch](#) to remove the CA91CX42 Universe-II driver, Arnd Bergmann is preparing to move the entire VME subsystem back to the staging area. As for the driver being removed outright, Bergmann noted, *"This is one of four remaining drivers using the ancient virt\_to\_bus() interface instead of the dma-mapping interface, making it incompatible with most modern machines. As nobody has cleaned this up, there is a high chance that this driver has no actual users. The chip was introduced in 1997 and only supports 32-bit legacy PCI. It was replaced by TSI148 in 2004, but that chip has since been discontinued, while a version of the older Universe II remains in production after 25 years. The vme\_vmivme7805 board uses Universe-II, so this also gets removed in the process, but PCI add-on cards based on TSI148 can still work in theory."*

As for moving the entire VME subsystem back to staging, [the patch](#) sums it up that maintenance of the drivers and subsystem itself effectively stopped in 2017, no hardware drivers remain in staging, and there is just the limited user-level access code that remains around.

### 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

Source: [https://www.phoronix.com/scan.php?page=news\\_item&px=Linux-VMEbus-Back-To-Staging](https://www.phoronix.com/scan.php?page=news_item&px=Linux-VMEbus-Back-To-Staging)



# Before we continue....

If you join an experiment / project where VMEbus is used right now:

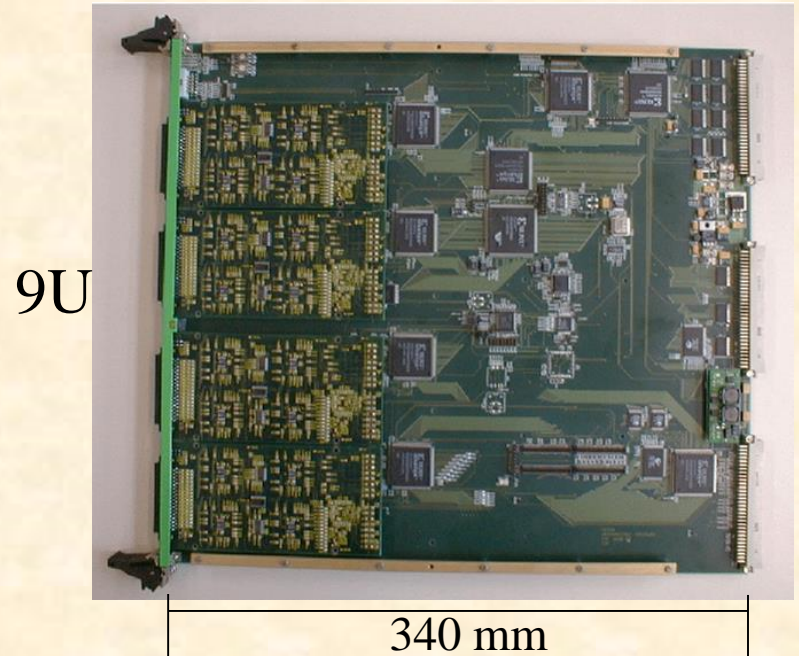
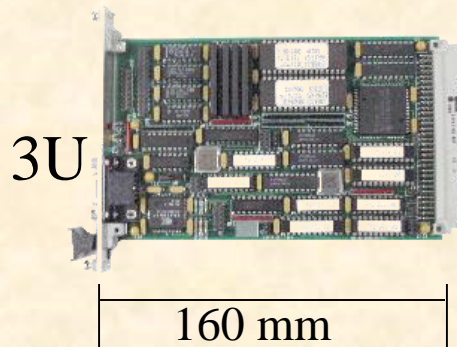
- Learn to **use it** (you will become an indispensable expert)
- Make sure new versions of the **OS** (64bit, etc.) will not be **incompatible** with your potentially old H/W
- If your experiment will last for another ~5 years (or more): Devise a **migration strategy** to e.g. MTCA or PCs

If you are starting a new experiment:

- Only use VMEbus as the basis of your DAQ system if you know what you are doing
- **MTCA** (see next lecture) or **PCs** may be a better choice

# VMEbus mechanics

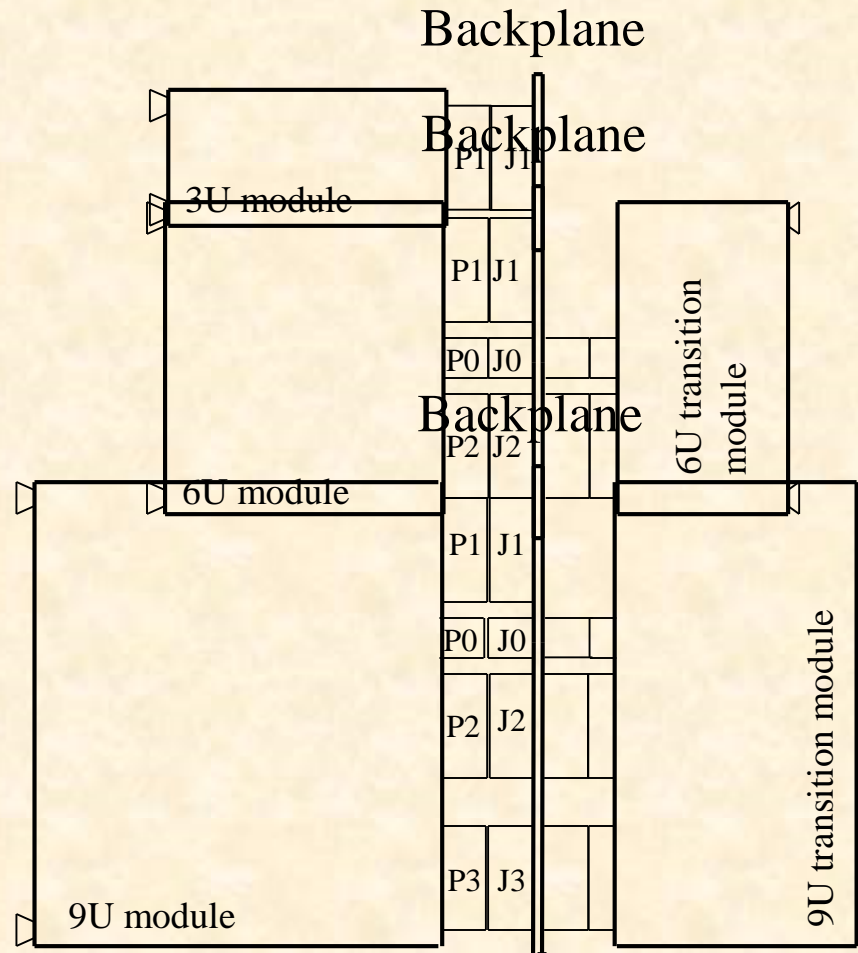
VMEbus cards exist in 3 standard heights: 3U, 6U and 9U (1U = 1.75 inch) and 2 depths: 160 mm (3U, 6U) and 340 mm (9U)



