HARDWARE- VS. SOFTWARE-DRIVEN REAL-TIME DATA ACQUISITION

Richard Powell, Sr. Principal Engineer, assisted by
Jeff Kuhn, Member of Technical Staff
L-3 Communications Telemetry & Instrumentation
San Diego, CA 92128

ABSTRACT

There are two basic approaches to developing data acquisition systems. The first is to buy or develop acquisition hardware and to then write software to input, identify, and distribute the data for processing, display, storage, and output to a network. The second is to design a system that handles some or all of these tasks in hardware instead of software. This paper describes the differences between *software-driven* and *hardware-driven* system architectures as applied to real-time data acquisition systems. In explaining the characteristics of a hardware-driven system, a high-performance real-time bus system architecture developed by L-3 will be used as an example. This architecture removes the bottlenecks and unpredictability that can plague software-driven systems when applied to complex real-time data acquisition applications. It does this by handling the input, identification, routing, and distribution of acquired data without software intervention.

KEY WORDS

Hardware-Driven Architecture, Deterministic, Real-Time Performance, Modular Architecture, Parallel Processing, Distributed Architecture

INTRODUCTION

Data acquisition systems come in two generic flavors: software-driven and hardware-driven. As used in this paper, these terms refer to how data is routed through the system (i.e., with software or with hardware). In both cases, the systems are data-driven, meaning that the operations are initiated by the arrival of data as opposed to being scheduled via software. But once the data is acquired, what mechanisms transfer the data from the input modules to the application processes that require the data? How is data input, identified, merged, correlated, routed to different processes, routed to different workstations, sorted, and routed to specific destinations? Are these tasks done primarily with software or with hardware? These questions form the basis of defining software-driven and hardware-driven architectures.

In a software-driven architecture, data movement is controlled by real-time software with hardware assistance from tools like DMA controllers. Data input is handled with software drivers that typically send the data from the hardware to some software distribution process (Data Manager). From here, it is sent to different applications (client processes) using inter-process communication techniques. Hardware can be involved (i.e., DMA controllers, Ethernet ports, etc.), but all data movement and coordination is controlled by, and requires, software processes. The system is still considered data-driven because the arrival of a block of data causes an interrupt to the software, and the software responds to the arrival of the data. But the software still manages all of the data movement through the system, making the system software-driven.

In a hardware-driven system, data is input into the system, routed to hardware modules and software processes, and sent to other systems ***without*** software involvement during real-time operations. Software is required to set up the data flow process, but is not used during the actual movement of data. Data is acquired, input, identified, routed through the system, and output without software intervention, making the system hardware-driven.

In real-time data acquisition systems, all of the incoming data must be handled as it arrives; otherwise, it is lost. For some applications, a software-driven architecture is completely adequate and appropriate. In other more demanding systems, this may not be the case. Because software-driven systems have inherent bottlenecks and latencies, there are many real-time applications for which software-driven systems are not appropriate.

As the number of inputs increase, as the data rates of the inputs increase, as the number of processes on the data increase, and as the number of clients for the data increase, real-time management using only software to drive the data through the system becomes more complex, more unpredictable, and more unreliable (i.e., non-deterministic). There is a point in complexity and performance where it is necessary to drive the data through the system using more hardware assistance.

This paper examines the two extremes of these approaches. The software-driven approach is exemplified by a standard computer system employing only a single bus for input/out (I/O) module data traffic. The hardware-driven approach is exemplified by a specialized system designed specifically for real-time data acquisition. It employs not only a second bus for I/O module data traffic, but also several other mechanisms for moving and routing data through the system and to other systems and applications.

TYPICAL SOFTWARE-DRIVEN DATA ACQUISITION PROCESS

A software-driven architecture is the traditional approach to acquiring real-time data. It involves installing one or more data input modules on a computer bus, servicing the input module with the main processor when data is ready, and distributing the data to the client applications requiring the data. These applications include storage, processing, display, and networking.

Figure 1 shows a typical Compact PCI hardware design used for software-driven architectures. In this case, the single board computer (SBC) board and the I/O modules plug into a Compact PCI backplane. The figure shows the main data paths through which acquired data will travel during the acquisition process. It also shows the CPU, memory, and main components, as well as their relationship to the data buses.

