An Engineering Approach to Determining Sampling Rates for Switches and Sensors in Real-Time Systems

Melissa Moy and David B. Stewart

Dept. of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
Tel. 301-405-3658; Fax. 301-314-9281; Email: {*msquared, dstewart}@eng.umd.edu*

Abstract: *An objective of our work is to find more systematic methods of designing and implementing real-time device drivers for embedded systems. As part of this objective, we answer the question "what is the best sampling rate to use for reading data from sensors that provide continuous data?" Our experiences in answering this question for digital switch and analog sensor inputs is described. We first present a model of a real-time device driver that enables the driver to execute as its own thread of control, independent of whatever control task is using the data. We then present an engineering approach towards determining a good sample rate for reading digital switches and analog sensors that provide continuous data. Rather than providing a single value for the sample rate, ranges that are based on application parameters are derived analytically. An application designer can use these equations to quickly determine the minimum and maximum sampling rates for their device driver task.*

***Keywords**: component-based device drivers, real-time operating system, experimental software engineering, digital control systems, real-time scheduling, switch matrix, analog-to-digital converter, software modelling, parallel I/O.*

1. Introduction

Most real-time scheduling algorithms rely on knowledge of a task's period and execution time to determine the CPU utilization and feasibility of a schedule. In most cases, the period of any task reading input is constant and provided as part of the specification. In our experience, this specification is often determined in an arbitrary manner, with no supporting evidence that the value is the best choice or even a good choice. Unfortunately, the practice of arbitrarily specifying periods of tasks often results in using more CPU resources than is necessary, or providing a level of performance or quality that is below what can be achieved using the same resources. In this paper, we focus on execution rate of input tasks; that is, tasks that read switches and sensors to obtain data that represents the current state of the environment.

For example, consider a speed control system with a velocity input, that is read through an analog-to-digital converter (ADC). The specifications state the velocity is to be read 50 times per second. Thus, the period for the task is set to 20 msec. Suppose a schedulability analysis is done on the entire system that included this task, and it is determined that the task set is not schedulable; what flexibility does the designer have to fix this? While a theoretical approach simply states this is not schedulable, implementation of such an application requires that a solution be found. The result is that the designer would use trial-and-error methods to fine-tune various parts of the system. For example, the designer may spend weeks to optimize part of the code to make the task set schedulable. Changing the task's period, however, is not acceptable, because 50 Hz is part of the specification.

An objective of our work is to find more systematic methods of designing and implementing real-time device drivers for embedded systems. As part of this objective, we answer the question, *what is the best sampling rate to use for reading data from sensors that provide continuous data?*

In general, there is no best answer for choosing a correct sampling rate. Rather, there are a variety of trade-offs, many of which are application-specific. Thus, decisions on adjusting real-time scheduling parameters can only be made after suitable determination of the needs of the software as it corresponds to the original application specifications [12]. Examples of parameters that are part of the trade-offs include CPU utilization, memory usage, real-time schedulability, data accuracy, data integrity, response time, preemption overhead, predictability, I/O hardware cost and complexity, and control system performance. Typically, trying to improve some of these parameters results in a necessary compromise of the other parameters. Correlating our results with the control sampling rates described in [2], becomes an end-to-end timing issue, which has been addressed in [5]. The results we present indicate that different switches and sensors often need to be polled at different rates; issues in developing control systems with such non-uniform sampling rates are presented in [1].

First, in Section 2, we present our model for I/O tasks, that we have designed especially for use in real-time systems. The I/O model is an improvement over traditional device driver approaches, as it allows for much greater real-time flexibility in both preemptive and non-preemptive multitasking environments. The model is sufficiently flexible to accommodate the trade-offs described in this paper.

Since trade-offs are application specific, it is not possible to provide a general umbrella solution that is suitable to all systems. Instead, in Section 3 we focus on digital input switches and in Section 4 on analog input sensors. These switches and sensors are representative of many of the input devices found in today's embedded applications.

2. Device Drivers

Device drivers are used to provide a layer of abstraction to hardware I/O devices, so that higher levels of software can access devices in a uniform, hardware-independent fashion. A class of device drivers is usually defined by the operating system as an interface specification, such that each instance of a driver ensures that interaction with a device conforms to the specification. Examples of classes include the POSIX *open/read/write/close* streams interface, printer drivers, network drivers, and display drivers.

While the use of device drivers in desktop computing has successfully enabled the creation of complex heterogeneous environments, real-time operating systems (RTOS) have failed to define a good specification for hardware-independent access to embedded I/O devices. For embedded systems, we define I/O as the interaction of the system with the environment. Aspects such as interprocessor communication (IPC), where one task outputs information to another, is not considered I/O, since it remains internal to the system.

The large variety of I/O on embedded processors makes it difficult to use a standardized model and application-programmer interface (API) for accessing the devices. In addition, low performance and limited memory often make the overhead of implementing a device driver unacceptable, especially when there are severe real-time constraints that require regular access to the devices at rates that range from 1 to 10,000 times per second. Thus, embedded system designers build application software that interacts directly with the microcontroller's I/O hardware. There are many problems with the existing device driver models, such as no concept of time, lack of real-time threading support, incorrect distinctions between the I/O port and the I/O device connected to a port, ad-hoc approach to interrupt handling, and lack of computer-aided software engineering (CASE) tools to aid in developing hardware-independent software.

As a result, nearly all of the software is hardware-dependent therefore preventing software reuse and portability to other target environments. Long development time is introduced since programmers need to learn the intricate low-level details of the microprocessor and I/O hardware and difficult debugging occurs since problems are hard to isolate in non-modular software.

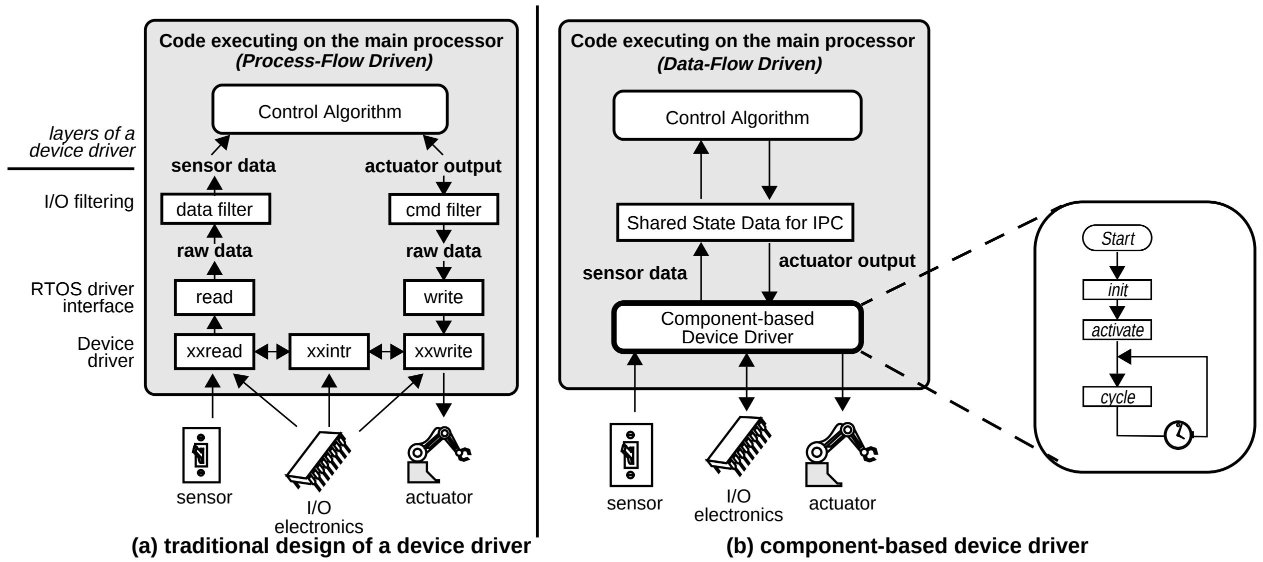
2.1 Device Driver Model for Embedded Systems

To address these problems, we have created a new device driver model and interface specification for use in embedded control systems. The model leverages the fact that real-time software can be implemented as a multitasking application, thus the driver itself can have its own thread of control.