All cards are 0.8 inch (20.3 mm) wide



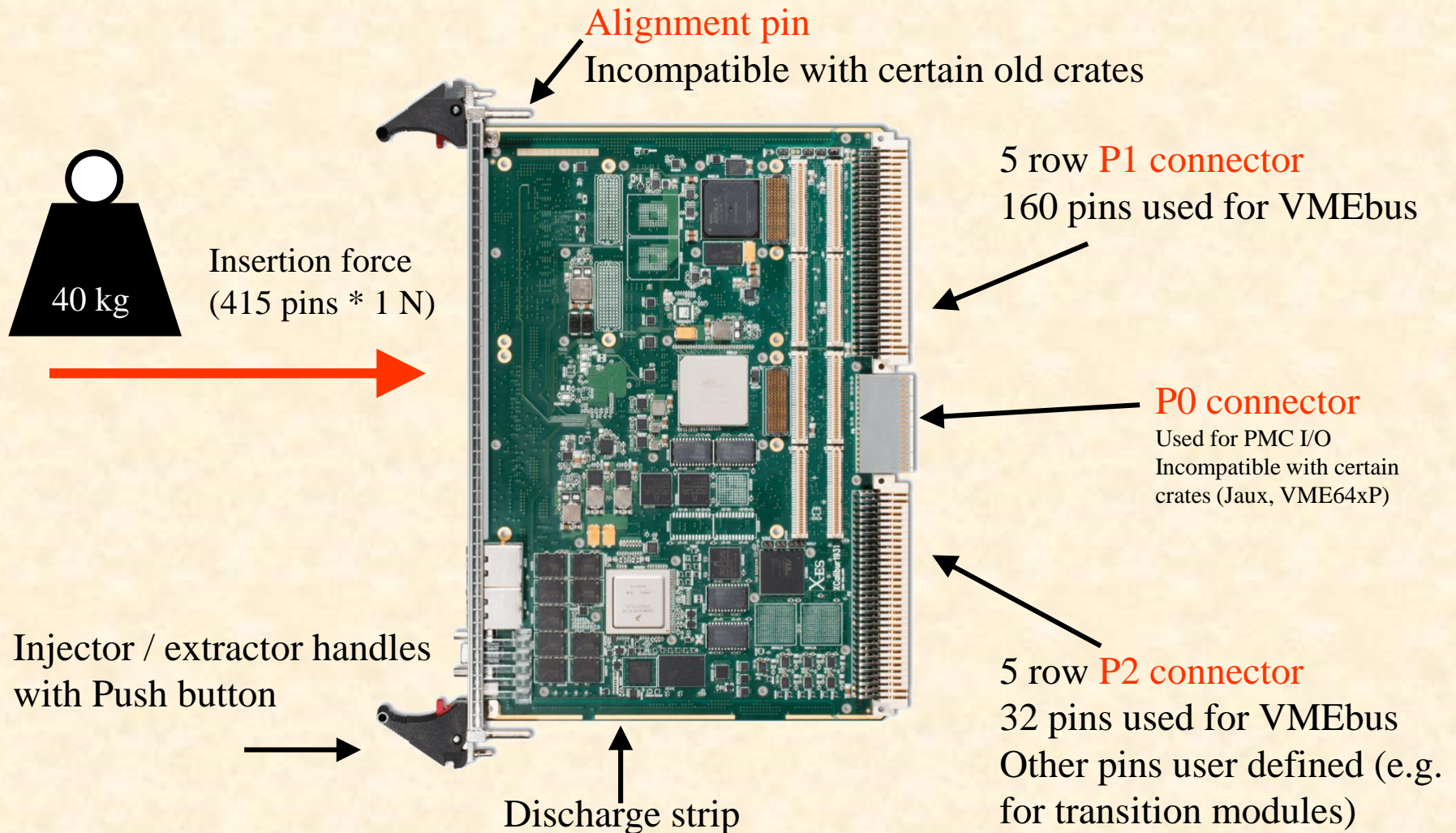
# VMEbus mechanics (2)



