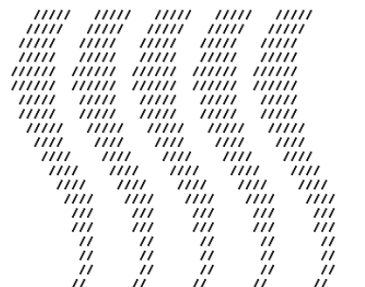




FYS 4220 / 9220 – 2011 / #8

Real Time and Embedded Data Systems and Computing

Real-Time / Embedded facilities

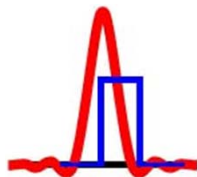


T O R N A D O
Development System
Host Based Shell
Version 2.0.2

Copyright 1995-1999 Wind River Systems, Inc.



PowerMIOAS M5000 (VME)



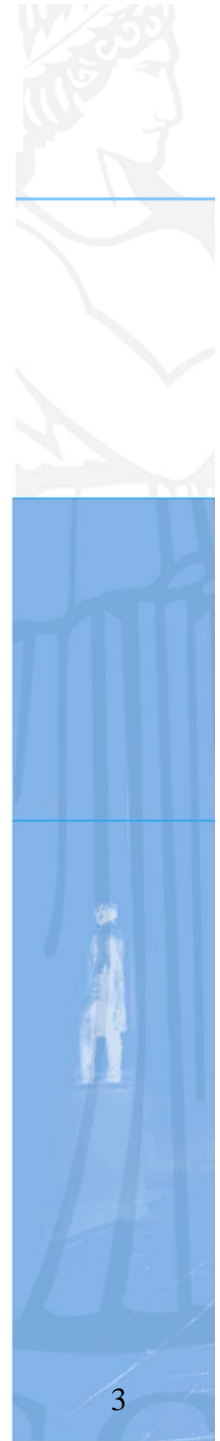


Real-Time / Embedded facilities

- The lecture will discuss the following software and hardware facilities for building Real-time and embedded systems:
 - Clocks and time
 - timer
 - watchdog
 - VxWorks timex
 - Instrumentation/bus/interconnect systems
 - I/O



UNIVERSITY
OF OSLO



CLOCKS



Real-Time clock and time facilities

- Interfacing to the passage of time of "the real world" is through "Clocks"
 - Absolute and relative time
 - Global time UTC
 - Delays
 - Timeouts
- Timing requirements:
 - Periodic execution of processes
 - Deadlines
- Timers and watchdogs
- Synchronization



Clocks, POSIX, time.h (1/3)

- The `<time.h>` header shall declare the structure **tm**, which shall include at least the following members:
 - int `tm_sec` Seconds [0,60].
 - int `tm_min` Minutes [0,59].
 - int `tm_hour` Hour [0,23].
 - int `tm_mday` Day of month [1,31].
 - int `tm_mon` Month of year [0,11].
 - int `tm_year` Years since 1900.
 - int `tm_wday` Day of week [0,6] (Sunday =0).
 - int `tm_yday` Day of year [0,365].
 - int `tm_isdst` Daylight Savings flag.

The value of `tm_isdst` shall be positive if Daylight Savings Time is in effect, 0 if Daylight Savings Time is not in effect, and negative if the information is not available.



Clocks, POSIX, time.h (2/3)

- The `<time.h>` header shall define the following symbolic names:
- NULL
 - Null pointer constant.
- CLOCKS_PER_SEC
 - A number used to convert the value returned by the `clock()` function into seconds.
- CLOCK_PROCESS_CPUTIME_ID
 - [TMR|CPT]
The identifier of the CPU-time clock associated with the process making a `clock()` or `timer*()` function call.
- CLOCK_THREAD_CPUTIME_ID
 - [TMR|TCT]
The identifier of the CPU-time clock associated with the thread making a `clock()` or `timer*()` function call.



Clocks, POSIX, time.h (3/3)

- The `<time.h>` header shall declare the structure *timespec*, which has at least the following members:
 - `time_t tv_sec` Seconds.
 - `long tv_nsec` Nanoseconds.
- The `<time.h>` header shall also declare the *itimerspec* structure, which has at least the following members:
 - `struct timespec it_interval` Timer period.
 - `struct timespec it_value` Timer expiration.
- The following manifest constants shall be defined:
 - `CLOCK_REALTIME`
 - The identifier of the system-wide real-time clock.
 - `TIMER_ABSTIME`
 - Flag indicating time is absolute. For functions taking timer objects, this refers to the clock associated with the timer. If one wants to work in relative time specify `TIMER_RELTIME`.
 - `CLOCK_MONOTONIC`
 - The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via `clock_settime()` and which cannot have backward clock jumps. The maximum possible clock jump shall be implementation-defined

some lines of VxWorks ...\target\h\time.h



```
/* time.h - POSIX time header */

/*
 * Copyright (c) 1992-2005 Wind River Systems, Inc.
 */

typedef int clockid_t;

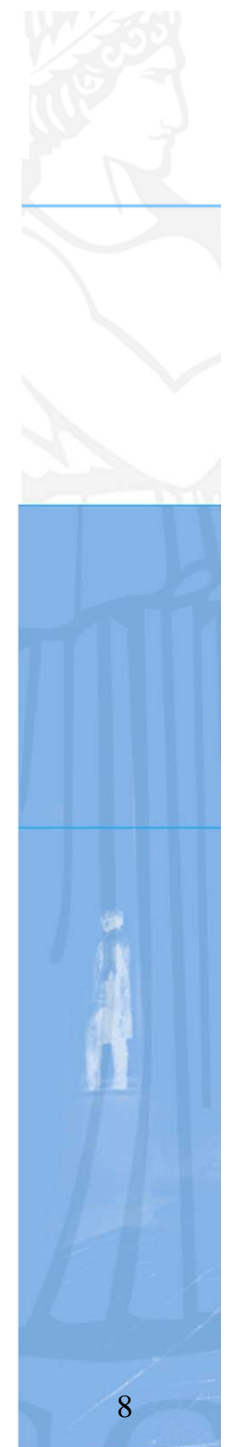
#define CLOCKS_PER_SEC          sysClkRateGet()
#define CLOCK_REALTIME          0x0          /* system wide realtime clock */
#define TIMER_ABSTIME           0x1          /* absolute time */
#define TIMER_RELTIME (~TIMER_ABSTIME)      /* relative time */

struct timespec
{
    time_t tv_sec;              /* interval = tv_sec*10**9 + tv_nsec */
    long tv_nsec;              /* seconds */
};

struct itimerspec
{
    struct timespec it_interval; /* timer period (reload value) */
    struct timespec it_value;    /* timer expiration */
};

struct tm
{
    int tm_sec;                /* seconds after the minute - [0, 59] */
    int tm_min;                /* minutes after the hour - [0, 59] */
    int tm_hour;               /* hours after midnight - [0, 23] */
    int tm_mday;               /* day of the month - [1, 31] */
    int tm_mon;                /* months since January - [0, 11] */
    int tm_year;                /* years since 1900 */
    int tm_wday;                /* days since Sunday - [0, 6] */
    int tm_yday;                /* days since January 1 - [0, 365] */
    int tm_isdst;               /* Daylight Saving Time flag */
};

/* function declarations */
extern uint_t _clocks_per_sec(void);
extern char * asctime (const struct tm * _tpr);
extern clock_t clock (void);
extern char * ctime (const time_t * _cal);
extern double difftime (time_t _t1, time_t _t0);
extern struct tm * gmtime (const time_t * _tod);
extern struct tm * localtime (const time_t * _tod);
```





VxWorks POSIX *clockLib*

clockLib - clock library (POSIX)

ROUTINES

clock_getres() - get the clock resolution (POSIX)

clock_setres() - set the clock resolution

clock_gettime() - get the current time of the clock (POSIX)

clock_settime() - set the clock to a specified time (POSIX)

DESCRIPTION

This library provides a clock interface, as defined in the IEEE standard, POSIX 1003.1b. A clock is a software construct that keeps time in seconds and nanoseconds. The clock has a simple interface with three routines: *clock_gettime*(), *clock_getres*(), and *clock_setres*(). The non-POSIX routine *clock_setres*() is provided (temporarily) so that **clockLib** is informed if there are changes in the system clock rate (e.g., after a call to *sysClkRateSet*()). Times used in these routines are stored in the timespec structure:

```
struct timespec
{
time_t tv_sec;           /* seconds */
long tv_nsec;           /* nanoseconds (0 -1,000,000,000) */
};
```

IMPLEMENTATION

Only one *clock_id* is supported, the required **CLOCK_REALTIME**.



clockLib - example

```
void    ClockResolution()
{
    int          status;
    clockid_t    clock_id = CLOCK_REALTIME;
    struct timespec res;
    char * array[2] = {"OK", "ERROR"};

    res.tv_sec = 0;
    res.tv_nsec = 0;

    status = clock_getres (clock_id, &res);
    if (status == ERROR) status = 1;
    printf("clock_getres status = %s\n", array[status]);
    printf("clock resolution = %d nsec = %f msec\n",
           (int)res.tv_nsec, (float)(float)res.tv_nsec/1000000);
    printCR;
}
```

Run the code:

```
-> ClockResolution
clock_getres status = OK
clock resolution = 16666666 nsec = 16.666666 msec
```



Delays (VxWorks)

taskDelay() - delay a task from executing

SYNOPSIS

STATUS taskDelay (int ticks /* number of ticks to delay task */)

DESCRIPTION

This routine causes the calling task to relinquish the CPU for the duration specified (in ticks). This is commonly referred to as manual rescheduling, but it is also useful when waiting for some external condition that does not have an interrupt associated with it.

