FYS 4220 – 2011 / #10
Real Time and Embedded Data Systems and Computing
Instrumentation, interconnects and I/O
Embedded / Real-Time systems and the real world

- An embedded system is useless if it cannot communicate with the external world!
- Some characteristics
  - Hardware
    - Bus systems (VME, PCI, etc)
    - Serial links, from RS-232 to RapidIO and others
    - Networks for loosely and tightly coupled distributed systems
    - Topologies
  - Protocols
    - Encoding, error detection and recovery
    - Packet structure
  - Software
    - Standard (serial) I/O
    - Block I/O, DMA
    - Asynchronous (non-blocking I/O)
Instrumentation

• These includes both the measuring components and the interface to those
  – Basics measuring components which connect to external instrumentation are:
    • Analog input => digital conversion
    • Analog output <= digital conversion
    • Digital IO
    • Counter/Timer
  – A large number of commercial products (COTS = commercial off-the-shelf) are available, and if it does not exist one has to make it (Custom design, much more fun)!
• Bus systems: we have previously presented the main characteristics of VMEbus, PCI and other interconnects.
Communication between devices

• There are three basic device classes – *Controllers*, *Sensors* and *Actuators*.
• We need to transfer a information from the sensors to the controller and from the controller to the actuators.
• For that, we use some transfer medium (wires, fiber, air..)
• And in order to interpret the information, we need a set of *protocols*

Ref. Svein Johannesen, KODE
Why use protocols at all?

• Even a “perfect” hardware solution may need some help since:
  – Almost all communication solutions have frame size limitations
  – Flow control may be necessary
  – Communication errors may occur
  – Source and destination may be on different hardware standards

• There are two basic flavors of communications: Peer-to-peer and Master/slave
  – Peer-to-peer is the democratic “everybody has a right to speak” model
  – Master/slave is the dictatorial “only speak when spoken to” model
  – Both models have their uses..

Ref. Svein Johannesen, KODE
Some A/D and D/A circuits, DSPs

- National Instruments

- For an overview of High Speed Sampling ADCs, see lecture note 2011-10_Sampling_ADCs.pdf

- The ALTRO ASIC developed for the CERN ALICE experiment is an example of a sampling ADC with programmable preprocessing and buffer storage. The 10-bit ADC has a maximum sampling rate of 20 MHz. Each ALTRO chip contains 16 ADCs. See following pages.

- Digital Signal Processor (DSP)
  - The architecture of a digital signal processor is optimized specifically for digital signal processing. See for instance
**Architecture and Main Components**

- **drift region**: L1: 6.5μs, 1 KHz
- **anode wire**: 88μs
- **pad**: 570132 pads

**PASA**

- 8 CHIPS x 16 CH / CHIP
- CUSTOM IC (CMOS 0.35μm)

**ADC**

- CUSTOM IC (CMOS 0.25μm)
- 10 BIT
- < 12 MHz
- GAIN = 12 mV / fC
- FWHM = 190 ns
- *GAIN EQUALIZ.*
- *LINEARIZATION*
- *BASELINE CORR.*
- *TAIL CANCELL.*
- *ZERO SUPPR.*

**Digital Circuit**

**RAM**

**FEC (Front End Card) - 128 CHANNELS (CLOSE TO THE READOUT PLANE)**

- 8 CHIPS x 16 CH / CHIP
- custom IC (CMOS 0.25μm)

**FEC (Front End Card) - 128 CHANNELS (CLOSE TO THE READOUT PLANE)**

- CUSTOM IC (CMOS 0.25μm)
- 10 BIT
- < 12 MHz
- GAIN = 12 mV / fC
- FWHM = 190 ns
- *GAIN EQUALIZ.*
- *LINEARIZATION*
- *BASELINE CORR.*
- *TAIL CANCELL.*
- *ZERO SUPPR.*

**MULTI-EVENT MEMORY**

- Power consumption: < 40 mW / channel

**ALTRO**

- **DDL (3200 CH / DDL)**
- L2: < 100 μs, 200 Hz

**DETECTOR**

- 1 MIP = 4.8 fC
- S/N = 30 : 1
- DYNAMIC = 30 MIP

- CSA
  - SEMI-GAUSS. SHAPER
  - GAIN = 12 mV / fC
  - FWHM = 190 ns

**FEE FOR THE NA49 AND STAR TPCs**

- analog memory in front of the ADC ⇒ readout time independent of the occupancy
- no zero suppression in the FEE ⇒ high data throughput on the detector data links
PRE-AMPLIFIER SHAPING AMPLIFIER (PASA)