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

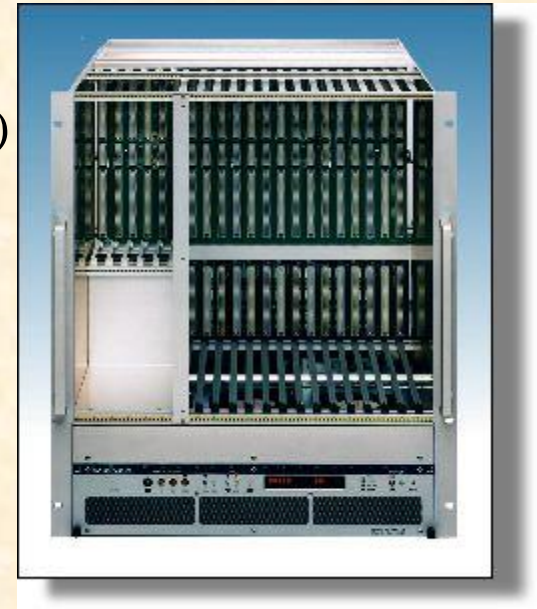




# VMEbus crates



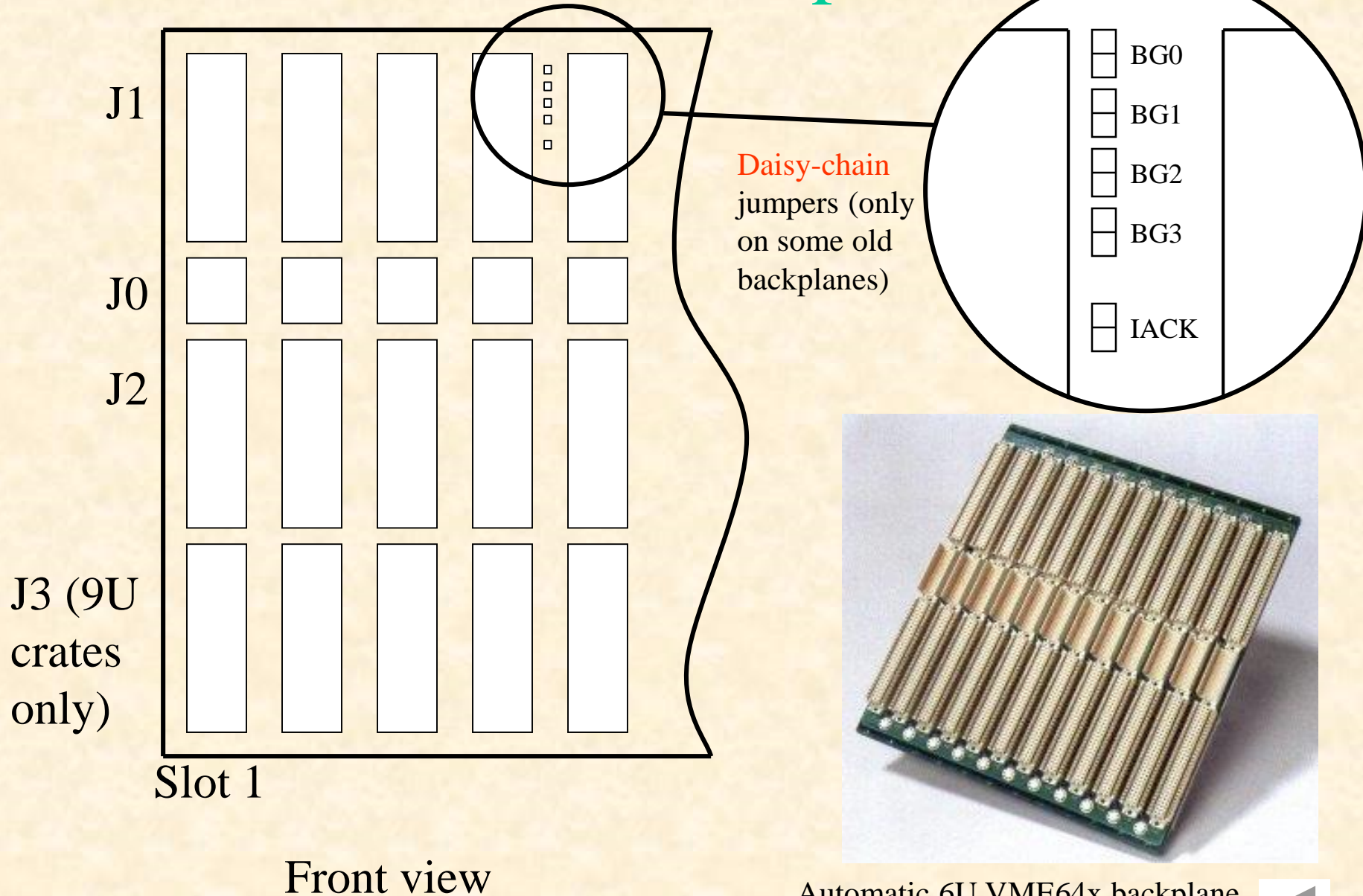
21 slot **9U** crate  
(with 6U section)  
for 19" racks



21 slot **6U** crate  
for 19" racks

- There are different types of power supplies (**5V**, **+/- 12V**, 3.3V, 48V) mounted locally or remotely
- The fan-tray unit allows to monitor parameters like voltages, currents, fan speeds, temperatures
- (Some) crates can be remotely controlled e.g. by a field bus (CAN)
- ATTENTION: The EMC gasket to the left of slot 1 may damage your VMEbus cards

# VMEbus Backplane




Automatic 6U VME64x backplane




# 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)
  - Some (but not all) VMEbus masters (e.g. VP-E24) have automatic byte swapping logic

# VMEbus basics (3)

- Main types of data transfers
  - Single cycles
    - Transfer 8, 16 or 32 bits of data (typically) under the control of the CPU on the master
    - Mnemonic: D8, D16 and D32
    - Typical duration: 1  $\mu$ s + 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 typically by slaves to signal a condition (e.g. data available, internal error, etc.)
    - Can (in principle) have 7 priorities
    - The interrupter provides an 8-bit vector on request of the interrupt handler to identify itself
    - ROAK (Release on Acknowledge) or RORA (Release On Register Access) 
- VMEbus addresses
  - Either 16, 24 or 32 valid bits. Mnemonic: A16, A24 or A32
  - A40 and A64 defined but very rarely used




# 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 and optimizing** VMEbus traffic by S/W (printf(), gdb, etc.) is difficult or even impossible
- A great help for fixing such problems are **VMEbus analyzers**
- In order to understand the output of such an analyzer you have to have some knowledge of the protocol

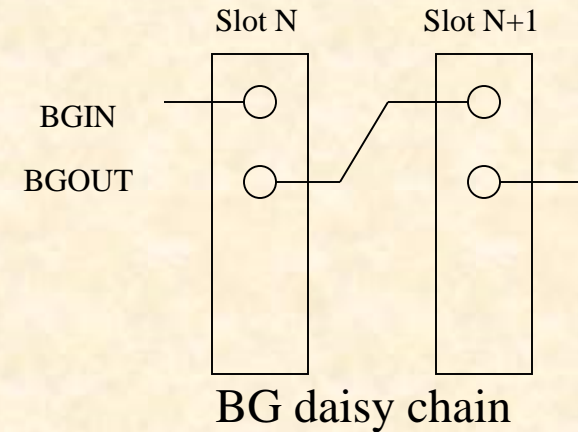
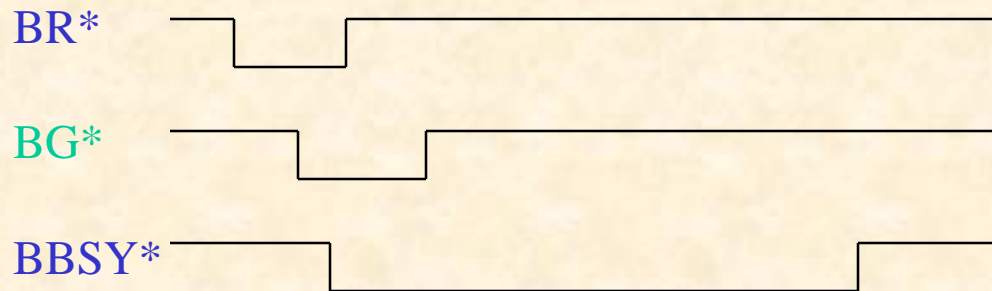
# Important signals