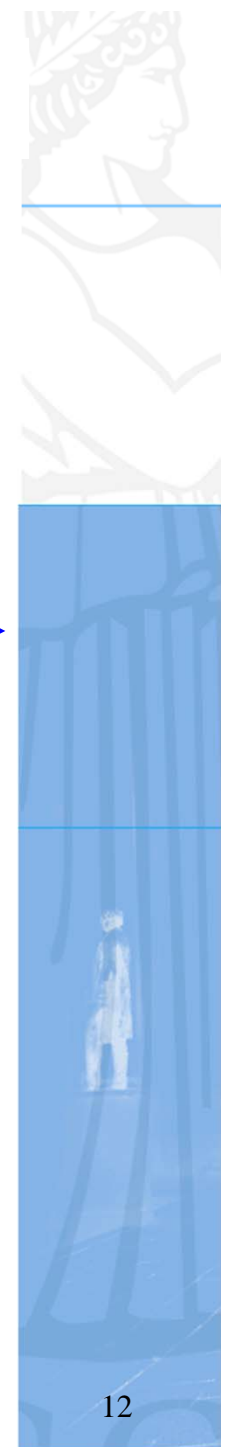
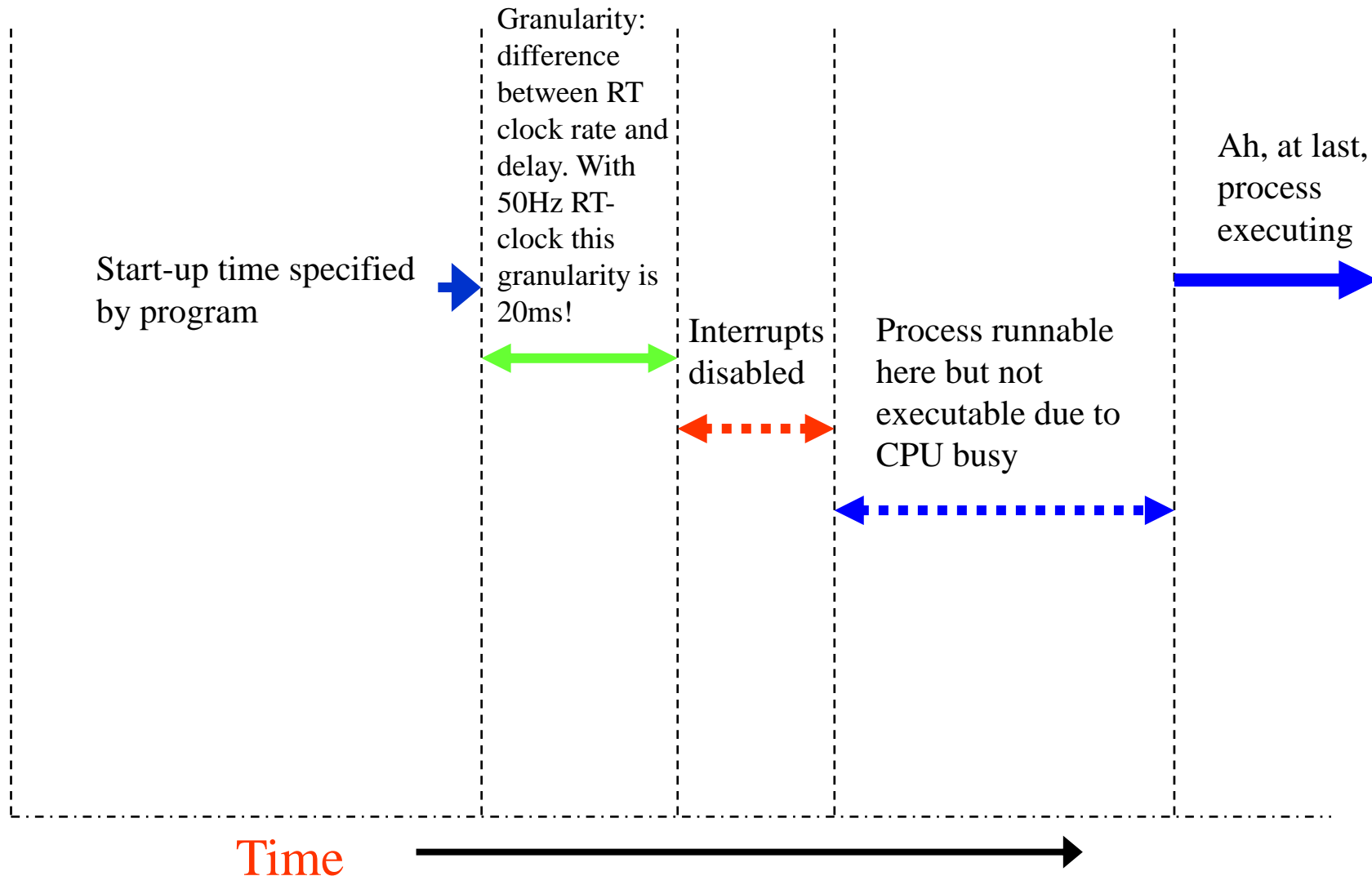
Note! If the calling task receives a signal that is not being blocked or ignored, *taskDelay()* immediately returns ERROR and sets **errno** to EINTR after the signal handler is run.

RETURNS

OK, or ERROR if called from interrupt level or if the calling task receives a signal that is not blocked or ignored.

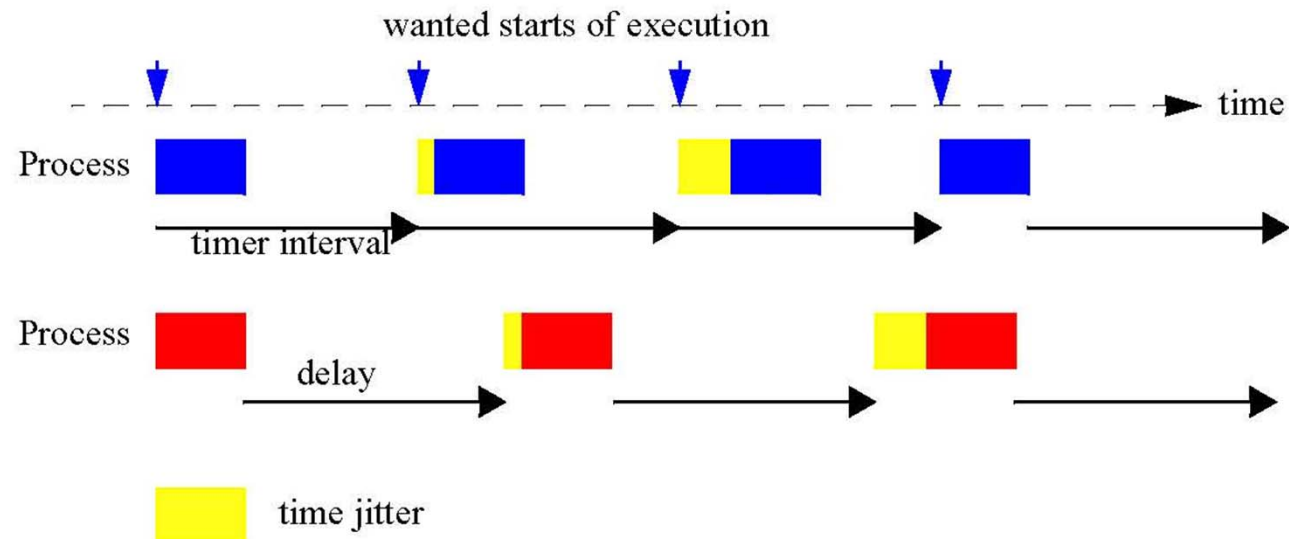


Time reference and time jitter



Time drift

- Cumulative time drift must be avoided when a process is executed periodically
- Using VxWorks `taskDelay()` instead of a timer will result in a cumulative drift!





POSIX timers (VxWorks)

- The POSIX standard provides for identifying multiple virtual clocks, **but only one clock is required--the system-wide real-time clock, identified in the clock and timer routines as `CLOCK_REALTIME`**. VxWorks provides routines to access the system-wide real-time clock; see the reference entry for **clockLib**. (No virtual clocks are supported in VxWorks.)
- The POSIX timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to **create, set, connect** and **delete** a timer; see the reference entry for **timerLib**. When a timer goes off, the default signal (**SIGALRM**) is sent to the task. **sigaction()** can be used to install a signal handler that executes when the timer expires. Alternatively, **timer_connect()** can be used.
- An additional POSIX function, **nanosleep()**, allows specification of sleep or delay time in units of seconds and nanoseconds



timerLib - timer library (POSIX)

ROUTINES

- timer_cancel* () - cancel a timer
- timer_connect* () - connect a user routine to the timer signal
- timer_create* () - allocate a timer using the specified clock for a timing base (POSIX)
- timer_delete* () - remove a previously created timer (POSIX)
- timer_gettime* () - get the remaining time before expiration and the reload value (POSIX)
- timer_getoverrun* () - return the timer expiration overrun (POSIX)
- timer_settime* () - set the time until the next expiration and arm timer (POSIX)
- nanosleep* () - suspend the current task until the time interval elapses (POSIX)

DESCRIPTION

This library provides a timer interface, as defined in the IEEE standard, POSIX 1003.1b.

Timers are mechanisms by which tasks signal themselves after a designated interval.

Timers are built on top of the clock and signal facilities. The clock facility provides an absolute time-base.

Standard timer functions simply consist of creation, deletion and setting of a timer.

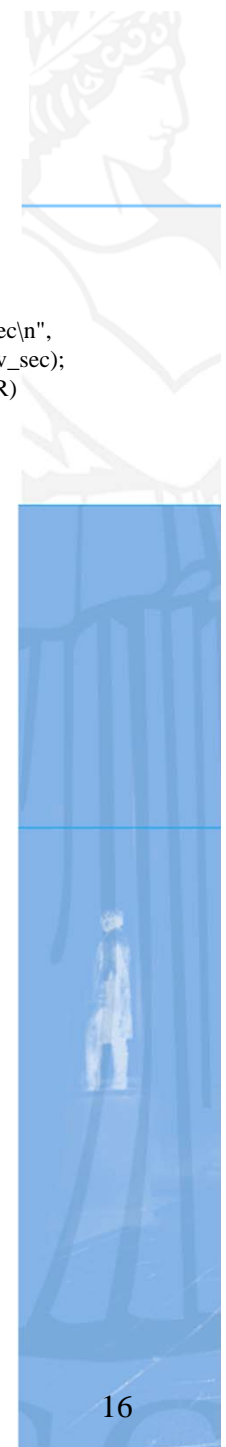
When a timer expires, *sigaction* () (see **sigLib**) must be in place in order for the user to handle the event.

The "high resolution sleep" facility, *nanosleep* (), allows sub-second sleeping to the resolution of the clock.

The **clockLib** library should be installed and *clock_settime* () set before the use of any timer routines.