- Gain: 12mV / fC
- FWHM: 190ns
- Noise: < 1000

REQUIREMENTS

- INL: < 0.3%
- Crosstalk: < 0.1%
- Power: < 20mW / ch
IMPULSE RESPONSE FUNCTION

FWHM = 190 ns

$Q = 149 \text{ fC}$

$A(t / \tau)^4 e^{-4(t/\tau)}$
**Input Signal**

- **ADC**
  - Baseline Correction I
  - Baseline Memory
  - Tail Cancel. Filter
  - Baseline Correction II
  - Zero Supression
  - Data Format

**Data Processor**

- **Correction of:**
  - Slow drifts and systematic effects
  - Non-systematic effects
  - Tail filtering
  - Data compression

**Memory**

- 40-bit back linked format
  - Channel address
  - Time stamp

- 5 kbyte
- 4 or 8 buffers

**Data Compression**

- 40-bit back linked format
  - Channel address
  - Time stamp

**TSA1001**

- 10-bit
- 25 MSPS
- 40 MSPS

**Trigger signals**

- L1: acquisition
- L2: event freeze

**Readout bus**

- 40-bit wide bus
- Bandwidth: 300 Mbyte/s

**Common Control Logic**

- Runs with Sampling Clock
- Runs with Readout Clock

T.B. Skaali, Department of Physics, University of Oslo
Baseline drift

Systematic perturbation

Baseline correction

Fixed pedestal

Slow drifts

Systematic perturbation

Combination

fpd = 0
Functions
- signal (ion) tail suppression
- pulse narrowing \(\Rightarrow\) improves cluster separation
- gain equalization

Architecture
- 3\textsuperscript{rd} order IIR filter
- 18-bit fixed point 2’sC arithmetic
- single channel configuration \(\Rightarrow\) 6 coefficients / channel

Diagram:
- 11 bits
- word extension
- 18 bits
- 3\textsuperscript{rd} order IIR filter
- 18 bits
- word rounding
- 11 bits
- Input
- 11 bit
- Output
- 11 bit
- 18-bit fixed point arithmetic
Characteristics:
* Corrects non-systematic perturbations during the processing time
* Moving Average Filter (MAF)
* Double threshold scheme (acceptance window)

Operation

1. Slow variations of the signal ⇒ Baseline updated
2. Fast variations of the signal ⇒ Baseline value frozen

Unsigned 11-bit FIR system

\[ bsl = \frac{1}{8} \sum_{n} din_n \]

After Tail Cancellation Filter

Double threshold

After Baseline Correction II

A fixed threshold can now be applied safely

\[ \begin{align*}
\text{for } & \text{dout < 0:} \\
& \text{dout = } (\text{din - bsl + offset}) - 1024 \\
\text{for } & \text{dout > 1023:} \\
& \text{dout = } (\text{din - bsl + offset}) + 1024 \\
\end{align*} \]
Zero Suppression Operation

above-threshold samples
pre-samples
post-samples
fill-in samples
rejected glitches
discarded glitches
dismissed samples

adjointed pre and post samples
merged clusters
NETWORKS / INTERCONNECTS
The Ethernet connection

- Ethernet is used more and more as interconnect in Real-Time systems
  - High speed and low latency with Gigabit and 10 Gigabit Ethernet
    - Very cheap, very simple to install
    - However, a WARNING. The IP (Internet Protocol) does not provide a 100% reliable communication facility as such. There are no acknowledgments either end-to-end or hop-by-hop. There is no error control for data, only a header checksum. No retransmissions. No flowcontrol (RFC 791).
  - Standard software, protocols TCP/IP and UDP.
    - The Media Access Control address (MAC address) is a unique identifier assigned to most network adapters.
    - In VxWorks the basis for intertask communication across a network is a socket. When a socket is created the protocol is specified.
    - TCP provides reliable, guaranteed, two-way transmission of data with stream sockets. TCP is often referred to as a virtual circuit protocol.
    - UDP provides a simpler but less robust form of communication. In UDP communications, data is sent between sockets in separate, unconnected, individually addressed packets called datagrams.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Start-of-Frame-Delimiter</th>
<th>MAC destination</th>
<th>MAC source</th>
<th>Ethertype/Lengde</th>
<th>Payload (nyttelast)</th>
<th>Integritetssjekk (CRC32)</th>
<th>Interframe gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 oktetter på 10101010</td>
<td>1 oktett på 10101011</td>
<td>6 oktetter</td>
<td>6 oktetter</td>
<td>2 oktetter</td>
<td>46-1500 oktter</td>
<td>4 oktetter</td>
<td>960 ns Fast Ethernet(100M))</td>
</tr>
<tr>
<td>64-1518 oktetter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72-1526 oktetter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Protocols – error detection - encoding

