

Digital Processing and Buffering

One of the critical considerations in configuring a data acquisition system involves moving the data from the ADC(s) or digital input device(s) into the computer for processing and generally saving the raw or processed data to some storage device. For control applications, the sequence is getting the data in, running the control algorithms, and outputting the control values. The initial and most obvious question in configuring systems is whether the data paths between the front-end to the processor, and the processor to the storage medium, will sustain the required throughput. What is often not adequately addressed are the sources and effects of latency and how they affect overall throughput. For example, a bus with 100 Mbyte/second throughput is useless for an application that needs to transfer 4bytes of data every 10 μ s (0.4 Mbyte/second throughput) if it shares the bus with a device that can acquire and hold the bus for 15 μ s—or, if a processor is involved, the interrupt latency of the processor may exceed the time between samples.

There are many techniques for addressing these issues. Included are buffering, the use of dedicated processors, dedicated direct memory access (DMA) controllers, and configuring system I/O buses to conform to the application requirements. This chapter will explore some of the issues and techniques for addressing these problems.

Throughput

Throughput is a measure of the rate that data that can be moved from one point to another in a system and is typically measured in bytes per second. Most resources, such as disks, processors, and buses in a data acquisition system, are *shared* at some level among various tasks, processes, and/or devices. When throughput is specified, it is usually given for a dedicated service. Thus, when a bus has a specified or *theoretical* throughput of 40 Mbytes/second, the specification is for a dedicated transfer between two devices on the bus with the bus limiting the transfer rate. It assumes that no other devices are contending for the bus and that devices respond to bus transaction requests in minimum allowed times. Generally, this throughput is fairly easily measured or calculated.

Latency

Latency is a measure of the time it takes for a device or process to *initiate* a transfer of data from one point to another in the system. There are two components to latency, the *inherent* time required to initiate the action in an otherwise idle situation, and the time required to acquire and access shared resources. The first is determined by the design of the device or process being considered, while the latter is typically dependent on the detailed configuration and application at hand. In many cases latencies resulting from shared resources dominate.

Latency is frequently *much harder* to quantify than throughput, since, in many cases, obtaining a precise signal indicating when a request is initiated and when the transfer starts is not easily obtained. To complicate the issue, latency is very dependent on what other processes and/or devices are performing concurrently. Also, since most processes

run asynchronous to each other, the effects of latency on a given process are usually statistical. Thus, to fully understand latency issues, making thousands of individual latency measurements is necessary and then histogramming the results. In terms of quantifying the results, such statistical measures as *average latency* or in some cases *worst case latency* under a given (known) loading condition is sensible.

In any case, latency has two major effects: it limits the *realizable* throughput and the guaranteed minimum interval that a device can be serviced. In data acquisition systems, worst case latency determines the depth of buffers needed to prevent loss of data, and average latency determines the *realizable* throughput.

In control systems worst case latency, along with throughput and control loop execution time, determines the minimum *guaranteed* time to close the loop. In many control applications it is frequently acceptable if the loop is closed "most" of the time within some specified interval. However, an occasional longer interval is acceptable if the loop is closed within some specified upper bound.

Operating System Latency

The *worst* latency offenders are, in fact, operating systems. The Operating System (OS), after all, is the arbiter of one of the most frequently shared resource elements, the CPU and its related resources such as memory, disks, network interfaces, and other I/O devices. The OS arbitrates between various applications and I/O processes within the computer system, and manages priorities between many concurrent processes as well as its resources. The data acquisition or control processes are just part of the larger picture. Even in a relatively dedicated application there are typically many related processes including I/O, data processing, data management, networking, and operator interface, each with its demands on a different set of resources under the OS.

The issue of interrupt latency is a relatively simple example. For typical real-time OS and processor, interrupt latency is usually under $10\mu s$ (average). Now consider a typical data acquisition application that is taking data, storing it to disk, and is likely connected to a network. Usually in such a scenario the worst case latency is several hundred microseconds to one millisecond, depending on the OS and the processor. This may seem excessive, but important factors are that the data acquisition interrupt may get delayed by the interrupt processing time of the disk, network, or system clock service routine; that the interrupt might be a result of some exception that requires extra processing at elevated priority; and that the system clock will usually have priority over any other interrupt and will almost always be serviced before your data acquisition interrupt.

A careful review of exceptional claims in this area is important. The computer sales business is *very* competitive and obtaining meaningful numbers from computer vendors is not easy. One can frequently get the "interrupt latency" but it's most likely the average for an unloaded system (best-case latency). Those who claim absolutely *super* numbers are using dedicated processors to achieve them (all the rest of the OS runs on another processor).

At the process level where one is writing relatively normal application code, latencies can be significantly higher.

Buffering and How It Helps Overcome Latency

In general, the basic data acquisition devices, such as ADCs, generate data (converted values) at rates that are based on a hardware clock. The rate may range from sub-microseconds to seconds, but when the "clock ticks," data is sampled, converted, and presented at the output following the conversion time. The data will generally remain **valid** either until the next clock or the end of the next conversion. In any case, there is a fixed time interval that *something* must respond and move the data to a storage device.

Depending on the architecture of the data acquisition system, this scenario is repeated multiple times, whether it be for a single sample from a single channel; a buffer of data corresponding to all the data from multiple channels at a single time interval; or a buffer covering multiple time intervals. In any case, there is a finite time interval where *something* must respond and move the data before the next block starts. Failure to respond in time means *lost data*.