We addressed the modeling and interface issues of device drivers in embedded systems, and developed a component-based device driver model. The driver uses a data-driven approach to interact with the rest of the application software, as opposed to the more traditional process driven approach, as shown diagrammatically in Figure 1.

The data-driven approach takes advantage of the fact that most real-time systems can be implemented as a collection of concurrent tasks that use IPC to exchange data. In contrast, the traditional driver model was designed for UNIX systems when each process was a single thread of control, and that one process would do a system call to invoke the driver's functions, then obtain the data as a return value to that call. Because the new model has less software layers, it can be implemented more efficiently on embedded processors.

Rather than defining an entirely new interface specification, we found the port-based object (PBO) interface specification, as detailed in [13], suitable for these component-based device driver. The PBO interface decomposes the software component into initialize, activate, cycle, sync, deactivate, and terminate methods. For our discussion, the most impor-



(a) traditional design of a device driver (b) component-based device driver

Figure 1: Comparison of traditional and new device driver designs.

tant method is the cycle method, which executes periodically as a real-time thread. See [13] for complete details. Transfer of data to the hardware independent code uses a state variable table mechanism, as described in [7]. The driver model should also be compatible with other reasonable real-time IPC mechanisms, such as shared memory with semaphores using the priority ceiling protocol. However, we have not experimented with other such mechanisms.

An important aspect of the new model is that the sampling rate is specified as one of the component's configurable parameters. This provides the flexibility needed to select the best sampling rate for any particular I/O device and application combination, as described in Section 3.

2.2 Real-time analysis of device drivers

Real-time scheduling analysis has focussed primarily on the scheduling of well behaved computational components, assuming an idealized system with no device access. Even work on aperiodic servers [11], which are characterized as random events occurring in response to externally generated interrupts, only take into account the CPU usage of the event handler, and not any of the overhead incurred in generating or handling the event. None of the prior driver models has even considered sampling rate as one of the factors that affects the performance and resource utilization of the device driver.

The component-based device driver model allows the same scheduling analysis to explicitly include the device drivers. Each device driver is analyzed simply as another task in the real-time system. Its period, execution time, and utilization then become the primary inputs to schedulability analysis. Enabling such analysis forms the basis for wanting to select the best sampling rates. In traditional systems, the rate at which an I/O device was read or written was the same as the module invoking the driver. Rather, device and application characteristics combined with application specifications can yield sampling rates for drivers that are independent of the execution rates of other processes.

3. Sampling Rates for Digital Inputs

In most embedded applications that we observed, sampling rates for reading or writing I/O devices were determined in an ad-hoc manner. Usually the rates are included in specifications. If the software appears to work with the specified sampling rates, then the rates would stay fixed for the lifetime of the application and the values never questioned. If the software does not work, then the sampling rate may be adjusted on a trial-and-error basis until an acceptable solution is obtained. While these methods may result in working systems, there is no guarantee that the software meets the application's specifications all the time, nor is there any indication as to whether the device driver software is using more CPU execution time than needed to meet those specifications.

The use of proper sampling rates for device drivers allows software to better meet application specifications, and may reduce the overall utilization by not executing the driver software any faster than necessary. In most cases, there is not a single answer as to the correct sampling rate. Rather, the sampling rate is usually a range of acceptable values, but different ends of the range result in trade-offs. For example, a customary trade-off is the potential error in the input or output signal versus execution time used by the driver software. The correct choice is then dependent on the application needs and resources available.

A systematic approach to determining sampling rate is the following:

- Measure sensor characteristics of application
- Identify constraints of I/O ports
- Compute the lower and upper bound for sampling and obtaining correct answers
- Identify the trade-offs between lower and upper bound
- Develop driver algorithm to read sensors
- Analyze the performance of the driver code
- Measure execution time of driver to experimentally validate the analysis
- Select period within sampling rate frame that is the best compromise of trade-offs and is schedulable

The methods combine experimental measurement with analytical understanding of the application needs, in order to engineer a good solution.

The simplest form of a digital input is a switch. When on, the switch produces a value of 1. When off, the switch provides a value of 0. The reverse can be true if negative logic is used; for purposes of this paper, we only use positive logic. When dealing with negative logic, the input from a switch can be inverted immediately upon reading the data. Through examples, we apply this systematic approach to digital inputs. The process is repeated in Section 4 for analog inputs.

Most embedded systems have one or more switches; sometimes dozens or even over a hundred. When a small number of switches is used (e.g. less than 10), the switches are usually directly connected to a digital input port (DIO), (also called a parallel port). With more switches, a switch matrix is commonly employed to reduce the hardware requirements. First, we discuss the simple switches that connect directly to DIO. In Section 3.5, we extend the analysis to large number of switches connected using a switch matrix.

3.1 Ideal and Real Switch Characteristics

An ideal switch provides a 1 when the switch is closed and 0 when the switch is open. The transition from one state to the other is instantaneous. In reality, there is rise and fall time. However, since these times are proportional to the amount of capacitance in a circuit, their value is generally negligible, as it is on the order of nanoseconds. For our analysis, we thus neglect the rise and fall time. When neglecting this time, some switches, such as optical and tightly constructed momentary switches, do exhibit ideal behavior.

Most mechanical switches, however, exhibit bouncing. When the switch is closed, the transition from 0 to 1 is not instantaneous, nor is it uniformly rising. Rather, Figure 2(a) shows the oscilloscope output of one such mechanical switch. However, our concern is the digital representation of the transitions, as shown in Figure 2(b). The extra pulses preceding (and following) the main pulse are commonly referred to as *bounces*, as they result from bouncing when contact is made between mechanical plates internal to the switch. When bouncing exists, an application will usually require filtering the input, also called debouncing.

The switch characteristics form the basis for the design of the algorithms for reading inputs and the necessary sampling rate of the tasks polling these inputs to meet the application specifications. We now focus on the experimental determination of relevant application parameters.

3.2 Switch Closure Times

One of the most important parameters to measure in order to determine the sampling rate is the minimum switch closure time, σ_{min}. If a switch is closed for at least this long, the software is guaranteed to detect it as a switch closure. If the actual pulse width of a switch closure is shorter than this, the software might miss the switch closure, but it is not necessarily considered a failure. As an example, one of our embedded applications was to design software for a pinball machine (one of our experimental testbeds [4]). The machine has several kinds of switches, each with a different set of characteristics. Some of the switches are shown in Figure 3.

Figure 3(a) are switches that must be polled quickly, because the velocity of the ball can be very fast. For these switches, we measured σ_{min} to be about 10 msec[1]. Note that this value is dependent on the environment; changing characteristics of the environment may yield a different value for the fastest switch closure time. It may be possible to experimentally or analytically determine the fastest the ball may travel across one of these switches, in which case σ_{min} can be derived indirectly as a function of the ball speed.

Figure 3(b) shows "medium-speed" switches. Due to a change in direction of the ball, there is a much lower bound on the maximum velocity of the ball as it travels across the switch. In our experiments, we measured the shortest switch closure time for these switches to be about 50 msec.

A "slow switch" is one that is guaranteed to remain closed until the control software detects it, and issues a command to re-open the switch. Figure 3(c) shows examples of such switches. In the first case, a ball is sitting in a saucer on the switch. When the software detects this, it fires a solenoid that kicks the ball out. In the second case, the targets are spring-loaded to fall when a ball hits it. Firing a solenoid is needed to re-raise the target. For the slow switches, the shortest switch closure time is a function of the control software used to fire the solenoid; in our testbed application, the solenoid firing process was executing at a rate of 10 Hz.

In general, we assume that a switch closing is not latched. Using latches is often not practical, and sometimes not possible such as in switch matrices (described in Section 3.5). If a latch were used, the response would then be similar to the switches shown in Figure 3(c), with the rate being a function of the task that generates the signal to clear the latch.

If the switch is not ideal, then the settling time, (which we call τ), must also be measured. The settling time is the amount of time the switch may bounce before settling to its steady state value that correctly represents the state of the switch as either closed or open. For the switches shown in Figure 3(a), we found that the rollover switch (left side of diagram) is not ideal. The opto-switch on the right side of the diagram, however, is an ideal switch and did not show any bouncing. For analysis purposes, we are especially interested in τ_{max}.

