A Predictable Device Driver Model for a Variable-Rate Software-Controlled Switch Matrix

David B. Stewart and Melissa Moy

Dept. of Electrical Engineering and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742; Email: *dstewart@eng.umd.edu*

***Abstract**: Our objective is to more systematically design and implement device drivers for embedded real-time systems. The specific goals are (1) to theoretically characterize the CPU utilization of device drivers and incorporate the analysis into the schedulability analysis of real-time systems; (2) propose alternate designs that improve the determinism, predictability, and performance, and experimentally compare them to common existing designs; (3) quickly analyze the effects on the overall schedulability of a real-time system of modifying parameters such as minimum interrupt arrival time, polling frequency, or driver execution time. As a first step towards more general analysis of device drivers, we present a case study of analyzing a switch matrix device driver.*

1. Introduction

The objective of our work is to more systematically design and implement device drivers for embedded real-time systems. The specific goals are the following:

- Theoretically characterize the CPU utilization of device drivers, and incorporate the analysis into the schedulability analysis of real-time systems.
- Propose alternate designs that improve the determinism, predictability, and performance, and experimentally compare them to common existing designs.
- Quickly analyze the effects on the overall schedulability of a real-time system of modifying parameters such as minimum interrupt arrival time, polling frequency, or driver execution time.

In this paper, as a first step towards more general analysis of device drivers, we characterize and analyze one specific device: a switch matrix. A switch matrix can be used in small scale applications such as keypads and keyboards, or in large scale applications such as building temperature control or alarm systems. We selected this device due to its significant usage of CPU time for polling large numbers of switches. The nature of the matrix hardware precludes having the switches directly generate interrupts, and requires synchronized polling of each column of the matrix.

2. Case Study: The Switch Matrix

A switch matrix is especially common for implementing keypads and keyboards [3]. Without a switch matrix, N digital input/output (I/O) ports are needed to interface to N switches, assuming a common ground for all switches. A software-controlled switch matrix can be 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 digital I/O 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 1(a) shows a 16-bit digital I/O (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 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, and activation of each switch can generate an interrupt. The disadvantage is the hardware cost. For example, if there are 64 switches, then much more wiring and a microcontroller with 64 DIO pins are needed.

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 1(b). The diodes in the switch matrix are used to prevent feedback current into inactive columns. Four of the DIO bits are configured for output; they are used to activate one of the four columns. The other four I/O bits are configured for input. The basic algorithm is to strobe a column, and read the corresponding value of each switch in that column.

The switch matrix is scalable. For example, an application with 64 switches needs an 8 by 8 matrix configuration, thus reducing the number of required I/O pins from 64 to 16. An application with 256 switches can be defined as a 16 by 16 matrix, requiring only 32 I/O pins.

Beyond using a switch matrix in keyboards, many applications require large numbers of binary sensors and switches. Examples include intelligent traffic light control, building temperature control, security and alarm systems, tactile skin for robots, and diagnostic subsystems. These applications differ from keyboards in that the I/O from a digital I/O 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 has a major impact on the overall CPU utilization of the driver.

Software that polls the switch matrix must select a column to poll, then wait for the signal to propagate through the amplifier, switch, and de-amplifier, then finally read the signal at the input port of the I/O device. As a simple demonstration of the effect of this delay: Suppose the propagation delay in an 8 x 8 switch matrix is 200 μsec, and the switches must be polled 500

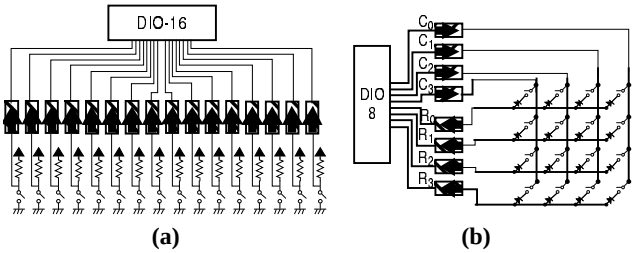

Figure 1: (a) 16-port DIO board connected to 16 switches. (b) 8-port DIO board connected to 16 switches by using a software-controlled switch matrix.

times per second. Assume another 50 µsec overhead for the interrupt handler and for writing to and reading from the I/O ports. It thus takes the software 250 µsec per column, or 2.0 msec to read all switches in the matrix. At a frequency of 500 Hz, this amounts to 100 percent CPU utilization. The switch matrix, however, is often not the only function that needs to be performed by the microcontroller.

In a multitasking environment where the switch matrix executes as a high-priority process, the 200 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 above 100 µsec, hence 200 µsec to swap out then 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.

2.1 Switch Closure Times

In many systems different kinds of switches may have different characteristics, such that the shortest amount of time any particular switch may be closed is not necessarily the same among all switches.