Timer code: see demo program next page

Using a timer to run *timerhandle()* periodically



```
/* POSIX timers */

#include "vxWorks.h"
#include "time.h"
#include "timexLib.h"
#include "taskLib.h"
#include "sysLib.h"
#include "stdio.h"

#define          TIMER_START      10
#define          TIMER_INTERVAL  5

/* timer is connected to timerhandle() */
void timerhandle(timer_t timerID, int targ)
{
    int i;
    printf("timerhandle invoked with targ = %d\n", targ);
    /* some CPU eating stuff */
    for (i = 0; i < 200000; i++) {};
}

/* run the demo from here */
int execTimer (void)
{
    timer_t timerID;
    struct itimerspec value, ovalue, gvalue;
    int t_arg = 12321;
    int i;

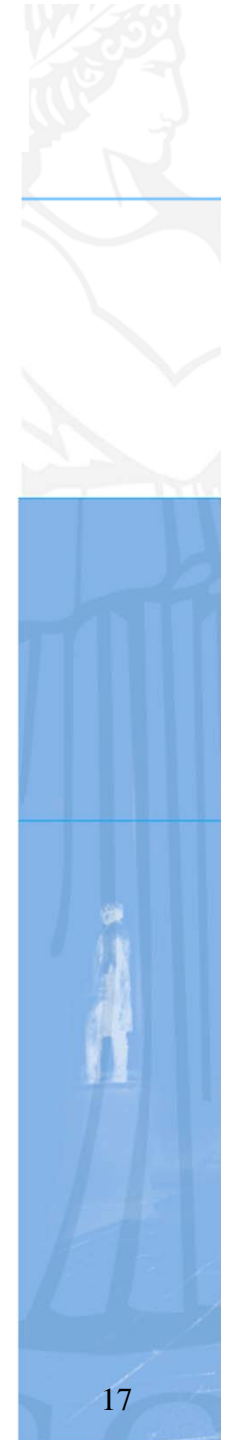
    if (timer_create (CLOCK_REALTIME, NULL, &timerID) == ERROR)
    {
        printf ("create FAILED\n");
        return (ERROR);
    }
    if (timer_connect (timerID, (VOIDFUNCPTR)timerhandle, t_arg) == ERROR)
    {
        printf ("connect FAILED\n");
        return (ERROR);
    }
}
```

```
value.it_value.tv_nsec = 0;
value.it_value.tv_sec = TIMER_START;
value.it_interval.tv_nsec = 0;
value.it_interval.tv_sec = TIMER_INTERVAL;
    printf("timer set up for start after %ld sec and interval %ld sec\n",
           value.it_value.tv_sec, value.it_interval.tv_sec);
if (timer_settime (timerID, TIMER_RELTIME, &value, &ovalue) == ERROR)
{
    printf ("timer_settime FAILED\n");
    return (errno);
}
/* some diagnostics during 25 sec */
for (i = 0; i < 25; i++) {
    if (timer_gettime (timerID, &gvalue) == ERROR)
    {
        printf ("gettime FAILED\n");
        return (errno);
    }
    printf("gvalue.it_value.tv_sec = %ld\n", gvalue.it_value.tv_sec);
    printf("gvalue.it_interval.tv_sec = %ld\n", gvalue.it_interval.tv_sec);
    taskDelay (CLOCKS_PER_SEC);
}

if (timer_cancel (timerID) == ERROR)
{
    printf ("cancel FAILED\n");
    return (errno);
}
if (timer_delete (timerID) == ERROR)
{
    printf ("delete FAILED\n");
    return (errno);
}
return (OK);
}
```



```
vxsim0@fyspc-elg045 - Host Shell
->
-> execTimer
timer set up for start after 10 sec and interval 5 sec
gvalue.it_value.tv_sec = 10
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 9
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 8
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 7
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 6
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 5
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 4
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 3
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 2
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 1
gvalue.it_interval.tv_sec = 5
timerhandle invoked with targ = 12321
gvalue.it_value.tv_sec = 5
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 4
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 3
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 2
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 1
gvalue.it_interval.tv_sec = 5
timerhandle invoked with targ = 12321
gvalue.it_value.tv_sec = 5
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 4
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 3
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 2
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 1
gvalue.it_interval.tv_sec = 5
timerhandle invoked with targ = 12321
gvalue.it_value.tv_sec = 5
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 4
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 3
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 2
gvalue.it_interval.tv_sec = 5
gvalue.it_value.tv_sec = 1
gvalue.it_interval.tv_sec = 5
timerhandle invoked with targ = 12321
value = 0 = 0x0
->
```





Watchdogs (voff-voff)

- “Watchdog” stands for a periodic activation of a process which gives a signal, for instance by lighting up a lamp, to show that a system is alive and operates correctly
- VxWorks includes a watchdog-timer mechanism that allows any C function to be connected to a specified time delay.
 - Watchdog timers are maintained as part of the system clock ISR. Normally, functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. However, if the kernel is unable to execute the function immediately for any reason (such as a previous interrupt or kernel state), the function is placed on the **tExcTask** work queue. Functions on the **tExcTask** work queue execute at the priority level of the **tExcTask** (usually 0). Restrictions on ISRs apply to routines connected to watchdog timers
 - Demo: see code next page



Watchdog example

```
/* This example creates a watchdog timer and sets it to go off in 3 seconds. */  
  
#include "vxWorks.h"  
#include "sysLib.h"  
#include "logLib.h"  
#include "wdLib.h"  
#include "taskLib.h"  
#include "tickLib.h"  
  
#define SECONDS (3)  
  
WDOG_ID myWatchDogId;  
  
int task (void)  
{  
    /* Create watchdog */  
  
    if ((myWatchDogId = wdCreate( )) == NULL)  
        return (ERROR);  
  
    /* Set timer to go off in SECONDS - printing a message to stdout */  
  
    if (wdStart (myWatchDogId,  
                sysClkRateGet( ) * SECONDS,  
                (FUNCPTR)logMsg,  
                "Watchdog timer just expired\n") == ERROR)  
        return (ERROR);  
  
    taskDelay (sysClkRateGet( ) * 10);  
    return (OK);  
}
```

Wind River Systems NUM Ln 2, Col 2

Output on console terminal: -> interrupt: Watchdog timer just expired



timexLib – execution timer facilities routines

timexInit() -	include the execution timer library
timexClear() -	clear the list of function calls to be timed
timexFunc() -	specify functions to be timed
timexHelp() -	display synopsis of execution timer facilities
timex() -	time a single execution of a function or functions
timexN() -	time repeated executions of a function or group of functions
timexPost() -	specify functions to be called after timing
timexPre() -	specify functions to be called prior to timing
timexShow() -	display the list of function calls to be timed

EXAMPLES

The routine **timex()** can be used to obtain the execution time of a single routine:

– > timex myFunc, myArg1, myArg2, ...

The routine **timexN()** calls a function repeatedly until a 2% or better tolerance is obtained:

– > timexN myFunc, myArg1, myArg2, ...

The routines **timexPre()**, **timexPost()**, and **timexFunc()** are used to specify a list of functions to be executed as a group:

– > timexPre 0, myPreFunc1, preArg1, preArg2, ...

– > timexPre 1, myPreFunc2, preArg1, preArg2, ...

– > timexFunc 0, myFunc1, myArg1, myArg2, ...

– > timexFunc 1, myFunc2, myArg1, myArg2, ...

– > timexFunc 2, myFunc3, myArg1, myArg2, ...

– > timexPost 0, myPostFunc, postArg1, postArg2, ...

The list is executed by calling **timex()** or **timexN()** without arguments:

– > timex or

– > timexN



timexLib example

```
/* timexLib: routines for timing the execution of tasks */
```

```
#include "vxWorks.h"  
#include "time.h"  
#include "timexLib.h"  
#include "taskLib.h"  
#include "stdio.h"
```

```
/* a function to be timed */  
void myFunc (int arg1)  
{  
    int i;  
    for (i = 0; i < arg1; i++) {};  
}
```

```
int execTimex (void)  
{  
    timexN ((FUNCPTR) myFunc, 99999999,0,0,0,0,0,0,0);  
    printf("Happy with this result?\n");  
    return (OK);  
}
```

```
- > execTimex
```

```
Timex: 1 reps. Time per rep = 616 +/- 16 (2%) millisec
```

```
Happy with this result?
```

Estimating Worst Case Execution Time (WCET) is very important , ref. lecture on Scheduling



Timing failures

- Detection of timing failures?
 - Overrun of deadline
 - Overrun of worst-case execution time
 - Timeouts
- And what could be the consequences?
 - Hard Real-Time: potentially disastrous
 - Soft Real-Time: can be accepted from time to another, provided that the overrun is not too large and does not occur too often (whatever that means)
- **POSIX (not VxWorks)**
 - Two clocks are defined: `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID`
 - These can be used in the same way as `CLOCK_REALTIME`
 - Each process/thread has an associated execution-time clock; calls to:
`clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);`
`clock_gettime(CLOCK_THREAD_CPUTIME_ID, &some_timespec_value);`
`clock_getres(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);`
 - will set/get the execution-time or get the resolution of the execution time clock associated with the calling process (similarly for threads)



Global timing and synchronization

- In wide area or global data acquisition system one needs access to a global clock if the registration of data must be time synchronized. The GPS system gives a very high accuracy in the nsec domain
 - To obtain this accuracy, the GPS signals are corrected for relativistic effects
 - However, not all regions on Earth are well covered by GPS
- IEEE 1588 Precision Time Protocol (PTP) [2002, 2008] is a protocol used to synchronize clocks throughout a computer network. On a local area network it achieves clock accuracy in the sub-microsecond range, making it suitable for measurement and control systems.
 - IEEE 1588 is designed to fill a niche not well served by either of the two dominant protocols, NTP and GPS. IEEE 1588 is designed for local systems requiring accuracies beyond those attainable using NTP. It is also designed for applications that cannot bear the cost of a GPS receiver at each node, or for which GPS signals are inaccessible.
 - The Network Time Protocol (NTP) [1985] is a protocol for synchronizing the clocks of computer systems over packet-switched, variable-latency data networks



UNIVERSITY
OF OSLO



INTERCONNECTS



Interconnects for the RT- and Embedded World

- There are many types of interconnect technologies for Real-Time and embedded systems,
 - The Interconnect market is in constant development, driven by:
 - Chip and bus technology
 - Need for higher and higher bandwidths
 - USB 1.0 : up to 12 Mb/s, USB 2.0 : 480 Mb/s and higher, USB 3.0 : 5 Gb/s
 - Electronic packing density – System-on-Chip
 - Larger and Complex systems, aeronautics is one example
 - One of the reasons for the delayed delivery of the Airbus A380 was the very complex (500 km!) cabling for everything from computer controls to in-flight entertainment!
 - Economy, time-to-market, etc
 - Interconnect paradigms: from busses to network based systems. "Switched fabrics"
 - Compared to a bussed system a point-to-point link topology has many advantages with respect to: physical distance, scalability, no load dependency, network topology, cost, and more
- Obviously, this is a domain where system software, computer architecture and distributed systems meet and overlap!



Front-end / Back-end topology

- A typical topology for Real-time systems is a front-end part which handles time critical tasks, such as reading sensor data with first order signal processing, interconnected with a «Back-end» computer, for instance a Linux system. Examples:
- A development board of the type used for the VHDL exercises contains a lot of facilities for Real-Time / Embedded projects:
 - On-board FPGA that can run both a soft-core processor and signal processing tasks
 - Memory of different types, Ethernet, clocks (programmable high resolution timers), connectors for clock inputs or outputs, small LCD screen, high-speed differential I/O connectors, Digital-to-Analog Converter, Analog-to-Digital Converter, switches, and you name it!
 - And the price is very low!
- A high-end Single Board Computer (SBC) like the MIDAS M5000 used in the VxWorks Workbench lab
 - This is professional stuff, and only for customers where money does not matter very much
 - It does not contain on-board AD or DA converters, if needed one can mount PMC mezzanine card onto the PCI slots (also expensive)



Two bus studies: VME and PCI bus

- The original VMEbus is a rather old design, but the VME technology is still going strong in the high-end (and high cost) market due to:
 - Steadily improved performance
 - Mechanical robustness
- PCI bus is the standard PC peripheral bus
 - A design for "plug-and-play"
- Whereas VME is a genuine multi-processor bus with possibility for sophisticated bus arbitration, PCI is what the name says: a Peripheral Component Interconnect
- Together, these two systems represent two interesting case studies
 - Ref. paper "A case for the VMEbus Architecture in Embedded Systems Education", IEEE Transactions on Education, Vol. 49, No. 3, August 2006
- After an summary of the VME and PCI bus, some other interconnects will be listed