To measure σ_{min} and τ_{max}, connect the switch between V_{cc} and GND (through a resistor) as shown in Figure 4, and connect a logic analyzer at V_{out}. Note that V_{cc} should be the same as would be used with this switch in the final application. Setup the logic analyzer to trigger on the rising edge.

Close and re-open the switch as fast as possible. If the switch is ideal (or near ideal), you should see a smooth switch transition from 0 to 1 and back to 0. If it is a bouncy switch, then the output would be similar to that shown in Figure 2. Repeat this experiment at least several dozen times, recording the values of τ_{max} and σ_{min} for each. When performing these

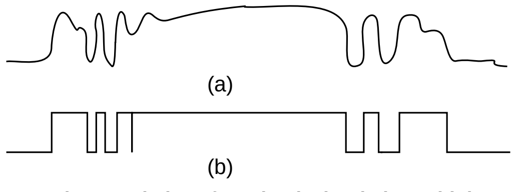


Figure 2: Characteristics of mechanical switches with bounces. (a) as viewed on an oscilloscope; (b) as viewed on a logic analyzer and as viewed on the input pins of a digital input port.

[1] We have more accurate measurements available. However, for sake of discussion, it is much easier to round to the nearest whole number.

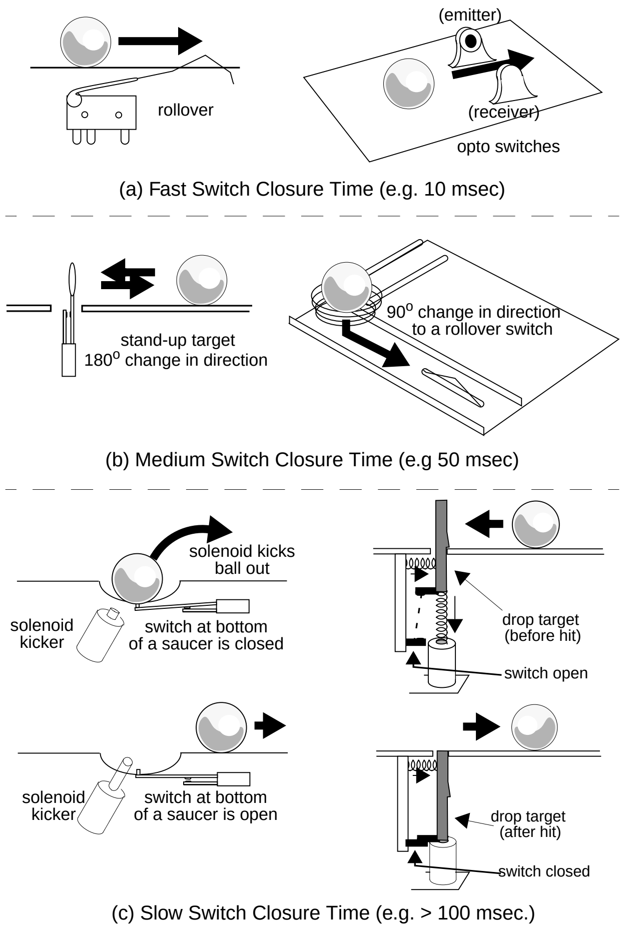

rollover
(emitter)
(receiver)
opto switches
(a) Fast Switch Closure Time (e.g. 10 msec)
stand-up target
180° change in direction
90° change in direction to a rollover switch
(b) Medium Switch Closure Time (e.g 50 msec)
solenoid kicks ball out
solenoid kicker
switch at bottom of a saucer is closed
drop target (before hit)
switch open
solenoid kicker
switch at bottom of a saucer is open
drop target (after hit)
switch closed
(c) Slow Switch Closure Time (e.g. > 100 msec.)

Figure 3: Switches that need fast, medium, and slow polling rates.

experiments, it is important to consider how the switch will be used in the final application. For example, a momentary switch that is designed for a human to press, might be pressed very differently depending on the person. It is thus necessary to repeat the experiments using a variety of touches. For example, a light tap may yield a fast settling time, but also a short σ_{min}. On the other hand, a heavy press might have a long σ_{min}, but also might have more bouncing and therefore a longer τ_{max}. It is important to record the minimum, average, and maximum σ_{min} and τ_{max} for your experiments.

3.3 Sampling Rate for Ideal Switches

For an ideal switch, τ_{max} will always be 0. The sampling rate to guarantee that all switch closures are detected must then be less than σ_{min}. While this seems simple, there is a trade-off. What if σ_{min} is 10 μsec? Must we poll the switch at 100,000 times per second? This would surely use all the available CPU resources. It is at this point that the application specifications must be considered and trade-offs performed. While it may be possible to get a switch closure that is only 10 μsec, that might only happen once in 1000 times (0.1%). In an application, we may find that 99.9% of the time, σ_{min} is greater than 5 msec. In such a case, sampling at 5 msec instead of 10 μsec is much more practical and uses a lot less CPU time. The question then, is whether or not it is acceptable for the application to detect a switch closure that is only 10 μsec. If the switch closure is associated with human input, we can assume that the switch was closed too lightly, hence they simply need to press harder. If the closure is one of the switches in our pinball machine, then we may conclude that the switch was not really closed. On the other hand, if the switch closure is associated with a toxic gas substance, then we want to capture it; in this latter case, we may choose to latch the switch, or dedicate a small processor to reading the switch at 10 μsec intervals.

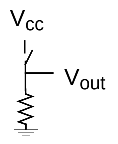


Figure 4: Circuit used to measure σ_{min} and τ_{max}

Lets suppose that it is acceptable in our application to only guarantee detecting switch closures with σ_{min} greater than 5 msec, thus being 99.9% accurate. What if the CPU is overloaded? We can halve the CPU utilization of this task if we poll at 10 msec instead of 5 msec. Based on experimentation, this might reduce accuracy to 99.0%. If that is still okay for the application, then the trade-off is acceptable. But if slowing the sampling rate to 10 msec reduces our accuracy to 70.0%, that might no longer be acceptable. It is for this reason that logging all results of the experimentation for determining σ_{min} should be noted. It allows for a trade-off between accuracy and CPU utilization.

The above discussion assumes an ideal switch. If there is switch bouncing, then this imposes additional constraints on selecting an appropriate sampling rate.

3.4 Sampling Rates for non-Ideal Switches

Let us reconsider the rollover switch in Figure 3(a). A sample of the output for this switch is shown in Figure 5(a), with the filtered output being shown in Figure 5(b). All of the bouncing is filtered, to provide a clean signal to the application code that acts in response to this switch closure. To perform this filtering, a debouncing algorithm is needed. A variety of debouncing algorithms exist (both in hardware and software) as discussed in [10].

In this paper, we use the simple debouncing algorithm shown as a state diagram in Figure 6. In words, this algorithm looks for two consecutive samples to be of the same value to consider that the switch has changed states. It is using this specific algorithm that we perform the following analysis. Should a different debouncing algorithm be used, the analysis would be different, and hence the specification of the correct sampling time would change. Implementation of this algorithm on an embedded processor is straightforward using boolean algebra. If variables are 8-bits wide, eight switches can be debounced in parallel.

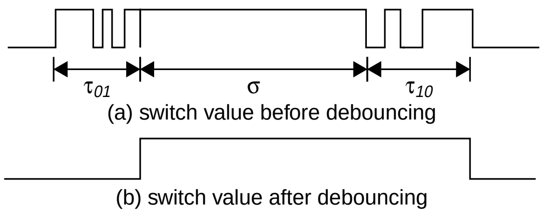
The switch closure must be sampled at least twice within the time σ_{min}, otherwise the switch hit will be filtered. This places an upper bound on the sampling period as $\sigma_{min}/2$.

To determine a lower bound, we consider the minimum case needed for the debouncing algorithm to mistake bounces for two consecutive switch hits. Such a case occurs if we obtain two samples showing 1, followed by two with 0, then two more with 1. Assume that only the last two 1's are the steady state. The sampling that would generate such a filtered output would require at least four samples during the time interval τ_{max}. To prevent such an occurrence, we must sample at most three times during the transient bouncing of the switch closure. Therefore, the sampling period must be greater than $\tau_{max}/3$.

