Planning a Computerized Measurement System

An introduction to digital processing of analog signals

Instead of peering at a wandering meter dial or at a squiggly line on a strip chart, why not let a computer get the eyestrain? Better yet, why not let the computer process that information as well as collect it?

The key to getting the most out of computerized measurement is careful planning. You need to: decide how often to sample the data signal and with what resolution, choose what computations will be performed by the computer and which ones by custom analog or digital circuitry, and decide what information to process in real time (while the measurements are being made) and what information to save for later processing.

Sampling Data

A measurement system must usually deal with a signal that is the voltage analog of some physical variable. To perfectly record such a signal, you would need to note its exact value at every moment. This isn't practical, so the task is to take samples of the signal with enough frequency and accuracy that you can store a reasonably close facsimile of the signal. However, how frequent is frequent enough?

The Nyquist sampling theorem says that the sampling rate (the number of data samples taken per second) should be at least twice the frequency of the sampled waveform. Consider, for example, a periodic signal with a fundamental frequency of about 3 Hz (e.g., the arterial blood-pressure waveform of someone exercising). If you were to accept a reconstruction consisting of the fundamental and up to the fifth harmonic (18 Hz), then you would need to sample the signal at least 36 times per second. As another example, consider a digital audio system. It would have to reconstruct signals of up to 20 kHz, so it would need to have a sampling rate of 40,000 per second.

Even a high sampling rate is not a guarantee of accurate signal reproduction. Accuracy also depends on the resolution of the A/D (analog-to-digital) conversion. In other words, if you have an A/D converter that cannot detect voltage differences of less than one-half volt, then you should not expect great accuracy when you feed it a signal that varies from 0 to 0.75 volt. A 12-bit A/D converter limits your ability to reconstruct a signal to 1 part in 4096 of the full range of the converter. Thus, if the 12-bit converter has an input range of −10 to 10 volts (so that it has a resolution of about 4.9 millivolts) and your signal varies from 0 to 0.5 volt, your maximum accuracy is about 1 percent.

The choice of sampling rates is one of the really critical parts of planning a computerized measurement system because the choice sets the limits on the amount of signal processing that any particular computer can do. Doubling the sampling rate extracts two penalties: it doubles the overhead associated with servicing the A/D coverter, saving raw data, and so on; and it can more than double the processing time associated with making computations on the data. Thus, you should always choose the lowest sampling rates that will allow acceptable resolution of the features of interest in signals.

Note that errors occur when your system monitors signals that have frequency components above twice the sampling frequency. Aliasing occurs when the sampling rate is slightly different than some multiple of a high frequency component of the signal. When that happens, a spurious low-

frequency signal appears to be present. Even if aliasing does not occur, frequencies in the signal above the sampling bandwidth appear as noise in the sampled data. The solution to these problems is not to increase the sampling rate, but rather to decrease the signal's bandwidth before it is sampled. I will discuss next how this and more may be accomplished.

Analog Preprocessing

Useful signal-processing systems existed before the advent of electronic digital computers. Millions of hours of effort have been directed at designing analog circuits that perform computations on the analog representation of measured variables; analog computers built from such circuits are still used for some types of modeling tasks.

Virtually all measurement systems do some analog processing. For example, the conversion of a physical variable into an electrical signal is an inherently analog process; this transformation is usually accomplished by a special-purpose device—a transducer—and associated electronics. The transducer and its electronics are often purchased as a package over which a user has little control. Consequently, most such packages do not produce the optimal signals for digestion by the computer in any particular setting.

The signal from the transducer electronics is unlikely to be the best size for your A/D converter, or you might be interested in only a restricted part of its full-scale range. Very often the transducer bandwidth will be too high for the appropriate sampling rate, or high-frequency noise will have crept into the signal. Thus, every A/D converter in a general-purpose measurement system should have some means of adjusting the range, offset, and bandwidth of signals. Figure 1 illustrates a simple analog circuit that performs these services for noncritical applications.

The operational amplifiers that you use in this circuit are not critical for the component values shown. The first stage of the circuit has a gain of negative one and provides input zero offsets from −10 to +10 volts. This

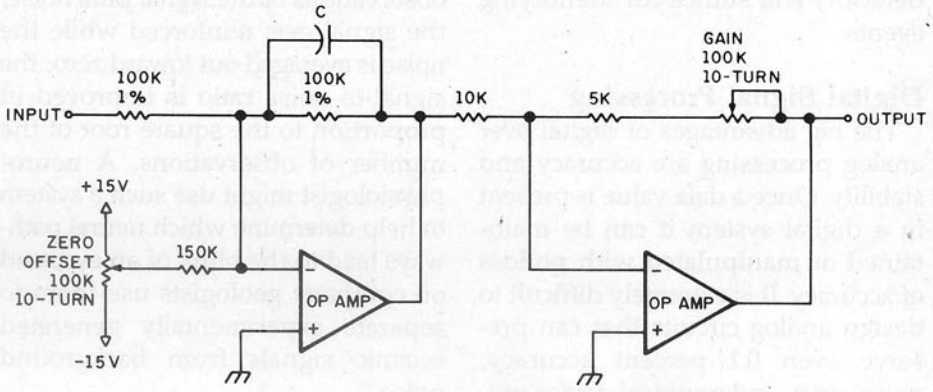

Figure 1: *An analog signal-conditioning circuit.*

stage also acts to limit high frequencies via the first-order, low-pass filter created by the capacitor, C, in the feedback loop; a value for C (in microfarads) of 3.2 times the sampling interval (in seconds) will result in 3-dB (about 30 percent) signal attenuation at one-half the sampling rate and 6-dB (one-half) per octave roll-off at higher frequencies. The second stage provides adjustable gain from 0.5 to 10; if zero offset is adjusted first, the gain and zero will not interact. Since each stage inverts the signal, the output has the same polarity as the input. Input impedance is 100 kilohms. For circuit-error analysis or more sophisticated designs, see references 3 and 4.

Analog processing can reduce the required sampling rate or relieve the computer's processing load. As an example where analog processing might be useful, consider a situation in which you need to know the peak pressure generated by an explosion. The required sampling rate might be more than 10 million samples per second and thus beyond the capacity of all but extremely expensive, special-purpose processors. An analog circuit called a peak detector would allow even a slow computer to sample the peak value at its leisure.

Analog processing can also improve accuracy in some cases. Suppose you are interested in the difference between two signals of almost equal magnitude. Using a computer to do the subtraction could leave only a few bits of accuracy left in the difference, and you have doubled the computer's work load by making it sample two signals instead of one. An analog differential amplifier would perform the subtraction and preserve computer processing time and accuracy.

Hundreds of analog circuits have been designed to do all sorts of useful processing, and for very fast computations they are usually the method of choice. As a general rule, where an analog circuit is available to perform some calculation, it will do it faster, but less accurately and less flexibly, than a computer. References 1 and 2 will get you started with analog-circuit design, and references 3 and 4 give a number of circuit designs for analog processing.

One other class of signal preprocessing—event counting and timing—needs to be discussed. Using a computer for event counting and/or timing is like using a howitzer to kill flies. There are inexpensive digital circuits designed expressly for the purpose of counting and timing events and then feeding the results to a general-purpose computer. Many A/D converter cards for popular microcomputers contain at least one such circuit, and expansion cards designed to count and/or time multiple events are also available. It is generally appropriate to use a computer for event logging only when you have computer power to burn or when an unusual amount of signal processing needs to be done to determine what constitutes an event. In most situations, the simple 1-bit A/D circuit called a comparator (i.e., a level

detector) will suffice for identifying events.

Digital Signal Processing

The big advantages of digital over analog processing are accuracy and stability. Once a data value is present in a digital system it can be maintained or manipulated with no loss of accuracy. It is extremely difficult to design analog circuits that can preserve even 0.1 percent accuracy; noise, drift, and nonideal device properties are a constant challenge.

Computations that require the accumulation of information over long time periods are especially difficult to implement with analog circuits because electronic storage elements (capacitors) are limited in size and number for practical designs. Digital processing is almost always the way to go when data needs to be held for more than about 30 seconds. I once needed to generate a very slow (20 minute) voltage ramp for controlling an experiment; the analog circuit to do this is theoretically trivial, but the practical solution was not. It was easier to use a digital counter to produce a numerical ramp combined with a digital-to-analog converter to get the voltage.

Because digital circuits are so good at holding data, signal processing tasks are now possible that were impractical when only analog delay-lines or tape loops were available for storing data. These tasks are those that require looking back in time. Thus, a digital processor can conveniently make computations with respect to an event based on signal values sampled substantially before the event.

An example of digital processing that illustrates most of its strengths is the computation of average transients. In many real-world situations, the desired signal is buried in noise that cannot be eliminated by analog filtering (if the signal and noise have similar frequency spectra, there is no general way to filter out the noise without filtering out the signal as well). The average transients technique comes to the rescue when there are multiple chances to observe the same signal. By averaging many observations of the signal plus noise, the signal gets reinforced while the noise is averaged out toward zero; the signal-to-noise ratio is improved in proportion to the square root of the number of observations. A neurophysiologist might use such a system to help determine which neural pathways lead to the blink of an eye, and oil company geologists use them to separate experimentally generated seismic signals from background noise.

Using a programmable computer to do digital processing confers two vast advantages over custom-designed digital circuitry—flexibility and ease of development. These advantages save so much time and money that only defense contractors are likely to be found designing custom digital circuitry when a programmable computer could do the job.

It is extremely difficult to design analog circuits that can preserve even 0.1 percent accuracy.

In the rest of this article I will assume that you are using a general-purpose mini- or microcomputer for signal processing. However, you should be aware that microprocessors designed specifically for real-time signal processing have been available for the past few years. These processors usually have on-chip 8-bit analog-to-digital and digital-to-analog converters, hardware multipliers, and very short cycle times. The processor designs are generally optimized for digital filtering (see reference 5), but they might also be used as an alternative to analog processing in numerous other applications where signals have frequency components up to about 100 kHz.

Real-Time versus Batch Signal Processing

Let us assume that you have decided what needs to be measured, applied appropriate analog and/or digital preprocessing, arranged for data sampling at close to the optimal (i.e., lowest) rate, and have some data processing in mind for the computer to do. At this point you need to decide what calculations to do as the data is being collected (in real time) and what needs to be saved for later processing in off-line or batch mode.

Any result used for feedback during a measurement session clearly needs to be calculated in real time. These results might be used in experimental control or perhaps to provide a visual display as the data is being collected. At the other extreme are the situations where you haven't decided just what sort of analysis you will want to apply to the data or where the hardware or software necessary for an analysis is not available. In these cases, data must be saved for later analysis. When the choice between real-time and batch data processing is not clear, the following points should be considered.

The advantages of batch processing of measured data are twofold. First, you do not need to decide before every measurement just what analysis will be done. In most cases, data processing that occurs in real time (whether done by analog circuits, digital circuits, or a computer) reduces the information content in the signal and thus precludes some types of later analysis. Saving raw data for batch processing can therefore allow much greater flexibility and can possibly save having to rewrite a real-time data collection program every time a new type of data analysis is desired. The second major advantage of batch data processing is the ability to perform computations that require more time than is available in real time. Complex digital filtering, statistical or correlation analysis, and most types of mathematical transforms, for example, all require more processing time than is available in many measurement situations.

The advantages of real-time data processing are also twofold. First, the amount of data that needs to be printed or saved on mass-storage devices is usually greatly reduced, since only results need to be saved rather than all of the data that had to be sampled to resolve the features to be analyzed. If your required sam-

pling rate exceeds your capacity to save data, this consideration will mandate at least some real-time data processing.