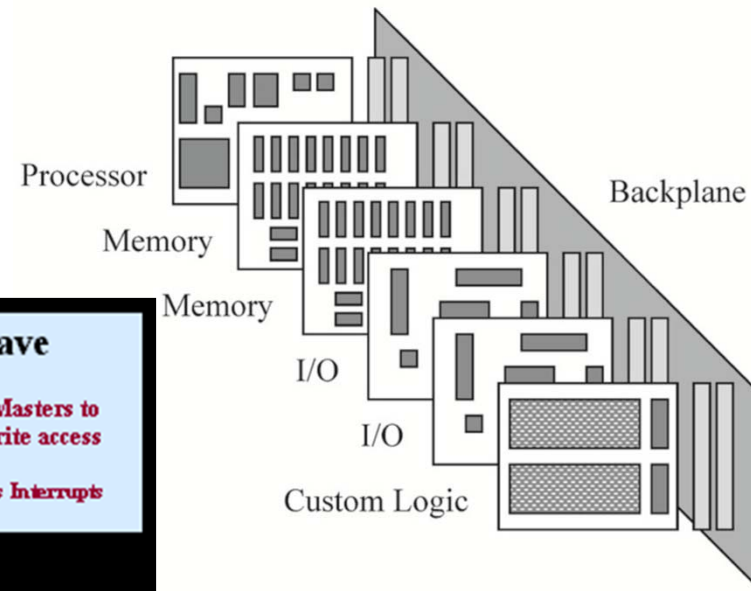


VMEbus basics vs. RT / Embedded systems

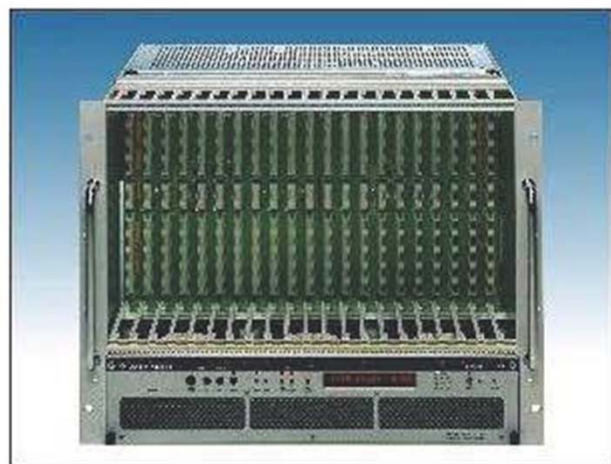
- Architecture:
 - the VMEbus is a computing system architecture consisting of the electrical specifications for a data bus and the mechanical specifications describing the backplane, bus connector, board sizes and enclosures
 - developed around 25 years ago by the companies Motorola, Mostek, Signetics and Thomson CSF as a non-proprietary bus
 - many extensions and improvements over the years, so despite its age, it is still a widely used platform for architecture for real-time systems
 - VMEbus is a shared system-bus architecture. The system bus resides on a backplane. The backplane has slots, 21 for a full 19-inch VME crate, where processor modules, memory modules or I/O modules connect to the bus
 - Many vendors provide a wide spectrum of VMEbus modules and components
- Applications:
 - for the professional market, primarily in industrial, military, aerospace, communication and control applications, in particular where robustness is required
 - however, rather expensive, and power hungry
- VITA
 - a non-profit organization for real-time and embedded computing systems
<http://www.vita.com>

VME backplane, modules, functionalities

- A VME processor module can usually be configured to incorporate all three functions: Controller, Master, Slave

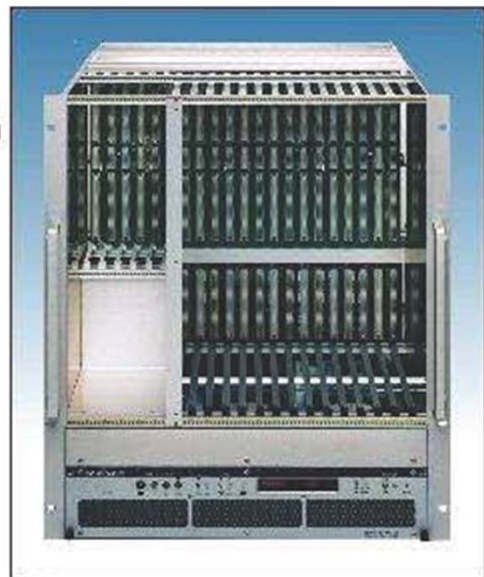


VMEbus crates



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 remote
- The fan-tray unit allows to monitor parameters like voltages, currents, fan speed, temperature
- (Some) crates can be controlled by a field bus (CAN)
- **ATTENTION:** The EMC gasket to the left of slot 1 may damage your VMEbus cards



Summary VMEbus general characteristics I

Item	Specification	Notes
Architecture	Master/slave	
Transfer Mechanism	<u>Asynchronous</u> , with both multiplexed and non-multiplexed bus cycles.	There is no central synchronization clock.
Addressing Range	16, 24, 32, 40 or 64-bit	<u>Address path width selected dynamically.</u>
Data Path Width	8, 16, 24, 32 or 64-bit	<u>Data path width selected dynamically.</u>
Unaligned Data Transfers	Yes	Compatible with most popular microprocessors.
Error Detection	Yes	Using BERR* signal.
Parity Protection	No	There are no parity signals on the backplane, but parity protected boards are quite common.
Data Transfer Rate	0 - 500+ Mbyte/sec	<u>See Table</u>
Interrupts	7 levels	Priority interrupt system with 8, 16 or 32-bit STATUS/ID (interrupt vector).



Summary VMEbus general characteristics II

Interrupts	7 levels	Priority interrupt system with 8, 16 or 32-bit STATUS/ID (interrupt vector).
Multiprocessing Capability	1 - 21 processors	Flexible bus arbitration with true peer-to-peer multiprocessing.
System Diagnostic Capability	Yes	Using SYSFAIL* signal and VME64x test & maintenance bus.
Geographical Addressing	Yes	Under VME64x
Live Insertion Capability	Yes	Using optional standards.
Control & Status Registers (Plug & Play Support)	Yes	Under VME64 & VME64x
Mechanical Standard	3U single-height Eurocard 6U double-height Eurocard 9U (optional standard)	160 x 100 mm Eurocard 160 x 233 mm Eurocard 367 x 400 mm Eurocard
User Defined I/O	Yes	Through the Front Panel and P2/J2 User Defined Pins
Conduction Cooled Version (Military)	Yes	Under IEEE 1101.2
Maximum Number of Card Slots in Backplane	21	The number of cards is limited by how many boards, located on 0.8" centers, can be placed into a 19" rack panel.

VMEbus basics

- 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 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 word at the lowest byte address (0x0)
 - PCI and Intel CPUs are little endian. They store the most significant byte of a word at the highest byte address (0x3)
 - Most VMEbus masters (e.g. VP110) have automatic byte swapping logic



VMEbus basics(2)

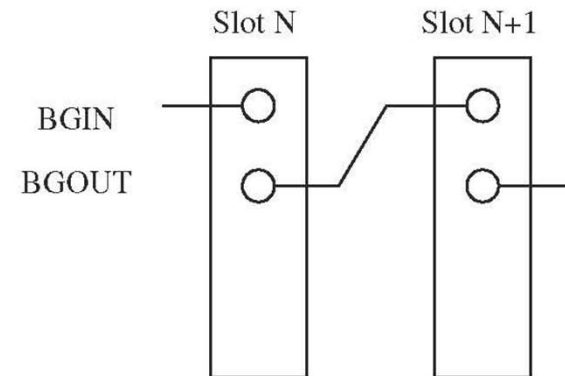
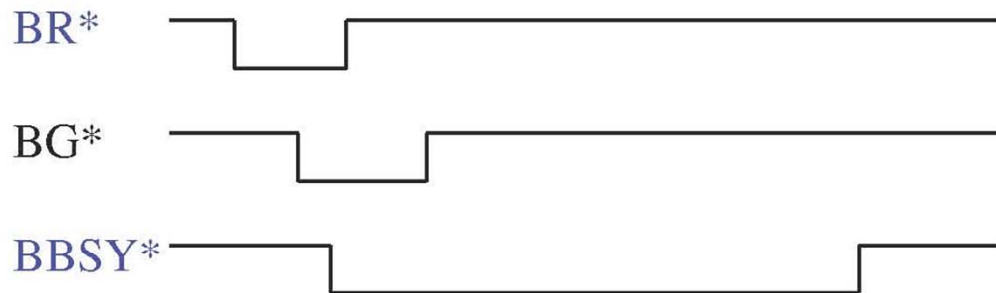
- Types of common modules (physical and logical)
 - Master
 - A module that can initiate data transfers
 - Slave
 - A module that responds to a master
 - Interrupter
 - A module that can send an interrupt (usually a slave)
 - Interrupt handler
 - A module that can receive (and handle) interrupts (usually 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

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
 - Typical duration: 1 us
 - Block transfer (DMA)
 - Transfer any amount of data (usually 32 or 64 bit at a time) under the control of a DMA controller (CPU independent)
 - 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)

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
 - The 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

VMEbus bus arbitration

- Two cases:
 - arbitrating for the bus then it is not in use
 - arbitrating when another MASTER is using it
- In both cases:
 - the new MASTER must request the bus by asserting one of the four bus request lines BR0* - BR3*, and waiting for the system controller in slot 1 to reply