Combining the lower and upper bounds, we have the following condition on the sampling period T_s and sampling rate f_s (where $f_s = 1/T_s$) for the input driver that debounces a digital input according to the state machine in Figure 6:

$$\frac{\tau_{max}}{3} < T_s < \frac{\sigma_{min}}{2}\text{; or } \frac{2}{\sigma_{min}} < f_s < \frac{3}{\tau_{max}} \qquad (1)$$

The range of possible values shows the range of acceptable trade-offs for the sampling rate. Suppose τ_{max} is 3 msec and σ_{min} is 10 msec. Then 1 msec < T_s < 5 msec. To minimize the



τ_{01}
σ
τ_{10}
(a) switch value before debouncing
(b) switch value after debouncing

Figure 5: Measurement of τ_{max} and σ_{min}.; τ_{max} = max(τ_{01},τ_{10}).

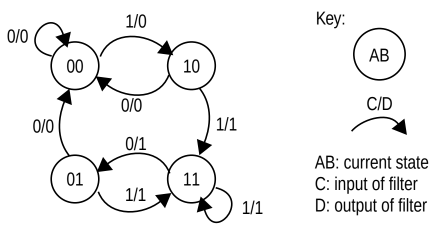


Figure 6: State diagram for debouncing algorithm

amount of CPU time that sampling the switches takes, we would select the sampling rate to be near 5 msec. On the other hand, suppose that the task set is being scheduled using the rate monotonic algorithm, the fastest task is executing at 3 msec, and all other tasks are multiples of this value (hence producing a harmonic task set). Then it may be desirable to use a bit more execution time, run the task at 3 msec, and thus keep the schedulable bound at 100%. We know that a 3 msec rate is acceptable because it is within the acceptable range. One may ask if there is any reason to ever execute with a sampling rate of 1 msec? One reason for doing so is that experimentally obtaining the value σ_{min} might not have yielded the lowest possible value. Or, if σ_{min} was selected to catch 99.0% of switch closures, using the faster sampling rate might raise this value even higher. But in no case should the sampling rate be faster than 1 msec to avoid mistaking a bounce for a switch closure.

For a particular bouncy switch, it might be possible that τ_{max} is 6 msec, and σ_{min} is only 4 msec. In such a case, there is no possible sampling rate that can guarantee the capture of the switch closure; the designer must choose other options. One is to use a different debouncing algorithm such as looking for two out of three 1's instead of two consecutive 1's. Another option is to consider the minimum inter-arrival time of switch closures. A third possibility is to either accept an occasional miss of a switch closure, in which case σ_{min} can be raised, or accept an occasional switch closure to be mistaken for two separate events, thus reducing τ_{max}. Regardless of the decision made by the designer, the choices can easily be documented, and should the decision not be the right one, modifying the design is simply a matter of modifying the algorithm in one module, or changing the sampling rate.

Although a single switch digital input is simple, it highlights many real issues for determining a correct sampling rate. Next, we focus on a more complex form of digital input: the switch matrix.

3.5 The Switch Matrix

A switch matrix is a more complex digital input device. It is used in applications with large number of binary sensors to reduce wiring requirements. A switch matrix is most commonly employed in keypads and keyboards [6]. It can also be used in more sophisticated applications, such as intelligent traffic light control, building temperature control, security and alarm systems, and tactile skin for robots. These applications differ from keyboards in that the I/O from a digital I/O (DIO) port of a microcontroller does not have sufficient drive current to propagate through the matrix. Instead, the computer signal must be propagated through power amplifiers, relays, and/or opto-isolators. The propagation delay is significant, often on the order of several dozens to hundreds of microseconds, and puts severe requirements on the software timing.

There is a trend towards using a serial bus, such as a controller area network (CAN) [14], to reduce wiring requirements. However, it raises the cost of the hardware and changes the design of the software, as each sensor or actuator node must have its local processor. In this paper, we do not debate the choice to use a switch matrix over other alternatives. Rather, we are interested in the switch matrix as an academic example of a complex device structure, to demonstrate an engineering approach to determining the sampling rate for a complex device driver.

Without a switch matrix, *N* digital input ports are needed to interface to *N* switches, assuming a common ground for all switches. A software-controlled switch matrix is used to reduce the number of I/O ports to $2*\log_2(N)$, thus reducing the overall cost of the hardware. Half of the I/O ports on a DIO board are used for selecting one of the $\log_2(N)$ columns of the matrix, and the other half of the ports are used to read the corresponding switches for each row in the active column.

As an example, Figure 7(a) shows a 16-bit DIO board connected to 16 binary switches. The software can obtain the values of the switches by reading the registers corresponding to the DIO's ports, all of which are configured for input. The advantage of this method is that software is very simple. A single read operation of the input ports is sufficient to collect all the data on a single cycle. Thus, each polling cycle of the entire switch matrix is about 100 µsec on an 8 MHz 8-bit microcontroller. The disadvantage is the hardware cost. For example, if there are 64 switches to poll, then a microcontroller with 64 DIO pins is required.

The DIO hardware requirements for reading 16 switches can be reduced to 8 by reorganizing the switches as a 4 by 4 matrix, as shown in Figure 7(b). The diodes in the switch matrix are used to allow the software to detect multiple switches that are depressed simultaneously by preventing feedback current into inactive columns. Four of the DIO bits

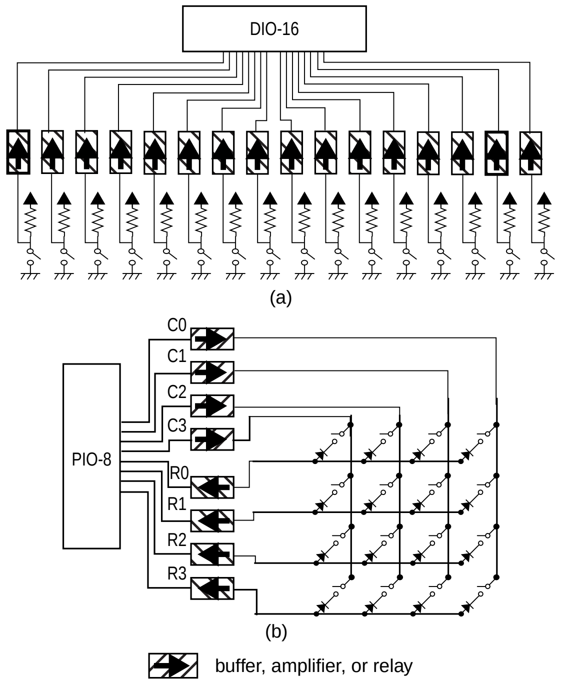


Figure 7: (a) 16-port DIO board connected to 16 switches. (b) 8-port DIO board connected to 16 switches by using a software-controlled switch matrix.

are configured for output; they are used to activate one of the four columns. The other four I/O bits are configured for input. Note that this is only one of many configurations for switch matrices. This particular matrix allows multiple switches to be closed simultaneously, and allows the software to detect precisely which switches are closed and which are open. Simpler switch matrices, as often used in keypads, usually cannot distinguish multiple simultaneous key presses. Regardless of the construction of the switch matrix, the analysis and algorithms described in this section are similar.

The switch matrix is scalable with complexity of $O(\log_2 n)$. For example, an application with 256 switches can be wired as a 16 by 16 switch matrix, to reduce the number of required I/O pins from 256 to 32. An application with 64 switches can be defined as an 8 by 8 matrix, requiring only 16 I/O pins.

The boxes with arrows shown in Figure 7 represent buffers, power amplifiers, opto-isolators and/or relays, (which for simplicity we will collectively call *buffers*) required to drive the lines with sufficient voltage and current. In a keyboard, line drivers with a few microseconds propagation delay is usually sufficient. In larger-scale applications, however, a separate power supply and opto-isolation between the computer's logic supply and this power supply is often needed. In our experimental testbed, we use Grayhill 70-ODC5 for output and 70IDC Industrial Control Modules for input. The data sheets specify a maximum turn-on time of 20 μsec for the output modules and 200 μsec for the input modules.