The second advantage of real-time data analysis is that it relieves you from having to do it later. This may seem obvious and unimportant, but it can have a major impact on overall computer usage. In one application, I was able to reduce the batch-processing time for a 3-hour experiment from 6 hours to a few minutes by saving only data that had been partially preprocessed in real time. (The apparent magic involved in doing 6 hours worth of processing in 3 hours of real time resulted from the fact that the real-time program had to be efficient, while the batch program didn't have to be and wasn't.) As a rule of thumb, I normally consider any measurement situation with more than 100,000 data values per hour to be a good candidate for some real-time processing to reduce the burden of data storage and subsequent batch processing.

How do you determine if a particular analysis task can be done in real time? Time constraints will determine the answer, and I have a few simple rules to help you make an estimate. The first step is to estimate the minimum sample cycle time. This is the time required to collect a sample, save it somewhere, and do all associated housekeeping. This time depends on both hardware and software; it might be as low as 10 microseconds for a fast analog-to-digital converter being controlled by a well-written assembly-language program, or it might be 10 milliseconds for a simple converter being controlled from an interpreted BASIC program. This time sets an absolute upper limit on the total sampling rates of all sampled channels and, thus, on effective measurement bandwidths.

Next, you have to estimate the average processing time you need to execute your data-analysis algorithm. It is often easiest to do this by writing a test program that applies the algorithm to a known quantity of dummy data. The average sample cycle time can then be computed as the sum of the minimum cycle time and the average algorithm-processing time needed for each sample. This time sets the absolute limit on the data sampling rates that can be maintained while still performing the desired analysis.

Finally, examine your algorithm and all real-time tasks that you plan to do to find the most time-consuming set of conditions that could be encountered; we'll call this the maximum sample cycle time. This time will normally be much longer than the average sample cycle time since occasional tasks, such as updating a screen display or dividing sums to get averages, can consume many machine cycles.

If you write only the most straightforward data-collection program, your sampling rates will be limited by the maximum sample cycle time. In this simplest programming approach, each sample is collected and analyzed before getting the next sample. Thus, you need to leave enough time between samples to do

your most time-consuming jobs or you will lose data. An alternate but trickier programming approach lets the sampling interval approach the average sample cycle time by collecting and saving samples asynchronously with respect to other processing tasks. For more information about efficient real-time programming techniques, see references 6 and 7.

A Real-World Example

To illustrate and summarize the principles outlined in this article, I will briefly describe BEAT, a real-time program I wrote to monitor experiments in cardiovascular physiology. The overriding characteristic of cardiac signals is that many of the most interesting things happen very fast (as the heart contracts) but not very often (once per cardiac cycle). BEAT was designed to capture this information from each heartbeat.

I measured blood flow and pressure in the ascending aorta (the big artery leaving the heart) using specialized transducers with associated electronics. Analog circuits scaled and filtered the signals as needed by the 12-bit A/D converter and the sampling rates. The 12-bit accuracy of the converter was more than sufficient, since the transduction and analog processing left the signals with an accuracy of only 1 percent.

Ascending aortic flow is nonzero for a relatively brief part of the cardiac cycle (for about the 150 milliseconds when the left ventricle is contracting and the aortic valve is open). Since the peak flow is stable to within 1 percent for only about 4 milliseconds, I chose a sampling rate of 250 per second for this channel (a bandwidth from DC to 125 Hz). Due to the dynamic characteristics of the arterial tree, aortic blood pressure changes much more slowly than aortic flow; a bandwidth of only 30 Hz allowed reconstruction of the pressure waveform to well within 1 percent accuracy. Thus, the arterial pressure channel was sampled at one-fourth the rate of aortic flow samples. The raw sample rates for these two variables plus others I measured were about 1000 per second. The raw data from a typical 5-hour experiment would

have required about 36 megabytes of mass storage. It was a clear candidate for real-time processing.

I needed to calculate: the cardiac cycle length, the integral of aortic flow minus its value just prior to a heartbeat (the stroke volume), the average pressure, the systolic and diastolic pressures, the maximum rate of change of pressure, and similar sorts of information about other measured variables. BEAT extracted this information from the raw data samples for each heartbeat and then saved the computed values on a mass-storage device.

The analog and custom digital circuits I would have needed to duplicate my computerized system would have been complex, inflexible, and difficult to calibrate. For instance, the stroke volume computation required looking back after detecting a heartbeat to the period when aortic flow was known to be zero. Current flow transducers suffer from slow zero drift, so an analog circuit to compute stroke volume would have been an endless source of frustration. Reference 8 outlines the problem in more detail and explains some other tricks we can do with the computer and aortic flow.

The processing time necessary to sample data, keep running sums, compute extrema, and so on, required about one-fourth of the available machine cycles. However, at the end of each cardiac cycle, BEAT had to perform tasks that required about 150 milliseconds. These tasks included preparing the extracted data for writing to mass storage and passing the data along to the device controller, computing a number of derived variables (heart rate, the current time, average pressure, etc.), converting the derived variables to engineering units and displaying them on a video monitor, echoing the derived values as analog voltages via a set of digital-to-analog converters, testing the keyboard for any special instructions from the user, and initializing in preparation for processing the next beat.

The time required for end-of-beat processing presented a programming dilemma. I might have ignored 150 milliseconds of the measured variables after the end of each beat (thereby missing the most important part of the cardiac cycle), or I might have forgone some or all of the end-of-beat processing. Both of these unpalatable alternatives were avoided, however, because even at the highest observed heart rates, only about 85 percent of the computer's time was needed to do all sample and end-of-beat processing. The problem was to somehow keep sampling data at the proper times even when the computer was busy with the time-consuming end-of-beat tasks.

The job of independent data sampling was relatively easy for BEAT because the A/D converter was a smart device that could be programmed to collect samples and save values in its own memory. Analog-to-digital converter systems with this sort of independent processing capability are becoming commonly available; for example, Data Translation makes them for the IBM PC that sell for $1200 and up. In the more usual

case where a smart analog-to-digital converter is not available, the same independent effect can be achieved with a simple service routine that is called via hardware interrupts generated by a clock. The usual time cost for such servicing might be 50 to 100 microseconds for one channel plus 10 to 30 microseconds for each extra channel sampled.

The BEAT program was largely written in a relatively inefficient compiled version of BASIC; only one small routine to maintain the sample buffers needed to be written in assembly language. This illustrates the point that you don't need to be an expert assembly-language programmer to write useful measurement-monitoring programs. Once you have written (or gotten someone else to write) the possibly tricky routines to sample signals and maintain a well-organized buffer of sampled values, then, with only reasonable care and programming skills, you can turn your computer into a flexible data-collection and signal-processing tool.■

References

1. Mileaf, H. *Electricity One—Seven*. Rochelle Park, NJ: Hayden, 1978.
2. Cirovic, M. *Integrated Circuits: A User's Handbook*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
3. Sheingol, D., ed. *Transducer Interfacing Handbook: A Guide to Analog Signal Conditioning*. Norwood, MA: Analog Devices, 1980.
4. Stout, D., M. Kaufman. *Handbook of Op Amp Circuit Design*. New York: McGraw-Hill, 1976.
5. Grappel, R. "Digital Filters Utilizing Microprocessors". Ch. 11 in *Microprocessor Applications Handbook*. Edited by D. Stout. New York: McGraw-Hill, 1982.
6. Mellichamp, D., ed. *Real-Time Computing—With applications to data acquisition and control*. New York: Van Nostrand Reinhold, 1983.
7. Wyss, C. "A Conceptual Approach to Real-Time Programming". BYTE, May 1983, page 452.
8. Wyss, C., T. Bennett, A. Scher. "Beat-by-beat control of cardiac output in awake dogs with atrioventricular block." *American Journal of Physiology*. 242 (Heart Circ. Physiol. 11): H1118-H1121. 1982.

Designing Systems for Real-Time Applications

Some pointers to keep in mind before you tackle a real-time design

A real-time system is one that responds immediately to your commands and processes data as soon as the data is produced. The opposite of a real-time system is a batch system, which batches commands together and processes them en masse, with no concern for immediate response.

Most personal computer users are familiar with applications that require some real-time operations. For instance, simple text editing or word processing must provide timely response to keystrokes and commands to prevent user frustration. Games require even more demanding real-time characteristics.

Applications that involve audio-frequency monitoring are quite demanding. To accurately sample data where the frequency is 15,000 Hz (moderately high fidelity—my stereo does 20 kHz, my ears do 12 kHz) requires 30,000 samples per second. Program-controlled sampling can often go no higher than 1000 to 5000 samples per second, so audio monitoring clearly requires specialized hardware and DMA (direct memory access) devices.

When designing a real-time system, you need to consider the performance of the hardware and the operating system. By hardware I mean the processor chip, the bus structure, the memory-management scheme, and the interfaces to the outside world. By operating system I mean the program that directly controls your hardware and acts as the interface between your application program and the hardware.

Hardware

Eight-bit processor chips are ideal for character processing and low-precision analog work. However, they have limited performance when greater precision or address space in excess of 64K bytes is needed.

Sixteen-bit microprocessors match the precision of the most common A/D (analog-to-digital) converters (with 10-, 12-, and 14-bit A/D converters most common). Unfortunately, microprocessors often run into address-space limitations when a large volume of data is needed.

The popular 32-bit microprocessors offer a large, linear address space, fast cycle times, and efficient program-processing characteristics. These are *essential* to success in real-time operations.

A factor in performance that goes beyond the speed of the processor chip is the delays the processor encounters in fetching data from memory. One example of this is the TRS-80 Model 16, which has a relatively slow 68000 processor but which encounters little delay in fetching data from memory because it has memory directly on board the processor. The Exormacs computer uses a higher-speed 68000 chip, but it has greater fetch delays due to the use of a standard (Versabus) memory interface and slower RAMs (random-access read/write memories). A second example is the Universe 68 computer. When using the memory bus (which is also a Versabus), it has one rate of processing. However, when using the 4K-byte cache memory, an onboard fast memory, it has double the instruction-processing rate.

The second hardware factor in designing a real-time system is the system bus. A good system bus allows a range of interfaces as well as sufficient bandwidth so that the system can accomplish all tasks within a reasonable time. If all the interfaces required are on the basic system (as in a single-board computer), the system bus is not an issue. But the variety of real-time applications and the special nature of some of the interfaces deem this unlikely.

Only three design choices are currently available for nonproprietary 32-bit buses: Versabus, VME, and Multibus. Versabus and VME have the largest number of available inter-

faces because they are older buses. Adapters permit the use of the smaller VME and Multibus boards in Versabus card cages, which gives this bus an additional short-term interfacing advantage.

A real-time system's memory-management design must be as carefully considered as the processor and system bus. Where performance is a critical issue, memory response time must be reduced. Some of the factors that influence memory response time are memory-management logic, bus drivers, dynamic-RAM refresh logic, and error-detection and error-correction circuits.

We shouldn't forget the real-time clock or timer circuit—traditional elements of real-time environments for good reasons. One of these is needed if time-related operations or rigid time intervals are involved. However, time-out logic is also needed to avoid deadlock and other error conditions. For example, in a game that uses multiple sprites (moving graphics image units), where the images associated with two different sprites attempt to affect the same item (say, eating a cookie) but only one can be allowed to do the task, a locking mechanism is needed. Either sprite A is locked out and sprite B has access, or vice versa. Given a bit more complexity, it's possible to have A locked out waiting for B, B locked out waiting for C, and C locked out waiting for A. With everything locked, all activity ceases. One way to detect this condition is a time-out that allows a program to regain control and determine if such a deadlock has occurred. Another way is for a task to release any items it has in case of time-out so that another task can continue.