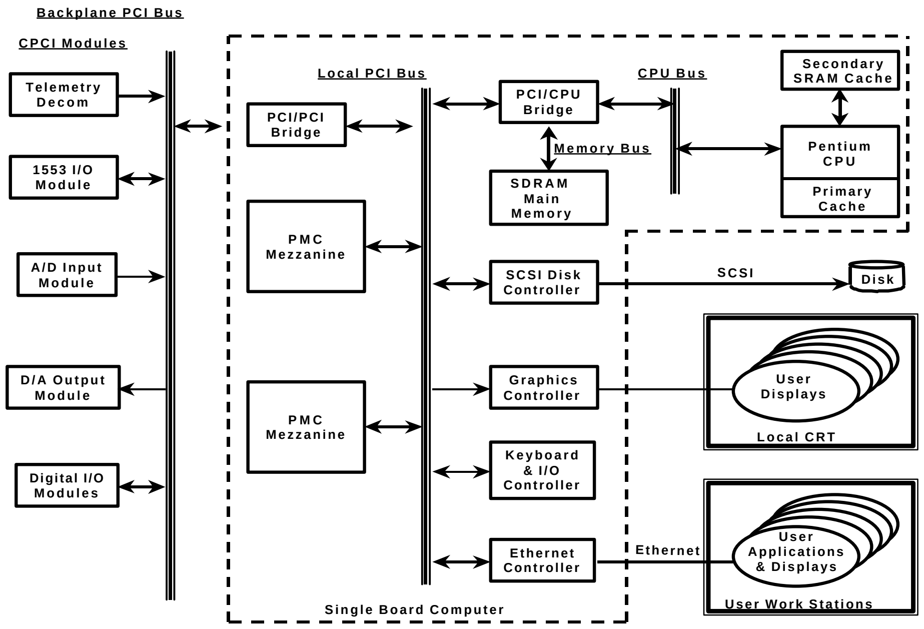

Figure 1. cPCI Hardware Architecture

Figure 2 shows a simplistic view of the software design required to support a real-time data acquisition application and drive the data through the system. Figure 3 shows a more complete, but still simplistic view of the same process. It shows the data paths though which the data must travel for various applications, and it shows the main software components and the interaction paths between them.

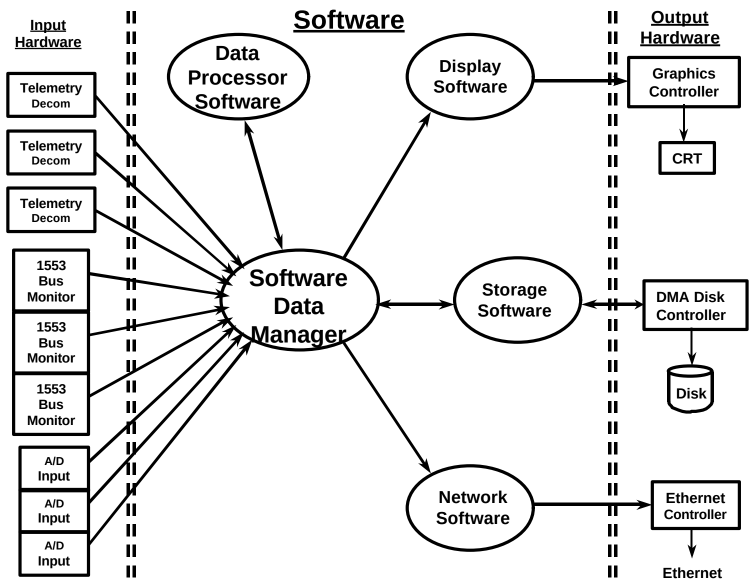

Figure 2. Simple Software Architecture Diagram

What follows is a simplified description of the software process required to support the acquisition of real-time data via a software-driven approach (see Figure 3). The system can be a standard computer, like a Windows NT-based PC, or it can be a more sophisticated system running a real-time operating system networked to a workstation.

1. Buffer Data on Input Module à Purpose: Collect Data and Reduce Interrupt Frequency

- Collect incoming data in on-board buffer.
- Buffer size is dependent on data rates and desired interrupt frequency.
- Interrupt frequency usually needs to be less than 100 Hz or 10+ milliseconds/interrupt. (This gives latency to the incoming data, with more latency at the beginning of the buffer.)

2. Interrupt Processor and Run Driver à Purpose: Move Data from Input Module to Main Memory

- Assert interrupt when buffer is full while continuing to collect data in second buffer.
- Halt current execution of CPU.
- Save context of current execution.
- Execute Module Driver ISR (interrupt service routine).
- Clear interrupt and schedule Module Driver.
- Execute Module Driver (sometimes at the task level).
    - Get pointer to next memory buffer.
    - Start DMA process.
    - Return to previous execution.

