Application Note

Designing with Shift Registers

Emrys Maier

ABSTRACT

Shift registers provide a simple, low cost, and flexible method for increasing the total number of input or output (IO) pins on a system controller. The fact that shift registers can be connected in series, or *daisy-chained* together, means that very few IO pins are required to support a large number of devices. The only two real caveats for their use are that they cannot provide a significant amount of power directly, and the IO signals must be much slower than the system clock.

Stepper motors require multiple relatively slow input signals that can be controlled through shift registers so long as the power for driving the motors comes from separate dedicated drivers. Appliances such as air conditioners, washers, dryers, and refrigerators commonly use stepper motors that benefit from shift registers. Industrial electronics such as servo motor drive controls for multi- or single-axis motors utilize shift registers for the same reasons.

It is also common to see shift registers in user interfaces for both LED control and keypad polling since humans are generally able to discern differences only on the millisecond time scale. Medical applications such as oxygen concentrators utilize shift registers for monitoring keys and lighting indicators. Similarly, fire control panels and elevator control panels commonly use shift registers for LED outputs and button inputs.

This document provides a complete description of the most common shift register functions (164, 165, 595), an explanation of how each common function operates, key design concepts to prevent issues when using them, and an example system design utilizing existing parts.

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Overview

1.1 Types of Shift Registers

There are three primary types of shift registers: Serial-In Parallel-Out (SIPO), Parallel-In Serial-Out (PISO), and so-called universal shift registers, which contain both parallel-in and parallel-out connections. Often these are shortened to parallel-in, parallel-out, and universal, as the parallel connections are the key concern. All shift registers have serial input and output connections to allow for series connection (aka daisy chaining) of devices.

Parallel-in shift registers are used to increase the number of inputs on a controller, an application often referred to as *input expansion*. Devices like the SN74HCS165 provide eight additional inputs per integrated circuit (IC).

Parallel-out shift registers are used to increase the number of outputs on a controller, an application often referred to as *output expansion*. Devices like the SN74HCS164 provide eight additional outputs per IC.

Universal shift registers provide the ability to switch between reading inputs into the internal registers or sending the internal data to outputs. They are not used as commonly, however, as they provide fewer storage bits than their dedicated input- or output-type counterparts. For example, a 16-pin universal shift register like the CD74HC194 can be used for inputs or outputs, however it can only contain up to 4 bits of data, while the aforementioned SN74HCS165 and SN74HCS164, both of which are also in 16-pin packages, can hold eight bits of data each.

1.2 Default State of a Shift Register

Shift registers, like many sequential logic devices, contain flip-flops which rely on the previous output state to determine the current output state. For example, see Table 1-1 for the common D-type Flip-Flop logical function table. The output when the clock is not being pulsed is purely dependent on the previous state, Q_0.

Table 1-1. Typical D-type Flip-Flop Function Table

D	CLK	Q
L	↑	L
H	↑	H
X	L, H, or ↓	Q_0

Because of this dependency on the previous state, the values stored in a shift register are unknown at startup, and remain unknown until data is loaded into the registers. This can be problematic for some systems that depend on the outputs of the shift registers to be in a certain state at startup.

There are three ways around this. First, many shift registers include a direct clear pin to force all internal registers to a logic LOW. This allows a system designer to have a power-on-reset (POR) signal clear the registers immediately when power is applied.

The second, and most common, method is to load in data to overwrite the unknown values. It is common practice to have an initialization routine for shift registers that pushes initial values into all registers to prevent undesired behavior.

Finally, some devices provide the ability to put the outputs into a high-impedance state, which allows pull-up or pull-down resistors to set something of a *default* value at the outputs. Alternatively, 3-state buffers can be added to any shift register's outputs to provide a high-impedance state. This should not be confused with changing the internal register values, however, as this method will only set the output values as long as they are in the high-impedance state. The internal registers are still unknown and must be overwritten before beginning operation.