In many real-time applications, the computer must be connected to a remote device (or another computer) via a serial port. This has become increasingly popular with the many low-cost processors being built into data-collection and display devices. RS-232C and RS-422A are the most common types of connections. RS-422A can operate over longer distances and with better noise immunity than can RS-232C. For example, 50 feet is the limit on an RS-232C interface operating at 9600 baud, but an RS-422A interface can easily run up to 4000 feet at this same rate.

The control lines in RS-422A and RS-232C cables are heavily used in real-time applications to provide essential flow control over data transmission. For many system configurations, read, write, and ground are the only required lines in a connection.

Some real-time system designs employ software routines such as the XON/XOFF sequence (usually Control-S/Control-Q) to control data communication flow. This routine sends a character back up the line to instruct the transmitting system to stop sending data. A second character indicates that data transmission is to resume. Full-duplex data transmission (simultaneous data reception and transmission) requires multitasking capability to monitor the receive data line while transmitting to ensure that a control signal or message is not received that could change or affect the transmission in progress. For those communications ports that do not support full-duplex communications (which prevents the use of the XON/XOFF controls), slower baud rates ensure successful transfers but hurt the real-time power of the system.

Some real-time tasks can be accomplished only by programs that directly control the computer.

The Operating System

Some real-time tasks can be accomplished only by programs that directly control the computer with no intervening operating system. However, as long as your task can tolerate the delay, it is better to use one of the standard operating systems now available.

The ideal operating system would have multiuser, multitasking, and real-time capabilities. Incidentally, multiuser development is advantageous even for single-user systems because it allows the execution of several simultaneous tasks, for example, editing one program while another is being compiled. Unix is an example of an excellent multitasking operating system, although it is not suitable for many real-time tasks. However, modified Unix and Unix-compatible systems that do provide real-time facilities are available. An example is the HP/Unix system, which has a real-time kernel written by Hewlett-Packard that interfaces at the system-call level to Unix. Another example is the Unos operating system.

For real-time tasks, the operating system must let the programmer have control of I/O (input/output) devices, physical memory, task priorities, exception processing, and data integrity. A programmer must control I/O devices to monitor and control the computer's peripherals. PEEK/POKE control is one form of this, but often a task is set up to run when an external condition changes, such as when the user pushes a button or the fire alarm sounds. For this, the system must provide an interrupt facility and the ability to pass control back to a specific user task.

Control over physical memory is needed to avoid swapping and to control data flow. Swapping, the automatic exchange of information between memory and a mass-storage device, can result in a change of memory location or, worse, in significant delays when that fire alarm sounds and the service task cannot be swapped back into memory. The ideal operating system would let the programmer inhibit swapping and identify and control physical-memory areas for buffering data from I/O devices. Sharing these areas between tasks can be essential when one task performs data input, a second scales the data, a third records the normalized data to disk, a fourth analyzes data for statistical evaluation, and a fifth uses the current statistics to control an external device. Shared memory and asynchronous operations are needed for this to work efficiently.

In the preceding example, the programmer must be able to control the priorities of tasks to make sure that

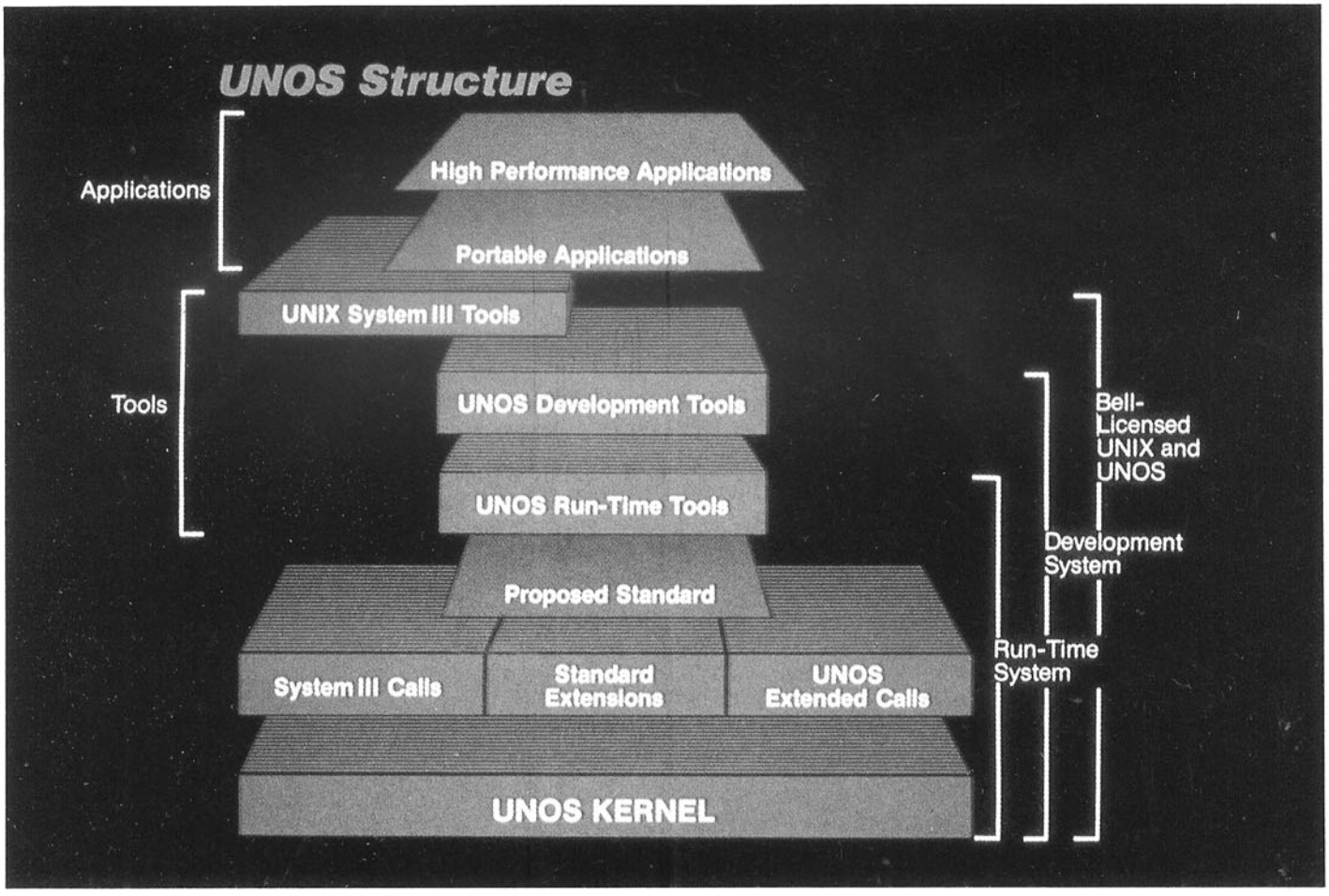

Figure 1: *An example of task scheduling by the Unos operating system.*

the real-time tasks are not stopped in favor of a less important task. Priority control mechanisms let the programmer decide and control which tasks get the processor. In the ultimate real-time system, each task is guaranteed a worst-case delay to ensure that the task has enough time to complete its job when activated. As with any other real-world situation, Murphy usually gets his way, and this has to be taken into consideration when setting task priorities. Some tasks simply may not be completed.

An operating system should provide dynamic priority-scheduling of tasks. In Unos, a Unix-like operating system, each task has a ceiling and floor priority with the numeric value of the ceiling greater than or equal to the floor. Normal timesharing users get a 100-10 assignment. A user can lower either value, but only privileged users can raise the values. If a timesharing task becomes compute-bound (completes a time slice without waiting for I/O), it is automatically dropped in priority so it will not interfere with interactive users. The example in figure 1 starts with the real-time task running at priority 200. A real-time task would have the floor and ceiling equal and swapping disabled for that task. When the task at the higher 255 priority is ready to run, it preempts the 200-level task and starts running. When both of these block, the processor is again available to time-slice between the timesharing users at priority 100. With these tasks blocked, tasks at levels 10 and 5 get an occasional shot. For tasks on the same level, round-robin scheduling is used, with either time slicing or blocking being the method of passing control. If a real-time task becomes compute-bound, it can use all the resources it needs to the exclusion of other tasks. This is essential in giving real-time tasks the control they need and explains why the raising of priority levels is a privileged operation.

If no abnormal conditions existed, exceptions would not occur, and real-time programming would be incredibly simplified. But the nature of real time ensures that many error conditions may occur. These require an additional level of asynchronous control by the programmer. In process-control-related tasks, lost control is fatal to the success of the application. This means checking for all obvious error conditions related to program operations (e.g., I/O errors and arithmetic overflow/underflow). Also, the program must be prepared to trap a number of other error conditions that are not related to its immediate operations. This includes conditions such as communications-line errors, disk and/or memory errors, and power failures.

Programmers can maintain control over data integrity if the operating system forces critical data to a disk in the event of a system failure. A power failure is only one example of an external event that can have a severe

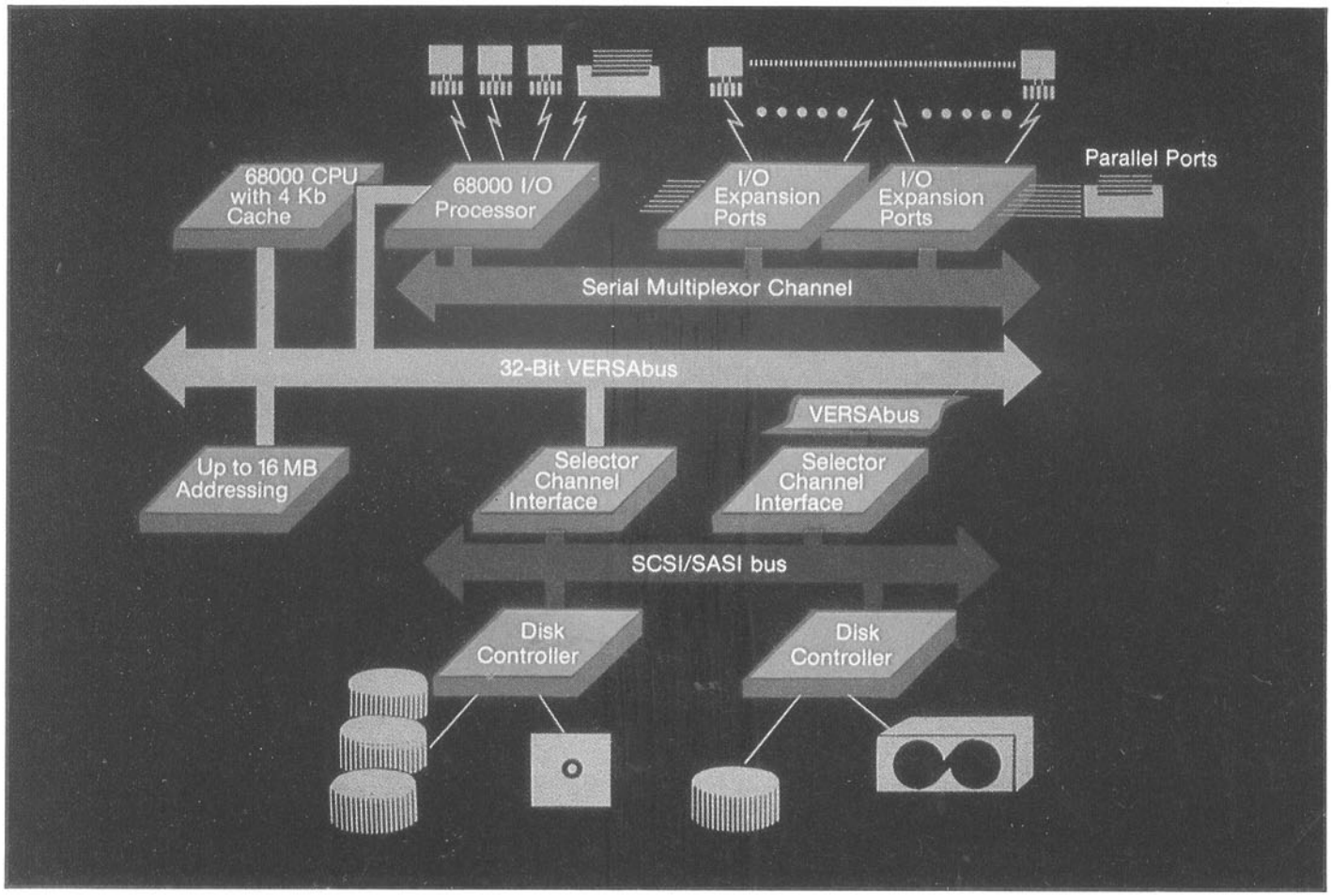

Figure 2: *The components of delay in a computer as it attempts to respond to a real-time event.*

impact on the integrity of the data being collected. You can expect to lose RAM to this or other failures; so, for real-time processes, a mechanism is needed for determining what data has been lost and what data is still valid. The worst case is exemplified by the many systems that have actually lost the integrity of their entire disk-file system after an abnormal termination. In that case, all data is lost, even that successfully written to and stored on disk. This can be avoided with data-integrity features in the operating system.