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) ⓘ



Color code: **Arbiter** - **Master**







# Arbitration (2)

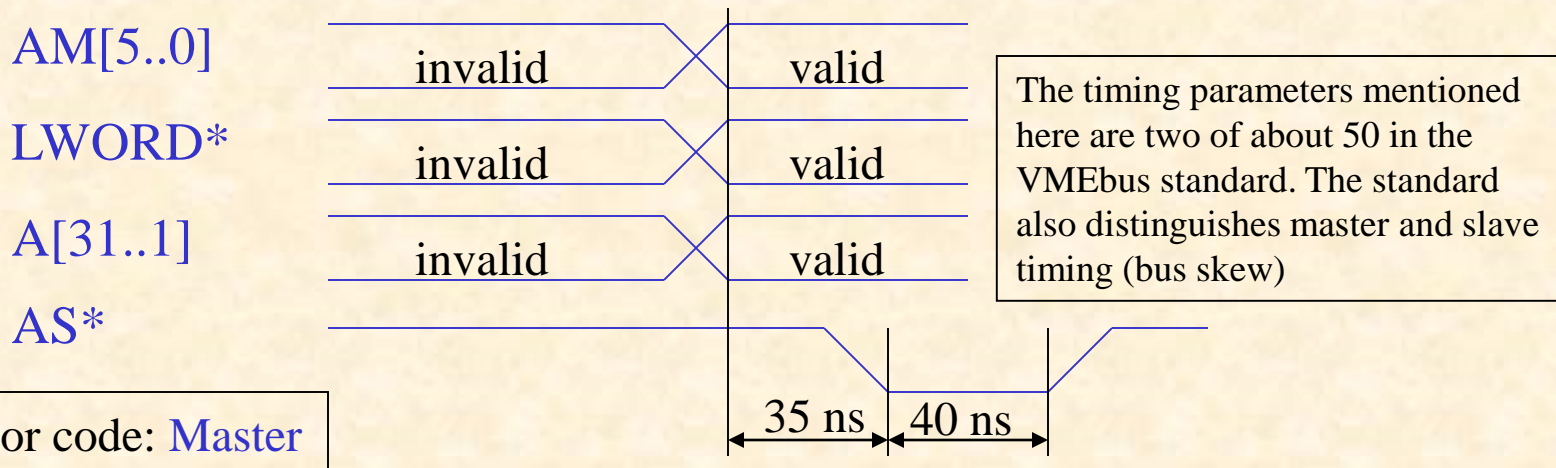
- The arbiter can use different schemes: **PRI** (priority based), **RRS** (round robin)
  - Not an issue for single master systems
- If two masters use the same bus request level the one closer to slot 1 inherently has a higher priority (because it detects BGIN first)
- Modern masters support “fair arbitration”. I.e. they delay their bus request if other masters are requesting the bus at the same level
- A master may get stuck if the BG daisy chain is not closed

# 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 16, 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 (A16, 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

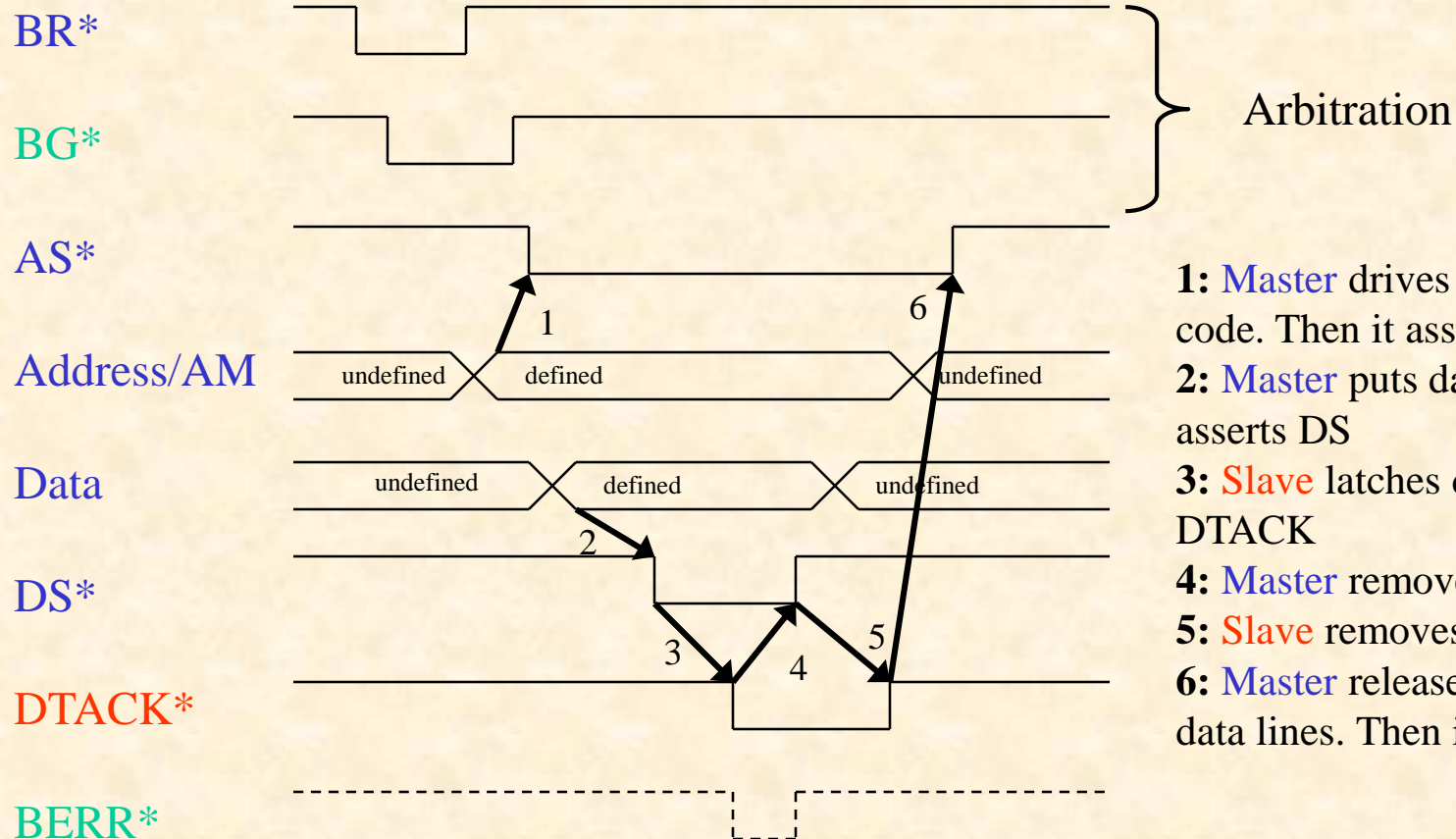
- First the **master** drives **AM**, **Address** and **LWORD\***. Then it waits 35 ns and finally drives **AS\*** to validate the information
- The **slave** has to decode the address information within 40 ns (even though most masters keep AS\* asserted much longer)
- The master does not know if a slave 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...