• Transmission on external media is subject to interference with possible corruption of data.
  – Each packet therefore contains a error detection element, like the CyclicRedundancyCheck in an Ethernet packet. An n-bit CRC, applied to a data block of arbitrary length, will detect any single error burst not longer than n bits (in other words, any single alteration that spans no more than n bits of the data), and will detect a fraction $1 - 2^{-n}$ of all longer error bursts. Errors in both data transmission channels and magnetic storage media tend to be distributed non-randomly (i.e. are "bursty"), making CRCs' properties more useful than alternative schemes such as multiple parity checks.
  – Encoding:
    • In order to transport digital bits of data across carrier waves, encoding techniques have been developed each with their own pros and cons.
      – In the well known 8/10 encoding each byte of data is examined and assigned a 10 bit code group. The 10 bit code groups must either contain five ones and five zeros, or four ones and six zeros, or six ones and four zeros. This ensures that not too many consecutive ones and zeros occurs between code groups thereby maintaining clock synchronisation.
      – In order to maintain a DC balance, a calculation called the **Running Disparity** calculation is used to try to keep the number of '0's transmitted the same as the number of '1's transmitted.
Tightly coupled systems (a computer architecture issue)

- In a tightly coupled system the interconnected processors can access a distributed, shared memory
  - Can be implemented over a common bus, but a bus represents a serious bottleneck
  - An example of a high-speed interconnect standard for shared memory multiprocessing and message passing is IEEE 1596 Scalable Coherent Interface (1995).
    - The goal was to create an interconnect that would scale well, provide system-wide coherency and a simple interface; i.e. a standard to replace buses in multiprocessor systems without the inherent scalability and performance limitations of buses. The working group soon realized that any form of buses would not suffice and came up with the idea of using point-to-point communication in the form of insertion rings as the right way to go. This approach avoids the lumped capacitance, limited physical length/speed of light problems and stub reflections in addition to allowing parallel transactions
- SCI shared memory access, nothing could be simpler

```c
/* map memory segment on remote node to local memory window and access remote memory as local memory. «ping-pong» latency less than 2 microsec
remoteBuffer = (volatile unsigned int *)remoteSegmentAddr;
for (j=0; j<nostores; j++) {remoteBuffer[j] = localBuffer[j]; }
```
I/O SOFTWARE INTERFACES
Headlines

• Non-blocking (Asynchronous) I/O
• Non-blocking POSIX message queues
• Direct Memory Access (DMA) transfer
Asynchronous (non blocking) I/O

- Asynchronous I/O (or non-blocking I/O), is a form of input/output transaction that permits other activities to continue while the transmission is in progress.
  - The alternative: synchronous I/O (or blocking I/O) would leave the process/task in a waiting state until the I/O terminates. As such it will leave system resources idle. When a process makes many I/O operations, this means that the processor can spend almost all of its time idle waiting for I/O operations to complete.
  - The benefit of AIO is greater processing efficiency: it permits I/O operations to take place whenever resources are available, rather than making them await arbitrary events such as the completion of independent operations. AIO eliminates some of the unnecessary blocking of tasks that is caused by ordinary synchronous I/O; this decreases contention for resources between input/output and internal processing, and expedites throughput.
Asynchronous I/O – OS support

• Operating systems implement asynchronous I/O at many levels
  – Implementations may support (from Wikipedia, no attempt to explain them further!) **Polling, Select loops, Signals, Callback functions, Completions queues, Event flags**
    • Spooling was one of the first forms of multitasking designed to exploit the concept of asynchronous I/O

• Oracle note 12 Oct 2007:
  – The performance of asynchronous I/O is heavily dependent on the operating system's implementation of the `aio_read()` and `aio_write()` system calls. **Kernelized asynchronous I/O** is greatly preferable to **threaded asynchronous I/O** but it is only available for raw devices and Quick I/O files.
    • **Kernelized** asynchronous I/O (KAIO): the kernel includes specific support for asynchronous I/O, whereas the **Threaded** implementation of asynchronous I/O just uses the kernel's light-weight process functionality to simulate asynchronous I/O by performing multiple synchronous I/O requests in distinct threads
    • KAIO: Solaris, HP-UX, AIX, Linux expected with 2.5 (?)
    • Threaded: Solaris, AIX, Linux, but **not** HP-UX
Asynchronous I/O - VxWorks

• The VxWorks AIO implementation meets the specification in the POSIX 1003.1b standard.