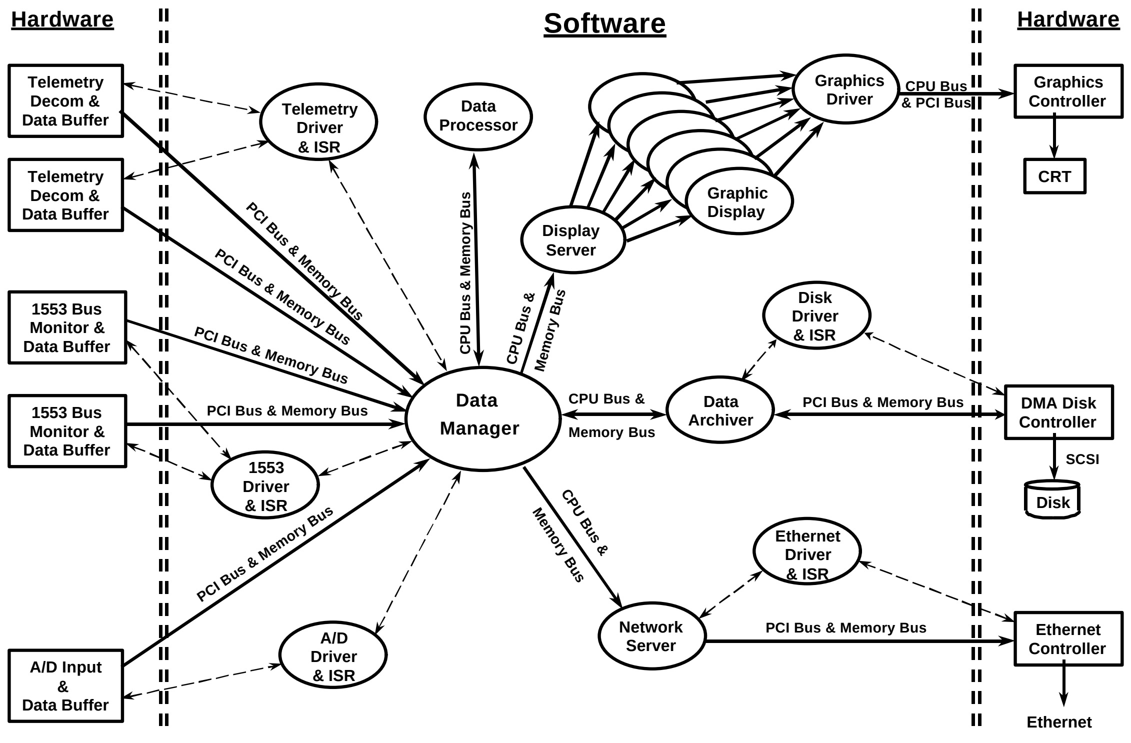

Figure 3. Detailed Software Architecture Diagram

3. Interrupt Processor and Run Driver à Purpose: Notify Data Manager of Arrival of New Data

- Assert interrupt when DMA is complete.
- Halt current execution.
- Save context of current execution.
- Execute Module Driver ISR (interrupt service routine).
- Clear interrupt and schedule Module Driver.
- Execute Module Driver (sometimes at the task level).
  - Send event to Data Manager with pointer to new data buffer.
  - Return to previous execution.

4. Run Data Manager à Purpose: Distribute Data to Client Processes

- Depending on priority of Data Manager relative to other tasks, Data Manager starts execution after previous task is suspended and scheduler starts Data Manager.
- Data Manager reads and identifies new block of data.
- Data Manager searches input lists of all client processes.
- For each client process that wants new data, Data Manager sends an event to each of those processes with a pointer to the new data block. Data Manager increments "use" count for the block for each client that gets the block.
- Data Manager checks to see if previous blocks have been freed by all client processes. If so, Data Manager frees the block for Module Driver to use again.

5. Run Client Processes à Purpose: Display, Archive, Network, and Process Input Data

- Depending on priority of each client process relative to other tasks, processes execute one at a time after previous task is suspended and Scheduler starts new process. (This involves a complete context switch to the new process.)
- Client process "copies" its data from Data Manager input buffer to its own memory and decrements "use" count.
- Client process reads ID block of data and searches its list of incoming data to determine what operation to perform.
- Client processes perform their operations on data and then send data to next process or output. (These sometimes involve calls to other drivers, more interrupts, more sending of events to other processes, sending processed data back to the Data Manager, more task switching, more inter-process communication, etc.)