# Single cycles

Example: (Simplified) **write** cycle



- 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

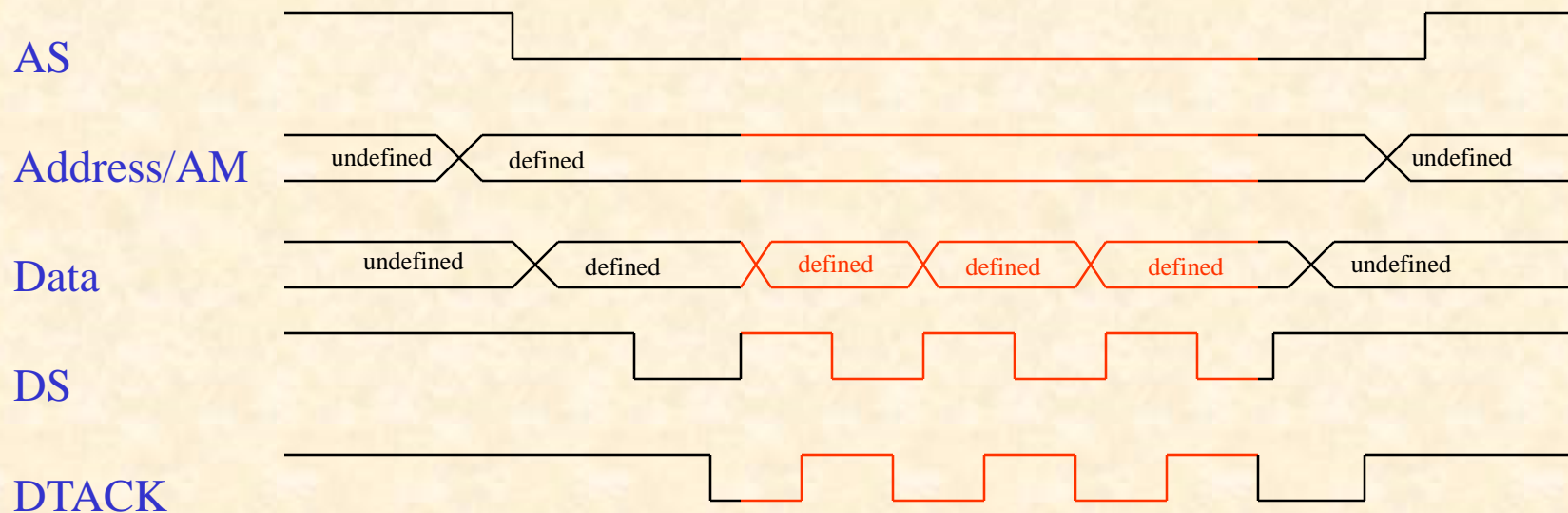
# Single cycles (2)

- 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 0x0000003 may not be a good idea
- VMEbus also supports (rarely used) read-modify-write cycles (useful for semaphores)
- 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)

- A master must not cross a **256 bytes** (D32) or **2048 bytes** (D64) address boundary respectively without releasing AS (transparent to the user)
  - This is to give other masters a chance to acquire the bus before too long
- Reading out **single address FIFOs** is not foreseen by the standard and requires special masters
- Designing a slave that terminates a block transfer from a FIFO with a bus error is legal but bad practice. It does not always work with the IDT Universe chip 
- VMEbus interface chips may require a relative alignment of the remote (VMEbus) and local (PCI) addresses
  - In case of the IDT 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  $\mu$ s 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)

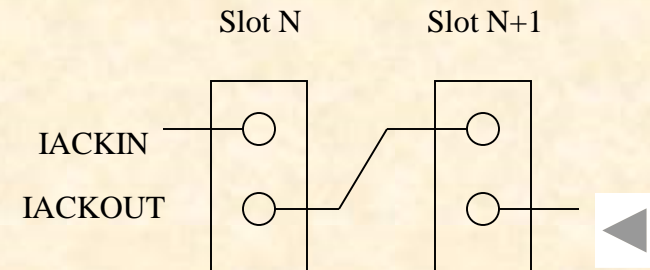
# Bus errors

- 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  $\mu\text{s}$ .
  - 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  $\mu\text{s}$ )
- There is no standard way for the delivery of a BERR from the VMEbus interface to the CPU of the master
  - On PowerPCs BERR is typically converted directly to a non-maskable interrupt and then converted to the SIGBUS signal by the operating system
  - Certain (old) Intel based SBCs ignore bus errors
  - Modern, Intel based SBCs convert it to a regular PCI interrupt. This interrupt is handled by the VMEbus driver and converted to the **SIGBUS signal** 



# Interrupts

- VMEbus provides 7 **interrupt levels** (= bus lines) to prioritize interrupts
- Each interrupter can use any level
- There must only be one interrupt handler for each 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  $\mu$ s** (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
- If two interrupters are active at the same time and on the same level the one closer to slot 1 will be serviced first (IACK daisy chain)



# 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:  
**$$\text{address} = \text{slot\#} * 0x80000$$**
  - 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.

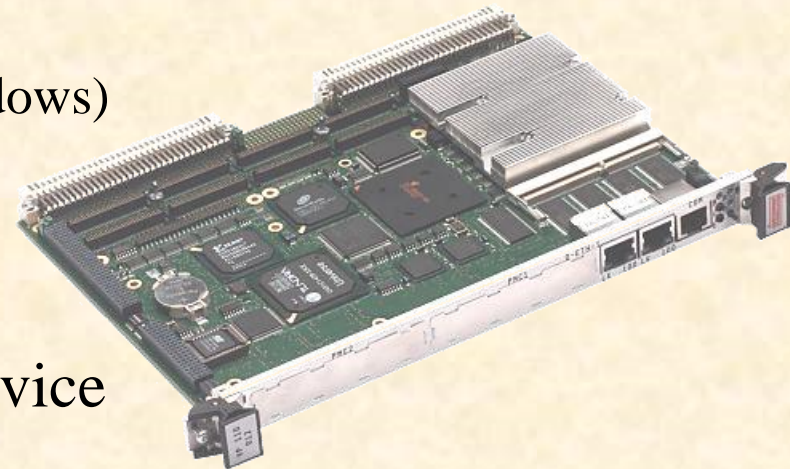
# 2eSST

- 2eSST = 2 edge Source Synchronous Transfer
- An **addition** to the VME64x standard (since 1999)
- It defines a **synchronous protocol** for VMEbus block transfers
- Three transfer speeds are defined: **160**, **267** and **320 MB/s** (8 bytes @ 20, 33.3 and 40 MHz)
- Allows for data **broadcast** and **multicast**
- Works only reliably on high quality backplanes (incident wave switching) and with special (Texas Instruments) driver chips
- There exist only a hand full of VMEbus modules built to this standard



# The VMEbus single board computer

- Usually this is the only **master** and **interrupt handler** in the crate
- It often 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
  - Typically **IDT Universe** / Tsi148
  - Depending on the model and the S/W used the VMEbus I/F has to be configured in the BIOS or at start-up by special programs
- Some SBCs can be equipped with mezzanines (PMC, IP) but this is another story