The POSIX AIO Routines

The VxWorks library `aioPxBLib` provides the POSIX AIO routines. To access a file asynchronously, open it with the `open()` routine, like any other file. Thereafter, use the file descriptor returned by `open()` in calls to the AIO routines. The POSIX AIO routines (and two associated non-POSIX routines) are listed in Table 3-4.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aioPxBLibInit()</code></td>
<td>Initialize the AIO library (non-POSIX).</td>
</tr>
<tr>
<td><code>aioShow()</code></td>
<td>Display the outstanding AIO requests (non-POSIX).</td>
</tr>
<tr>
<td><code>aio_read()</code></td>
<td>Initiate an asynchronous read operation.</td>
</tr>
<tr>
<td><code>aio_write()</code></td>
<td>Initiate an asynchronous write operation.</td>
</tr>
<tr>
<td><code>aio_listio()</code></td>
<td>Initiate a list of up to LIO_MAX asynchronous I/O requests.</td>
</tr>
<tr>
<td><code>aio_error()</code></td>
<td>Retrieve the error status of an AIO operation.</td>
</tr>
<tr>
<td><code>aio_return()</code></td>
<td>Retrieve the return status of a completed AIO operation.</td>
</tr>
<tr>
<td><code>aio_cancel()</code></td>
<td>Cancel a previously submitted AIO operation.</td>
</tr>
<tr>
<td><code>aio_suspend()</code></td>
<td>Wait until an AIO operation is done, interrupted, or timed out.</td>
</tr>
</tbody>
</table>
Asynchronous I/O - VxWorks

• Run the VxWorks demo programs, ref. VxWorks Programmers Guide. The source is also on the FYS4220 lab web
  – aioExSig.aio.c
  – aioEx.c
aioExSig.c

AIO complete signal
POSIX message queues

- Can be set up for non-blocking operation
- `mq_receive()`
  - If the message queue is empty and `O_NONBLOCK` is not set in the message queue's description, `mq_receive()` will block until a message is added to the message queue, or until it is interrupted by a signal. If more than one task is waiting to receive a message when a message arrives at an empty queue, the task of highest priority that has been waiting the longest will be selected to receive the message. If the specified message queue is empty and `O_NONBLOCK` is set in the message queue's description, no message is removed from the queue, and `mq_receive()` returns an error.

- However, this is not really AIO!
POSIX message queues - notify

• Notifying a Task that a Message is Waiting
  – A task can use the `mq_notify()` routine to request notification when a message for it arrives at an empty queue. The advantage of this is that a task can avoid blocking or polling to wait for a message.
  – The `mq_notify()` call specifies a signal to be sent to the task when a message is placed on an empty queue. This mechanism uses the POSIX data-carrying extension to signaling, which allows you, for example, to carry a queue identifier with the signal (see POSIX Queued Signals).
  – The `mq_notify()` mechanism is designed to alert the task only for new messages that are actually available. If the message queue already contains messages, no notification is sent when more messages arrive. If there is another task that is blocked on the queue with `mq_receive()`, that other task unblocks, and no notification is sent to the task registered with `mq_notify()`.
  – Notification is exclusive to a single task: each queue can register only one task for notification at a time. Once a queue has a task to notify, no attempts to register with `mq_notify()` can succeed until the notification request is satisfied or cancelled.
  – Once a queue sends notification to a task, the notification request is satisfied, and the queue has no further special relationship with that particular task; that is, the queue sends a notification signal only once per `mq_notify()` request. To arrange for one particular task to continue receiving notification signals, the best approach is to call `mq_notify()` from the same signal handler that receives the notification signals. This reinstall the notification request as soon as possible.

• Example program: `ex-2-10.c` (also on FYS4220 web)
Asynchronous I/O - DMA

• A *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk. In VxWorks, the term *block* refers to the smallest addressable unit on the device. For most disk devices, a VxWorks block corresponds to a *sector*, although terminology varies.
  – Block devices in VxWorks have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, block device support consists of low-level drivers that interact with a file system. The file system, in turn, interacts with the I/O system.

• Data transfer is executed as Direct Memory Access (DMA), using a DMA engine (controller) which competes with the CPU on accessing bus/memory. (“Cycle stealing”)
MIDAS M5000 DMA transfers

• The Universe PCI-VME bridge includes a built-in DMA controller that enables high-speed block transfers between MIDAS PCI and VME, without the involvement of the CPU

• Both **direct DMA** transfers and **chained DMA** transfers are supported
  
  – Direct mode transfers a single block of data between the PCI bus and the VME bus (or more correctly, PCI and VME memories)
  
  – Linked list (chained) mode transfers one or multiple blocks of data between the PCI bus and the VME bus. The DMA engine uses DMA command packets to describe how to transfers each block of data.
  