Disabling the outputs with a POR signal combined with writing in known values can create a very effective initialization routine for a system with many shift registers. The POR just needs to last long enough for data to be written into the registers at system startup.

For details on creating a POR signal, please see this video: Generate a Reset Signal at System Power On

1.3 164 Function Shift Registers

The 164 function is an 8-bit parallel-out shift register. The serial data inputs are logically ANDed together to provide an easy method for disabling the register. If either A or B is in the LOW state, then the data input to the shift register is LOW, and each rising-edge clock input will load a LOW, irrespective of the state of the opposite input. To permanently enable the serial data input, either A or B can be tied directly to V_{CC}, and the opposite input can be used as the serial data input.

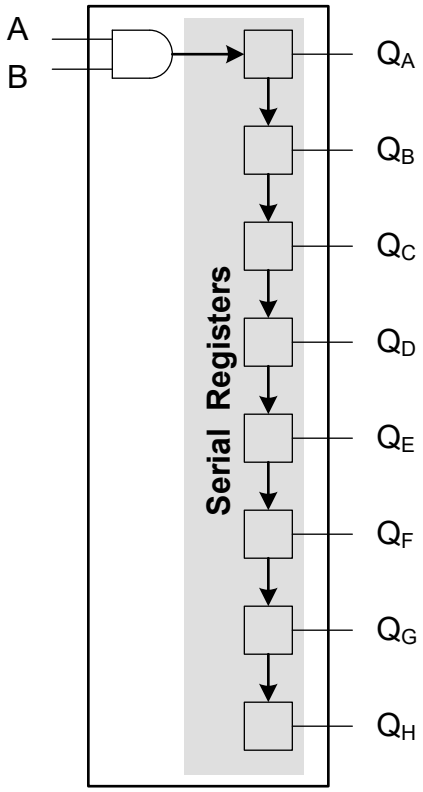


Figure 1-1. Visual Representation of the Effect for Clock Input of 164 Function Shift Registers

When a rising edge is detected at the CLK input, the eight registers are loaded with the data value indicated by the arrows in Figure 1-1. To be clear, the result of A AND B is loaded into Q_A, while the value that was in Q_A is loaded into Q_B, the value that was in Q_B is loaded into Q_C, and so on. The last value, which was in Q_H, is *shifted out*, or, in other words, it is overwritten.

The 164 function comes with a dedicated asynchronous active-low clear pin ($\overline{CLR}$) which allows forcing the internal register (and thus the output) values to zero. If this pin is held low during startup, the outputs will remain low as soon as the device is within the operating voltage range.