Performance Measurement

Two major areas of performance measurement are interrupt latency and context-switching time. Interrupt latency is the longest time that the computer takes to act on a single interrupt. Only the highest-priority interrupt (which is usually called nonmaskable because it must be acted upon, that is, it cannot be "masked out") is always acknowledged within the interrupt-latency period. Lower-priority interrupts must wait if a higher-priority interrupt is active.

Context-switching time is the time required to stop execution of task A, save that task state, and restore the state of and start task B. This switch might occur as a result of an interrupt, a synchronization signal between tasks, or a time slice. Note that the number quoted in some microprocessor ads simply reflects the switch between the user mode and system mode and does not include the overhead needed to enter a second user task.

If the real-time application requires sampling data at a high rate, the context-switch transition can be a limiting factor. A millisecond (ms) response time would limit processing to 1000 samples per second. To improve this, data can be buffered in the driver (with 50-microsecond (μs) overhead, or a limit of 20,000 samples per second) and the application task given control on every nth sample. Notice that the limits of 1000 or 20,000 samples per second imply that the system performs no other functions, no time-of-day updates, no disk I/O, and no computations. This, then, does not provide a practical limit but rather a way to measure the percentage of utilization that a specific application requires.

The interrupt latency and context switching, along with other factors such as the application program, determine the elusive and overused term—response time. It entails a few measurable time delays and a number of application-code-related delays. The result is a response-time measure that becomes quite dependent on the actual application, and it cannot be predicted in advance.

In figure 2, the delays encountered by an external interrupt are

1. The hardware delay in signaling the processor. Daisy-chaining of interrupts can delay this, as can the current bus cycle. Interrupt

masking, disable logic, and priorities can all add delays. If the interrupt is not of high enough priority, or if it is currently masked or disabled, it will wait.

2. Saving the previous state of the processor on a system stack somewhere. Operating systems often take control at this point to save even more information to make sure the previous state can be recovered and continued.
3. The device driver, which uses some convention to indicate the occurrence of an event. It then releases control, with some additional overhead needed to restore the previous state of the system. The previous state may be the service of another interrupt; this could continue for a while, with various delays and additional interrupts, before any processing occurs at the user's task level.
4. Operating-system housekeeping. These delays can be quite substantial. (One minicomputer "real-time" operating system I have used quite often can produce multiple minutes of delay at this point!) The highest-priority task that is ready to run needs to be identified now, and this may require a context switch. The context switch saves the state of the previous operating task and allows a new (presumably higher-priority) task to take control. This time is one that can be measured.
5. The user's task itself. If swapping from disk is required to load the user's task, the delay may be greater than 100 ms.

A number of techniques can improve performance. By moving the sampling logic to a DMA controller and moving blocks of data to main memory with interrupts every 100th or 1000th sample, the host processor has more time for its operations. Similarly, addition of hardware to perform floating-point calculations or array processing can provide additional performance boost. Finally, the disk-interface hardware can limit the data logging and output rate. For a disk, system facilities that allow an application direct control over disk blocking, and the creation and use of contiguous disk files, can significantly increase performance. For some microsystems, all the files are contiguous and are limited in size and dynamic growth by this characteristic. In larger systems such as Unix, this reverses, with dynamic growth given the preference and little or no facility for contiguous disk-file control. Contiguous files are favored because the use of physically adjacent blocks of disk space allows the application to transfer data with just rotational delays, usually 2 to 10 ms, rather than the 10- to 100-ms seek delays. Faster disks and disk caching can also provide a way to deal with this limit on performance.

Conclusion

Both the systems manufacturer and the application developer must fully realize the relationship between the hardware and operating system if they expect to develop products for the real-time world. While processor performance is only one element of this, careful hardware design using higher-performance buses and accelerators such as cache, memory management, and arithmetic processors is quite critical. Once the hardware is defined, the operating system (and language used) must be able to take advantage of that hardware and give the programmer efficient control over the real-time environment. Finally, the application program must take into consideration not only all the hardware characteristics but those exceptional conditions that are almost guaranteed to occur. This will ensure that the application does not get caught in slowdowns related to memory control, addressing, priority, device drivers, abnormal conditions, and/or other applications that should not be affecting the real-time process. ■

James Isaak (983 Concord St., Framingham, MA 01701), the director of product marketing for Charles River Data Systems, has an M.S.E.E. from Stanford University. He enjoys camping, skiing, and canoeing. He also likes to spend some of his leisure moments with personal computers as applied to genealogical work.

Laboratory Data Collection with an IBM PC

A versatile hardware/software combination

You have a new IBM Personal Computer (PC) and you want to use it in the laboratory to collect data from a scientific instrument. How do you do that with a small investment of time on your part and still get a product that is a powerful, useful tool in the laboratory?

I faced that same problem almost two years ago when our chemistry department received its first IBM PC, and I wanted to interface it to a variety of chemical laboratory instruments. We had only one of each type of instrument, so I was faced with the possibility of designing a custom interface for each of 10 or more instruments.

Fortunately, I had interfaced single instruments to a DEC LSI 11/23 and to an Apple, so I knew from my own previous mistakes that a little advanced planning would make this a much simpler project. Specifically, I realized that interfacing can be made much easier by using two simple concepts: first, buy commercially available hardware where possible, and, second, develop general-purpose software that can be used for almost any instrument.

By utilizing these two concepts, I found that even undergraduate chemistry students with little previous computer experience can produce research-quality interfaces, with complete software, in less than one week. If you follow the suggestions provided here, you should be able to design and implement an interface to the instrument of your choice in less time. All you need to do is be able to program in BASIC, FORTRAN, or some other language that allows the use of assembly-language subroutines.

The essential elements of this system are a commercially available data-collection board that fits in one of the slots of the IBM PC, a preamplifier and filter for conditioning the signal from the instrument, and a set of BASIC and assembly-language routines to perform tasks common to all of the instruments to be interfaced.

The utility of this approach arose fairly naturally from some initial design decisions. My major criteria for selection of equipment and software were ease of development and ease of use. Therefore, I judged it to be not cost-effective to spend time developing special-purpose A/D (analog-to-digital) converters, timers, or other equipment. Similarly, I chose to use BASIC for all purposes except the data-collection process itself because of the ease of programming, even for novices; when the programs are completely tested, they are converted to compiled BASIC to greatly increase their execution speed.

In order to encourage a variety of users, I put the (now several) IBM PCs on carts so that the computers can be wheeled from experiment to experiment. Each cart contains a 64K- or 128K-byte IBM PC with a color-graphics monitor adapter and green monitor; dual 320K-byte disk drives; a combination board containing an A/D converter, D/A (digital-to-analog) converter and programmable clock; and a preamplifier and filter combination. A typical system in use is shown in photo 1.

Each of the components on the cart is designed to accommodate interfaces to a variety of instruments. If you are attempting to develop a similar system, it may help to have a description of why I selected each component.

Data-Acquisition Board

Several different manufacturers now market general-purpose data-

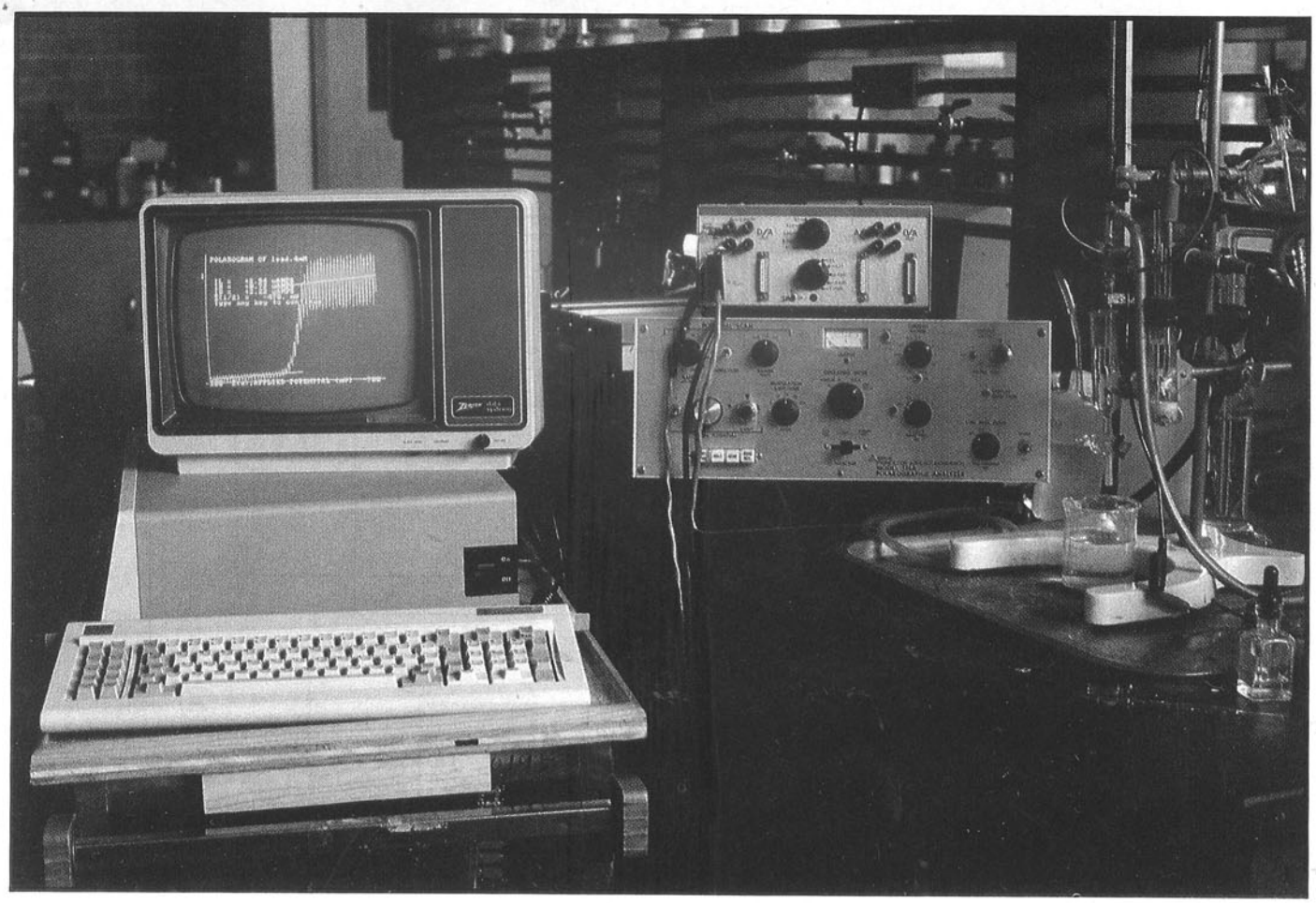

Photo 1: *The general-purpose laboratory interface station can be moved easily from instrument to instrument because it is on a laboratory cart. The IBM PC contains a color/graphics monitor board and a Tecmar Lab Master interface board. The preamplifier box is perched on top of the larger control device for the polarograph, in the center. The electrodes for the polarograph are at the right side of the photo.*

acquisition boards (see reference 4). These usually include a multichannel A/D converter, one or more D/A converters, and a programmable clock as standard features, with options such as programmable gain, higher acquisition rates, and DMA (direct memory access). For most scientific applications, a 12-bit A/D conversion is necessary; 8-bit A/D converters simply do not provide adequate resolution.