This is the procedure for receiving one block of data from one source. In each step of this procedure, there are caching, virtual memory, context switching, and higher priority tasks to contend with, not to mention other data sources. These tasks contribute to added latency, non-determinism, and lower performance. All processes and data movement must use the same resources (i.e., CPU, PCI Bus, main memory, CPU data bus, etc.). Even with multiple processors, there remain many bottlenecks and resource issues as described below.

Caching

All data and instructions must be in the primary cache for the CPU to access them. If needed data or instructions are not present, they are copied in from the secondary cache or main memory, thereby flushing other data or instructions out. If the data/instructions are still not in the secondary cache, then access from main memory is required, which is much slower.

Virtual Memory

Each software process needs a certain amount of memory for data and instructions. As new processes are started (displays, user applications, etc.), more memory is needed. In virtual memory systems, when more memory is needed than is physically present, the hard disk is used for additional memory. In this case, if the needed data/instructions are not in memory, there is a much greater delay in the processing of a single data block since disk access is also required to complete the operation.

Context Switching

Every time a different process (thread) runs, the system requires a call to the Scheduler, a full context switch, often primary and secondary cache updates, data copying, and sometimes swapping of memory in and out of disk.

Task Priorities

Every process (task) has a priority (at least this is true in real-time operating systems). If the Data Manager or client processes have lower priority than other tasks, they can be preempted or delayed in running. This again adds to non-determinism and lower performance.

Other Data Sources

With multiple data sources, even of the same type, the number of interrupts, calls to the Data Manager, events to the client processes, context switches, etc. multiplies proportionally. However, because of the interdependence of the application, and the contention for resources, the increased loading can be greater than a simple proportional increase and can actually increase exponentially.

Determinism and Predictability

Every time a new display is opened, more processing is required (not to mention the startup time of that process). Every time the user holds down the mouse button, the loading on the system is increased. Every time a data packet or a broadcast packet is received from the network, the loading of the system momentarily increases (including loading on the CPU, loading on the buses, and additional processes waiting to run). Every time a data rate changes or a block size changes, the loading on the system changes and the performance of the system changes. The number of interrupts per second, the input data rates, the storage rates to disk, the processing rates on the data, the quantity of data to be displayed, and how quickly data can be displayed are all interdependent and very difficult to determine in a software-driven architecture. Every new input requires more inter-process and/or inter-processor communication and sharing of the same resources.

OTHER FACTORS

Time Stamping of Data

Unless the time source is located on the hardware, it is not possible to accurately time stamp incoming data to better than a few milliseconds. Time stamping can only occur after the Module Driver is called. Normally, a driver talks to its own hardware and no other. Therefore, to get time, it must make a call to another driver (time board) and wait for the time to be returned. This process is subject to all the delays and latencies cited above. In some systems, it is possible to violate this convention, but even when the input module driver reads the time directly, it can still involve tens to hundreds of microseconds' worth of delays and errors. Also, since the data is blocked, time is assigned to the block, not to individual data words. Deriving the time for individual data words can be difficult and requires even more processing resources.

Data Merging, Correlation, and Synchronization

Data arrives in blocks and in time intervals of tens to hundreds of milliseconds. Data from different sources, even when they are of the same type, cannot be easily merged and correlated with data from different sources. Synchronizing events and data becomes difficult, processing-intensive, and usually low in fidelity.

Replaying or Output of Data in Correct Time Intervals

Because data in a software-driven architecture is blocked and driven by interrupts and software tasks that cannot run continuously, it is impossible to output data in the same time sequence and interval in which it was received. This makes it difficult to output acquired data to analog outputs (for strip charts, etc.) or for replaying the acquired data for post-analysis, which requires original timing.

Distributed Applications

Spreading the application across multiple processors or across a network to multiple workstations increases the loading on the front-end acquisition system. The bottleneck of the input section of the software-driven architecture is now further burdened by having to distribute more data to more destinations, while not being relieved of the loading required for servicing the input sources.

Parallel Processing

Parallel processing is difficult because it requires the software to distribute, coordinate, and synchronize the application, all adding to the processing load.