Software that polls the switch matrix must select a column to poll, then wait for the signal to propagate through the buffers, then finally read the signal at the input port of the I/O device. The pseudocode for polling is the following:

```
for $i = 0$ to $N_{col}$ do
    Activate column $C_i$ by turning on the bit leading to that column, and
        turning off all other output bits.
    Wait $D_p$ seconds, where $D_p$ is the sum of the propagation delay for both
        the input and output modules. (We assume that the propagation
        delay of the wires that connect the switches is negligible).
    Read the row. If a switch in column $C_i$ is closed, then the bit in the row
        of that switch will be on, otherwise it will be off. Any number of
        switches in the column can be closed simultaneously.
end
```

To demonstrate the effect of propagation delay, suppose the delay in an 8 by 8 switch matrix is 150 μsec and the switches must be polled 500 times per second. Assume another 100 μsec overhead for context switching and for writing to and reading from the I/O ports. It takes the software 250 μsec per column, or 2.0 msec to read all switches. At a frequency of 500 Hz, this amounts to 100% CPU utilization. The switch matrix, however, is often not the only function that needs to be performed by the microcontroller. On a slower processors the utilization may rise to above 100%.

In a multitasking environment where the switch matrix executes as a higher-priority process, the 150 μsec delay may be too short to make it worthwhile to switch contexts to another process. A context switch on a microcontroller is expensive, often longer than the amount of busy waiting needed; thus the time to swap out and back in does not provide any improvement. Experienced embedded system engineers combat the problem by replacing the delay with unrelated but useful instructions into the code. Although this improves on CPU utilization, it eliminates any form of modularity and prevents creating a standard device driver model for a switch matrix since the polling software cannot be encapsulated. The resulting software is extremely difficult to analyze, and often the source of difficult to fix timing errors.

Table 1: Overhead of switch matrix polling operations on an 8 MHz 8088

Variable	Description	μsec
C_{csw}	context switch overhead	88
C_{wr}	Write to column	15
C_{rd}	Read from row	24
C_{mod}	Increment and modulo	11
C_{lo}	Base loop overhead	15
C_{lv}	Additional overhead per loop iteration	8
C_{if}	Simple If comparison	15

3.5.1 Modelling the Switch Matrix Driver

The primary requirements for polling a switch matrix are conflicting:

• poll fast enough to catch every switch closure

• poll as slow as possible to minimize CPU utilization

We have studied five variations for the switch matrix driver, three of which are based on a fixed polling rate for all switches, and two are based on a configurable variable rate. For each variation, we provide the pseudocode, a mathematical characterization of the utilization, measurements of the execution time of our implementations to validate our analysis, and a comparison with the other methods. We conclude that for a fixed-rate method using an interrupt-driven timer, it is best to poll only a single column of the matrix per cycle, unless the interrupt overhead is greater than the propagation delay. In cases where only some of the switches need to be polled at the fastest rate, we present a device driver for polling different columns of a matrix at variable rates that can significantly reduce the overall CPU utilization of the microcontroller.

To capture the overhead of polling and implementing the individual instructions, we use the method that Katcher [8] used to characterize real-time scheduling overhead. To quantify our analysis, we measured the values for several microcontroller platforms. In this paper, we present results from our experiments using the 8 MHz 8088.

The overhead for various basic operations used in our equations is shown in Table 1. The context switch overhead, C_{csw}, includes scheduling time. C_{wr} is the time to select a column by writing to an appropriate output port. C_{rd} is the time to read the row via an input port. The time to perform the ***mod*** operation is C_{mod}. C_{lo} and C_{lv} quantifies the loop overhead. C_{if} is the time to perform the simple comparison for the variable-rate algorithm (Section 3.5.3).

The overhead was measured by instrumenting the code to toggle unused bits on a DIO board. Before an operation, the bit is set to 1. After an operation, the bit is set to 0. The signal was then viewed on a logic analyzer with better than 1 μsec resolution. Note this method is fairly accurate for microcontrollers with no cache nor pipeline, but not necessarily as accurate for modern RISC architectures.

The context switch overhead cannot be measured directly using the method described above. Instead, it is measured by taking three measurements, as shown in the timing diagram of Figure 8. A bit on Channel 1 is continuously toggled within a low priority task (e.g. the idle task). A bit in Channel 2 is set to 1 as the first instruction in the higher priority switch matrix polling task, and reset to 0 as the last instruction. The pulse width on Channel 2 represents the length of the sampling rate code. The measurements are of t_1, t_2, and t_3, where t_1 represents the length of time it takes to execute the switch matrix polling plus overhead for calling the handler, t_2 represents the length of only the handler code, and t_3 represent the execution time of one cycle of the while loop. We compute the overhead of the context switch code as $C_{csw} = (t_1 - t_2) - t_3$. Note that this same overhead analysis can be used if the polling code is placed within a periodic interrupt handler; the schedulability analysis, however, might change.

The CPU execution time used by one iteration of the sampling code, including all overhead, is C_s. The period is T_s. Thus the utilization of the switch matrix polling software, is

$$U_s = \frac{C_s}{T_s} \qquad (2)$$

The analysis assumes that the polling software is executed at a rate that matches the shortest switch closure time. In many applications, however, it may be desirable to poll at least twice as fast, to take into account issues such as switch debouncing.

3.5.2 Fixed-Rate Switch Matrix Polling Algorithms.

Fixed-rate algorithms are commonly employed in switch matrix control software. Every column of switches is polled with the same frequency. We first present the most common algorithm for reading a switch matrix, then show two variations. For each version, the CPU utilization is computed as a function of the shortest switch closure time and of the propagation delay.

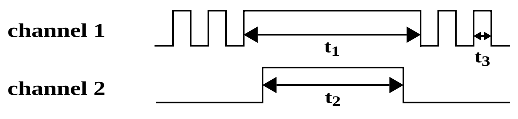


Figure 8: Measuring the context switch overhead.

All switches every cycle

This first algorithm is used to scan a switch matrix in which multiple switches may be closed simultaneously, and the software must detect all such closures. Pseudocode for ALGORITHM F1 is shown in Figure 9.

The execution time for ALGORITHM F1 is computed as

$$C_s = C_{lo} + n_{col}(C_{lv} + C_{wr} + C_{rd} + D_p) + C_{csw} \qquad (3)$$

Since every column is polled on every cycle, $T_s = \sigma_{min}$; therefore, the CPU utilization, U_s, is

$$U_s = \frac{C_{lo} + n_{col}(C_{lv} + C_{wr} + C_{rd} + D_p) + C_{csw}}{\sigma_{min}} \qquad (4)$$

An alternative to polling every column on each cycle, is to poll only a single column per cycle, but increase the rate of the interrupt handler. This leads to the next algorithm.

One column per cycle

ALGORITHM F2, detailed in Figure 9, is another fixed rate method that can be used to spread out the overhead of reading the matrix, such that each switch is only polled once every n_{col} cycles.

The execution time for ALGORITHM F2 is

$$C_s = C_{wr} + C_{rd} + D_p + C_{csw} + C_{mod} \qquad (5)$$

As compared to ALGORITHM F1, there is no loop overhead but additional overhead for computing C_{mod}. However, since only one column is polled on each cycle, the rate of the interrupt handler must be increased accordingly, such that $T_s = \sigma_{min}/n_{col}$. Therefore, the utilization is

$$U_s = \frac{C_{wr} + C_{rd} + D_p + C_{csw} + C_{mod}}{\sigma_{min}} \cdot n_{col} \qquad (6)$$

A comparison of the utilization between this algorithm, ALGORITHM F1, and the other algorithms still to be shown in this section is illustrated in Figure 10 and Figure 11. Figure 10 compares the results assuming the propagation delay D_p=200 µsec, but the shortest switch closure time σ_{min} varies from 5 to 50 msec. Figure 11 shows σ_{min} constant at 10 msec, but D_p varies from 0 to 300 µsec. Note that although this algorithm uses consistently more CPU time than ALGORITHM F1, we use it as the basis for ALGORITHM F3.