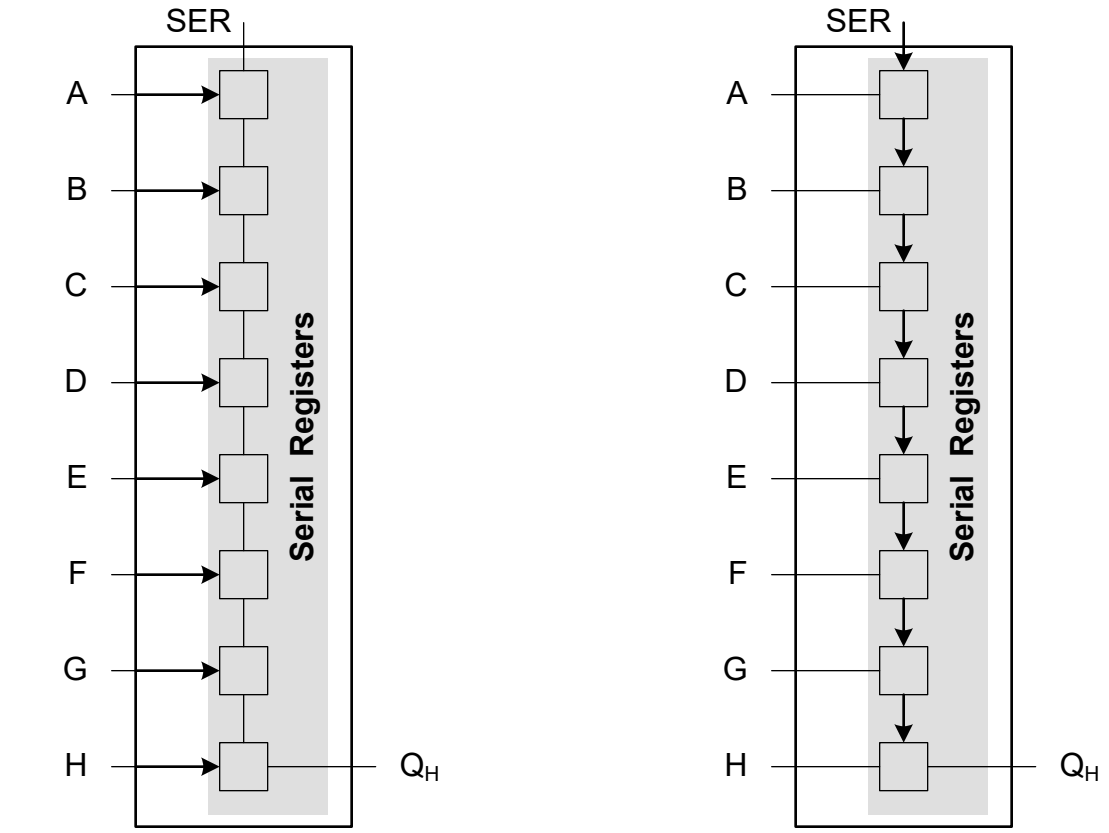
For normal functionality, eight bits of data are loaded into the serial data input one at a time with eight clock pulses. With each rising edge at the clock input, the outputs will change immediately to match the values inside the serial registers.

The output Q_H comes directly from the last internal shift register, which allows it to be used for daisy-chaining devices together.

1.4 165 Function Shift Registers

The 165 function is an 8-bit parallel-in shift register. The parallel data inputs are logically connected to the internal serial registers. The shift or load (SH/$\overline{LD}$) input pin determines the operating state of the device.

In the *load* mode, the inputs are asynchronously copied into the internal registers. The clock (CLK) and clock inhibit (CLK INH) have no effect in this mode. The Q_H output will immediately take on the value of data at the H input.



A low state at the SH/$\overline{LD}$ input loads data into the serial registers

A rising edge at the CLK input shifts the data in the serial registers

Figure 1-2. Visual Representation of the Primary Operating States for the 165 Function.

In the *shift* mode, when a rising edge is detected at the CLK or CLK INH input, assuming the opposite input is in the LOW state, the eight registers are loaded with the data value indicated by the arrows in right hand diagram of Figure 1-2. To be clear, the value at SER is loaded into A, while the value that was in A is loaded into B, the value that was in B is loaded into C, and so on. The last value, which was in H (and is accessible at the output Q_H), is *shifted out*, or, in other words, it is overwritten.

For normal functionality, the values are loaded into the register, then seven clock pulses are applied to read all eight internal values from the Q_H output pin.

The output Q_H comes directly from the last internal shift register, which allows it to be used for daisy-chaining devices together.

