L2 Object I/O Format

• L2 needs to read and write objects **online** and **offline**
  ➤ pre-processors transfer their outputs to L3 and L2 global
  ➤ simulation will need to read and write L2 objects in an offline environment
  ➤ offline analysis will need to read L2 trigger objects

• Code must be flexible enough for **offline** but efficient enough for **online**
Requirements

• Changing the format should be simple
  ➢ the format will evolve over time and so should not be fixed in stone

• Must support multiple formats offline (but not necessarily online)

• Must be portable
  ➢ needs to run on all offline platforms

• Must read from a block of alpha byte-ordered memory
Solutions

• Hand code routines
  ✔ very flexible
  ✗ not easy to change format
  ✗ probably need two versions of reader for off and online
  ✗ read and write code independent = BUGS!

• XDR using rpcgen
  ✔ easy to change format
  ✔ machine independent
  ✔ read and write code automatically generated
  ✗ not easy to add different formats (but can be done)
  ✗ performance and compatibility issues with Alpha board
Solutions II

- CORBA
  - ✗ too unwieldy for use online!
- Python/Perl script to parse header file and generate I/O routines to compile
  - ✔ fully flexible
  - ✔ read and write code linked
  - ✔ changing format simple
  - ✔ can generate online and offline version of reading code
  - ✗ possibly more work to write initially? May pay off in reduced maintenance work?
class Electron {  // Version=3.5
    L2OBJECT(Stream,Electron);
private:
    float emFrac;  // Min=0. Max=1. Size=1
    float ET;  // Min=0. Max=1000. Size=2
    Eta eta;   // Version<=2.0 ....
};

Stream &operator>>(Stream &str,Electron &ele) {
    int8 tmp8bit;          // temporary 8bit variable
    str >> ele.version;   // read electron version info
    str >> tmp8bit;
    ele.emFrac=(float)((tmp8bit*(1.-0.)/256)+0.);  // Similarly for ET......
#ifdef OFFLINE
    if(ele.version <= 2.0)
        str >> eta;
#endif ....
}
Release Environment

• Attempt to follow standard DØ environment
  ➢ SoftRelTools
  ➢ CVS
  ➢ CTEST

• Status
  ➢ currently trying to set up ups and upd at MSU
  ➢ L2 code already in CVS
  ➢ hope to update to SoftRelTools once ups online (and understood!)
  ➢ ready within a month
Software Libraries

Utilities
L2util
L2utilTest

Emulator
L2emu
L2hw

{OR}

Common Code
L2common
L2GblAdmin
L2GblWorker
L2 Global

Hardware
Test Code
L2common Test
L2 Code Structure

- **Choice of release procedure:**
  - Link releases of data transport and user code together to give a single L2 version number.
  - Separate data transport and crate code so that each has a separate version number.
Version Numbering

- **Single version number**
  - ✔ easy to keep track of exactly what code was running when
  - ✗ updating **ANY** crate’s code means **ALL** crates have to have a new version number

- **Multiple version numbers**
  - ✔ each crate can update its code independently
  - ✔ only data movement code change will update all crate code versions
  - ✗ harder to keep track of multiple versions
  - ✗ compatibility issues now that data movement and crate code can change independently
Coding Guidelines

• Follow DØ draft standard
  ➢ obtainable from WWW
  ➢ explains how to write clear, consistent C++ code

• Use DOC++ comments to document the code
  ➢ only change from DØ draft standard
  ➢ every function, variable, typedef etc. should have a DOC++ entry explaining its purpose AND its parameters/return value
  ➢ documentation on WWW “DØ Computing Tools” page
// File: EventMessage.hpp
// Purpose: EventMessage class for L2 Trigger code
// Created: 24-SEP-1998 by Roger Moore
//
// Comments:
//    Header file for EventMessage class. See Doc++ comments for
//    full documentation.
//
// Revisions:
//
#ifndef _L2UTIL_EVENTMESSAGE_HPP
#define _L2UTIL_EVENTMESSAGE_HPP
#include...
//================================================
// CLASS : EventMessage
/** Class which reads and writes an admin event message to a worker. The format of the message as send is:
\begin{center}
\begin{tabular}{|r|l|} \hline
Byte & Function \hline
0 & message ID (Message::EVENT_MSG) \hline
1 & buffer number of event processed \hline
2 & event processed=0/failed=1/passed=2 \hline
8-15 & 128 L2 trigger bits \hline
\end{tabular}
\end{center}
@author Roger Moore (moore@pa.msu.edu)
@version 0.1 24-SEP-1998 */
//================================================
Example L2 Class
class EventMessage : public Message

Class which reads and writes an admin event message to a worker

Inheritance:

![Message](EventMessage)