In addition, most laboratories now use nonintegrating A/D converters rather than integrating types because of the slow speed of the latter. The primary advantage of the integrating A/D converter is the reduction of noise; however, this can be accomplished instead through appropriate software used with the nonintegrating type. The A/D converters on almost all of the general-purpose data-collection boards now available are of the nonintegrating type.

While not essential, a programmable clock is highly recommended. Although timing can be controlled by carefully timed program loops, usually in assembly language, it is much more easily and accurately achieved in hardware.

For these reasons, I chose to use a Tecmar (6225 Cochran Rd., Cleveland, OH 44139, (216) 349-0600) PC-Mate Lab Master board with a 16-channel, 12-bit nonintegrating A/D converter with no programmable gain and a general-purpose clock/timer. The board also contains two D/A converters and a digital I/O (input/output) section that I do not routinely use, but which you may need if you plan to control the operation of your instrument as well as collect data from it.

Connecting the Interface

In order to use the hardware interface in your lab, you must first connect the interface to the instrument. If the instrument has a recorder output, this is very easy to do; simply connect the A/D converter input to the recorder output wires. For signals below 1 volt maximum, the preamplifier should be interposed between the A/D converter and the instrument.

Often, particularly on more recently designed instruments, both a recorder output and a BCD (binary-coded decimal) or other computer-compatible output exist. If there is a computer output, no A/D converter is needed; instead, a digital I/O board, serial interface, or other hardware is required. Unfortunately, I found that the documentation provided by Tecmar on the digital I/O section of the interface board is almost no help to those who are not already familiar with this type of hardware.

Alternatively, if no suitable output

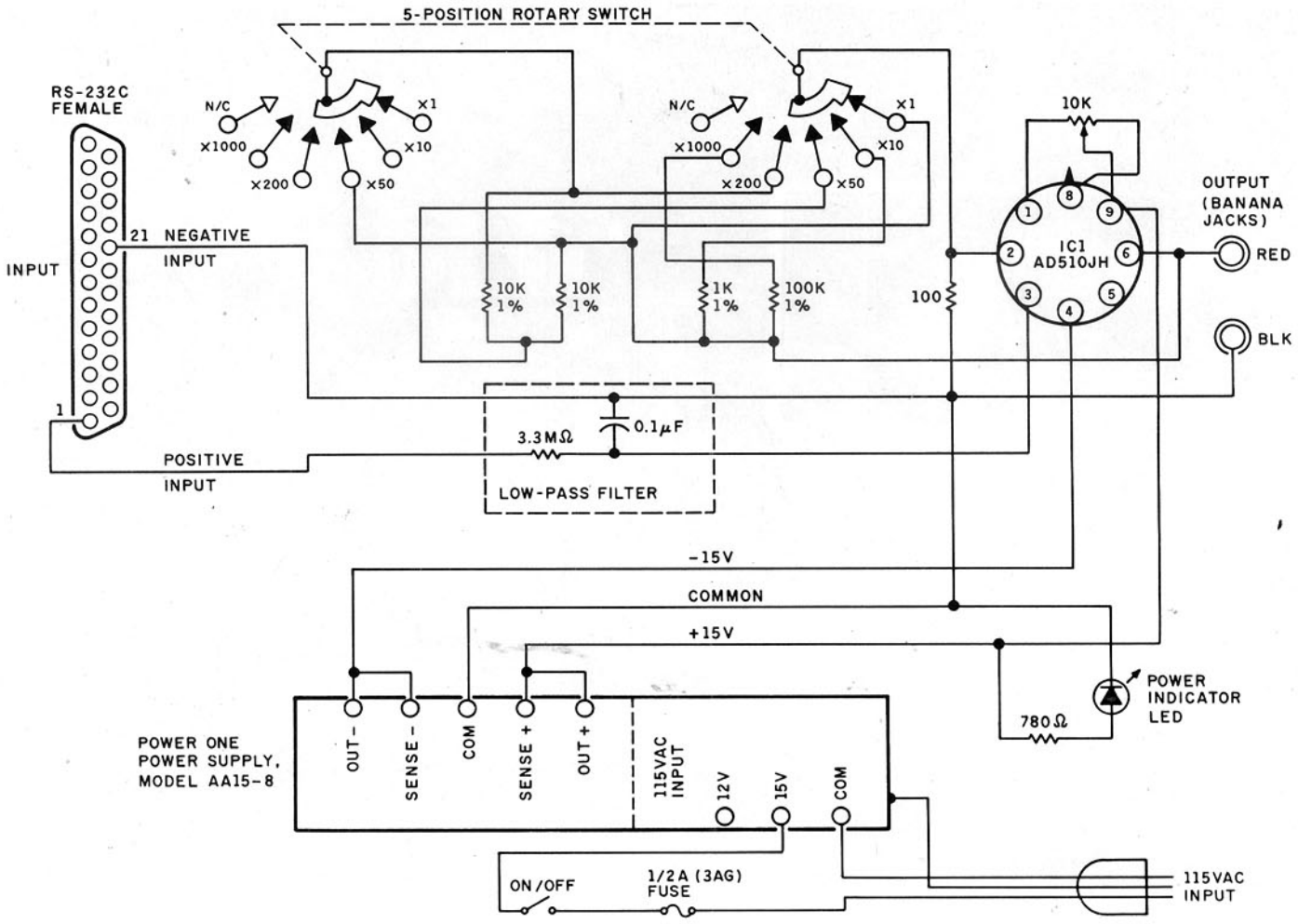

Figure 1: *The schematic diagram for the preamplifier described in the text. The low-pass filter is optional.*

is provided, it may be necessary for someone with knowledge of the electronics of the instrument to locate for you the portion of the circuitry needed to provide a suitable voltage output to the A/D converter. Where possible, this voltage should be in the volt range, rather than in millivolts (mV) or microvolts (μV). Fortunately, most instruments have recorder outputs and consequently are very easy to interface.

Preamplifier

Depending upon the instrument being interfaced and the A/D board being used, varying amounts of preamplification are needed. I designed our system to accommodate a wide variety of possible inputs; hence, a simple amplifier circuit was included to permit five different gains between 1 and 1000. The amplifier schematic is shown in figure 1.

Alternatively, a programmable-gain A/D board may be desirable, although that option is usually much more expensive than a separate amplifier. There is another reason for separate preamplifiers, however. Instruments with full-scale outputs of under 10 mV are common in scientific laboratories because of the widespread availability of 10-mV strip-chart recorders. For these instruments, your best alternative is to build the preamplifier into the instrument itself, or at least to connect it so that it is as near as possible to the instrument. This reduces the amount of noise picked up by the low-level signal lines that, in effect, act as antennas to the various sources of electronic noise in the environment. In general, the shorter the distance between the instrument and the A/D board, the better the signal-to-noise ratio will be in the final data.

Filter

The most general solution to noisy signals is software filtering, because the filter can be varied to best match the noise level. However, particularly for low-level signals and low data-collection rates, e.g., 1-mV signals at 60 Hz (hertz), I have found it useful to have a hardware filter because of the large amount of computation time required for extensive software-based filtering. For such instruments, I use the simple, passive, low-pass filter included in figure 1. This filter has a cutoff frequency of approximately 0.5 Hz, which is adequate for filtering out the most common noise signals that are 60 Hz or higher in frequency. More expensive filters, including active and notch filters, may be desirable for specific applications. Almost any "electronics for scientists" text can be consulted for more details.

Data Collection

One aspect of interfacing that texts often neglect is the need for general-purpose programs to collect, plot, and process the instrumented data.

However, by having a suitable library of general-purpose routines, you can shorten the development time for your specific interface considerably. By using the general-purpose data-collection, smoothing, and display routines described here, you can concentrate all of your efforts on developing the device-specific portion of the software and end up with a higher quality product in a much shorter time than if you "reinvent the chip" for each new interfacing project.

In order to provide high data-collection rates and a real-time plot, I wrote a data-acquisition routine in assembly language. The routine illustrated in listing 1 provides rates up to 2400 Hz with a real-time plot, and up to 20 kHz without plotting. Even faster rates are possible with special hardware settings of the standard Tecmar board, and rates up to 125 kHz are available as an optional feature. However, very few instruments will require higher rates than 20 kHz.

The routine in listing 1 assumes the use of the Tecmar Lab Master data-acquisition board, so that some of the code is device-specific and would need to be modified for use on other systems.

Although the listing is fully documented, several comments are required. First, using the excellent procedure suggested by Rollins (see reference 2), the routine begins with a header section to enable it to be converted by EXE2BIN to a binary file that can be loaded into memory with a BASIC BLOAD command. Second, high-resolution plotting is done using the BIOS VIDEO_IO routine, which is invoked with interrupt 16 (10 hexadecimal).

Three different clock rates are used, depending upon the desired data-collection rate. This is done to achieve maximum precision. For high data rates, the 1 MHz clock in the Tecmar board is used directly. For rates below 31 Hz, a 10-kHz subfrequency of the clock is used; to use the 1-MHz clock directly would require chaining several of the counters together. Rates of less than 1 Hz are counted with a 100-Hz subfrequency.

