Accelerator Magnet Control from an SMP Workstation:

S. Herb and H. G Wu
Deutsche Elektronen Synchrotron,
Notkestrasse 85, 22603 Hamburg, Germany

Introduction:

We report on the development of a prototype accelerator magnet control system based on an Axil 311 (Sparc 10 clone) multiprocessor workstation running the Solaris 2.4 operating system, a UNIX variant supporting soft real time scheduling and response in an SMP (symmetric multiprocessing) environment. The motivation for this work comes from several related considerations:

• The introduction of real-time capabilities in desktop operating systems may reduce the need for specialized real-time systems for driving front end hardware and permit task development and execution in more comfortable and standardized environments.

• Control of large groups of 'tightly-coupled' equipment, such as accelerator magnets, can be localized in self-sufficient modules which perform device coordination and error handling and present integrated device interfaces to clients, rather than having to export thousands of 'channels'.

• Multiprocessor workstations appear to be an excellent platform for combining the high-rate real-time I/O required for the front-end control of a large number of devices with the flexible higher-level tasks concerned with 'integration' and communications. One benefit of using more than one processor is that the complications and inefficiencies resulting from context switching and pre-emption can be drastically reduced. Another is that efficient intertask communication using shared memory can be available to a larger group of tasks.

• Commercial developments in this direction are very rapid, which ensures competitive costs and effective software support.

System Description:

The Sparc station has at present two 55 MHz HyperSparc CPUs, each with 256 kB of external cache. It is connected to a VME crate via an SBUS to VME adapter (Performance Technologies SBS 915). In the VME crate are Motorola MVE 162 CPU cards, each driving four of the DESY specific 'Sedac' fieldbus lines used for control of the HERA power supplies. The present test setup has four lines; 16-20 lines will be required to drive all of the ~1300 magnet power supplies in the HERA electron and proton rings.

The MVE162 cards run simple queue processing tasks written for convenience using VxWorks. The Sparc VME interface is the sole VME bus master; it writes packets of fieldbus 'telegrams' into the VME memory queues, and later (asynchronously) collects the results returned. The writes and reads are performed on the Sparc side by a *Device Server* task. This task, described later in more detail, monopolizes one of the Sparc processors by running with Real Time priority in a polling loop.

Controls tasks running on the other processor include a *Network Server* which provides access for remote clients and '*Bump*' tasks which permit the control of groups of magnets. It is planned later to also include alarm and data logging tasks. These tasks communicate with the Device Server using two mechanisms built on Unix System V shared memory. The first is a system of shared memory queues and data buffers used for passing API messages (commands) between the tasks. The second is a real time shared memory database, which is continually refreshed by the Device Server and can be read out, via API calls, by the other tasks (read time 4 µsec). The architecture is sketched in Figure 1. An important point is that the Sparc station will not serve as a console machine; whether X-windows access will be permitted at all during real operation is an open question.

The structures and tasks have all been implemented in C++. The shared memory queues permit write access from multiple tasks and therefore include mutex protection of the queue tail. Most of the low level structures are persistent, to avoid performance issues associated with dynamic object management. Experience so far with C++ is that it is well suited to this relatively low level programming and that it is superior to C (at least) for producing reliable and understandable code. A minor problem is that the creation of shared C++ objects in the shared memory becomes non-trivial once virtual methods are defined.

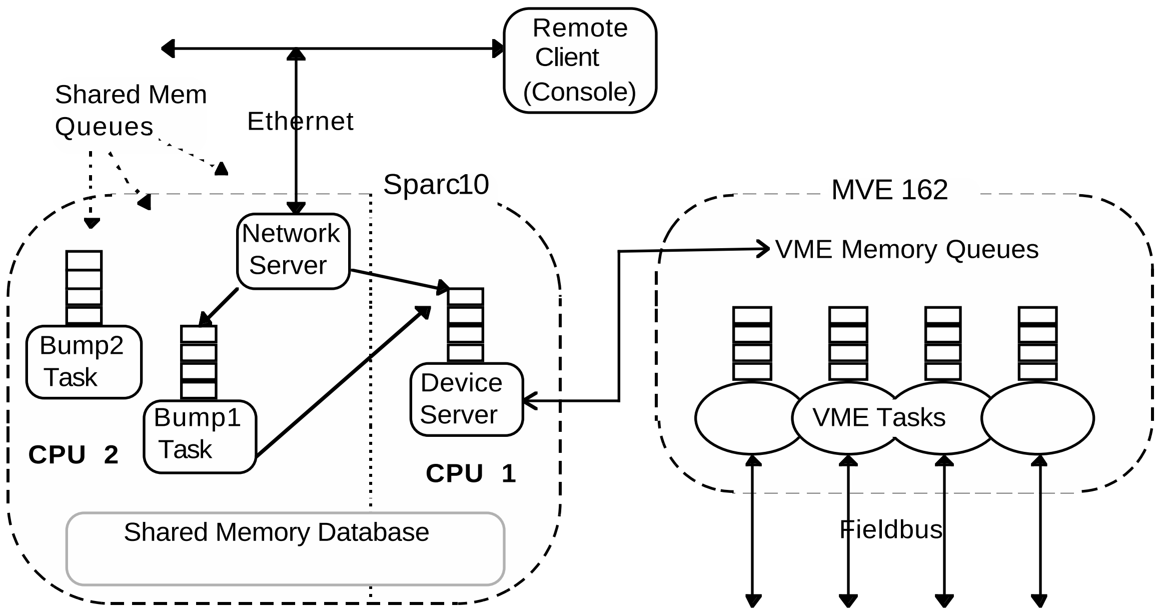

Figure 1 - Schematic drawing of the system architecture.