  – Next pages from the MIDAS M5000 manual, more about M5000 I/O in a lab exercise
Direct DMA

**uniDmaDirect**

```
STATUS uniDmaDirect
(
UINT32 vmeAdrs,
UINT32 pciAdrs,
UINT32 byteCount,
UINT32 vmeAmCode,
BOOL pci64,
int direction
)
```

**Synopsis**

This function commands the Universe DMA engine to transfer a single DMA block.

- `<vmeAdrs>` is the VME bus address of the DMA block.
- `<pciAdrs>` is the PCI bus address of the DMA block.
- `<byteCount>` is the number of bytes in the DMA block.
- `<vmeAmCode>` is the VME Address Modifier code to be used when transferring the DMA block.

If `<pci64>` is TRUE, then PCI dual address cycles are enabled.

- `<direction>` is either UNI_DMA_V2L (0) meaning VME to PCI, or UNI_DMA_L2V (1) meaning PCI to VME.

**Returns**

OK or ERROR
Linked list DMA

```
uniDmaChainCmdPktCreate

UNI_DMA_CHAIN_CMDFPT_NODE * uniDmaChainCmdPktCreate
{
  UINT32 vmeAdrs,
  UINT32 pciAdrs,
  UINT32 byteCount,
  UINT32 vmeAmCode,
  BOOL pci64,
  int direction,
  UNI_DMA_CHAIN_CMDFPT_NODE *prev,
  UNI_DMA_CHAIN_CMDFPT_NODE *next
}
```

**Synopsis**

`<uniDmaChainCmdPktCreate>` allocates a new DMA chain command packet and initializes it according to the arguments given.

- `<vmeAdrs>` is the VME bus address of the DMA block.
- `<pciAdrs>` is the PCI bus address of the DMA block.
- `<byteCount>` is the number of bytes in the DMA block.
- `<vmeAmCode>` is the VME Address Modifier code to be used when transferring the DMA block.

If `<pci64>` is TRUE then PCI Dual Address Cycles are enabled.

- `<direction>` is either UNI_DMA_V2L (0) meaning VME to PCI, or UNI_DMA_L2V (1) meaning PCI to VME.

If the newly created command packet is to be part of an already existing chain of DMA command packets, `<prev>` should point to the `<UNI_DMA_CHAIN_CMDFPT_NODE>` structure representing the packet in front of the new one, and `<next>` should point to the `<UNI_DMA_CHAIN_CMDFPT_NODE>` structure representing the next packet in the chain. Both `<prev>` and `<next>` may point to NULL.

**Description**

**Returns**

Pointer to newly created `<UNI_DMA_CHAIN_CMDFPT_NODE>` structure on success or NULL on failure
6.4 Linked-list Mode

Unlike direct mode, in which the DMA performs a single block of data at a time, linked-list mode allows the DMA to transfer a series of non-contiguous blocks of data without software intervention. Each entry in the linked-list is described by a command packet which parallels the DMA register layout. The data structure for each command packet is the same (see Figure 16 below), and contains all the necessary information to program the DMA address and control registers. It could be described in software as a record of eight 32-bit data elements. Four of the elements represent the four core registers required to define a DMA transfer: DCTL, DTBC, DVA, and DLA. A fifth element represents the DCPP register which points to the next command packet in the list. The least two significant bits of the DCPP element (the PROCESSED and NULL bits) provide status and control information for linked list processing.

The PROCESSED bit indicates whether a command packet has been processed or not. When the DMA processes the command packet and has successfully completed all transfers described by this packet, it sets the PROCESSED bit to 1 before reading in the next command packet in the list. The PROCESSED bit must be initially set for 0. This bit, when set to 1, indicates that this command packet has been disposed of by the DMA and its memory can be de-allocated or reused for another transfer description.

DCTL = DMA transfer control register
DTBC = DMA transfer byte count register
DVA = DMA VMEbus address register
DLA = DMA PCI address register
Linked list DMA – Universe PCI-VME bridge

Figure 16: Command Packet Structure and Linked List Operation

The NULL bit indicates the termination of the entire linked list. If the NULL bit is set to 0, the DMA processes the next command packet pointed to by the command packet pointer. If the NULL bit is set to 1 then the address in the command packet pointer is considered invalid and the DMA stops at the completion of the transfer described by the current command packet.