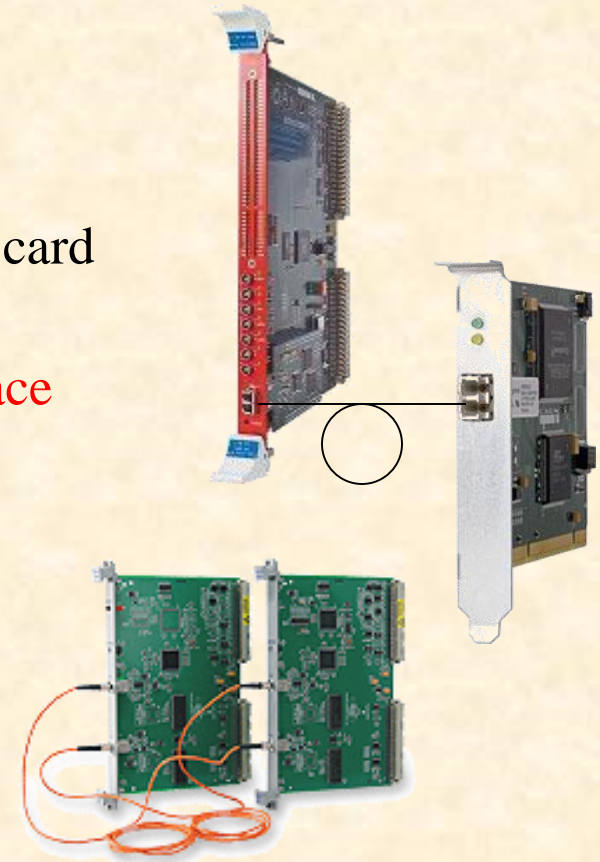


# The VP-E24 SBC (used in the exercises)

CPU	Intel® Atom E3800
RAM	4 GB
VMEbus interface	IDT Universe
PMC sites	Two. 32 or 64 bit, 33 or 66 MHz, (can be configured for 5V or 3.3V signaling)
Mass storage	A SATA or CompactFlash hard disk can be installed on-board. This takes one of the PMC slots
Network interface	Two channels, 1 Gbit/s, RJ45 on front panel
Mechanics	VME64x compliant: 5-row P1 and P2, P0 (optional), front panel with alignment pin and injector / extractor handles (alternative solution for 3-row VME backplanes exists), solder side cover
Terminal connection	DVI (monitor) and USB

# Other VMEbus masters and interfaces

- **PC to VMEbus interfaces**
  - Available from several manufacturers
  - A set typically consists of a PCI card, a VMEbus card and a cable (copper or optical fiber)
  - Also available: VMEbus master with **USB interface**
- **VMEbus repeaters**
  - Allows a master in **crate 1** to access a slave in **crate 2**
  - A set consists of two VMEbus cards and a cable
  - There is usually a performance penalty
- **VMEbus to CAMAC interface**
  - Allows a master in a VMEbus crate to control a CAMAC crate
  - A set consists of a VMEbus slave, a CAMAC crate controller and a cable




# Subsystem buses

The P0, P2 and P3 connectors have a number of user defined pins. They can be used to implement specialized communication channels independently from the VMEbus protocol. Examples:

- **VXS** (-> next lecture)
  - A recent but **unsuccessful** addition to the standard
  - It allows each VMEbus card to connect to a switch fabric.
  - Initially it uses the Ethernet protocol but other technologies (e.g. PCIe) are possible as well
- **Custom P3**
  - In 9U crates the P3 is totally user defined
  - Special backplanes are possible too
- **VSB** (VME Subsystem Bus)
  - Obsolete.
  - Provides a 32 bit bus for up to 8 adjacent cards
  - Was used to interface to **FASTBUS**



# 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**
- The drivers typically 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 **memcpy()**; 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**

# Debugging tools

- H/W (Examples)
  - 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



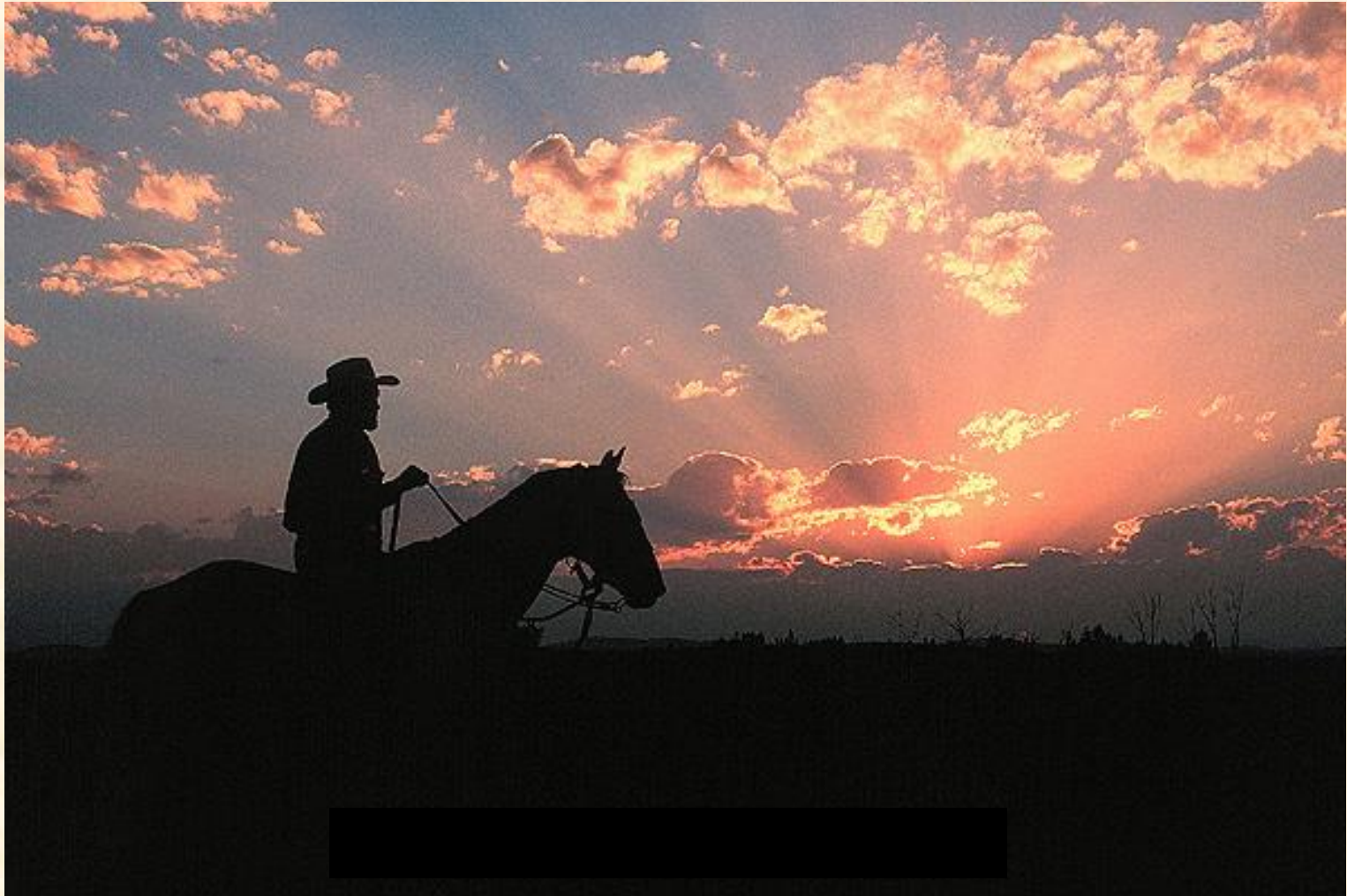
Note: This slide contains product placement



# Links for further information

- VMEbus standard
  - [www.vita.com](http://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
  - <https://www.kernel.org/doc/html/latest/driver-api/vme.html>
  - <http://www.vmlinux.org/> (Webpage last changed on May 24, 2003)
- VMEbus market overview
  - <http://www.vita.com>