Figure 2 shows examples of different switches on the pinball machine that we are using as an experimental testbed. The pinball machine was initially built as an educational project sponsored by Lockheed Martin Corporation [2]. Figure 2(a) are switches that must be polled quickly, because the velocity of the ball can be very fast. For this kind of switch, we measured the fastest switch closure time to be about 10 msec. Note that

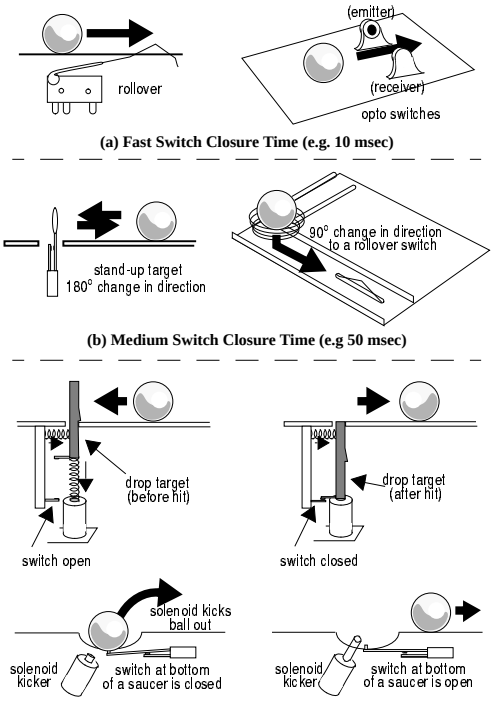

Figure 2: Example showing switches that need fast, medium, and slow polling rates.

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

Variable	Description	Measured Value (in µsec)
C_i	Interrupt handler overhead	88
C_{wr}	Write to column	15
C_{rd}	Read from row	24
C_{mod}	Increment and modulo	11
C_{l0}	Base loop overhead	15
C_{lv}	Additional loop overhead per iteration	8
C_{if}	Simple If comparison	15

this value depends on both the application and the environment; changing either of these may yield a different value for the fastest switch closure times.

Figure 2(b) shows "medium-speed" switches. Due to a 90-degree change in direction of the ball, there is a much lower velocity of the ball as it travels across the switch. We measured the shortest switch closure time to be approximately 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 2(c) shows examples of the 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.

3. Modeling the Switch Matrix Driver

The primary requirements for polling a switch matrix are conflicting:

- poll every switch fast enough to catch every switch closure
- poll switches as slow as possible to minimize CPU time.

In this paper, we assume the polling software runs as a high-priority interrupt handler.

To capture the overhead of calling the interrupt handler and implementing the individual instructions, we use the method that Katcher used to characterize real-time scheduling overhead [4]. To quantify our analysis, we measured the values for the 8 MHz 8088 processor embedded in our pinball machine testbed. These results are shown in Table 1. Although for our experiments we use a low-performance microprocessor (as typically found in embedded systems), the analysis and procedures we use apply to any processor, although the results of which algorithm is best may differ. The propagation delays, however, are independent of the speed of the processor.

The interrupt handling overhead, C_i, includes saving/restoring registers, initialization local variables, and disabling and re-enabling interrupts. 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. C_{mod} is the time to perform the *mod* operation. The loop overhead is quantified through C_{l0} and C_{lv}. C_{if} is the time to perform an increment and compare.

The CPU execution time used by one iteration of the interrupt handler, including all overhead, is C_{cyc}. The period of the interrupt handler is T_{cyc}. The utilization of the interrupt handler, and hence of the switch matrix polling software, is $U_{cyc} = C_{cyc}/T_{cyc}$. We define σ_{min} as the shortest switch closure time, and n_{col} as the number of columns in the switch matrix. In fixed-priority algorithms, $T_{cyc}=\sigma_{min}$ if all columns

polled on same cycle, or $T_{cyc}=\sigma_{min}/n_{col}$ if only a single column is polled on each cycle.

The analysis assumes that the interrupt handler 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 perform mechanical switch debouncing.

3.1 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. In Figure 3, we present three different fixed-rate algorithms. Algorithm F1 is the traditional implementation, where every column is polled on each timed interrupt cycle. Algorithm F2 and F3 represent an interrupt handler that executes more frequently, but only polls one column per cycle.

We computed the utilization for each of these algorithms.