Public Methods

- static bool check(byte *msg)
  Checks the given memory address for a valid message format
- void eventBuffer(Buffer &buf)
  Sets the buffer number of the event processed
- EventMessage(void)
  Constructor to create an event message

Documentation

Class which reads and writes an admin event message to a worker. The format of the message as send is:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>message ID (Message::EVENT_MSG)</td>
</tr>
<tr>
<td>1</td>
<td>buffer number of event processed</td>
</tr>
<tr>
<td>2</td>
<td>event processed=0/failed=1/passed=2</td>
</tr>
<tr>
<td>8-15</td>
<td>128 L2 trigger bits</td>
</tr>
</tbody>
</table>

- EventMessage(void)
  Constructor to create an event message. The constructor requires the MBus address to write the message to. It also calls the reset method to ensure the class is fully reset.
Additional Rules

• Restrictions on online code
  - no dynamic memory allocation
  - no C++ style exceptions
  - no RTTI
  - no Standard Template Library

• Additional guidelines
  - only use virtual functions where **ABSOLUTELY** needed
  - can be pretty free with use of “inline” since entire program will sit in cache and branching slows down Alpha processor significantly
Prototype Boards

• Two environments
  1. Alpha Linux
  2. bare system (no OS)

• Three setups to test with
  – 1, 2 and 3-4 nodes

- Ethernet
- Serial
- Floppy drive
- Hard drive
- Alpha Card (128Mb)
Single Node I

Get the boards to boot!

• Setup both software environments (Linux and DBM)
• Test hardware devices
  ➔ VME interface
  ➔ M Bus interface
• Test existing low level code
  ➔ interrupt handler
  ➔ Real Time Clock handler
• Test hardware monitor port
  ➔ needs new FPGA program
Single Node II

• **Run “dumb” Administrator**
  - no MBT, Workers, VBD etc
  - only respond to Bit3(TCC)

• **Run “dumb” Worker**
  - no MBT, Admin, VBD etc.
  - responds to Bit3(TCC)

• **Stabilize “dumb” Worker**
  - perform simple timing tests on Pre-processor/ Global worker code

• **Add and test interfaces to VBD and MBT**
Twin Nodes I

- Test inter-alpha MBus communication
  - MBus speed tests (?)
  - more comprehensive test of broadcast engine (?)

- Test inter-alpha VME communication
  - setup of the VME mapping by a remote node

- Test simple Admin-Worker interactions
  - no MBT, VBD
  - responds to Bit3(TCC)
Twin Nodes II

• Add MBT and VBD to crate
  ➡ perform full test of single worker system
  ➡ no interrupts on MBus (can we poll MBT?)
  ➡ send simulation events through entire chain and examine output
  ➡ perform preliminary speed tests
Multiple Nodes

• Setup crate with 2+ workers
  ➔ use Bit3 to start with
  ➔ go to MBT and VBD later

• Test different Administrator setups
  ➔ lockstep OR non-lockstep
  ➔ pre-processor OR global
  ➔ shadow nodes?

• Test Calorimeter PP
  ➔ three workers running in one crate
How far can we go with Linux?

- Linux useful for debugging code “offline”
  - runs on Alpha board
  - same source code (or even executable!) behind a UNIX interface
  - nice, cosy development environment

- Can we conceive of using Linux at run-time?
  - what about performance?
  - interrupt latency?
  - preemptive multi-tasking?
The Linux Kernel

Kernel Memory Space

Interrupt Handler

Hardware

Virtual Memory space for processes

Physical Memory

Timer interrupts used by kernel to perform “book keeping” tasks

Device interrupts passed to drivers which request them
L2 Run-time Kernel

Hardware

Switch turns off all or some interrupts to the kernel

Communicate with process using reserved physical memory

L2 Interrupt Handler

Interrupt Handler

Kernel Memory Space

Reserved Physical Memory

L2 Executable Code

Physical Memory

(possibly PAL code)
Run-time Linux

• **Reserved Physical Memory**
  - used to store event Buffers
  - used to communicate between interrupt handler & executable

• **Interrupt Handler**
  - switch can turn off interrupts to Linux kernel
  - executable disables timer interrupts to kernel and so cannot be preempted
  - updates FIFOs in reserved memory when new event arrives
Linux Performance

- Linux environment will incur some overhead
  - interrupt handler has an extra ‘if’ statement to select L2 or kernel
  - executable runs in virtual memory so it’s very hard to ensure no L3 cache overwrites
  - initialization may be a little slower since code has to do more (load executable into virtual memory, turn off kernel etc.)
So what does Linux give you?

- **Core Dumps!**
  - easy stack unwinding since executable runs as a Linux process
  - able to store multiple core dumps on local disk so several crashes can be examined

- **Full UNIX environment**
  - run debugger on same machine as executable
  - remote access simple

- **Same development and run-time environment**