# *The End*



# Additional slides

The slides below provide additional information at a more detailed level. Some of them are based on the VMEbus S/W that was developed at CERN for the ATLAS and ALICE experiments. Keep in mind that the library functions and applications of this S/W package are different from other (commercial or public domain) packages for VMEbus access

# 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/lsmmod`
- In what state is my driver?
  - `more /proc/<name>` (value of name depends on the driver used.  
ATLAS: “vme\_rcc”, “cmem\_rcc” or “io\_rcc”)
- Is the driver currently used
  - `/sbin/lsmmod`
  - Check the “Used by” number in the third column.
- How to restart a driver
  - “Used by” has to be 0
  - `su` (then enter root password)
  - `cd /etc/rc.d/init.d`
  - `<name> restart` (e.g. “vme\_rcc restart”)



# (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

# Other information

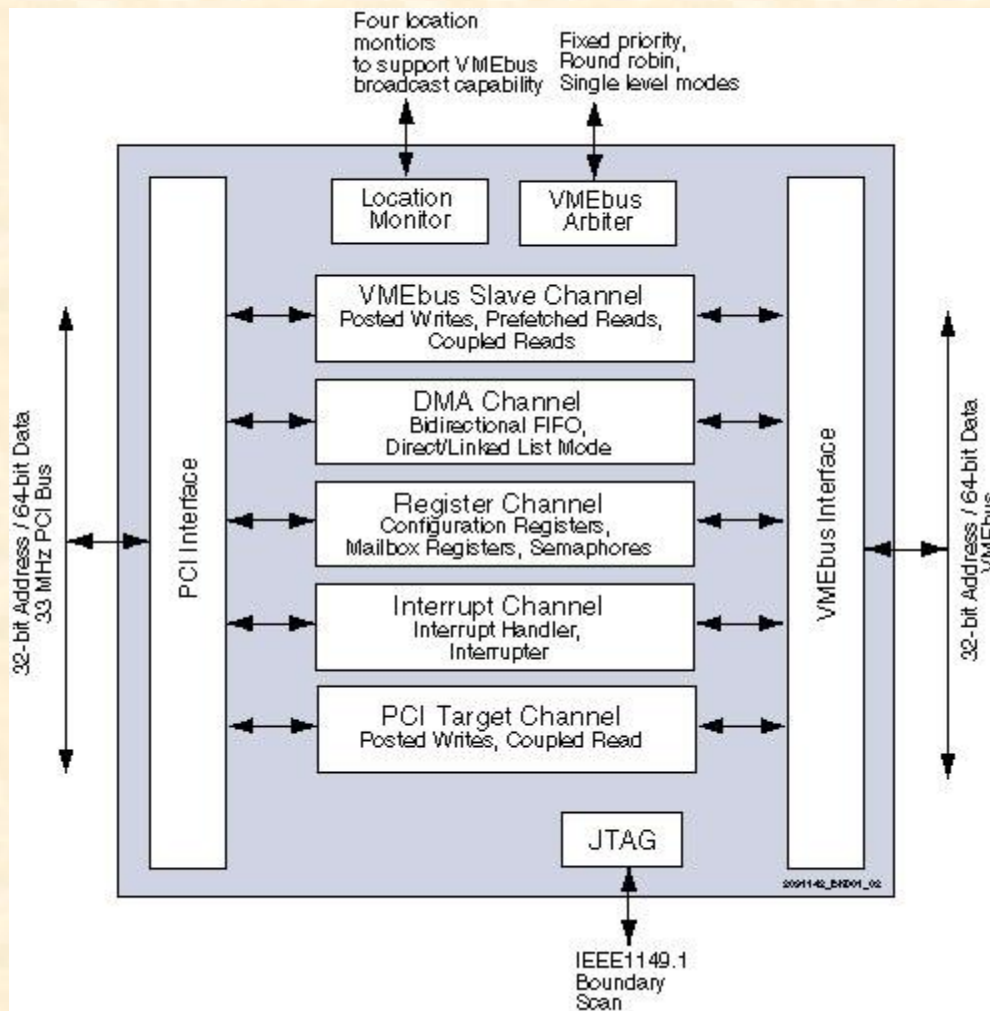
- For the ATLAS VMEbus library there is a C++ wrapper
  - <https://edms.cern.ch/file/325729/4/wrapper.pdf>



# 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

# The IDT Universe II ASIC



- Bridges VMEbus to PCI
- Used on most of the commercially available VMEbus processors



# The ATLAS vme\_rcc package

- The vme\_rcc package contains the default VMEbus driver and library
- Documentation: <https://edms.cern.ch/document/325729/4>
- Supported H/W: Most SBCs of Concurrent Technologies
- The source code of both the driver and the library has been fully developed at CERN and tested on Linux releases up to CC7 (it already compiles for CC8)
- Why not using an existing VMEbus driver instead of developing a new one?
  - We wanted to have a sufficiently generic API that could easily be implemented for any VMEbus interface on a SBC
  - Most drivers for the Universe chip lack support for some features
  - External code is not necessarily optimized for your type of applications

# Debugging tools

- S/W: The ATLAS vme\_rcc package (other packages may have similar tools)
  - Look at the `/proc/vme_rcc` file of the driver
  - In the vme\_rcc package you find special applications
    - scanvme: Scan VMEbus for modules
    - vme\_rcc\_test: Use the functions of the library interactively. This program is also a good programming example
    - cctscope: Decode and dump the configuration of the Universe 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 Universe chip

- The API used in the vme\_rcc package is generic enough to fit (almost) any type of VMEbus interface. So far there exists only an implementation for the Universe and TSI148 chips
- The IDT Universe chip is used on most current SBCs but has a number of limitations:
  - There are only 8 map decoders for master and slave pages respectively
    - 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
  - Some BERRs (posted write) are difficult to catch. The SBCs from Concurrent Technologies have extra logic to cope with that
  - There is no H/W byte swapping (required on little endian CPUs). The SBCs from Concurrent Technologies have extra logic for that purpose

# vme\_rcc: Overview

- The package provides:
  - A VMEbus driver
    - To be dynamically installed in the Linux kernel
    - You need the **cmem\_rcc** and **io\_rcc** drivers as well
    - If you are not using the standard CERN kernels you may have to compile the drivers yourself
  - A library
    - There are about 60 functions.
  - Utility programs (selection)
    - vmeconfig
    - cctscope
      - Dump the register of the Universe chip and some other resources of the SBC in human readable form
    - scanvme
      - Scan the VMEbus for slave cards at unknown addresses