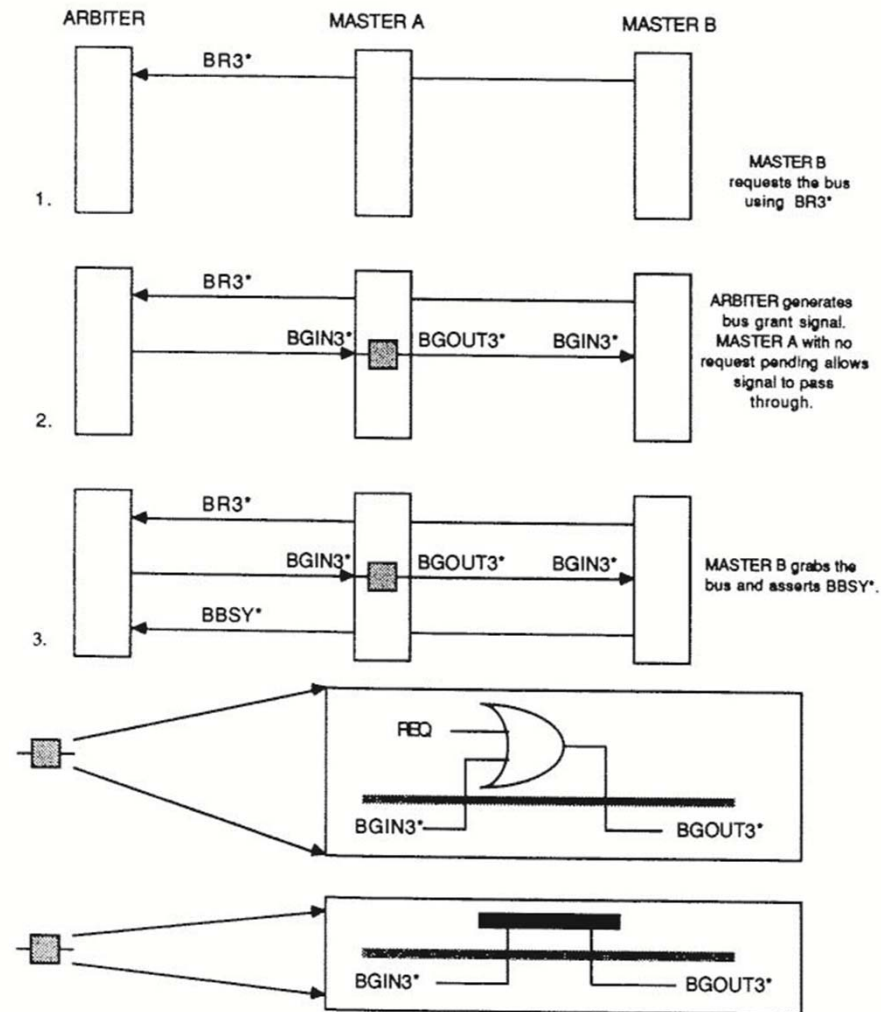
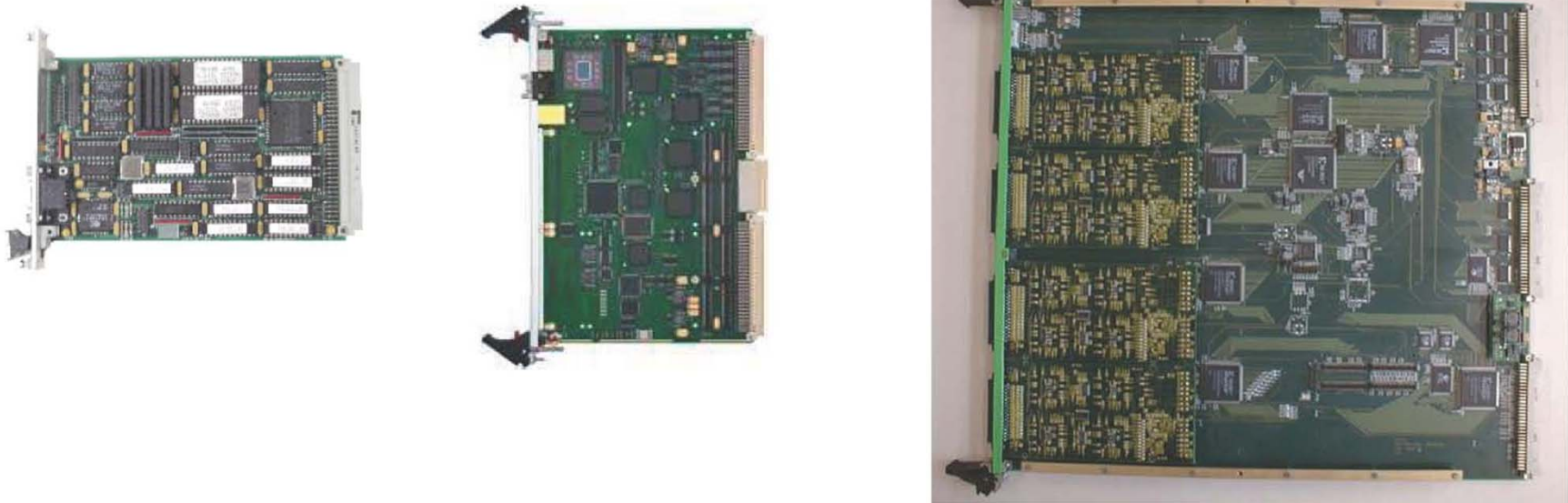


Figure 2.9 VMEbus DTB arbitration slot priority

VMEbus mechanics

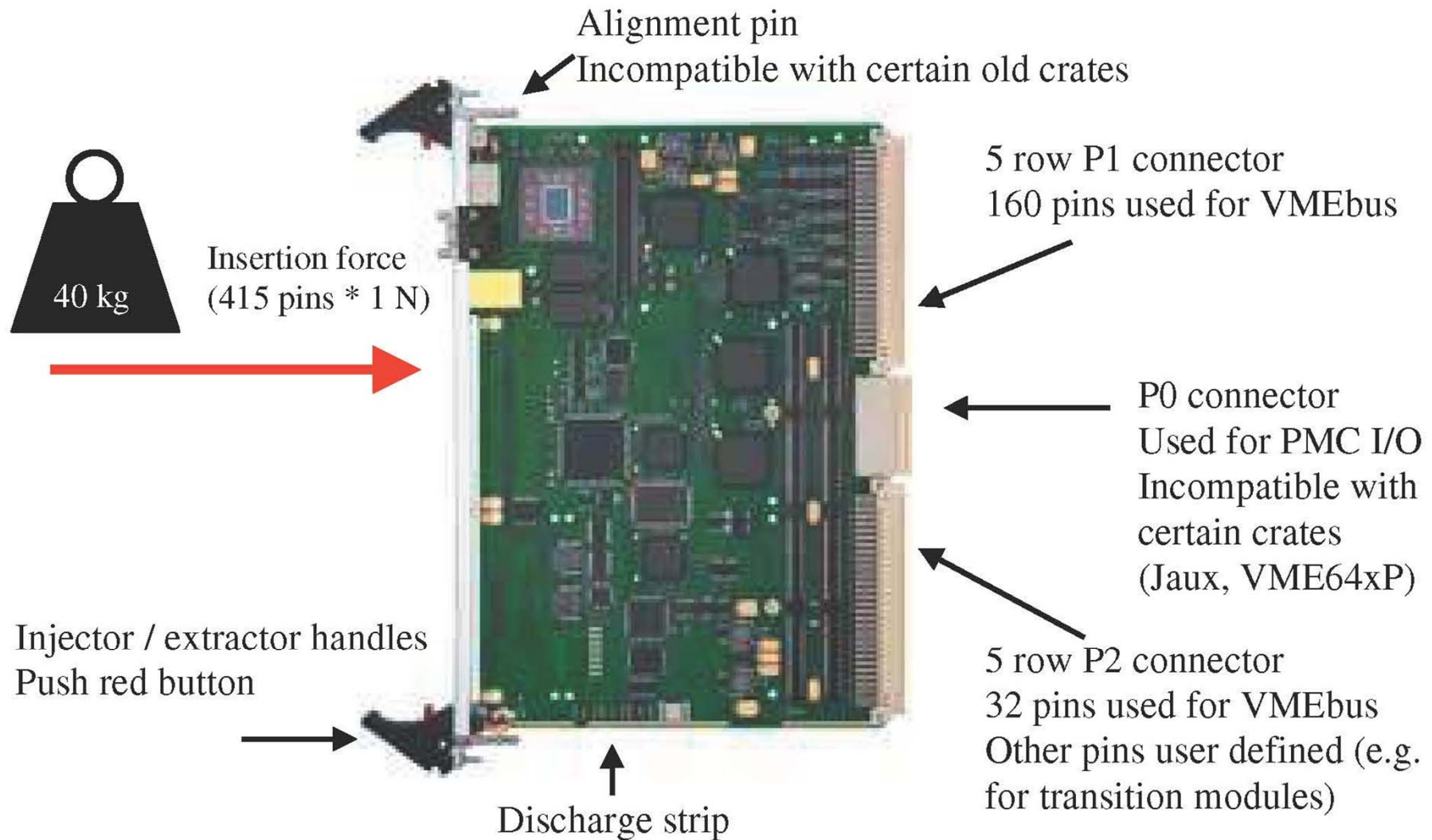
VMEbus cards exist in 3 standard heights: 3U, 6U and 9U
Definition: 1U = 1.75 inch



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



Lab target: MIDAS M5000 VME version



UNIVERSITY
OF OSLO





MIDAS M5000 VME version

1.1 Main features:

- A PowerPC processor subsystem (440GX from IBM) with up to 256MiB local DDR-SDRAM memory, 32MiB of FLASH memory, four Ethernet ports (two 10/100/1000Mbps and two 10/100Mbps) and two serial ports.
- Two standard PPMC sites located on separate PCI segments (64-bit, 33/66MHz PCI, 66/100/133MHz PCI-X). In addition, PMC#1 can be MONARCH
- Dual 2Gib Fibre Channel I/O Controller (ISP2312 from QLOGIC) (Optical)
- Single 10/100 Ethernet port (front panel connector)
- Dual serial ports configurable as two RS-232 ports or one RS-232 and one RS-422
- PCI-to-VME bridge (Universe IID from Tundra)
- Three PCI(X)-to-PCI(X) bridges which connects the different PCI(-X) segments together (PCI6540 from PLX)
- Optional PCI-to-RACEway bridge (PXB++ from Mercury)
- Optional I/O Spacer extension via a built-in connector, thus adding up to 3 Ethernet ports (One Fast Ethernet 10/100 SMII and two Gigabit Ethernet 10/100/1000 RGMII) and an I2C bus. Note: Adding more than one Gigabit Ethernet port will reduce the number of Fibre Channel ports
- Optional mezzanine extension via a built-in connector, thus adding 3 PMC sites to the Quaternary PCI segment. These PMC sites are 64-bit, 33MHz PCI, 5V signaling.
- One PIM I/O board slot, in accordance to VITA36.

PCI SIG

DOWNLOAD SPECIFICATIONS | MEMBERS LOG IN | CONTACT US

SEARCH GO SHORTCUTS About Us GO

HOME SPECIFICATIONS EVENTS DEVELOPERS MEMBERSHIP NEWSROOM

Home > Specifications > PCI Conventional

Conventional PCI 3.0 & 2.3: An Evolution of the Conventional PCI Local Bus Specification

Members-only download:

- [Conventional PCI 3.0](#) (3.3MB PDF)
- [Conventional PCI 3.0 with Change Bar](#) (4.4MB PDF)
- [Change Summary - PCI 2.3 to 3.0](#) (9k PDF)
- [Conventional PCI 2.3](#) (4.3MB PDF)
- [Summary of changes from PCI 2.2 to PCI 2.3](#) (16k PDF)

Members and non-members download:

- [PCI 3.0 Compliance Checklist](#)
- [PCI 2.3 Compliance Checklist](#)
- [Interrupt Line Register Usage ECN](#) (64k PDF)
- [Generic Capability Structure for SATA Host Bus Adapters Draft ECN](#) (174k PDF)
- [PCI/PCI-X Connector Contact Finish Changes ECN](#) (165k PDF)
- [Errata against PCI Conventional 2.3](#) (6k PDF)
- [SATA Class Code ECN](#) (117k PDF)
- [MSI-X ECN Against PCI Conventional 2.3](#) (340k PDF)
- [Conventional PCI 3.0 FAQs](#) (90k PDF)
- [Conventional PCI Advanced Caps ECN](#) (76.9k PDF)

Conventional PCI Revision 3.0

Revision 3.0 completes the evolutionary migration plan for the PCI Local Bus Specification, migrating the PCI bus from the original 5.0V signaling to a 3.3V signaling bus. Revision 2.3 began that evolution by removing support for the 5V keyed add-in card, while providing support for both the 5V and 3.3V keyed system board connectors. Revision 3.0 takes the next step of removing support for the 5V keyed system board connector. Revision 3.0 continues support for the 3.3V keyed system board connector, which supports the 3.3V and Universal keyed add-in cards. PCI 66, PCI-X, Mini PCI, and Low Profile PCI also support only 3.3 volt signaling on 3.3V keyed system board connectors and 3.3V and Universal keyed add-in cards.