Algorithm F1:

$$U_{cyc} = \frac{C_{l0} + n_{col}(C_{lv} + C_{wr} + C_{rd} + D_p) + C_i}{\sigma_{min}} \quad (1)$$

Algorithm F2:

$$U_{cyc} = (C_{wr} + C_{rd} + D_p + C_i + C_{mod})\frac{n_{col}}{\sigma_{min}} \quad (2)$$

Algorithm F3:

$$U_{cyc} = (C_{wr} + C_{rd} + C_i + C_{mod})\frac{n_{col}}{\sigma_{min}} \quad (3)$$

A comparison of the utilization for these algorithms is shown in Figures 4 and 5. Figure 4 compares the results assuming the propagation delay D_p=200 µsec, with the shortest switch closure time σ_{min} varying from 5 to 50 msec. Figure 5 shows σ_{min} constant at 10 msec, but D_p varies from 0 to 300 µsec. Algorithms V1 and V2 are variable-rate algorithms, described next.

3.2 VARIABLE-RATE Switch Matrix Polling Algorithms

Given that not all switches must be polled at the same frequency, we developed a driver model for polling columns at different rates. It is essential that the implementation of the model is efficient, such that additional overhead of selecting and scheduling which column to poll 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 speeds 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 device driver viewpoint.

First, we need to determine an appropriate schedule given the shortest closure switch time for each column k (σ_k). Our method is based on building a cyclic schedule, as commonly used for job shop scheduling [1,5]. Note that we assume it is acceptable to poll any column faster than the specified switch closure time. 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 developer. From this value, σ_{min} and σ_{max} are computed as,

$$\sigma_{min} = \min(\sigma_k)\Big|_{k=0}^{7}, \quad (4)$$

$$\sigma_{max} = \left\lfloor \frac{\max(\sigma_k)\Big|_{k=0}^{7}}{\sigma_{min}} \right\rfloor \sigma_{min} . \quad (5)$$

The length of the schedule, L_s, is computed as the ratio of σ_{max} to σ_{min} as, $L_s = \sigma_{max}/\sigma_{min}$, assuming that the frequency of the interrupt handler will be set to $1/\sigma_{min}$. Equation (5) 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. σ_{max}, σ_{min}, or both can be lowered 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 in units of number of iterations of the interrupt handler, ρ_k, is computed as follows:

$$\rho_k = \left\lfloor \frac{L_s}{\lceil \sigma_{max}/\sigma_k \rceil} \right\rfloor \quad (6)$$

Algorithm F1 *All columns every cycle*	*Algorithm F2* *one column per cycle* *write before read*	*Algorithm F3* *one column per cycle* *read before write*

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
	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]*
		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 3: Fixed-rate switch polling algorithms.

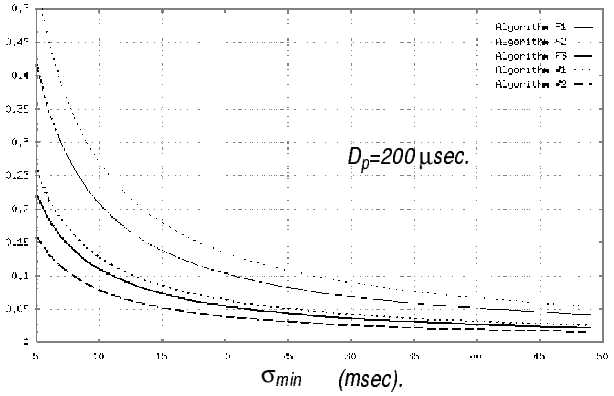

Figure 4: Comparison of utilization vs. shortest switch closure time for switch polling algorithms. D_p=200 µsec.

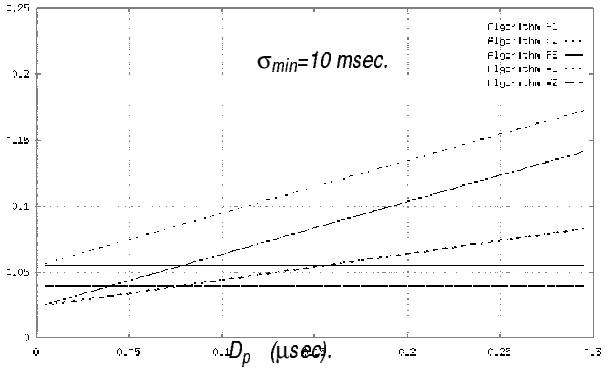

Figure 5: Comparison of utilization vs. propagation delay for switch polling algorithms. σ_{min}=10 µsec.

The polling period T_k for each column is then computed as $T_k = \rho_k \sigma_{min}$. T_k and ρ_k are used to ensure that the polling times for individual columns are harmonic and are a multiple of the interrupt handler. If necessary, polling occurs more often than defined by σ_k, but never less often. The polling time T_k is used to build the schedule *S*.