At very high data rates, it is possible that a conversion may take place

Listing 1: *An assembly-language data-collection routine for use with the IBM PC and the Tecmar Lab Master board.*

```
                TITLE   TIMER
;       S.C. GATES  DEPARTMENT OF CHEMISTRY, ILLINOIS STATE
;                    UNIVERSITY, NORMAL, IL   61761
;       SUBROUTINE TO DO TIMED DATA COLLECTION FROM TECMAR BOARD
;       CALL FROM BASIC WITH CALL OF FORM:
;               CALL TIMER (A%(1),F%,P%,N%,C%,S%)
;       WHERE   A% IS ARRAY WHERE DATA ARE TO BE STORED
;               F% IS OVERRUN FLAG--SET TO ZERO UPON NORMAL EXIT
;                  OTHERWISE SET TO VALUE OF CX REGISTER TO GIVE
;                  NUMBER OF POINTS NOT COLLECTED
;               P% IS 0 TO OMIT REAL-TIME PLOT, OTHER TO PLOT
;               N% IS NUMBER OF POINTS TO BE COLLECTED
;               C% IS CHANNEL NUMBER OF A/D
;               S% IS NUMBER OF DATA POINTS PER SECOND
;                  S% MUST BE <= SPEED OF A/D
;                  IF S% <0 THEN MEANS WANT THAT MANY SEC/POINT
;
CSEG    SEGMENT
        ASSUME CS:CSEG,  DS:NOTHING
HEADER:
        DB      0FDH            ;CODE FOR BLOAD FILE
        DW      0
        DW      0
        DW      RTN_LEN
TEMP    DW      ?               ;TEMP. STORAGE
PLOT    DW      ?               ;PLOT FLAG
TEMPSI  DW      ?               ;TEMP STORAGE FOR SI REGISTER
OVRUN   DW      ?               ;OVERRUN OF A/D FLAG
;DEFINITIONS:
ADD0    =1808                   ;BASE OF TECMAR BOARD
ADD4    =ADD0+4                  ;A/D CONTROL BYTE
ADD5    =ADD0+5                  ;A/D CHANNEL NUMBER
ADD6    =ADD0+6                  ;A/D START
ADD8    =ADD0+8                  ;CLOCK DATA PORT
ADD9    =ADD0+9                  ;CLOCK CONTROL PORT
;
TIMER   PROC    FAR
        PUSH    BP              ;SAVE BP
        MOV     BP,SP           ;SET BASE PARAMETER LIST
        MOV     DI,[BP]+6       ;GET DATA POINTS/SEC
        MOV     AX,[DI]         ;   INTO BX REGISTER
        MOV     BX,AX
        MOV     DI,[BP]+8       ;GET CHANNEL NUMBER
        MOV     AX,[DI]         ;   AND STORE AS AX
        MOV     DX,ADD5         ;   AND OUTPUT TO A/D
        OUT     DX,AL           ; (USE ONLY LOWER BYTE)
        MOV     DI,[BP]+10      ;GET NUMBER OF DATA POINTS
        MOV     CX,[DI]         ;  STORE IN CX REGISTER
        MOV     DI,[BP]+12      ;GET PLOT FLAG
        MOV     AX,[DI]         ;  STORE IN MEMORY
        MOV     PLOT,AX
        MOV     AL,128          ;SELECT A/D MODE (DISABLE AUTOINCREMENT,
        MOV     DX,ADD4         ;   EXTERN. START CONVERSION, ALL INTERRUPTS
        OUT     DX,AL           ;   GAIN=1)
        MOV     AX,0            ;SI IS X-VALUE OF POINT TO BE
        MOV     TEMPSI,AX       ;  PLOTTED--SAVE FOR LATER
        MOV     AX,6            ;SET UP HIGH-RES GRAPHICS MODE
        INT     10H
        MOV     DX,ADD6         ;RESET DONE FLIP-FLOP OF A/D
        IN      AL,DX
        MOV     DX,ADD9         ;SET DATA POINTER TO MASTER MODE REGISTER
        MOV     AL,23
        OUT     DX,AL
        MOV     DX,ADD8         ;SET MASTER MODE REGISTER FOR SCALER CONTROL=
        MOV     AL,0            ;   BCD DIVISION, ENABLE INCREMENT, 8-BIT BUS,
        OUT     DX,AL           ;   FOUT ON, DIVIDE BY 16, SOURCE=F1,
        MOV     AL,128          ;   COMPARATORS DISABLED, TOD DISABLED
        OUT     DX,AL
        MOV     DX,ADD9         ;SET DATA POINTER TO COUNTER MODE OF
        MOV     AL,5            ;   REGISTER 5
        OUT     DX,AL
        MOV     DX,ADD8         ;SET COUNTER 5 FOR COUNT REPETITIVELY,
        MOV     AL,33           ;BINARY COUNT,COUNT DOWN, ACTIVE HIGH
        OUT     DX,AL           ;TC, DISABLE SPECIAL GATE, RELOAD FROM LOAD,
        CMP     BX,31           ;CHECK IF >= 31 POINTS/SEC
        JGE     FAST            ;IF SO, JUMP TO FAST
        CMP     BX,0            ;CHECK IF > 0 POINTS/SEC
        JG      MED             ;IF SO, JUMP
;  BRANCH TO HERE IF POINTS/SEC < 0, MEANS THAT WANT LESS THAN
;  ONE POINT/SEC.
SLOW:   MOV     AL,15           ;SET TO 100 HZ (NO GATE, RISING EDGE
        OUT     DX,AL           ;   OF F5)
        NEG     BX              ;GET ABSOLUTE VALUE OF BX
        MOV     AX,BX           ;AND MULTIPLY BY 100 TO GET COUNT
        MOV     DI,100
        MUL     DI
        JMP     GO
;BRANCH TO HERE FOR 31 TO 20,000 POINTS/SEC--USE 1 MHZ CLOCK
FAST:   MOV     AL,11           ;COUNT AT 1 MHZ (NO GATE, RISING
        OUT     DX,AL           ;    EDGE OF F1)
        MOV     AX,10000        ;DIVIDE 1,000,000 BY PTS/SEC BY
        MOV     DI,100          ;  GETTING 10E6 INTO DX+AX
        MUL     DI
        DIV     BX              ;BX=PTS/SEC; RESULT IN DX+AX, BUT
                                ;   IGNORE DX, SINCE DX=0
```

Listing 1 continued:

```
        CMP     AX,200          ;DISABLE INTERRUPTS IF >=5000
        JG      FAST2           ;   POINTS/SEC
        CLI
FAST2:  JMP     GO
;BRANCH TO HERE FOR 1 TO 30 POINTS/SEC--USE 10 KHZ CLOCK
MED:    MOV     AL,13           ;COUNT AT 10 KHZ (NO GATE, RISING
        OUT     DX,AL           ;   EDGE OF F3)
        MOV     AX,10000        ;CALCULATE NUMBER OF TICKS OF 10,000 HZ CLOCK
        CWD                     ;   PER DATA POINT BY DIVIDING
        DIV     BX              ;   10,000 BY POINTS/SEC
;START CLOCK TICKING AT DESIRED RATE
GO:     MOV     DX,ADD8         ;   AND LOAD COUNTER 5 WITH TICKS
        DEC     AX              ;(COUNT TO ZERO, SO DECREMENT AX
        OUT     DX,AL           ;   FOR CORRECT COUNT)
        MOV     AL,AH
        OUT     DX,AL           ;   8 BITS AT A TIME
        MOV     DI,[BP]+14      ;GET OVERRUN FLAG ADDRESS
        MOV     WORD PTR [DI],0         ;ZERO THE FLAG
        MOV     OVRUN,DI        ;AND STORE THE FLAG ADDRESS
        MOV     DI,[BP]+16      ;GET ADDRESS OF DATA ARRAY
        MOV     DX,ADD9         ;LOAD COUNTER 5 FROM LOAD REGISTER
        MOV     AL,112          ;   AND ARM (START COUNTING)
        OUT     DX,AL
        MOV     DX,ADD4         ;ENABLE EXTERNAL START (PINS 3 + 4 OF
        MOV     AL,132          ;  CONNECTOR J2 MUST BE CONNECTED)
        OUT     DX,AL
;BEGIN DATA COLLECTION; COLLECT UPON EXTERNAL START TRIGGER
DONE:   MOV     DX,ADD4         ;CHECK IF DATA READY
        IN      AL,DX
        CMP     AL, 128         ;BY CHECKING READY BIT (BIT 7)
        JB      DONE            ;LOOP UNTIL READY
        TEST    AL,64           ;SEE IF DATA OVERRUN FLAG SET
        JNE     ERRMESS         ;IF SO, NOTIFY BASIC PROGRAM AND EXIT
        MOV     DX,ADD5         ;YES, DONE, SO GET LOW BYTE OF DATUM
        IN      AL,DX
        MOV     [DI],AL         ;AND STORE IT
        INC     DI              ;GO TO NEXT LOCATION IN ARRAY (1 BYTE LATER)
        MOV     DX,ADD6         ;GET HIGH BYTE AND STORE IT
        IN      AL,DX
        MOV     [DI],AL
        INC     DI
        CMP     PLOT,0          ;DON'T PLOT IF PLOT FLAG=0
        JZ      NOPLOT
;PLOT ROUTINE STARTS HERE
        MOV     TEMP,CX         ;SAVE CX FIRST
        MOV     AH,AL           ;GET HIGH BYTE JUST TAKEN
        MOV     AL,[DI-2]       ;AND LOW BYTE FROM STORAGE SO AX=DATUM
        ADD     AX,2047         ;CALCULATE Y-VALUE TO PLOT =
        CWD                     ;     199-((DATUM+2047)/21)
        MOV     BX,21           ;DIVIDE BY 21--QUOTIENT IN AX
        DIV     BX
        MOV     DX,AX           ;RESULT INTO DX
        NEG     DX              ;NEGATE AND ADD TO 199
        ADD     DX,199
        MOV     SI,TEMPSI       ;GET X-VALUE OF LAST POINT ON SCREEN
        INC     SI              ;GO TO NEXT LOCATION ON SCREEN
        CMP     SI,640          ;TEST IF AT RIGHT EDGE OF 640 X 200
        JL      M1              ;  SCREEN
        MOV     SI,0            ;IF SO, GO TO LEFT EDGE TO PLOT
M1:     MOV     CX,SI           ;GET X-VALUE INTO CX
        MOV     TEMPSI,SI       ;SAVE X VALUE
        MOV     AX,3073         ;AH=12,AL=1 TO WRITE DOT TO SCREEN
        INT     10H             ;PLOT POINT
        MOV     CX,TEMP         ;RESTORE CX
NOPLOT: LOOP    DONE            ;DECREMENT CX AND LOOP IF >0
;BRANCH TO HERE UPON FINISH OR OVERRUN
NOGO:   MOV     DX,ADD4         ;TURN OFF A/D
        MOV     AL,0
        OUT     DX,AL
        STI                     ;RESTORE INTERRUPT SERVICE
        POP     BP              ;RESTORE BP
        RET     12              ;6 ARGUMENTS IN CALL X 2=12
ERRMESS: MOV    DI,OVRUN        ;SET OVERRUN FLAG SINCE A/D GOING
        MOV     WORD PTR [DI],CX    ;TOO FAST
        JMP     NOGO
TIMER   ENDP
RTN_LEN         EQU     $-TEMP  ;LENGTH OF ROUTINE FOR HEADER
CSEG    ENDS
        END     HEADER          ;NEEDED FOR A .BIN FILE CONVERSION
```

Listing 2: *General-purpose data-collection, graphing, and smoothing program in IBM PC BASIC (DOS 1.10).*

```
10 REM GENERAL PURPOSE DATA COLLECTION PROGRAM
20 REM S. GATES, DEPARTMENT OF CHEMISTRY, ILLINOIS STATE UNIVERSITY, NORMAL, IL
30 REM
40 REM Some FOR...NEXT loops are compressed to speed execution
50 CLEAR,31000: BLOAD "TIMER.BIN",31000: TIMER=31000      'Get timer routine
60 DIM A%(1000),B(1000),SG%(9)
70 WIDTH 80:CLS
80 INPUT " Do you wish to process data that have already been collected?";Y$
90 IF Y$="y" OR  Y$="Y" THEN Y=4: GOTO 300
100 INPUT "Enter your name, please"; NAM$
110 D$=DATE$: T$=TIME$: INPUT "Enter the sample identification, please."; S$
```

Listing 2

before the previous data point has been read from the A/D converter. This is referred to as an overrun. Thus the program must check for the occurrence of an overrun. Upon finding one, the assembly routine sets a flag that can be read by the BASIC program once the data collection is finished.

At very high data-collection rates, interrupt-driven processes occurring in the computer, such as interrupts by the system clock, may interfere with data collection. Indeed, initially this program was limited to 6 kHz until I realized that the interrupts from the system clock were taking too much time. For this reason, at rates above 5 kHz, the subroutine turns off interrupts with a CLI (clear interrupt flag) instruction; when data collection is completed, the interrupts are again enabled by using an STI (set interrupt flag) instruction.

At low-to-moderate data-collection rates, it is useful to have a real-time plot. This is done for each data point collected by loading the low and high bytes of the data point into a register and converting it so that the screen displays a −10-volt A/D reading at the bottom and a +10-volt reading at the top—i.e., so that the full screen is used for the display.