OTHER CONVENTIONAL PCI SPECS

Download the Specifications

- [Mini PCI](#)
- [PCI to PCI Bridge Architecture](#)
- [PCI Hot Plug Specifications](#)
- [PCI Bus Power Management Interface 1.2](#)
- [PCI Mobile Design Guide 1.1](#)
- [PCI Firmware 3.0](#)

<http://www.pcisig.com>

Over to
PCIbus



PCI bus basics

- The PCI bus was originally developed by Intel. PCI stands for Peripheral Component Interconnect
- PCI exists in many flavours: PCI Conventional (i.e. the standard PC card version) 32 bit 33 MHz (max 132 MB/s), 64 bit 33 MHz (max 264 MB/s), 64 bit 66 MHz (max 528 MB/s), PCI Express (PCIe) and Compact PCI
- PCI is a **synchronous** bus where Address and Data are time multiplexed on the same lines
- PCI has been through several technical revisions, current PCI Conventional is 3.0
- Complete PCI specifications are available from <http://www.pcisig.com> : the homepage for the PCI Special Interest Group (if you are a member!)



Reflected wave switching

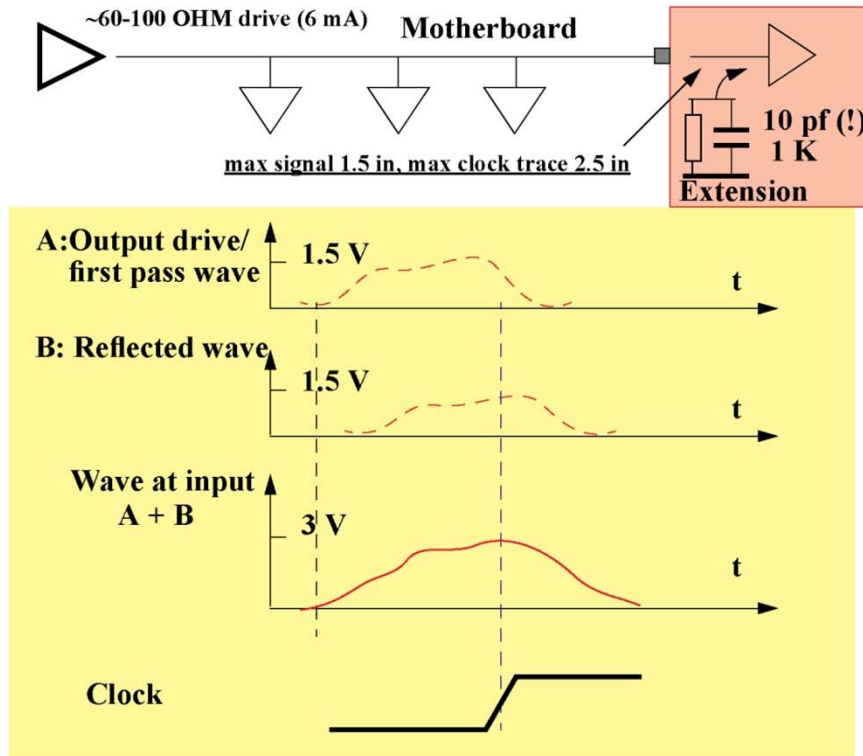
- In high-frequency environments such as PCI, conventional incident-wave switching on a terminated bus using drivers with large driving capability would create a number of problems. As such frequencies each trace (bus line) will act as a transmission line, and the electrical characteristics of the trace must also be taken into account when selecting the output driver.
 - Using strong drivers to switch (by brute force) a bus system at high frequency will present a number of problems, such as: i) very difficult to decouple, ii) spikes, increase EMI (electromagnetic interference) and iv) crosstalk.
- The PCI bus is a low power “green” bus, exploiting the reflection of a signal on an unterminated line. The PCI bus is unterminated and uses wavefront reflection to an advantage. A relatively weak output driver drives the signal halfway to the desired logic state, say 1.5V. When the wavefront arrives at the unterminated end of the bus, it is reflected back and doubled (3V)!
 - The drawback of this method is that the maximum length of a PCI bus can not exceed around 15cm, i.e. for 4 cards. If a longer bus is needed then it must be built from segments interconnected by PCI bridges.



CMOS reflected wave technology

PCI was initially intended as CMOS interchip bus without extension

**Weak Voltage drivers & unterminated lines !
less spike/decoupling problems**



Consequences of critical timing reflected wave technology:

No DC current since no termination resistors

**Input clamp diodes in devices needed
limited total signal trace length: few inch**

**high imp. PCB (thin traces far from GND)
low capacitance packages (plastic)**

Hans Muller CERN ECP

The reflected wave concept requires a fixed and short length of a signal trace. Maximum number of cards on a PCIbus is 4.

PCI signals

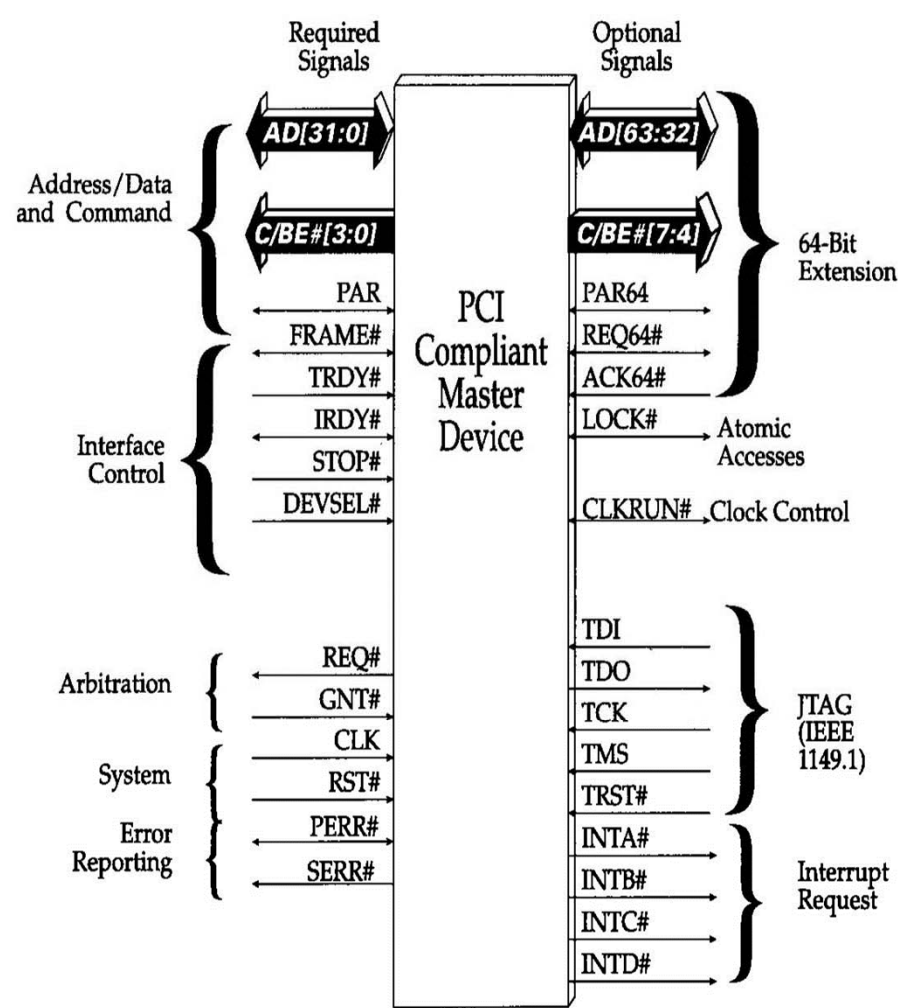


Figure 5-1. PCI-Compliant Master Device Signals

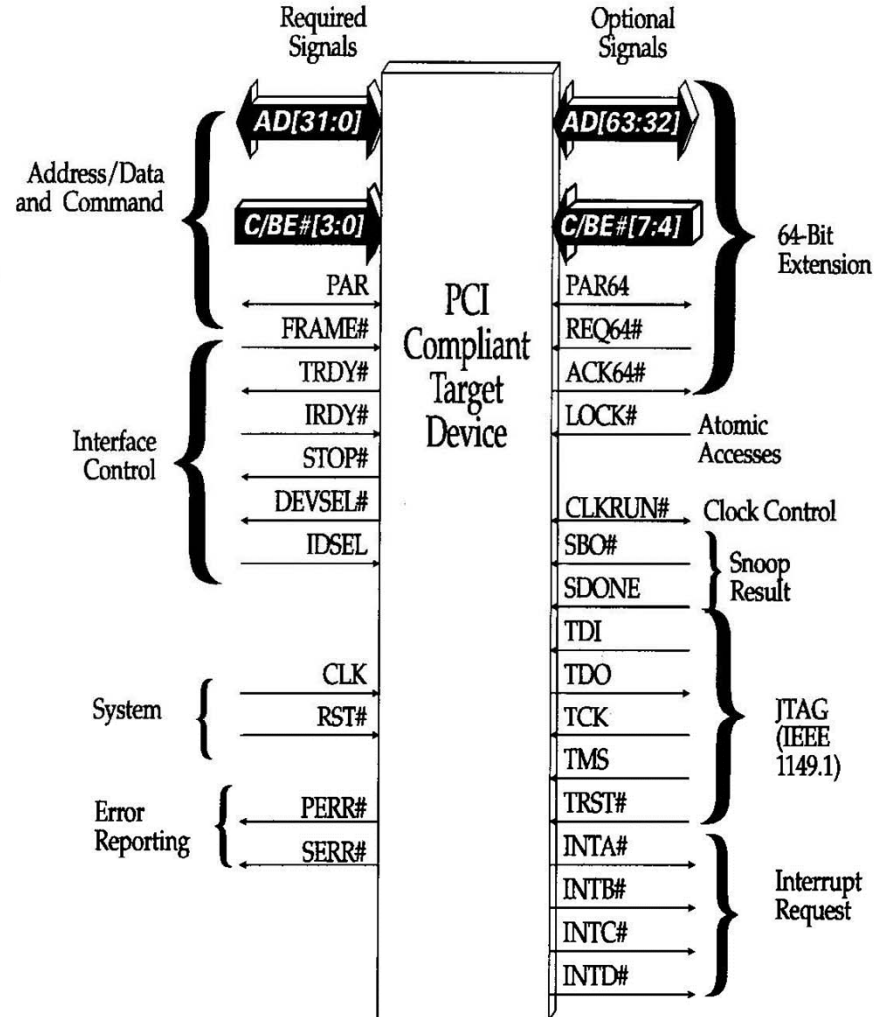


Figure 5-2. PCI-Compliant Target Device Signals



PCI Burst transfers

- A burst transfer is one consisting of a single address phase followed by two or more data phases. The bus master only has to arbitrate for bus ownership one time. The start address and transaction type are issued during the address phase. The target device latches the start address into an address counter and is responsible for incrementing the address from data phase to data phase.