Modularity

Modularity is limited because of the interdependence of the system and the inherent bottlenecks involved. All input modules must interact with other resources of the system. Client processes must know the structure of the data buffers for each input, which must remain fixed in real time and is different for every input source. Modularity has a very limited definition in a highly interdependent, software-driven architecture.

Expandability

Adding more data sources, increasing data rates, or adding applications that use the data all impact the loading and throughput of the system. All of these require complicated and detailed analysis and usually require software changes, while still having limited expansion capability. Because of the interdependence of the system, doubling the data rate or number of input sources can more than double the loading on the system and the system's complexity.

Complexity and System Reliability

Because of the interdependence of the software-driven architecture, as more data sources and applications are added, the complexity of the system increases significantly. Determining the performance limits of the system is difficult and usually can be done only through thorough testing with all possible scenarios. Predicting performance limits ahead of time is almost impossible. The complexity and lack of predictability cause system reliability to be questionable when high performance is required. Increasing input sources or data rates compounds this problem.

Development and Maintenance Costs

Development and maintenance costs are often grossly underestimated in software-driven real-time data acquisition systems. Because of the complexity and non-predictability of a software-driven system, budgeting, scheduling, and designing such a system is extremely difficult and takes a great deal of experience. Customers and suppliers, as well as managers and engineers, all tend to underestimate the complexity and unpredictability of a software-driven architecture applied to a high-performance real-time requirement. There is a tendency to over-simplify the problem and assume that one can buy a few off-the-shelf data acquisition modules, install them in a high-end computer, write some glue software for the vendor-supplied drivers, and easily integrate them into the desired application.

There have been numerous cases where the long-term cost of a system was tens or hundreds of times more than the cost of the initial inexpensive COTS hardware. Sometimes the system never does what it was expected, and frequently it cannot be upgraded to do more. Users often spend fortunes in vain attempting to modify their investment to do what it was supposed to do at the outset. The problem is that there are applications where systems just do not have the correct architecture to ever do what is needed. The only answer is to switch to (or start with) a system that is designed and architected for real-time data acquisition; one that does not have the inherent bottlenecks, complexity, and unpredictability of a software-driven system.

HARDWARE-DRIVEN ARCHITECTURE

To eliminate the bottlenecks and interdependencies associated with a typical software-driven design, L-3 designed a real-time data acquisition system using a hardware-driven architecture. It was designed to have all those characteristics required to reliably perform real-time data acquisition from multiple high-speed data inputs of various types. The characteristics of the system include the following:

- Predictable and deterministic
- High-performance
- No bottlenecks and minimal shared resources
- Non-interrupt-driven I/O
- No data blocking, except for storage
- No unnecessary data copying
- Very low latency
- Accurate time stamping of data
- Able to merge, correlate, and synchronize data
- Able to output processed data in correct time sequence (strip charts)
- Simple software interfaces
- No unnecessary inter-process and inter-processor communication
- Supports distributed applications
- Parallel processing and parallel data paths
- Modular, expandable, and reconfigurable
- Able to play back data in its original form

Figure 4 illustrates the basic system design. It shows I/O modules of various types plugged into a high-speed backplane, along with the basic support hardware for processing, archiving, and exporting the data.

