A real time, hardware in the loop, parallel, Avionics Test System.

Stephen Bissell BA (Hons) Pg Dip
Introduction

- This presentation will take you briefly through the parallel computing issues involved in TEST of Avionics Control Systems.
- It is based on several years experience in test of avionics systems for large civil aircraft.
- I’ll be presenting the nature of the problem to be solved, the solutions chosen, and the lessons learned during the development.
In brief...

I’ll be taking you through:

- The problem domain
- Scaling the solution via parallelism
- Tools and platforms
- Language / Code issues
- Summary
The problem...

Are you sure you tested this, Dad?

Trust me, you’ll be fine!

We may be very confident indeed of our design... but... there is a very human need to PROVE our confidence BEFORE we get too “carried away”.
Test without flight

- Testing our DESIGN.
- “Fooling the system”

• The job of the Test Bed is to convince the System Under Test that it really is flying....
What kind of “System Under Test”?
Zoom in a little...

- Control, via high bandwidth comms
- Modular Architecture
- Conversion of control from digital to/from lower level IO

To (a lot of) IO
What is all this “I/O”?

- AFDX (Avionics Full DupleX Ethernet, 100MB/s)
- ARINC 429 – 100kb/s
- “Discretes” – input, output
- “Analog” – input, output
- LVDT / RVDT measurement of excitation, and of output (throttle simulation).
- “Tachometer” card – simulating speed of rotation.
- ARINC 825 (aka CANbus for aerospace).
And how much are we talking about?

- Each Major LRU has nearly 400 pins.
- A typical Test Rig may accommodate all the LRUs for a given function, including control and IO concentrators.
- Several AFDX network ports, multiple switches.
- Numerous ARINC 429 busses (100kBit serial).
- Several hundred discretes.
- Numerous Analog and other specialised IO types for fast loop closure.
- Additional layers of wiring replicating that found on the aircraft.
- Around 200,000 distinct named data entities distributed over tens of thousands of named data packets and hundreds of individual wires.
Requirements for our Test Environment.

- Simulation of physical components and physical laws.
- Transport of information from simulations to/from IO.
- Abstract the simulation from the underlying IO (models are 3rd party).
- Loop closure at up to 1ms rates.
- Scenario modelling – “Virtual Flight Test”.
- Visualise data in a customisable way.
- Allow users to design automated tests, or test interactively.
- Integrate with client MATLAB / Simulink toolset or other proprietary standards.
- Log (record) data.
- In short, a full “Test Tool Platform”: TMS is a primarily a software product.
The scale of the problem demands distribution of computing power across a number of processors.

Real time constraints, specialised IO types, and the scale of IO demand distribution of IO acquisition.

Visualisation must be scalable to meet the needs of testers.
Scaling Parallelism: what do we have?

Generally:
- Multiple layers (hardware/firmware/driver layer/middleware ...).

From the software perspective, we have:
- Parallel execution of client code and middleware tasks.
- Distributed across various OS and network types.
- Mixed IO semantics (IRQ, polled, DMA, serial/parallel, autonomous/non-autonomous).
- Distributed Integration Platform for client simulations.
We need the following primitives:

- Data Exchange.
- Synchronisation (Action Y follows event X: dependencies).
- Mutual Exclusion.
- Appropriate level of determinism.
- Appropriate prioritisation (pseudo “automatic” parallelism).

And we need these:

- At many of the layers.
- Accessible to the client simulation toolchain.
- Available on, and distributed across, the various OS / CPU / hardware types.
Parallel computation: device semantics

- RCC (GUI)
- DT (GUI)
- DT (GUI)

- Win32 on PC
- Ethernet
- Fibre network
- RTOS on VME CPU cards
- Multiple VME backplanes
- VME I/O cards
- PCI busses
- PMC modules
- Conditioned I/O
- Unconditioned I/O

Signal conditioning/Fault insertion/routing/"Break-out"
bottlenecks
Tactics - delegating “scatter gather”

- Inputs arrive asynchronously from the various parts of the SUT into multiple parallel channels on multiple IO cards.
- Extensive use of largely autonomous IO cards with onboard DSP / CPU + RAM + DMA Engines to offload CPU and backplane.
- Acquisition semantic – “polled”, “queued” or “events” can then be configured according to need.
- Autonomous “current value table” IO cards provide parallel output management.
Tactics: distribute parallelism in real time

- Systematic reliance on RFM device semantics: distributed memory “for free” with built in event mechanism.
- Synchronisation within/between nodes achieved by extensive use of publisher / subscriber pattern by means of RFM IRQ.
- Mutual Exclusion between nodes: centralised resource broker by means of RFM IRQ.
Parallelism for the modeller...