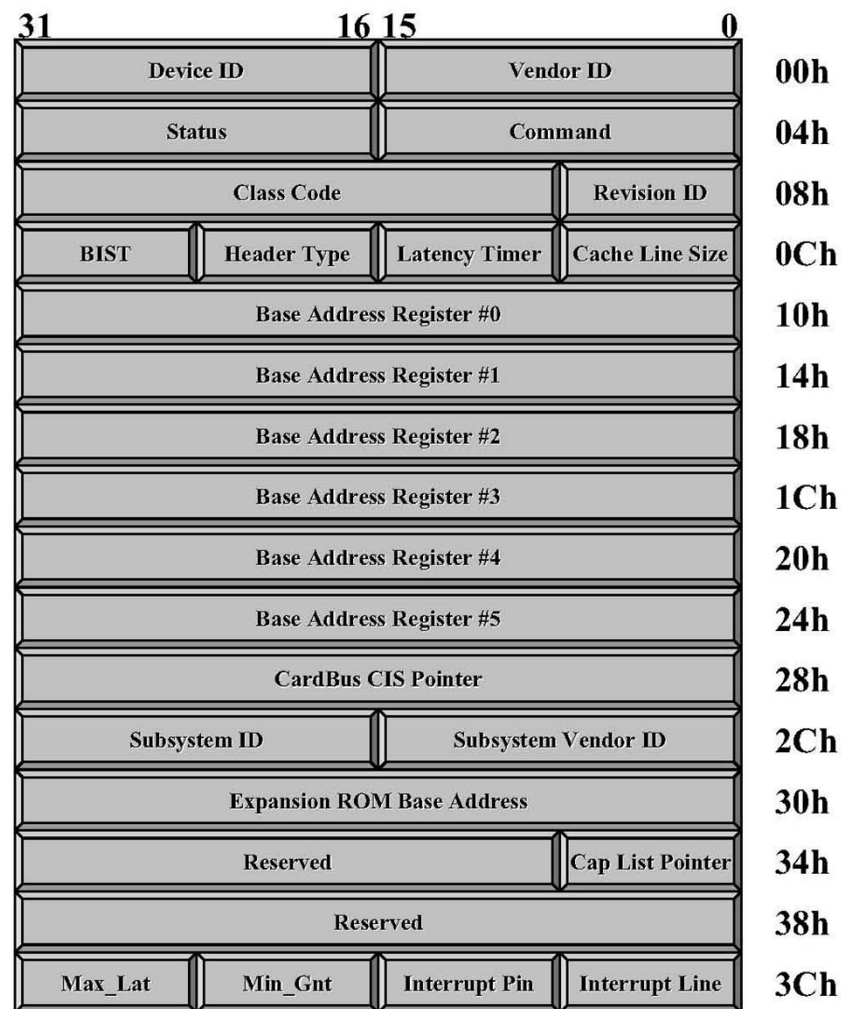


Configuration registers

- PCI bus is a "plug and play" system!
- A PCI device is described by the content of its Configuration space
- Each functional PCI device possesses a block of 64 configuration doublewords (32-bit) reserved for its configuration registers.

The Plug-and-Play Concept

- ◆ To make PNP possible in PCI, each PCI device maintains a 256-byte configuration space
 - The first 64 bytes (shown here) are predefined in the PCI spec and contain standard information
 - The upper 192 bytes may be used to store device-specific information



PCI Commands

- ◆ **PCI allows the use of up to 16 different 4-bit commands**
 - *Configuration commands*
 - *Memory commands*
 - *I/O commands*
 - *Special-purpose commands*

- ◆ **A command is presented on the C/BE# bus by the initiator during an address phase (a transaction's first assertion of FRAME#)**

PCI Commands

C/BE#	Command	
0000	Interrupt Acknowledge	
0001	Special Cycle	
0010	I/O Read	Memory
0011	I/O Write	I/O
0100	Reserved	
0101	Reserved	
0110	Memory Read	Configuration
0111	Memory Write	Special-Purpose
1000	Reserved	
1001	Reserved	Reserved
With IDSEL	1010	Configuration Read
With IDSEL	1011	Configuration Write
	1100	Memory Read Multiple
	1101	Dual Address Cycle
	1110	Memory Read Line
	1111	Memory Write and Invalidate



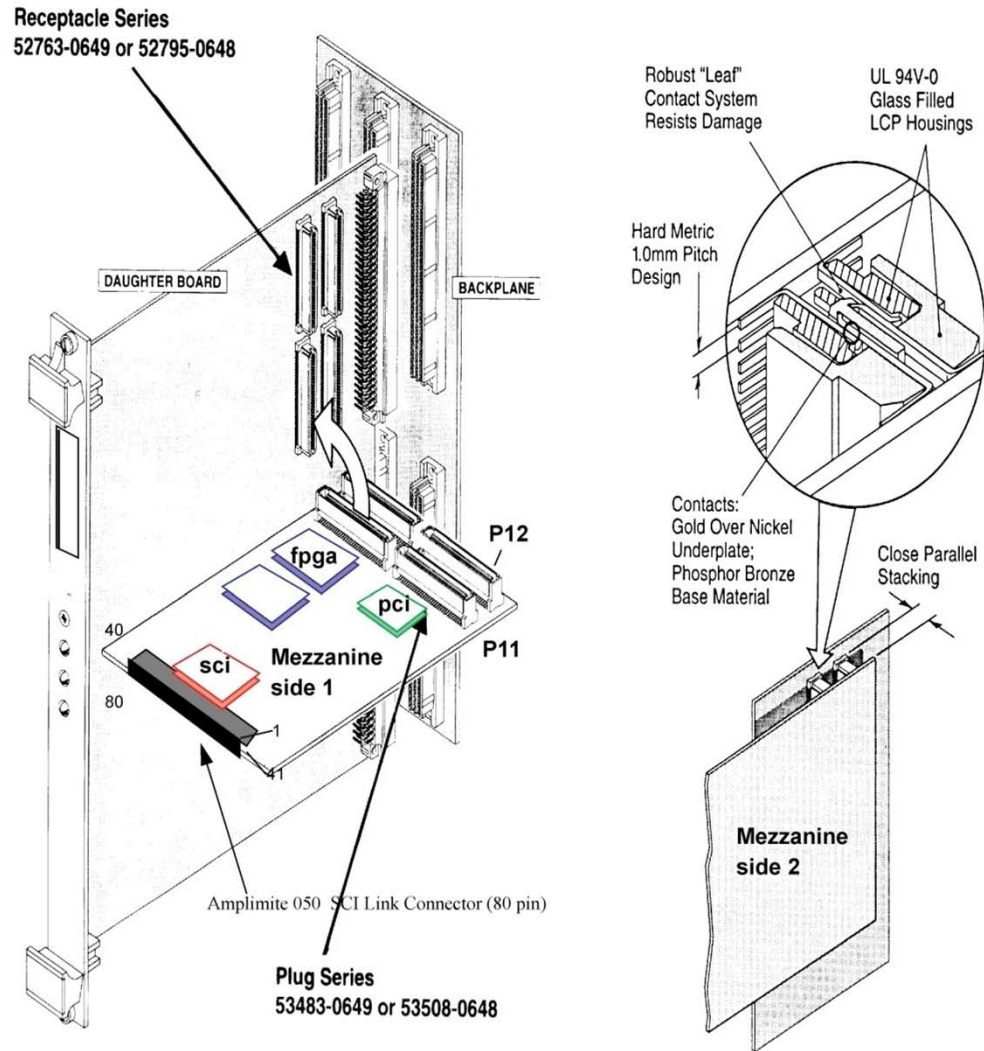
More from the PCI zoo

- Common Mezzanine Cards (CMC) and PCI Mezzanine (PMC) standard, used on VME cards
- Compact PCI
- PCIe (PCI express)

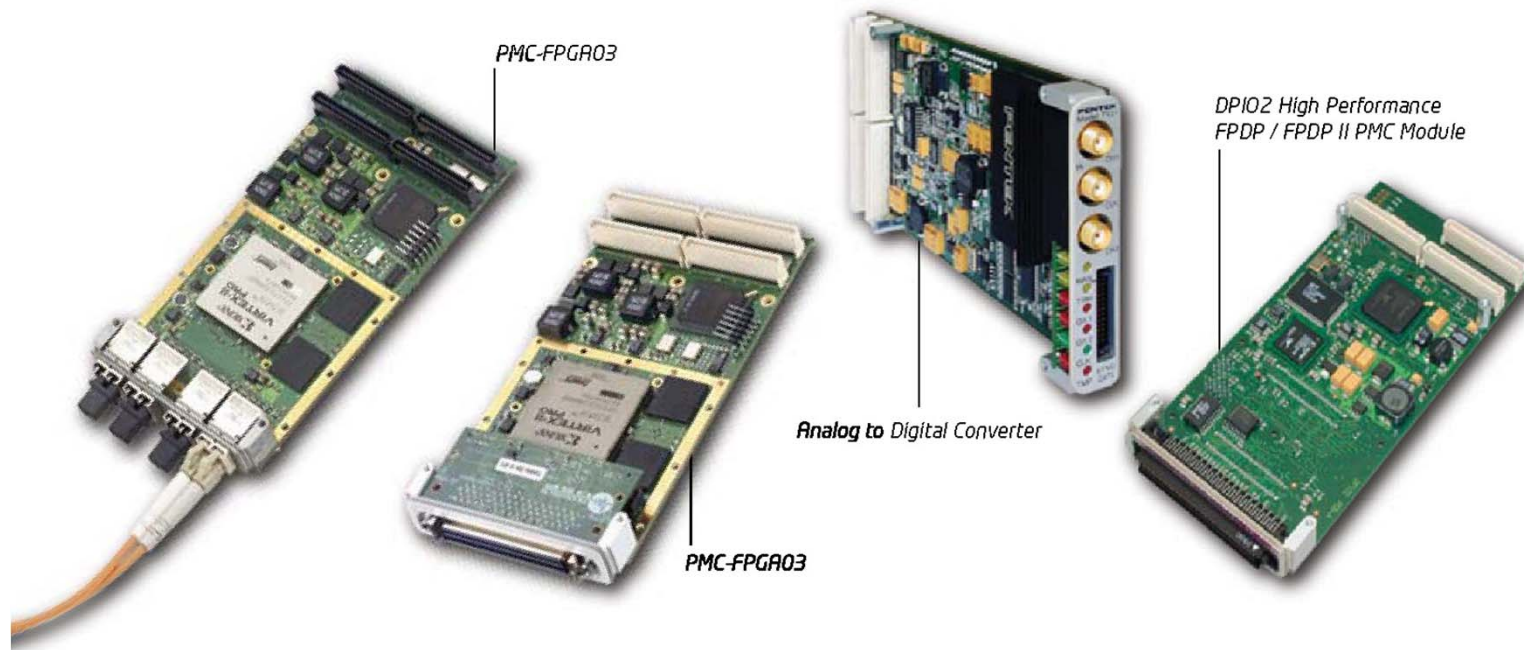


Common Mezzanine Card Environment

for VMEbus, Futurebus, ...
(PMC = physical PCI Mezzanine Card)



Examples of PMCs



PowerMIDAS C5000 used in a System Slot

System slot functionality is designed into the PowerMIDAS C5000 board with small systems in mind. The typical target application is a data recorder system consisting of one (or more) PowerMIDAS C5000 board(s) and one (or more) CompactPCI A/D converter boards, with the backplane bus being used for initialization, setup and synchronization. Such configurations do not require a separate system slot board.