The simplest solution is to interrupt the processor each time the ADC has a value and have the CPU place the value in a buffer (block of main memory), and when it is full write the buffer to disk or other permanent storage. This technique works until the interval between ADC conversions approaches the worst case latency of the application software in the CPU. From this discussion of latency, this data rate is relatively low. Buffering of data at the hardware level is the technique that permits data rates approaching the I/O bandwidth of the CPU or storage device, whichever comes first.

In modern data acquisition systems, several buffering techniques are used and may occur at multiple levels, including at the ADC module, at the I/O chassis controller level, at the host computer interface level, within the host computers main memory, or within the disk controller. The purpose of buffering is simply to provide a place to store data for a period that is *long* compared to latency associated with accessing the affiliated device.

For example, if the ADC conversion time is short compared with the worst case bus latency that the ADC uses to transfer data, then *the ADC must include sufficient buffering to insure no data is lost*. If the device that initiates the transfer of data from the ADC is a processor, then the amount of buffering must be consistent with the worst case latency of the processor plus I/O bus.

Buffering Techniques

A number of buffering techniques exist. The simplest is double buffering. In this case two buffers are used. When one is filled, a flag or interrupt is posted indicating that the first buffer is full, and the device, e.g., the ADC, begins filling the second buffer while the first is being emptied.

Multibuffering

A generalization of double buffering is *multibuffering*, where a number of buffers N with $N > 2$ of equal size are defined. The I/O device fills them in order, signaling the processor as each is filled. It is the processor's responsibility to ensure that each buffer is emptied before the I/O device is ready to fill it again. The advantage of multibuffering is that when dealing with *very long* worst case latencies, but relatively shorter average latencies, the processor can act on the data faster *on the average*.

Circular Buffering

Circular buffering is a variation on multibuffering, where each of the buffers appear sequentially in memory. The data acquisition device simply treats the multibuffers as a giant single buffer and resets itself to the head of the buffer when it reaches the end. The advantage is a simplification of the hardware with no loss of generality from the software side. In this configuration, a good practice is to provide a flag for each buffer segment so the hardware can set the flag when the buffer is filled; software can clear the flag when it has emptied the data; and the hardware (and software) can detect a buffer over-run condition should the software fail to keep current.

Ping-Pong Buffering

For some configurations, "Ping-Pong" buffering can be advantageous. This is simply a form of double buffering the data associated with each "tick" of the sample data clock. For example, in a multiplexed ADC with a 10 kHz sample rate per channel, when the 10 kHz sample clock "ticks," the ADC presents one side of the Ping-Pong buffer to the I/O bus while it digitizes values into the other side. Since ADCs do not digitize values instantaneously, this technique gives the processor that is collecting and buffering the data a *full clock period* to move the data.

The Ping-Pong technique can be equally effective on a multi-channel Digital-to-Analog (D/A) converter, in that the processor can load the next set of values for all the channels, and then at the next "clock tick," all the updated values become true. Again, the processor has a full clock period to update the values.

FIFO Buffering

First-In-First-Out (FIFO) buffering basically uses a dual-ported memory where data is clocked in one port and clocked out the second port. These memory devices are typically implemented with RAM memory and two address pointers that keep track of where the next input is to be stored and from where the next output is to be taken. Flags are typically provided for *full*, *half-full*, and *empty*, FIFOs are particularly useful for buffering internal to hardware and can be effective as buffers in data acquisition.

FIFOs have two limitations. First, they are generally only available in limited depths. The other limitation is that if the data acquisition process ever gets out of synchronization or the FIFO overflows, the only way to recover is to stop the data acquisition process, clear

the FIFO and restart. In other buffering techniques, simply skipping over the buffer in error to regain synchronization is sufficient.

Using Dedicated Processors to Overcome Latency

In some applications, primarily control, a necessary step is to *initiate action* based on current real-time data. Buffering techniques *do not* solve this problem. The required procedure is to access the input data, make some determination based on the data, and generate a response within some fixed known time interval. When this time interval is shorter than the worst case system latency (e.g., the computer OS latency), then a dedicated processor may be a viable solution.

Several approaches are possible. One approach is to use a dedicated processor with a real-time kernel OS. Here the "dedicated" processor can frequently be used for several related real-time tasks. The real-time kernel operating systems are generally more deterministic and exhibit somewhat lower latencies than larger general purpose operating systems, but one must remember that, *they also have significant latencies and ultimately are subject to the same phenomena that cause latency in general purpose operating systems*. The primary advantage of the real-time kernels is that the user has better control over the operating system and its behavior.

The ultimate solution to latency is the *truly dedicated processor* where the application is the *sole* process being executed. Typically, these applications have either no operating system, or just enough of an OS to get the application loaded and to provide some very basic services like I/O drivers and possibly some debugging tools.

In some cases (especially where very fast responses are required) the only solution may be hardwired logic. For example, it may be required to trigger an event within microseconds of an analog signal exceeding a specified threshold. This is a relatively simple feature to implement in an ADC. Such a feature could be used to provide a *trigger* to initiate the storing of the data in memory, or as a signal to initiate a shutdown of a critical process.