SubSystem

Parallel start

multi CPU Setup

Synchronised starts

Communication
Integration into the application layer
A customer example.
Platform considerations

- **PC**
  - Cheap
  - Huge software and skill base (for Windows)
  - Rapidly evolving hardware – multi core now standard
  - Obsolescence / Maintenance issues for RTOS

- **Wind River vxWorks + VME**
  - Expensive
  - Long roadmaps
  - Exotic / large scale, autonomous IO cards
  - Deterministic, priority based
  - Excellent visualisation tools
Choice of platform

- Until recently, the choice of platform was fairly straightforward: RTOS for IO and models; Windows for display and command.

- Scale and PC CPU performance plus overhead of migrating RTOS BSP favours devolving some FP intensive tasks towards the PC: “appropriate determinism”
Parallel software: code constraints

- Third party driver software supplied in C / C++ or even C compiled to binary.
- Where source is supplied, modifications may be needed.
- OS primitives universally in C (possibly POSIX).
- Differing native OS semantics for mutual exclusion / synchronisation / forking.
- Differing OS models for process address space privacy.
- Very small developer team.
Some solutions employed.

**Tools / Languages**

- Reduce language count: main system middleware and some apps in **C++**.
- **C** interface for client toolchain / model development.
- Less time critical GUI apps in **C#** or **Tcl/Tk** panels.
- **Tcl** scripting for rig control.

**Programming Model**

- Explicit parallel programming, with priority preemption “faking” a certain degree of automatic parallelism.
Using C++ in parallel real time.

- Naturally extends existing C / C++ device / OS semantics.
- Good abstraction, but can still get “close to the machine”.
- **User defined types** and class hierarchies to hide differing semantics for mutual exclusion, synchronisation, distributed memory and device semantics.
- Further class hierarchies providing higher level pattern interfaces, such as publish / subscribe.
- Extensive use of RAII pattern for all resources that have a “take / release” semantic ensures automatic, reliable, deterministic release of resources.
RAII in distributed systems

- Distributed systems may make use of resources that must be "mutually exclusive".
- These resources should be accessible from all nodes.
- Release must be guaranteed and deterministic.
- RAII is a pattern available in the C++ language that offers precisely this control via user-defined types.

```cpp
class mutex {
    mutex() { //get the appropriate mutex }
    ~mutex() { //release it }

    private:
    sometype mHandle;
}

void foo() {
    mutex get; //get the mutex
    ...
    if (a) return; //released here
    ...
    if (b) throw "oops"; //or here
    ...
    return; //or here
}
```
C++ and the future of parallel real time.

- Too many pointers – can be very unproductive.
- No native constructs for parallelism.
- C legacy means that syntactically it can be esoteric, further reducing productivity.
- Disciplined use of user defined types can help with the above.
- Much useful functionality is “bolted on” via the STL and other libraries.
- Alternative languages, such as D?
Summary

- Parallel design - right the way up from the hardware.
- Patterns for parallelism are needed at many layers.
- Language support ideally should support the patterns, but allow user defined implementations appropriate to the layer / interface / system.
- These patterns need to be accessible to the toolchain level.
- But embedded / real-time will always need “bare metal” access.
- Meanwhile, embedded systems keep growing: this is the challenge.
Thank you for your attention
Alternatives to C++?

The **D** language, recently stabilising at version 2.0, seems to offer most of C++ and C language features:

- “Bare metal” efficiency, including inline assembler.
- C RTL / OS compatibility.

But adds:

- Auto type inference.
- Garbage collection OR explicit memory management.
- Reduced need for pointers.
- Automatic member initialisation.
- Built in RTTI

D is aimed at being “as productive as Python, but as powerful as C”.
Gotchas

- Still no explicit parallelism constructs.
- “Automatic” conversion of C++ to D seems unlikely, so existing code would need some intelligence in porting.
  - No multiple inheritance.
  - No namespaces (but “better equivalent” exists).
  - No support for bitfields of arbitrary size.