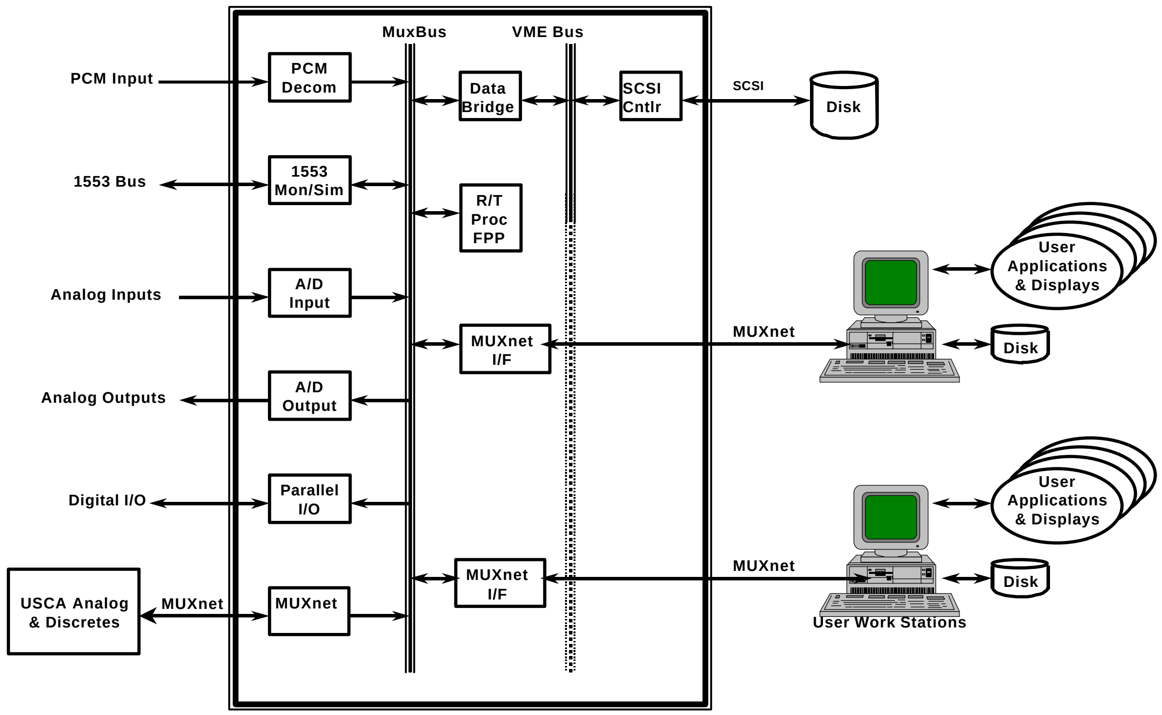

Figure 4. Real-Time Hardware-Driven Distributed Architecture

Figure 5 illustrates the data flow and software design of the hardware-driven architecture. Note the minimal software required to drive the data through the system. The only data path requiring software support is for high-speed storage to disk. But in this case, the driver only sets up the DMA process on the disk controller; it never touches the data. The data is transferred to the disk without any other software involved, and all data transfers between the Data Bridge and the disk controller are handled via hardware.

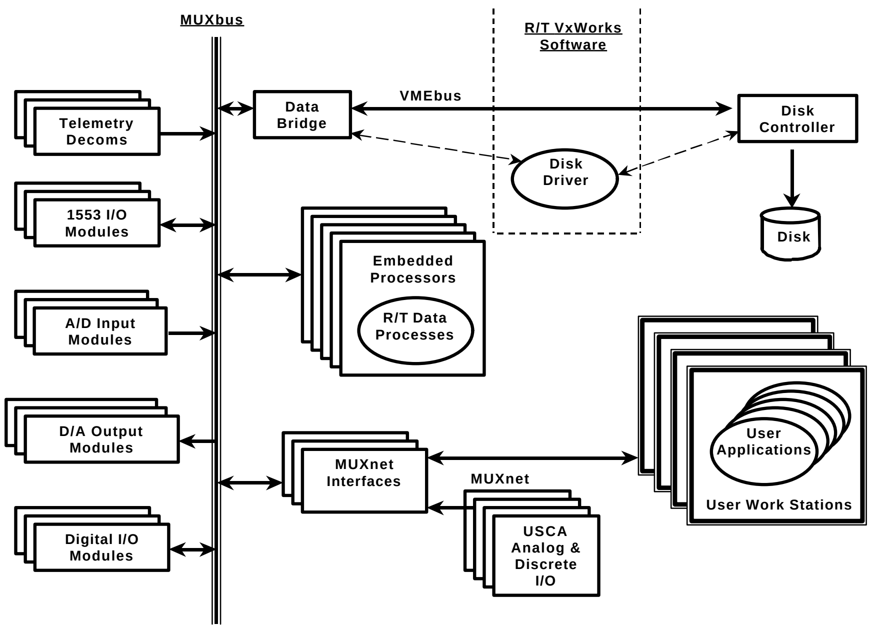

Figure 5. Hardware-Driven Architecture

The system's data acquisition process is described below:

1. Input module acquires and outputs data to the high-speed backplane (MUXbus) one sample at a time.

- Each data measurement to be received by an input module is assigned a 16-bit ID tag.
- When a data word is received, the input module immediately requests an output cycle to the MUXbus.
- With a guaranteed maximum latency of one microsecond, the data is output to the MUXbus with the 16-bit ID tag.
- Input modules do not buffer input data. Data is identified and output to the MUXbus when it is received by means of hardware only and no software interaction.