When this assembly-language routine is linked to a higher level program, such as a compiled BASIC or FORTRAN program, only minor changes are required. The Header section must be removed, so that the code starts at Temp. The Timer procedure must be made Public, and the last line of the routine must include an End statement instead of an End-Header statement. After assembly, the subroutine is linked to the calling program using Link in the normal fashion; EXE2BIN does not need to be run in that case.

Sample BASIC Program

A short interpreter BASIC program for the IBM PC that uses the assembly-language routine is shown in listing 2. The program sets aside a region of memory for the routine; the location chosen in line 30 may vary depending upon the amount of memory available in the system. The

value 31500 is correct for a 64K-byte system using Advanced BASIC.

After the data has been collected, the overrun flag is checked, and the data is displayed in the high-resolution graphics mode. The data is scaled to fill the entire screen.

Once the data has been collected and displayed, you usually will need to remove high-frequency noise. A simple method for doing this in software is shown in listing 2. It uses a

Listing 2 continued:

```
120 PRINT "Please enter 3 lines of experimental description, including":
    PRINT "Sample preparation, instrument settings, etc."
130 FOR I=1 TO 3: LINE INPUT L$(I): NEXT I
140 INPUT "Enter the channel number (0 to 15)"; C%
150 INPUT "Enter the number of data points to collect.";N%
160 INPUT "Enter the number of data points/second desired. "; S
170 S%=S: IF S< 1 THEN S%=-1!/S   'Convert to proper format for timer routine
180 PRINT "Type any key to start count-down for data collection."
190 I$=INKEY$:IF I$="" THEN 190
200 CLS: FOR I=10 TO 0 STEP -1: LOCATE  12,40 : PRINT I: FOR J=1 TO 500:NEXT J:
    NEXT I                        'Count down;  J loop is delay between counts
210 F%=0                          'Initialize overrun flag
220 P%=1:IF S% > 2000 THEN P%=0   'Plot if < 2000 pts/sec
230 CALL TIMER(A%(1),F%,P%,N%,C%,S%)    'Collect data; all variables MUST be
                                                                    INTEGER!
240 IF F%<> 0 THEN PRINT "Warning--data taken too fast": N%=N%-F%
250 FOR I=1 TO N%
260     IF A%(I) > 32767 THEN A%(I)=A%(I)-65535!
270     A%(I)=A%(I)/.2047:      'Store input as mV, assuming -10 to 10V range
280 NEXT I
290 CLS: PRINT " Enter a 1 to plot data on the screen":
    PRINT " A 2 to store the data in a file.": PRINT " A 3 to smooth the data":
    PRINT " A 4 to get another file":PRINT " A 5 to exit": INPUT Y
300 ON Y GOSUB 340,510,670,770,890
310 GOTO 290
320 '******* SUBROUTINES *********
330 REM Screen plotting routine
340 SCREEN 2 :KEY OFF
350 DEF FNSCALE(Z%)=190-190*(Z%-YMIN)/(YMAX-YMIN)
360 INPUT "Enter the label for the graph",LAB$
370 CLS:YMAX=A%(1): YMIN=A%(1)
380 FOR I=1 TO N%
390     IF A%(I)<YMIN THEN YMIN=A%(I) ELSE IF A%(I)> YMAX THEN YMAX=A%(I)
400 NEXT I
410 YPLOT=FNSCALE(A%(1))
420 PSET (60,YPLOT),0                  'Go to first point
430 FOR I=2 TO N%: XPLOT=60+579*(I-1)/(N%-1): YPLOT=FNSCALE(A%(I)):
    LINE -(XPLOT,YPLOT): NEXT I
440 LOCATE  25,40 : PRINT LAB$;: LOCATE  1,1 : LINE (60,0)-(639,190),,B 'label
    and box plot
450 LOCATE  25,8 : PRINT "1";: LOCATE  25,75 : PRINT N%;: LOCATE  1,1 :
    PRINT YMAX;: LOCATE  24,1 : PRINT YMIN;      'Label axes
460 LOCATE 6,1: PRINT "Type any key to continue";
470 Y$=INKEY$:IF Y$="" THEN 470
480 RETURN
490 REM ********
500 REM Subroutine to store data in a file
510 INPUT "Enter the name of the file in which the data are to be stored.";FILN$

520 OPEN FILN$ FOR OUTPUT AS #2
530 WRITE #2,NAM$,D$,T$                'Save name, date, time
540 WRITE #2, S$                       'Save sample description
550 FOR I=1 TO 3: WRITE #2,L$(I): NEXT I     ' and conditions
560 WRITE #2,N%,S%             'number of points, sampling rate
570 FOR I=1 TO N%: WRITE #2,A%(I): NEXT I
580 CLOSE #2: RETURN
590 REM ********
600 REM Subroutine to compute second-order 9-point Savitzky-Golay smooth
610 REM     including smooths at both beginning and end of data
620 REM It computes a "smoothed" value for each point by adding together
630 REM       the 4 points on either side of it, plus itself, each multiplied
640 REM       times the corresponding coefficient.
650 REM It then computes the "smoothed" value for each successive point
660 REM       using the original data array.
670 DATA -21,14,39,54,59,54,39,14,-21: 'Savitzky-Golay coefficients
680 RESTORE
690 FOR I=1 TO 9:READ SG%(I):NEXT I     'Get coefficients
700 FOR I=1 TO N%: B(I)=0:DF%=0: FOR J=-4 TO +4: IF I+J< 1 OR I+J>N% THEN 720
710     B(I)=B(I)+A%(I+J)*SG%(J+5):DF%=DF%+SG%(J+5)
720 NEXT J:B(I)=B(I)/DF%:NEXT I          'Divide by sum of coefficients used
730 FOR I=1 TO N%: A%(I)=B(I): NEXT I  'Store back in original array
740 RETURN
750 REM ********
760 REM Subroutine to read previously collected data from disk file
770 INPUT "Enter the name of the file to be processed. " ; FILN$
780 OPEN FILN$ FOR INPUT AS #2
790 INPUT #2, NAM$,D$, T$
800 INPUT #2,S$
810 FOR I=1 TO 3: LINE INPUT #2,L$(I): NEXT I
820 INPUT #2,N%,S%
830 PRINT NAM$,D$,T$: PRINT S$: FOR I=1 TO 3: PRINT L$(I): NEXT I
840 PRINT "Number of points= "; N%, "Points/sec= "; S
850 FOR I=1 TO N%: INPUT #2,A%(I): NEXT I
860 CLOSE #2: PRINT "Type any key to continue"
870 Y$=INKEY$: IF Y$="" THEN 870
880 RETURN
890 END
```

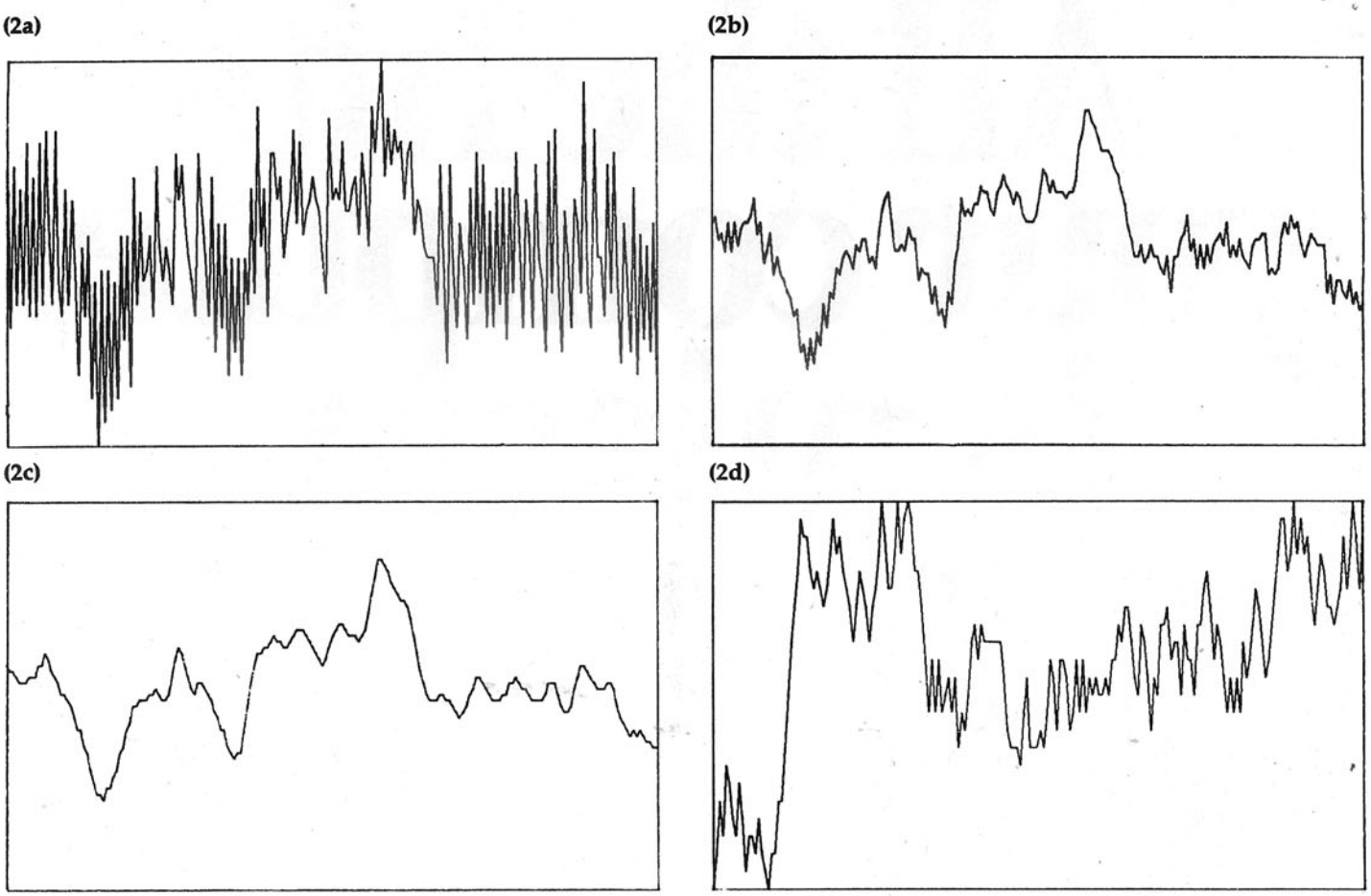

Figure 2: *The effect of filtering on noise levels can be very significant. Figure 2a shows 200 data points taken from an instrument over a period of 20 seconds. Ideally, the signal should be a straight line, but instead shows both long-term and short-term noise. In figure 2b, data from the same instrument is passed through a digital (software) filter once; in figure 2c, it is passed through the filter twice. In figure 2d, data from the same detector is passed through a hardware low-pass filter.*

"Savitzky-Golay" type smoothing algorithm (see reference 3), which is a rapid, easily implemented smoothing technique that is equivalent to fitting a least-squares line through the data. The order of the fit and the number of points included in the fit can be modified to provide varying amounts of smoothing. A second-order, 9-point smooth is the one most often used in my lab. In picking which software filter to use, you may find an article by Cram et al. quite useful (see reference 1). For severe noise problems, other techniques such as ensemble averaging or filtering using fast Fourier transforms may prove useful.

The usefulness of the filtering process is illustrated in figure 2. Figure 2a shows data collected from the detector of a high-performance liquid chromatograph, without filtering. In figure 2d, data was collected from the same detector, but with the low-pass hardware filter being used. In figure 2b, the data is exactly the same as the unfiltered data (figure 2a), except that it has been passed through the Savitzky-Golay second-order, 9-point filter contained in listing 2. In figure 2c, the data from figure 2a has been passed through the Savitzky-Golay filter twice; the reduction in the noise is striking. I often use a combination of hardware and software filtering for optimum results.