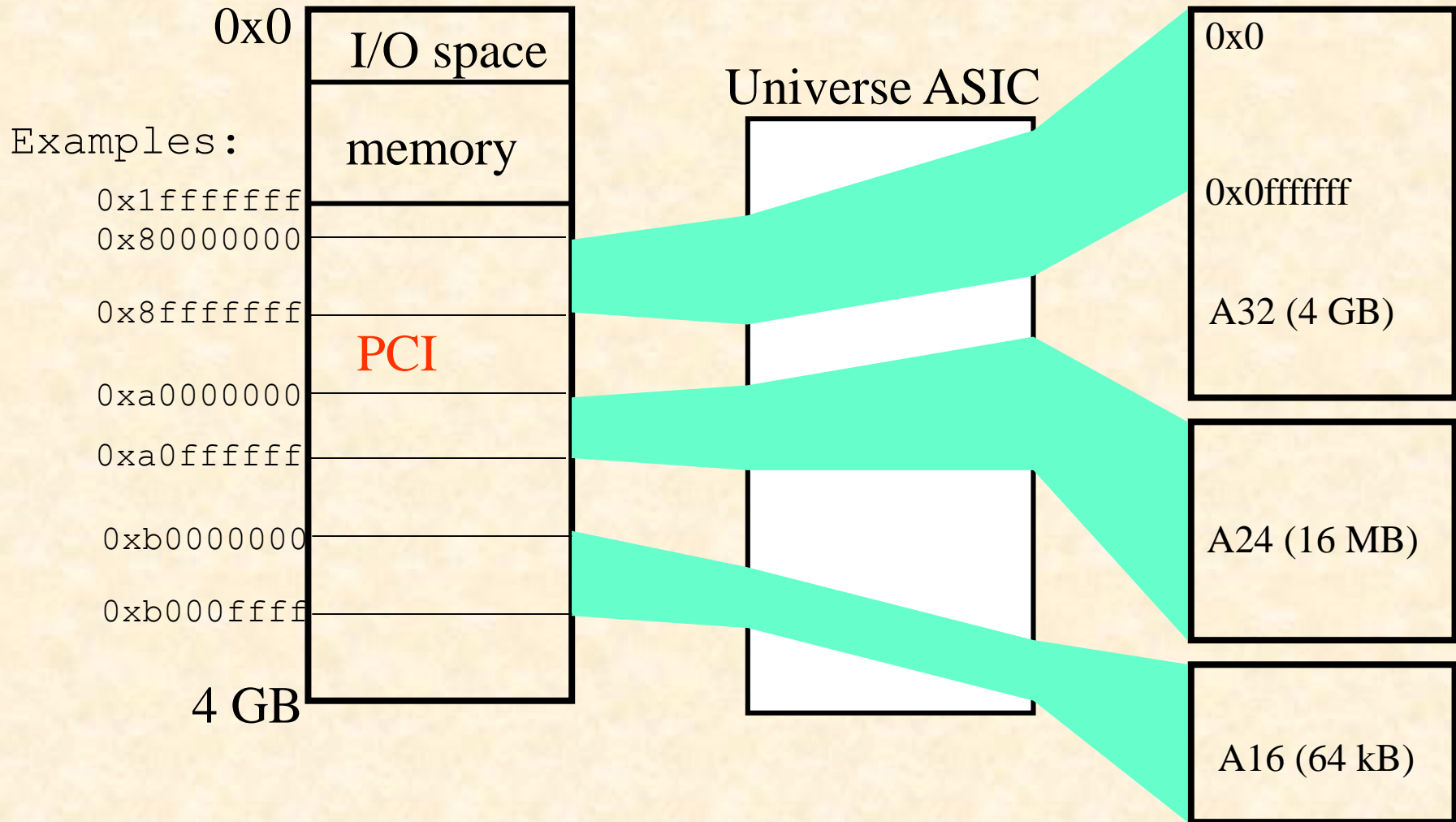
# System initialization

- Before the first VMEbus cycle can be made one has to program (at least) one of the 8 map decodes with appropriate parameters
- This basically means **mapping** a range of **PCI** addresses (MEM space) **to** an equally large window of **VMEbus** addresses
- In many drivers this can be done dynamically at run time via a function call (but the VMEbus and PCI base addresses still have to be provided by the user)
- `vme_rcc` is different. The library cannot modify the set-up of the map decoders. This is the job of **`vmeconfig`**
  - Using a static set-up has the advantage that one can not run out of map decoders in the middle of an application
  - This policy enforces some discipline and is therefore not liked by everybody. We are, however, convinced that it helps to reduce problems

# Address spaces

CPU addresses

VME addresses





# vmeconfig

- If called in the form “vmeconfig –a vmetab” it loads a user set-up into the Universe chip. This typically happens automatically at boot time
- Called as “vmeconfig –i vmetab” it allows you to edit the configuration file (vmetab)
- The configuration file is a binary. It cannot be modified with a text editor
- The (most important) user parameters are:
  - Master and slave mapping
  - Interrupts
  - Byte swapping
  - Arbitration and bus request modes
- In order to run vmeconfig you need a number of dynamic libraries from the ATLAS TDAQ release. Use the command “ldd vmeconfig” to check
- Have a look at the on-line help in vmeconfig

# vmeconfig (2)

## Example: How to set up a master mapping

Use option 4 to program a map decoder. vmeconfig will ask you to enter a number of parameters:

```
Enter number of map decoder          <0..7> [0] :
Enable map decoder                   <0=no 1=yes> [1] :
Select VMEbus base address           [0x00000000] :
Select PCI base address              [0x90000000] :
Select the window size (bytes)       [0x00010000] :
Select Write posting                 <0=no 1=yes> [0] :
Select address space
  <0=A16, 1=A24, 2=A32, 5=CR/CSR, 6=USER1, 7=USER2> [0] :
Select cycle type                    <0=User, 1=Supervisor> [0] :
Select cycle type                    <0=Data, 1=Program> [0] :
```

Defining the PCI base address is the artistic part because you have to “guess” it. For an educated “guess” use these guidelines:

- DRAM addresses grow from 0x0 upwards
- PCI devices are mapped by the kernel from 0xffffffff downwards
- Addresses in the range 0x50000000 – 0xbfffffff should be safe
- Never use addresses that are listed in /proc/iomem

When you are done upload the configuration into the Universe and save your changes. It is recommended to check the new set-up with cctscope (function 2/2) for errors like address overlaps

# 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
- All functions return **error codes** in the format defined in the `rcc_error` package
- There exists a C++ wrapper in a separate package (`RCDVme`)

# 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);  
ldata = *lptr;
```


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

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

```
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);
}
```



# 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 chip for which nobody has requested support so far

# Service packages

- **cmem\_rcc**
  - Driver and library for the allocation of contiguous memory (e.g. for block transfers) either via the `get_free_pages()` kernel function or the BigPhysArea patch
  - Used by some of the test programs in the vme\_rcc package
  - <https://edms.cern.ch/document/336290/3>
- **io\_rcc**
  - Driver and library for the access to PCI and PC I/O registers from user code
  - Used by some of the test programs in the vme\_rcc package
- **rcc\_error**
  - A simple library for error reporting