Read-before-write, one column per cycle

An alternative to ALGORITHM F2 eliminates the busy-waiting time D_p, as long as the $T_s > D_p$. Note, the modification to the initialization is to ensure that a good value is read the first time *swCycle* is called. Pseudocode for ALGORITHM F3 is shown in Figure 9. The CPU utilization for ALGORITHM F3 is the following:

```
ALGORITHM F1
  swInit:

  swCycle:
    for col = 0 to $n_{col}$–1 do
      write (1 << col) to $P_{out}$
      delay $D_p$
      read $P_{in}$ to swmx[col]
    end for
```

```
ALGORITHM F2:
  swInit:
    col = 0;

  swCycle:
    col = (col+1) mod $n_{col}$
    write (1 << col) to $P_{out}$
    delay $D_p$
    read $P_{in}$ to swmx[col]
```

```
ALGORITHM F3:
  swInit:
    col = 0;
    write (1<<col) to $P_{out}$
  swCycle:
    read $P_{in}$ to swmx[col]
    col = (col+1) mod $n_{col}$
  write (1 << col) to $P_{out}$
```

Figure 9: Fixed-Rate Switch Matrix Polling Algorithm.

$$U_s = \frac{C_{wr} + C_{rd} + C_{csw} + C_{mod}}{\sigma_{min}} \cdot n_{col} \qquad (7)$$

Even though it is necessary to run the *swCycle* routine n_{col} times faster than ALGORITHM F1 to read the switches at the same rate, it uses less CPU time for cases where $D_p > C_{csw}$, as shown in Figure 11. The utilization for ALGORITHM F3 as a function of σ_{min} with D_p=200 µsec is consistently better than both ALGORITHM F1 and ALGORITHM F2, as shown in Figure 10.

3.5.3 Variable-Rate Switch Matrix Polling Algorithms

Given that not all switches must be polled at the same frequency, we want to develop a driver model for scheduling which columns of the switch matrix are polled on each periodic execution of *swCycle*. It is essential that the implementation of the schedule is efficient, such that the overhead does not counteract the benefits of using a variable-rate polling method. Note that when wiring the switch matrix for variable-rate polling, it is desirable to include switches with similar closure times on the same column, since the column must be polled at a rate fast enough to read the switch with the shortest closure time. In contrast, assigning switches to specific rows makes no difference from the software perspective.

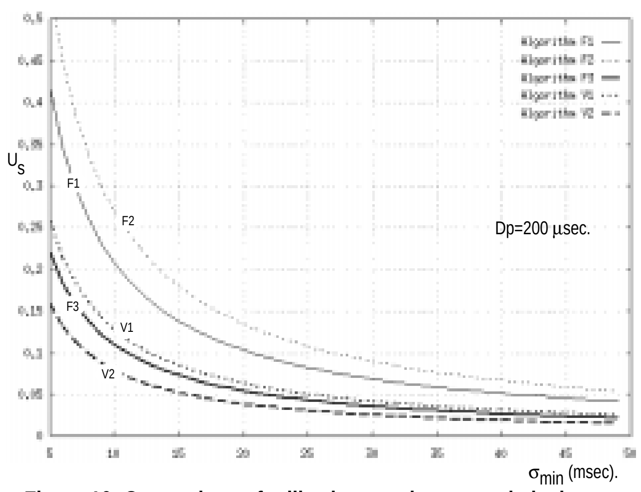


Figure 10: Comparison of utilization vs. shortest switch closure time for switch polling algorithms. Propagation delay D_p=200 µsec.

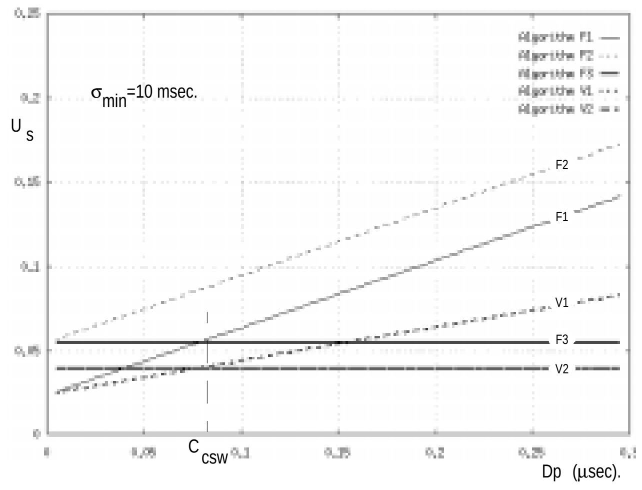


Figure 11: Comparison of utilization vs. propagation delay for switch polling algorithms. σ_{min}=10 msec.

We now describe a method for determining an appropriate schedule given the shortest closure switch time (σ_k) for each column k. The method is based on building a multi-rate cyclic schedule, as commonly used for job shop scheduling [3,9]. Note that we assume it is acceptable to poll any column faster than the specified switch closure time; although if switch debouncing is used, the fastest polling time may be bounded as was discussed in Section 3.4. This allows us to adjust the polling time to optimize the cyclic schedule.

To build the schedule, the shortest closure time of any switch in each column k, σ_k, must be supplied by the application specification, or obtained through experimental measurements to gather the data.

From this value, we can compute σ_{min} as

$$\sigma_{min} = \min(\sigma_k)\Big|_{k=0}^{7} \qquad (8)$$

We also define σ_{max} as the largest value of *min*(σ_k), scaled to a multiple of σ_{min}, as follows:

$$\sigma_{max} = \left\lfloor \frac{\max(\sigma_k)\Big|_{k=0}^{7}}{\sigma_{min}} \right\rfloor \sigma_{min} \qquad (9)$$

The length of the schedule, L_s, can be computed as the ratio of σ_{max} to σ_{min} as,

$$L_s = \frac{\sigma_{max}}{\sigma_{min}} \qquad (10)$$

assuming that T_s will be set to σ_{min}. Equation (9) ensures that σ_{max} is a common multiple of σ_{min}. Note that the more factors of σ_{max} that are also multiples of σ_{min}, the more flexible the algorithm we present below for building a schedule. Note also that σ_{max}, σ_{min}, or both can be lowered – implying that it is acceptable for a switch to be polled faster than necessary – to increase the number of factors of σ_{max} that are multiples of σ_{min}, without affecting the correctness of the switch matrix operation.

The polling period of each column in units of number of iterations of the interrupt handler, ρ_k, is computed as follows:

$$\rho_k = \left\lfloor \frac{L_s}{\left\lceil \frac{\sigma_{max}}{\sigma_k} \right\rceil} \right\rfloor \qquad (11)$$

The polling period T_k for each column is then be computed as,

$$T_k = \rho_k \sigma_{min} \qquad (12)$$

Equations (11) and (12) are used to ensure that the polling times for individual columns are harmonic and are a multiple of T_s. If necessary, polling occurs more often than defined by σ_k, but never less often. T_k is used to build the schedule S.

The schedule S is an array of L_s elements of n_{col} bits each, that is used to store the schedule. Setting bit k in element S_j means to poll column k on cycle j, where on each cycle, *j=(j+1)* ***mod*** L_s. In creating a schedule, the number of bits for any one cycle j should be minimized, thus minimizing the number of columns that must be polled on a single cycle.

The schedule is built either offline or during initialization, using the following algorithm, where the function *bits(x)* returns the number of bits set in the value *x*. The pseudocode for creating the scheduler is shown in Figure 12.

Using ALGORITHM S1, the maximum number of columns that must be polled on any one cycle, m_{cc}, is computed as,

$$m_{cc} = \left\lceil \frac{\sum_{k=0}^{n_{col}-1} \left\lceil \frac{\sigma_{max}}{T_k} \right\rceil}{L_s} \right\rceil \quad (13)$$

Given the above schedule, we now present an algorithm to poll the switch matrix.