Examples of Use

I offer a four-week course to science students that teaches them to interface to a variety of scientific instruments using the techniques described in this article. Students spend one week learning BASIC, two weeks learning the concepts of interfacing and writing simple programs, and one week interfacing the computer to a specific chemical laboratory instrument.

Although the students learn to write data-collection and display routines in BASIC, for their final project they use the Timer routine in listing 1. Using the standardized interfacing system, in one week's time they have written complete data-collection and analysis programs for a number of different instruments, including a pH meter, a UV (ultraviolet)-visible spectrophotometer, a differential scanning calorimeter, a high-performance liquid chromatograph (HPLC), and a polarograph. Even though these programs were written in one week's time, each of these programs is now in routine use in our teaching or research laboratories.

I'll use two examples to show how quickly and easily instruments can be interfaced using this approach.

One student interfaced an IBM PC to a polarograph, using the circuitry shown in figure 1. The polarograph already has a sophisticated preampli-

Using the Tecmar A/D Board

The Tecmar board can be given instructions, and have information read from it, in one of two ways: either the I/O (input/output) mode or the memory-mapped mode can be used. In the I/O mode, various functions of the board are accessed through ports, which are addressed with INP and OUT instructions in BASIC, or IN and OUT instructions in assembly language. In the memory-mapped mode, the functions are accessed at a series of consecutive memory locations; this requires PEEK and POKE instructions in BASIC, or any memory-addressing instruction in assembly language, such as MOV or TEST.

The choice between these two modes is largely a matter of personal preference. The memory-mapped mode is slightly faster but the board is configured at the factory for the I/O mode, which is probably the simpler mode to program. In either mode, you must select the base address, which is the first of 16 consecutive addresses used to communicate with the various functions on the board. The base I/O address set at the factory is 1808. However, other base addresses, as well as the memory-mapped mode, may be selected using the appropriate jumpers or switches.

Other options available on the board include auto-incrementing of the A/D (analog-to-digital) converter (automatically switching the channel from which data is being taken), and the range of the signals coming from or going to the instrument. In addition, three types of inputs to the A/D converter are selectable by appropriate jumper settings: single-ended, pseudo-differential, and true differential. The single-ended setting is normally used, but the differential modes are particularly useful with low-level signals in environments with large amounts of electromagnetic noise. It is also possible to use interrupts to signal the computer when the A/D board has data ready for storage.

The system described in the text uses a −10-V to +10-V bipolar range for the A/D board, clock triggering of the A/D board, and a single-ended input. Only one instrument is normally connected, so the auto-incrementing feature is disabled, as are interrupts. Timer 5 is used to trigger the A/D board.

The clock portion of the Tecmar board provides a general-purpose mechanism for timing various events or for providing timed pulses for triggering various events. At least 18 different modes of operation are possible, each with several options. To the average user, this number of possibilities can prove highly confusing at best.

For triggering the A/D board at specific intervals, however, the process is fairly straightforward. The clock circuitry contains a 1-MHz clock, which is further subdivided either by powers of 10 (BCD scaling) or by powers of 16 (binary scaling), depending upon the option selected. Any one of five counters can be loaded with a count, which is then either incremented or decremented every time the clock "ticks."

For example, with a BCD scaling of divide-by-100, the clock provides a 10-kHz output. Assuming the count is in a downward direction, then the 16-bit counter can be loaded with a value of 99 to provide an output pulse to the A/D board every 0.01 second (i.e., 10 kHz ÷ 100 = 100 Hz). Note that the counter provides an output to the A/D board when it attempts to go below zero (called the "terminal count"); hence, the counter is set to 99 rather than to 100.

To connect the counter pulses to the A/D converter, the output from the specific counter must be directed to the trigger input of the A/D converter. Because of the pin placement on the Tecmar board, the easiest method for doing this is to connect the output of counter 5 to the A/D converter by jumpering pins 3 and 4 of connector J2.

All of the functions of the clock are controlled using two I/O ports accessible to any program. Although these ports are termed control port and data port, both ports are needed to set up the correct timing sequence. In a typical use of the timer, the control port is first directed to point to an internal register called the master mode register. You then select the various control options by loading a 16-bit word into the master-mode register via the data port; this selects options such as whether an 8- or 16-bit I/O bus is being used, what is to be used as a source of the clock frequency, and so forth.

Most of the information, however, is loaded into another internal register, the "counter-mode register." There is one such register for each of the five counters. Hence, the program uses the control port to select which counter-mode register is to be used; in this case, the one for register 5 is selected. The counter-mode register is then loaded, through the data port, with the various options selected for that register. Options include whether to count up or down, whether to count in binary or BCD, and which subfrequency of the clock is to be used. Special options are available if the counters are to be used as a time-of-day clock.

When the program is ready to begin collecting data, the appropriate counter must be loaded with the correct count and "armed," i.e., started counting. Assuming that the A/D converter has been set to recognize the signal from the clock as a trigger by enabling the external start bit, the A/D converter will automatically initiate a conversion (data collection) every time the counter register goes to zero. Hence, the program only needs to wait until the A/D converter signals that it has completed a conversion and then store the data; no timing loops need to be written. The A/D converter will continue to be triggered by the clock until the clock output is turned off by the program.

fier system, so a 10-volt signal could be readily obtained. Hence, the student set the preamplifier on the interface cart to a gain of 1, attached it to the recorder output of the polarograph, used no filtering, and set the Timer routine to collect data for a period of time determined by the potential range scanned.

The major task of the student, then, was to understand the theoretical basis of the instrument readings and to design a program in BASIC to analyze the data. In order to accomplish this, the student had to fit a least-squares line to a sawtooth wave function, determine the inflection point in the curve, and calculate the distance between the two least-squares lines at the inflection point. The A/D readings were then converted to current values in microamperes and the time scale was converted into the applied potential in millivolts.

Students in the analytical chemistry class now use data collected with this system from a series of standard lead samples to calculate the amount of lead in leaded gasoline. Photo 2 shows data collected by a group of students for a standard sample of lead.

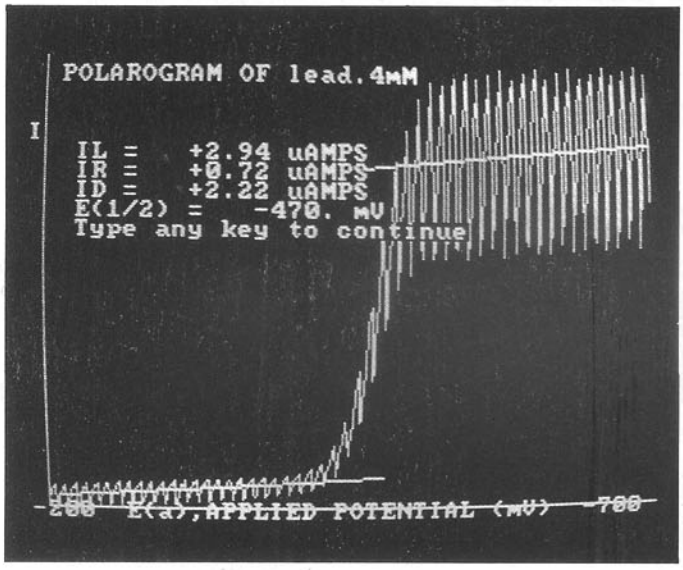

Photo 2: *Students taking our analytical chemistry laboratory course analyze the amount of lead in gasoline using the interface described in the text. The diffusion current (ID on the display) is proportional to the concentration of the lead in the sample.*

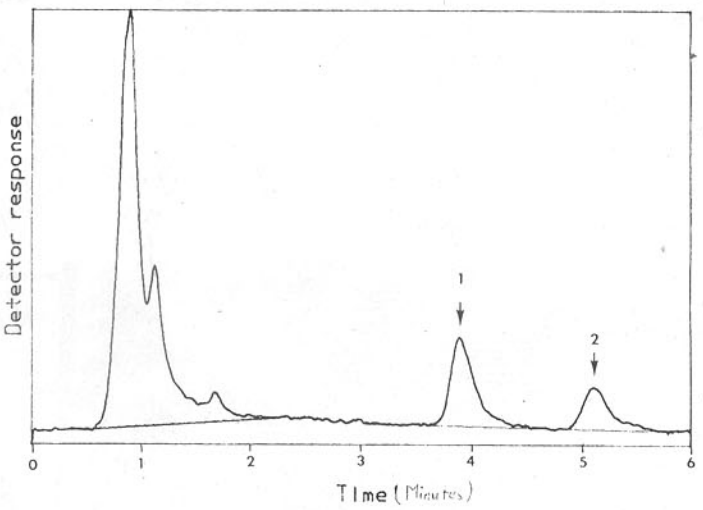

Figure 3: *A common problem in chemical laboratory work is to measure the areas of peaks. Each peak in this figure is integrated by the computer program; the peaks of interest are peaks 1 and 2, which are caffeine and benzyl alcohol, respectively. The benzyl alcohol peak serves as an internal standard for measuring the caffeine. The straight lines under each peak are the baselines determined by the computer during the integration process. The large initial peak is a group of unidentified substances. The sample is a cup of instant coffee.*

A second example of an instrument that students have interfaced is a high-performance liquid chromatograph. The normal output of the HPLC is a 10-mV signal displayed on a strip-chart recorder; hence, the preamplifier was set to a gain of 1000 to provide a 10-volt signal to the A/D converter.

The student writing the program divided it into two sections: a data-collection portion and a data-analysis portion. In the data-collection portion, all of the parameters of the instrument and the sample to be analyzed are recorded, thus providing a permanent record of the conditions of the analysis. The program also asks for the names of the substances being analyzed, if known, and whether an internal standard is being used.

The data collection is done using the assembly-language routine, with a real-time plot of the data. If more than a predefined number of points are collected, the data is "bunched," or averaged, together. The Savitzky-Golay smooth is then performed, and the smoothed data and identifying information are stored in a disk file.

In the second section of the program, the peaks in the data are integrated, and the area of each peak is compared to that of an internal standard. Proper integration involves deciding where each peak starts and stops and then selecting the appropriate baseline to be subtracted from each peak. The results of this process are shown in figure 3. Again, the program is used routinely in our analytical laboratory course; figure 3 shows an analysis of caffeine in coffee performed by a group of students in that course.

Conclusions

One of the many advantages of the revolution in "home" computers is that powerful but inexpensive computers can be used in scientific or industrial laboratories, even by those with relatively limited computer skills. Utilizing off-the-shelf components and simple programming languages, extremely sophisticated data-collection and data-processing systems can be developed very rapidly.

The system described here represents a hardware and software solution to the problem of data collection and analysis in a wide variety of commonly encountered laboratory situations. By making only minor modifications, you should be able to adapt it to other types of hardware and to other types of instrumentation with an extremely wide range of applications, not only in chemistry, but in other scientific and industrial areas as well. ■

References

1. Cram, Stuart P., S. N. Chesler, and A. C. Brown III. "Effects of Digital Filters on Chromatographic Signals." *Journal of Chromatography*, volume 126, Amsterdam, Netherlands: Elsevier Scientific Publishing Company Inc., 1976, page 279.
2. Rollins, Dan. "The 8088 Connection." BYTE, July 1983, page 398.
3. Savitzky, Abraham and Marcel J. E. Golay. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." *Analytical Chemistry*, volume 36, Washington, DC: American Chemical Society, 1964, page 1627. Minor corrections were also published in *Analytical Chemistry*, volume 44, 1972, page 1906.
4. Welch, Mark J. "Expanding on the PC." BYTE, November 1983, page 168.

Stephen C. Gates, Ph.D. (Department of Chemistry, Illinois State University, Normal, IL 61761), is assistant professor of biochemistry. He teaches a course in computer interfacing and does research on computerized chemical analysis of biological samples.