The schedule *S* is stored in an array of L_s elements of n_{col} bits each. Setting bit *k* in element S_j means to poll column *k* on cycle *j*, where on each interrupt, $j=(j+1) \bmod L_s$. 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 using the algorithm shown in Figure 6, where the function *bits(x)* returns the number of bits set in the value *x*. Using this algorithm, the maximum number of columns that must be polled on an interrupt, m_{cc}, is computed as,

$$m_{cc} = \left\lceil \frac{\sum_{k=0}^{n_{col}-1} \lceil \sigma_{max} / T_k \rceil}{L_s} \right\rceil \quad (7)$$

Given the above schedule, Figure 7 shows two separate algorithms for an interrupt handler to implement the schedule.

The CPU utilization for these algorithms is the following:

Algorithm V1:

$$U_{cyc} = \frac{C_i + C_{l0} + n_{col}(C_{lv} + C_{if}) + m_{cc}(C_{wr} + C_{rd} + D_p + C_{mod})}{\sigma_{min}} \quad (8)$$

Algorithm V2:

$$U_{cyc} = \left(C_i + C_{l0} + \frac{n_{col}}{m_{cc}} \cdot (C_{lv} + C_{if}) + C_{wr} + C_{rd} + C_{mod} \right) \frac{m_{cc}}{\sigma_{min}} \quad (9)$$

As can be seen in Figure 4 and Figure 5, these algorithms (plotted for σ_k={10,10,20,80,120,240,20,80}) perform similar or better than the best fixed rate algorithm.

To experimentally verify the correctness of the analytically-derived plots, we measured the actual execution time for the interrupt handler for all five implementations; the results are shown in Table 2. Although the measured times tend to be higher than the theoretical estimations computed above, the results do support our claim that a variable-rate switch matrix handler can significantly reduce CPU utilization for applications where not all switches need to be polled at the same rate as the switch with the shortest closure time.

```
sort Tk in increasing order
for each column k (in order of Tk) do
   b=0; valid = FALSE;
   while (valid == FALSE)
      for j = 0 to ρk do
         if bits(Sj nkol) == b then
               // try to fit in as b^th column to poll on every ρk^th cycle.
            valid = TRUE;
            for i = j to Ls step ρk do
               if bits(Sj nkol) > 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 Sj, Sj+ρk, Sj+2ρk, etc.
   for i = j to Ls step ρk do
      set bit k in Si
   end for
end for
```

Figure 6: Algorithm to build harmonic schedule for variable-rate switch polling algorithms.

Table 2: Experimental measurements of utilization for σ_k={10,10,20,80,120,240,20,80} (All times in μsec).

Algorithm	T_{cyc}	Measured C_{cyc}	Measured U_{cyc}	Theoretical U_{cyc}	Eq. No. for theoretically computing U_{cyc}
F1	10,000	2550	26 %	21 %	(1)
F2	1250	402	32 %	27 %	(2)
F3	1250	151	12 %	11 %	(3)
V1	10,000	1442	14 %	13 %	(8)
V2	2500	202	8 %	5 %	(9)

4. SUMMARY

We present an exact characterization of various algorithms for a switch matrix device driver. Our case study demonstrates that a significant improvement in the CPU utilization can be obtained through systematic analysis and refinement of design. The theoretical analysis aided in comparing various designs, then selecting the best design for the given application.

More important than the results of the case study, which are specific to applications that use switch matrices, is the demonstration of methods that can be used to systematically analyze and configure real-time device drivers. In the near future, we will address other classes of device drivers, such as serial input/output, analog-to-digital converters, and low-level network interfaces. Our ultimate objective is to create a device driver framework that will aid in reducing development time and improve the quality, functionality, and reusability of device drivers for embedded systems [6].

5. REFERENCES

[1]A. Burns, N. Hayes, M.F. Richardson, "Generating feasible cyclic schedules," *Control Engr. Practice*, v.3, n.2, pp. 151-62, Feb. 1995.

[2]"Engineering Students Become Pinball Wizards," *The Chronicle of Higher Education*, v.44, n.25, Feb. 27, 1998. For project details, see *http://www.ee.umd.edu/pinball*.

[3]D. Hammond, "Microprocessor chip scans keyboard without hardware interface," *Electronics*, v.50, n.1, pp.110-12, Jan. 6 1977.

[4]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.

[5]J.O. McClain, W.W. Trigeiro, "Cyclic assembly schedules," *IIE Transactions*, v.17, n.4, p. 346-53, Dec. 1985.

[6]D.B. Stewart, "An I/O device driver model and framework for embedded systems," in *Proc. of IEEE Workshop on Middleware for Distributed RTSS*, San Francisco, Dec. 1997.

Algorithm V1 ***m_{cc} columns per cycle***	***Algorithm V2*** ***one column per cycle***

swInit:
initialize L_s (from ()) and S_j from
data generated by algorithm in Fig 6.
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
	swInit:
	initialize m_{cc}, L_s and S_j
	j = 0; *m*=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 7: Variable-rate switch polling algorithms.