Variable-rate switch matrix polling, write before read

The basic variable-rate polling algorithm is a variation of the Algorithm F1 (See ALGORITHM V1 of Figure 13). The CPU utilization for ALGORITHM V1 is the following:

$$U_s = \frac{C_{csw} + C_{lo} + n_{col} \cdot (C_{lv} + C_{if}) + m_{cc} \cdot (C_{wr} + C_{rd} + D_p + C_{mod})}{\sigma_{min}} \quad (14)$$

As can be seen in Figure 10, this algorithm (plotted for σ_k={10,10,20,80,120,240,20,80}) performs similar to the best fixed rate algorithm. Note that results may vary significantly depending on σ_k. With a known set of minimum switch

```
ALGORITHM S1
    sort $T_k$ in increasing order
    for each column $k$ (in order of $T_k$) do
        $b$=0;
        valid = FALSE;
        while (valid == FALSE)
            for $j$ = 0 to $\rho_k$ do
                if bits($S_j$,$n_{col}$) == $b$ then
                        // try to fit in as $b^{th}$ column to poll on every $\rho_k^{th}$ cycle.
                    valid = TRUE;
                    for $i$ = $j$ to $L_s$ step $\rho_k$ do
                        if bits($S_i$,$n_{col}$) > b
                            valid = FALSE;     // does not fit, try with next cycle j.
                    end for
            end for
            if valid == TRUE exit while loop;
            $b$++;
        end while
        // starting cycle for column k is j. Set bit k in $S_j$, $S_j+\rho_k$, $S_j+2\rho_k$, etc.
        for $i$ = $j$ to $L_s$ step $\rho_k$ do
            set bit $k$ in $S_i$
        end for
    end for
```

Figure 12: Scheduling algorithm

```
ALGORITHM V1:
    swInit:
        initialize $L_s$ (Eq. (10)) and $S_j$ from
            data generated in ALGORITHM S1.
        $j$ = 0;
    swCycle:
        for col = 0 to $n_{col}$ do
            if (1<<col) & S[j] then
                write (1 << col) to $P_{out}$
                delay $D_p$
                read $P_{in}$ to swmx[col]
            end if
        end for
    $j$ = ($j$+1) mod $L_s$
```

```
ALGORITHM V2:
    swInit:
        initialize $m_{cc}$, $L_s$ and $S_j$
        $j$ = 0;
        col=0;
    swCycle:
        read $P_{in}$ to swmx[col]
        while not (1<<col) & S[j] do
            increment col;
            if col == ncol then
                $j$ = ($j$+1) mod $L_s$
                col=0;
            end if
        end while
    write (1 << col) to $P_{out}$
```

Figure 13: Variable-rate switch matrix polling algorithms

Table 2: Experimental measurements of utilization for sk={10,10,20,80,120,240,20,80}on an 8 Mhz 8088. (All times in msec).

Algorithm	T_s	Measured C_s	Measured U_s	Theoretical U_s	Equation for computing U_s
F1	10,000	2550	26	21	(4)
F2	1250	402	32	27	(6)
F3	1250	151	12	11	(7)
V1	10,000	1442	14	13	(14)
V2	2500	202	8	5	(16)

closure times, we can use any graphical plotting package to display the graphs and select the best performing algorithm for the specific application.

Variable-rate switch matrix polling, read before write.

In the fixed-rate method, we reduced utilization by performing only a single column per cycle and switching the order of the write column and read row operations. The compromise is that the rate of the interrupt handler had to be increased by a factor of n_{col}. Nevertheless, it still resulted in reduced utilization.

We can perform the same modification to ALGORITHM V1 to reduce CPU utilization further. (See ALGORITHM V2 of Figure 13).

The execution time of *swCycle* is

$$C_s = C_{csw} + C_{lo} + \frac{n_{col}}{m_{cc}} \cdot (C_{lv} + C_{if}) + C_{wr} + C_{rd} + C_{mod} \quad (15)$$

Since only one column is polled on each cycle, the rate of the interrupt handler must be increased. However, since we only have to cycle through m_{cc} columns and not all n_{col} columns, the rate only needs to be increased to T_s=σ_{min}/m_{cc}. Therefore,

$$U_s = \frac{C_{csw} + C_{lo} + \frac{n_{col}}{m_{cc}} \cdot (C_{lv} + C_{if}) + C_{wr} + C_{rd} + C_{mod}}{\sigma_{min}} \cdot m_{cc} \quad (16)$$

As can be seen in Figure 10 and Figure 11, this algorithm (plotted for σ_k={10,10,20,80,120,240,20,80}) performs significantly better than any other algorithm.

To substantiate our analytically-derived plots, we measured the actual execution time for our implementation of *swCycle* on our 8088 implementation. The results are shown in Table 2. Although the measured times tend to be higher than the theoretical estimation, the results do support our claim that a variable-rate switch matrix handler can significantly reduce CPU utilization for applications where switches need not all be polled using the sample sampling rate.

To compare the effect of the variable rate algorithms, we compare the CPU utilization of different switch matrix configurations each with n_{col}=8, as shown in Table 3. The different configurations are obtained by assuming different sets of switch closure times σ_k. The schedule *S* for each configuration is then obtained by applying ALGORITHM S1. The values L_s, m_{cc}, ρ_k, T_k, and T_s, are also computed using the equations above, and included in the table. The schedule lets us know which columns are polled during a given cycle. Utilization for each configuration and algorithm is computed assuming the use of an 8088 microcontroller. Based on the calculated utilizations, we can configure a device driver's real-time behavior to suit the application's needs as well as to use only as much CPU utilization as necessary. Note that the second row in this table corresponds to the schedule used in generating the graphs in Figure 9 and Figure 10, as well as for the measurements in Table 2.

4. Sampling Rates for Analog Input

Analog inputs provide data to the processor through an analog-to-digital converter. The sampling rate refers to the number of times the data is read from the ADC and passed along to other application components that use the data. The sampling rate directly affects the temporal resolution of the input signal, much in the same way as the number of bits of resolution in the ADC affects the spatial resolution.

The maximum error is a function of the sampling rate. We define the error $\varepsilon(t)$ as the difference between the real sensor value, and the value used by the control value, at any time t. Note that t is continuous, thus as t increases between sample periods, the input value is constant, but the error typically increases. This is shown in Figure 14.

In signal processing, Nyquist criteria is used to determine the sampling rate. Specifically, the sampling must be at least twice as fast as the highest frequency component in the input signal. Given this sampling rate, the original input signal can then be reconstructed.

Unfortunately, the Nyquist criteria cannot be used in most embedded applications. Reconstruction of the original signal requires significant computational power; thus the need for digital signal processors. On the other hand, in embedded control applications, the analog input does not need to be reconstructed. Rather, the input is typically used to provide sensory input as the basis for feedback control. Thus, only the most recent data is needed. It is important to keep the error within the maximum bounds specified by the application.

For microcontrollers, as when using Nyquist criteria, the highest frequency component in the system can be used to determine the minimum sampling rate σ_{min}. Let's call the highest frequency component f_{max}. We define $\omega=2\pi f_{max}$. The worst-case changes in the analog input can be modelled as

$$g(t)=2^{n-1}\sin(\omega t) \qquad (17)$$

where n is the number of bits on the ADC. The maximum rate of change for $g(t)$, which we call G, occurs when $\frac{d}{dt}g(t)$ is maximized. Thus

$$G = \left.\frac{d}{dt}g(t)\right|_{max} = \omega 2^{n-1}\cdot\cos(\omega t)\big|_{\omega t=0} = \omega 2^{n-1} \qquad (18)$$

For many applications, the maximum rate of change might already be specified as a maximum slope, such as "1 degree per second" for a digital thermometer. This can be converted to the form of $\Delta a/\Delta t$ (ADC units per second) through simple scaling based on the range of the ADC. Other times, the maximum can often be estimated reasonably through experimentation. For example, suppose an analog velocity sensor is connected to a motor. Then the maximum rate of change of the velocity occurs when the motor exhibits maximum acceleration, which will occur when full power is applied to the motor. The measurement can use the analog sensor as input, by sampling as fast as the processor is capable of sampling and recording the values of the velocity in ADC units. Suppose the sampling rate is Δt, reading the log of measurements will yield a max(Δa) during a time interval Δt, where Δa is the difference between two successive ADC readings. We can then compute $\Delta a/\Delta t$. If significant digits on the input Δa becomes an issue because Δt is too small, then we can combine multiple samples, and instead compute $k\Delta a/k\Delta t$, where k is the number of samples and $k\geq 1$. Note that this Δt is for this experimentation phase only; the objective is to find T_s, which is the best value of Δt for the final application, to ensure sufficient accuracy while minimizing resource usage.

Since $\Delta a/\Delta t$ is a specification of the maximum slope, which by definition is also G, we note the relationship:

$$G = \frac{\Delta a}{\Delta t} = \omega 2^{n-1} = 2^n \pi f_{max} \qquad (19)$$

Let E be the maximum desired error, specified as a percentage of the maximum range of the signal. That is, for a maximum five percent error, E=0.05.