PMC - the PCI Mezzanine card

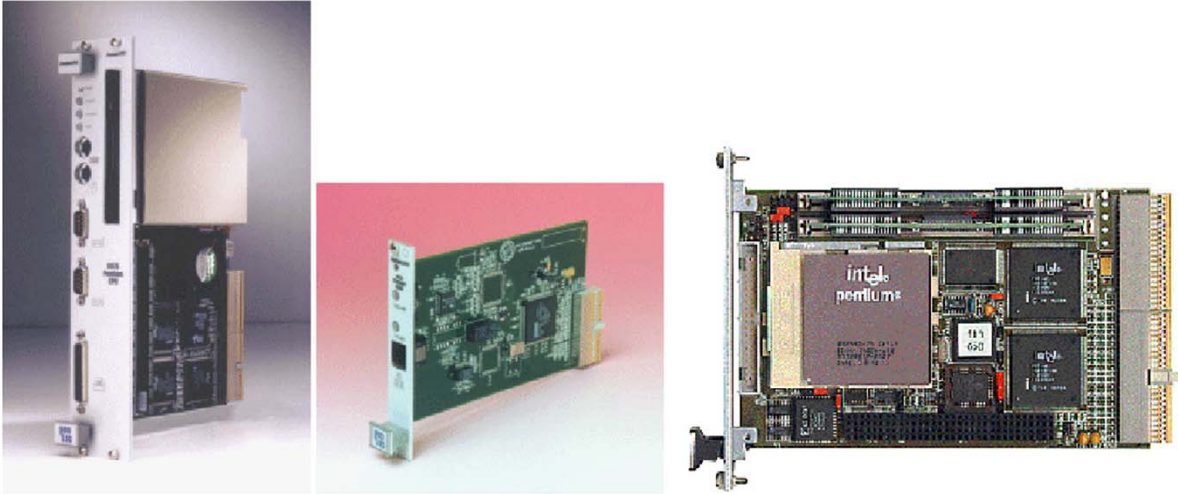
PMCs replace entire VME boards

Due to the high level of integration offered by PCI ASICs typically found on PMC modules, a single PMC module may in many cases replace an entire VME or CompactPCI board. Traditionally, functions like I/O, memory and special processors like DSPs have occupied a full board, but now these functions may take only a neat PMC module to implement.

CompactPCI

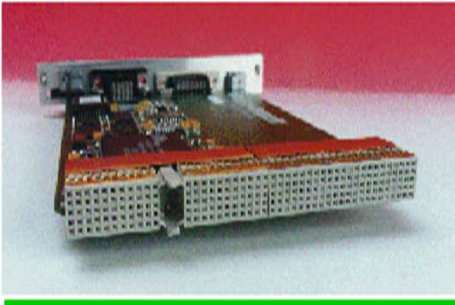
Introduction

The newest standard for PCI-based industrial computers is called CompactPCI. It is electrically a superset of desktop PCI with a different physical form factor. CompactPCI utilizes the form factor popularized by the VME bus.



Defined for both 3U (100mm by 160 mm) and 6U (160mm by 233 mm) card sizes, CompactPCI has the following features:

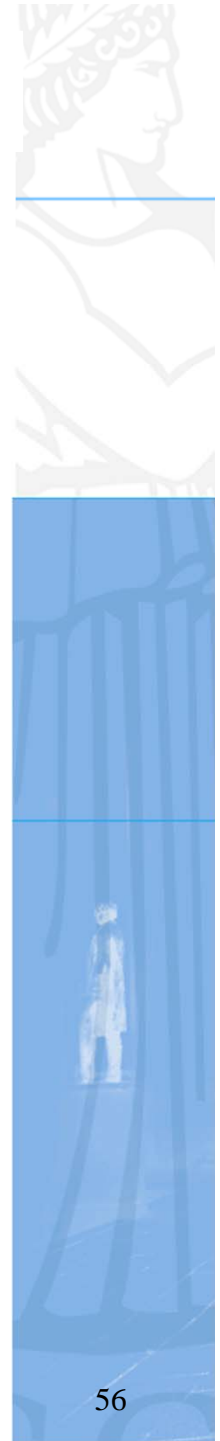
- Standard Eurocard Dimensions (complies with IEEE 1101.1 mechanical standards)
- High Density 2mm Pin-and-Socket Connectors IEC approved and Bellcore qualified)
- Vertical Card Orientation for good cooling
- Positive Card Retention
- Excellent Shock and Vibration Characteristics
- Metal Front Panel
- User I/O Connections on Front or Rear of module
- Standard Chassis available from many Suppliers
- Uses Standard PCI Silicon Manufactured in Large Volumes
- Staged Power Pins for Hot Swap Capability (Future)
- Eight Slots in Basic Configuration. Easily expanded with Bridge Chips



The CompactPCI Connector



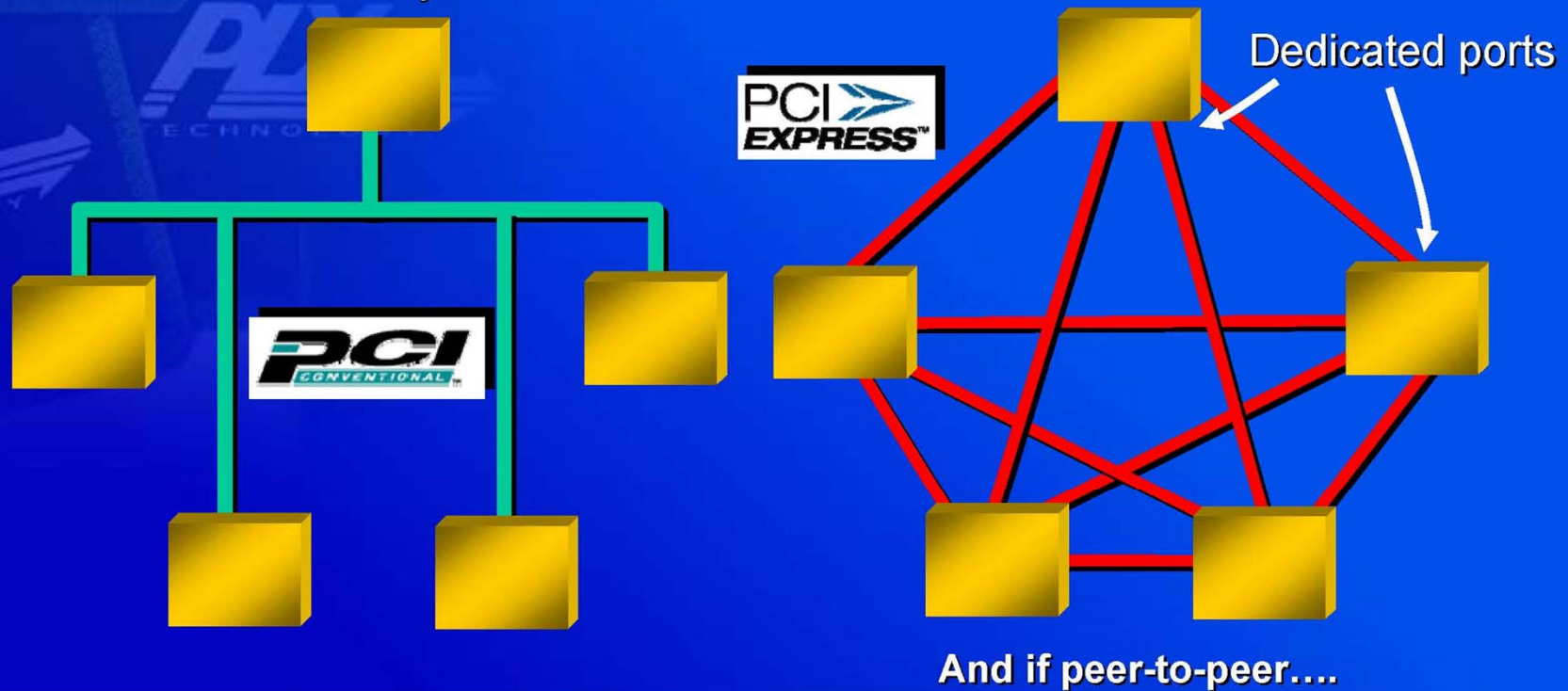
UNIVERSITY
OF OSLO



PCI EXPRESS

PCI vs. PCI Express

- Unlike PCI, PCI Express is point-to-point
 - This requires each native PCI Express device to include a dedicated port for every other device it must communicate with in the system



PCI Express Performance (Gen 1)

Link Width	X1	X2	X4	X8	X12	X16	X32
Bandwidth in Gbits/s (raw, aggregate)	5	10	20	40	60	80	160
Bandwidth in GB/s (raw, aggregate)	.625	1.250	2.500	5.000	7.500	10	20
Throughput in GB/s (aggregate)	.5	1	2	4	6	8	16
Throughput in GB/s (per direction)	.25	.5	1	2	3	4	8



PCI 32/66

PCI or PCI-X 64/66

PCI-X 64/133

Make the RapidIO[®] Connection



RapidIO in Critical Embedded Systems

Jan 2007

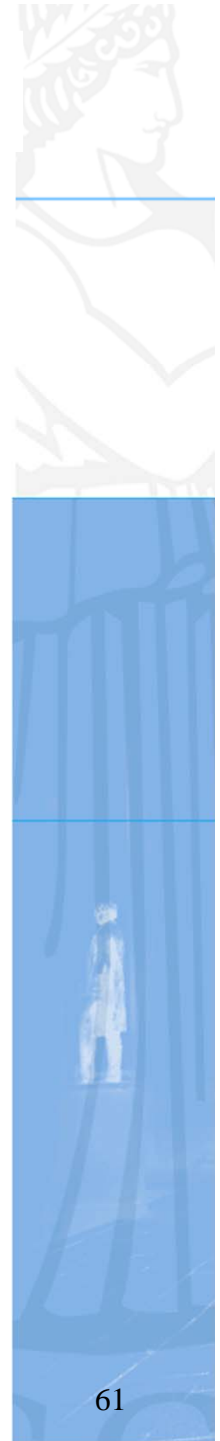


The Embedded Fabric Choice

Critical Embedded Systems

REQUIRE -System-Level Fault Tolerance

- There are six key elements to system-level fault tolerance:
 - No single point of failure
 - No single point of repair
 - Fault recovery
 - 100% fault detection
 - 100% fault isolation
 - Fault containment
- RapidIO supports all of these critical System-Level Fault Tolerance elements!



INTERCONNECT MATRIX