Device Server Description:

The Device Server contains C++ objects representing each of the hardware devices controlled over the fieldbus lines and is responsible for the timely activation of each of these objects when a command for it is received. This is achieved by a 'command processing engine' operating on a single FIFO input queue, which for each command calls the requested device and method of generating the fieldbus operations and later, asynchronously, hands back the fieldbus results. ‘Timely’ activation is ensured by limiting commands at this level to single devices and single step operations and maximizing throughput. The goals are a maximum command throughput of roughly 10 kHz (including 3 kHz of data poll commands for the real time database refresh) and typical return times for requests from local clients (with contact to the shared memory) of less than 20 msec, permitting local clients to control groups of devices with a 50 Hz rhythm, if desired. An obvious limitation of such a system is that this rhythm may contain significant jitter, which is not a problem for our magnet control applications.

This architecture matches well the polling loop operation of the Device Server and could be characterized as ‘loosely deterministic’ rather than ‘tightly deterministic’, in that timings and priorities of individual commands are not rigidly ordered. The complete absence of context switching and interrupt handling permits significant simplication in the software design and the ‘wasted’ time in the polling loop is of little interest if the remaining CPU(s) can satisfy the other system needs.

System Design Considerations and Goals

There is a great deal to be said about the considerations which led to this design; we content ourselves here with with a mixed list summarizing (or repeating) some of the points we found important. We should:

• like to control ~1300 Power Supplies for the HERA e and p rings from a single platform
• be able to drive the ~16 fieldbusses at a significant fraction of their full bandwidth
• be able to build magnet bumps from any of the various magnet types sitting on arbitary fieldbusses
• have many magnet bumps active at once - use *‘linear superposition’* rather than *‘ownership’* as principle
• keep error handling for multi-magnet groups close to the hardware
• use command communication structure based on message passing and queues
• satisfy most reads from real time database rather than fieldbus access (esp. display update traffic to consoles!)
• use object description of hardware with common functions implemented via inheritance and polymorphism
• support local ‘compound devices’ (such as bumps) which offer remote clients control over groups of magnets
• support local supervision tasks (avoid unnecessary data transfer over the network.)
• use API protocol close to the 'standard model'
• build a Network Server as an independent module to permit easy modification or replacement

System Performance:

A typical test run configuration has been included:

- a *Device Server* running in a polling loop in the RT (real time) class with the highest priority, with all pages loaded and locked into memory and with 100 ms time slices (essentially the rescheduling frequency)
- a *Network Server* running in the RT class with lower priority and 'waking up' every 50 ms to send data refreshes (read from the real time database) to a remote client.
- several local *'Bump'* tasks running in the RT or TS (time-sharing) class, 'waking up' every 20 ms to test the return status of commands submitted during the previous cycle, then submitting new commands to the *Device Server*.
- Assorted user jobs running in the TS class, either for supervision, or in an attempt to test for interaction with the controls jobs (performing a compile or starting xemacs is a good way to generate some disk activity).

These tests indicate that we can achieve the desired command throughput and that there is no problem in running many 'bump' tasks at the 50 Hz rate. We also looked in more detail at the behaviour of the tasks running in the RT class; we measure and histogram the time for the device server to pass through the polling loop using the Solaris 2.x *gethrtime()* call, which returns (in 3 μsec) a 64-bit non-wrapping time with sub-microsecond resolution. The maximum computational work for one pass through the loop takes certainly less than 500 μs. We may therefore assume that loop tranversal times of 1 ms or longer represent deadtimes related to system activity.

The response of Solaris 2.4 is usually specified in terms of the interrupt dispatch latency, which should be between one and two ms, and is related to the non-pre-emptible kernel code sections. What this means for a polling task in an SMP environment is less than clear; the RT task has a priority higher than all system tasks, but lower than all interrupt handlers. In any case, we see a significant number of traversal times in the 1-2 ms range, representing roughly a 0.002% duty factor. More disturbing are measurements taken over time intervals which include heavy disk access activity for TS class tasks, occasioned by the start of Xemacs etc. when we then see several events in the 10-20 ms bin. Occasional 10 ms gaps would not be a severe problem for our magnet control but could be extremely unpleasant in some other real time environments. A recent exchange on usenet suggests that this is an artifact from a scheduler problem which has been identified and fixed in Solaris 2.5.

Another point is the performance of time-share class jobs in the presence of real-time jobs using one processor and part of the second. Performance in executing tasks seems reasonable; an amusing point is that the response of the windowing system on the parent console, on which the real time tasks have been started, is rather sticky, while the response for remote X-windows login is normal.

Conclusions:

The term ‘Distributed Control System’, covers a wide range of possibilities. The model proposed here is of a node powerful enough to drive a large group of devices which functionally belong together, while in addition supporting a variety of higher level tasks. This permits running flexible supervisory and maintenance tasks, as well as tasks which provide remote clients with interfaces to groups of devices, as opposed to ‘channel oriented’ systems which emphasize low level data transfer. In general, we are trying to move more intelligence and self-sufficiency down close to the front end.

SMP workstations with real time capabilities appear well suited to this model. Very interesting in this regard is the progress of the POSIX.4 (more properly, 1003.1b + c,d,...) standards for real-time calls, which have already been incorporated in hard real-time systems such as LynxOS and VxWorks and are now gradually moving into the mainstream operating systems so that real-time programming is becoming a less esoteric activity.

Acknowledgements:

We would like to acknowledge the ESRF, CESR, and HERA PKTR control systems as sources for some of the prejudices which have gone into the system here; obviously they are in no way to blame for the results. Special thanks to K. H. Mess and F. Willeke at DESY for their support, to K. Rehlich, M. Clausen and P. Duval for discussions and to comp.unix.solaris and comp.lang.cpp for many useful hints.