Table 3: Examples of various switch matrix configurations, corresponding schedule for variable-rate algorithms, and utilization for each algorithm presented

n_{col}	σ_k	σ_{min}	σ_{max}	L_s	m_{cc}	ρ_k	T_k	Schedule S (in hex)	T_s	U_{f1}	U_{f2}	U_{f3}	U_{v1}	U_{v2}
8	50	10	80	8	4	4	40	S_0 to S_3: 39,52,34,d0	10	.21	.27	.11	.13	**.08**
	60					4	40	S_4 to S_7: 31,52,34,d0						
	70					4	40							
	80					8	80							
	10					1	10							
	20					2	20							
	30					2	20							
	40					4	40							
8	10	10	240	24	4	1	10	S_0 to S_3: 0f,c3,17,63	10	.21	.27	.11	.13	**.08**
	10					1	10	S_4 to S_7: 07,43,07,43						
	20					2	20	S_8 to S_{11}: 0f,c3,07,43						
	80					8	80	S_{12} to S_{15}: 07,43,17,43						
	120					12	120	S_{16} to S_{19}: 0f,c3,07,43						
	240					24	240	S_{20} to S_{23}: 07,43,07,43						
	20					2	20							
	80					8	80							
8	5	5	150	30	3	1	5	S_0 to S_3: 07,61,0b,31	5	.42	.54	.22	.20	**.13**
	10					2	10	S_4 to S_7: 83,21,07,21						
	30					6	30	S_8 to S_{11}: 03,21,03,61						
	100					15	75	S_{12} to S_{15}: 07,21,03,21						
	150					30	150	S_{16} to S_{19}: 03,29,07,a1						
	10					2	10	S_{20} to S_{23}: 03,61,03,21						
	50					10	50	S_{24} to S_{27}: 07,21,03,21						
	80					15	75	S_{28} to S_{29}: 03,21						

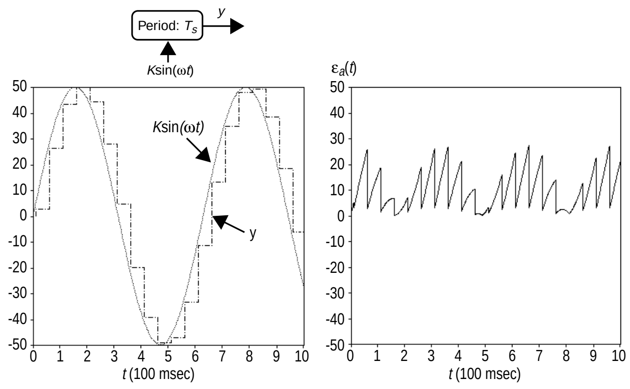


Figure 14: Example of error that results from sampling an analog signal. $\varepsilon(t) = |y - K\sin(\omega t)|$

The minimum sampling rate is then computed as

$$f_s = \frac{G}{E \cdot 2^n} = \frac{(\Delta a / \Delta t)}{E \cdot 2^n} = \frac{\omega}{2E} = \frac{\pi f_{max}}{E} \qquad (20)$$

For example, if an application has a maximum rate of change that is equivalent to sampling a 50 Hz sine wave, an 8-bit ADC is used to read a sensor, and the maximum error is 5%. Then, f_s=50π/0.05=3.1 KHz. The sampling period $T_s=1/f_s$= 318 μsec.

Interesting to note, the minimum sampling rate is not a function of the number of bits on the ADC. Rather, it creates a lower bound on the maximum error:

$$E \leq 2^{-n} \qquad (21)$$

This also leads to asking the question, if maximum error is specified as 1 ADC unit (i.e. maximum error is $\Delta a/\Delta t$=1), then what is the sampling rate that would yield this result. To compute that value, we set $E=2^{-n}$ in Eq. (20), to provide

$$f_{\Delta a/\Delta t} = f_1 = G = 2^n \pi f_{max} \qquad (22)$$

For a specific ADC device and application, it does not make sense to sample any faster than this value, as the error is already limited by the resolution of the device.

The sampling rate for an analog input has a lower bound as defined in Eq. (20) and an upper bound as defined in Eq. (22). To summarize, given f_{max},

$$\frac{\pi f_{max}}{E} \leq f_s \leq 2^n \pi f_{max}, \qquad (23)$$

or if given $\Delta a/\Delta t$,

$$\frac{(\Delta a / \Delta t)}{E \cdot 2^n} \leq f_s \leq \frac{\Delta a}{\Delta t} \qquad (24)$$

Note that these equations provide a bound that assumes a maximum error based on the full range of the ADC, which is from 0 through 2^{n-1}. In some applications, the full range of the ADC might not be used, in which case slower sampling rates might be possible. The slower rates can still be determined analytically, by adjusting the amplitude of the model sine wave in Eq. (17).

5. Summary

In this paper, we first present a model of a real-time device driver that enables the driver to execute as its own thread of control, independent of whatever control task is using the data. The interface specification of the device driver is compatible with the port-based object model of software components, thus it can be used in reconfigurable systems.

We then present an engineering approach towards determining a good sample rate for reading digital switches and analog sensors that provide continuous data. Rather than providing a single value for the sample rate, ranges that are based on application parameters are derived analytically. An application designer can use these equations to quickly determine the minimum and maximum sampling rates (or periods) for their device driver task.

The input devices we considered are representative of devices in many embedded systems, but are far from being complete. We plan to continue with this work, and to incorporate results into a tool that will enable designers to quickly configure their systems. We will also investigate output devices, for which there are similar issues with regards to determining good update rates.

6. Acknowledgements

The research described in this paper has been supported in part by the National Science Foundation and in part by the Dept. of Electrical and Computer Engineering and the Institute for Advanced Computer Studies at University of Maryland. The pinball machine that was used as an experimental testbed was built as part of a capstone design course and sponsored by Lockheed Martin Corporation; details of this project are online at *http://www.ece.umd.edu/pinball*.

7. References

[1] P. Albertos,, A. Crespo, “Real-time control of non-uniformly sampled systems,” *Control Engineering Practice*, v.7, n.4 p. 445–458, April 1999.

[2] M.C. Berg, N. Amit, J.D. Powell, “Multirate digital control system design,” *IEEE Trans. on Automatic Control*, v.33, n.12 p. 1139-1150, Dec. 1988.

[3] A. Burns, N. Hayes, M.F. Richardson, “Generating feasible cyclic schedules,” *Control Engineering Practice*, v.3, n.2, p. 151-62, Feb. 1995.

[4] “Engineering students become pinball wizards,” *The Chronicle of Higher Education*, v.44, n.25, Feb. 27, 1998. For project details, see *http://www.ece.umd.edu/pinball*.

[5] R. Gerber, S. Hong, M. Saksena, “Guaranteeing end-to-end timing constraints by calibrating intermediate processes,” in *Proc. Real-Time Systems Symposium*, pp.192–203, Dec. 1994.

[6] D. Hammond, “Microprocessor chip scans keyboard without hardware interface” *Electronics*, v.50, n1, pp.110-2, Jan 1977.

[7] M. Hassani and D.B. Stewart, “A mechanism for communicating in dynamically reconfigurable embedded systems,” in *Proc. High Assurance Systems Engineering Workshop*, Washington, DC, pp. 215–220, August 1997.

[8] D. Katcher, H. Arakawa and J. Strosnider, “Engineering and analysis of fixed priority schedulers”, *IEEE Trans. on Software Engineering*, v.19, n.9, Sept. 1993.

[9] J.O. McClain, W.W. Trigeiro, “Cyclic assembly schedules,” *IIE Trans.*, v.17, n.4, p. 346-53, Dec. 1985.

[10]A. Smith, “The merest flick of a switch,” *Practical Electronics*, v.27, n.4 pp. 24–29, April 1991.

[11]B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for hard real-time systems,” *J. of Real-Time Systems*, v.1, n.1, pp. 27-60, November 1989.

[12]S. A. Steele, “System software design trade-offs for real-time data measurement and control systems,” *Int’l J. of Mini and Microcomputers*, v.4, n.2 pp. 28-32, 1982.

[13]D.B. Stewart, R.A. Volpe, and P.K. Khosla, “Design of dynamically reconfigurable real-time software using port-based objects,” *IEEE Trans. on Software Engineering*, v.23, n.12, pp. 759–776, December 1997.

[14]H. Zeltwanger, “An inside look at the fundamentals of CAN,” *Control Engineering*, pp.81–87, January 1995.