2. The MUXbus broadcasts the data to all I/O modules on the bus.

- Using a broadcast mode, the MUXbus broadcasts the data word to all modules in the system on the same bus cycle on which it was received by the input module.
- The MUXbus uses a rotating priority arbitration, which guarantees every input module between 1M and 8M output cycles per second, depending on how many other input modules are in the system. Each input module receives 1/N of the 16M cycle/sec MUXbus, where N is the number of input modules.
- Each data word occurs on the MUXbus only once. In that one cycle, the data word is received by all receiving modules taking data from the bus. These receiving modules include embedded processors (FPPs), a data bridge to the VMEbus, MUXnet (real-time network) I/O modules, analog and digital

output modules, as well as many others. Multiples of each of these modules can be present without any change in MUXbus performance.

3. Each MUXbus receiving module inputs only the data it needs from the MUXbus.

- All receiving modules have hardware-decoding RAMs for selecting which data words to take off the MUXbus. These RAMs are programmed to select only those data words needed by the module.
- The modules compare the ID tag of the data word with the contents of its tag decoder RAM, and, if there is a match, the data is input into the receiving module's input FIFOs.
- The data is immediately processed by the receiving module. Output modules, like the Parallel I/O and MUXnet I/O, output the data immediately without any processing. Other modules, like the FPP and the A/D module, process the data immediately.

4. Embedded processors (FPPs) perform real-time processes on the data.

- The FPP constantly monitors its input FIFO for new data. It is not interrupt-driven and does not wait for more than one data word to arrive before it starts processing.
- For performing real-time processes, the FPP vectors immediately to the algorithm needed by the data word that was read from its input FIFO.
- As soon as the data word is processed, it is output back to the MUXbus with a new 16-bit ID tag.
- The FPP contains canned algorithms, like EU conversion, limit checking, bit manipulation, etc. It can also be programmed with user algorithms written in C.
- Multiple FPPs can be installed in a system. The existence of an FPP has no impact on the performance of other FPPs. This provides an efficient parallel processing environment with a linear increase in computing power as additional FPPs are added.
- FPPs can perform different processes on the same data words or the same/different processes on different data words.
- All data routing to and from FPPs is handled in hardware, with no software intervention.

5. Embedded processors (FPPs) collect data for network distribution.

- FPPs are also used to collect data for distribution on Ethernet, FDDI, or ATM.
- One sample, all samples, or a statistical sample is collected for each data word needed by the network.
- The real-time (VxWorks) VME CPU periodically reads the data from the FPP.
- The VME CPU outputs the data directly to the client processes on the network.

6. The Data Bridge buffers data for storage.

- The Data Bridge has its tag decoder RAM programmed to collect those data words that are to be archived to disk.
- For efficiency and performance, the Data Bridge buffers data in a 4 MB RAM FIFO.
- When 2 MB of data have been collected, the Data Bridge interrupts the VME CPU.
- The VME CPU programs the SCSI Disk Controller to read the 2 MB block from the Data Bridge to the disk.
- The SCSI Controller then DMAs the data directly from the Data Bridge to itself across the VMEbus and out to disk.
- The sustained data rate to disk is 36 MB/sec and is independent of the number of input, output, and processing modules required by the system.

7. The MUXnet Module outputs data words directly to external workstations and PCs.

- The MUXnet Module is typically set up to collect all data on the MUXbus.
- When the data word and ID tag are received from the MUXbus, they are immediately output on the MUXnet. No processing or network protocols are involved.
- Latency from the MUXbus to the MUXnet Receiver Module of the workstation is about 1 microsecond.
- The receiving workstation can filter the data with its own on-board tag decoder RAM for input into its local FIFOs, or it can read the most recent value of any desired data word in the Current Value Table of the MUXnet Receiver Module.
- Using the MUXnet Output Module, data can be distributed in real time to an unlimited number of workstations over distances up to several miles.
- MUXnet is a bi-directional interface, so workstations can also output data back to the MUXbus.
- The MUXnet Module handles all I/O in hardware; no software is involved. The number of MUXnet modules in a system, or the number of workstations connected to MUXnet, has no impact on the performance of other modules.
- The MUXnet eliminates all bottlenecks in distributing real-time data to real-time applications running on a distributed network.

Figure 6 illustrates a high-end data acquisition system with multiple types of inputs and with multiple client workstations and applications. Because data movement in the system is hardware-driven and independent of software, the components of the system operate independently. This makes the system extremely predictable and deterministic. The storage rate to disk is 36 MB/sec, independent of the number of input sources, types of input sources, number of output destinations, number of embedded processors, etc. The storage rate is always predictable because it is independent of software and other hardware modules.

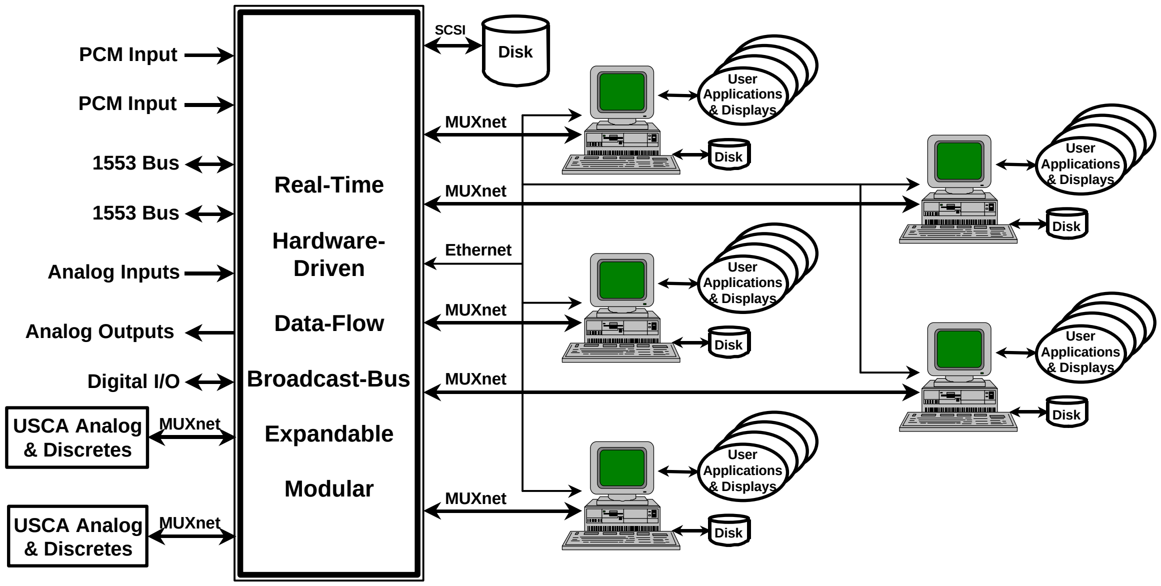

Figure 6. High-End Data Acquisition System with Multiple Data Sources & Client Workstations

Likewise, one input source has no impact on another input source, and destinations for the data have no impact on the input sources. Input sources require no software for identifying the data and placing it on the broadcast bus. Input sources also need no knowledge of the destination of the data, thereby completely decoupling them from any interaction associated with where the data is going. Modules that receive the data need not know where the data originated or the format in which the data was buffered. Receiving modules only need to know the ID tag of the data, thereby decoupling them from the input sources.

The only shared resource is the MUXbus itself, which is extremely predictable. It provides 16M data samples per second, with all input modules guaranteed an equal share of the bandwidth. All of these factors combine to make the system predictable, deterministic, flexible, and very high in performance.

The system also provides other features. The MUXbus automatically merges and time correlates data because each data word is immediately identified and broadcast onto the MUXbus as soon as it is received, with an overall latency on the order of one microsecond. This allows the data to be accurately time stamped in hardware as it arrives on the MUXbus. Since the data is not blocked, the merged data from all sources can be played back from the hard disk in the same time sequence and time intervals as it occurred in real time. This enhances post-processing and playback to strip chart recorders.

CONCLUSION

The independence of the input, output, and processing modules, with no required inter-process communication, makes hardware-driven systems very modular and makes system software very simple. If new capabilities or higher rates are needed, the required modules can be added with known results and without added complexity. As new modules are required, the impact on the system and the added performance are predictable. Parallel processing, parallel data paths, and distribution of data to other systems is easily accomplished with deterministic results.

These factors permit configuring a system that can be easily matched to almost any application's requirements. Because of the modularity and independence of modules, various configurations are made with no change to software. This provides low-risk and cost-effective solutions for many real-time data acquisition applications. It also lowers the cost of future upgrades and long-term support.

In summary, for high-performance and/or complicated real-time data acquisition requirements, a system with a hardware-driven architecture eliminates the bottlenecks, unpredictable performance, and configuration and/or data rate restrictions that are typical in conventional computer-based software-driven architectures.