1.5 595 Function Shift Registers

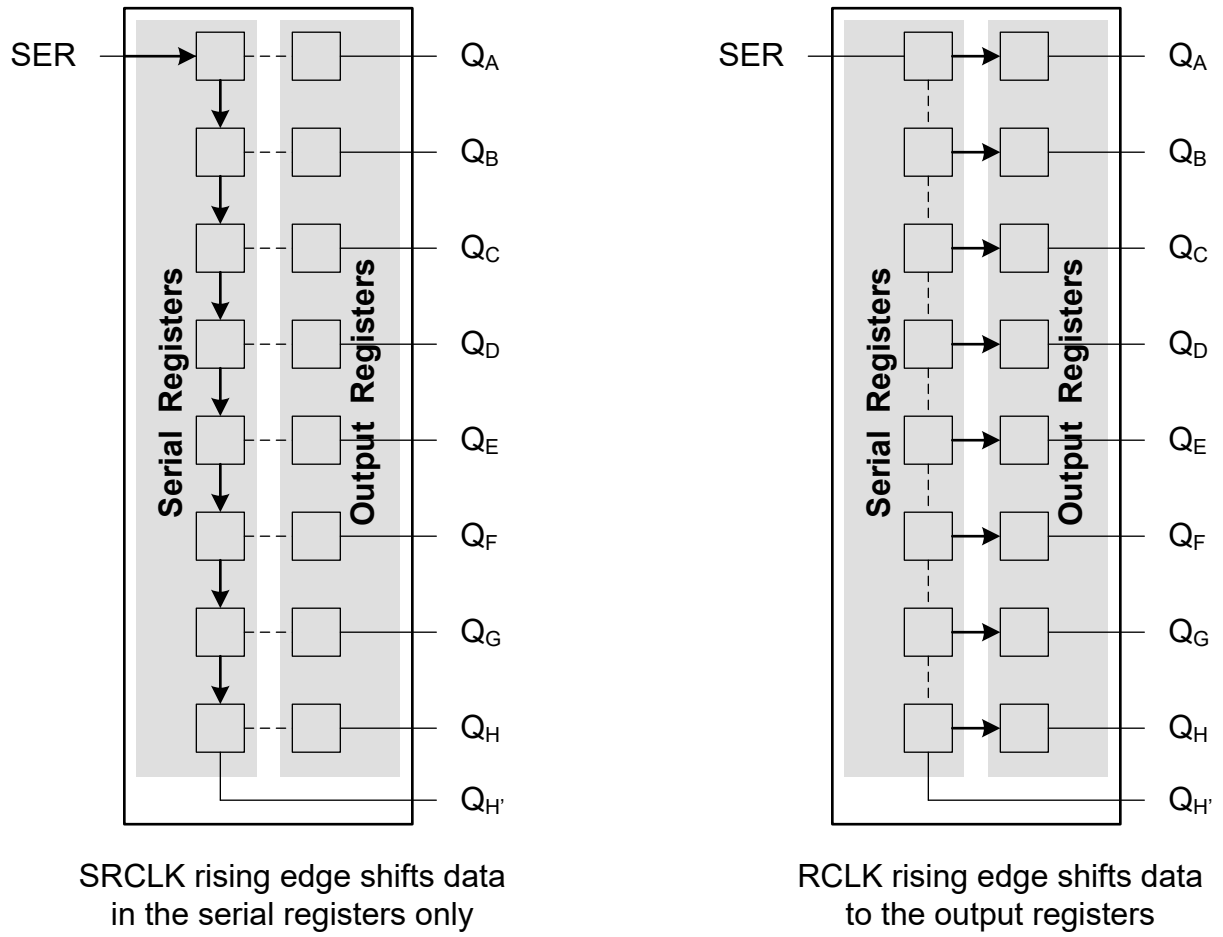


Figure 1-3. Visual Representation of the Effect for Each Clock Input of 595 Function Shift Registers

The 595 function includes latched output registers, which allow the serial shift registers to change while holding the outputs constant. Figure 1-3 illustrates the operation of the 595 function. On the rising edge of the SRCLK input, the data at SER will be loaded and the serial registers will shift by one location as indicated by the arrows in the left diagram of Figure 1-3. To be clear, the value at the SER input will be loaded into the first serial register, the value that was in the first serial register will be loaded into the second serial register, the value that was in the second serial register is moved to the third serial register, and so on. The value that was in the last register, which was previously accessible from the $Q_{H'}$ output pin, is “shifted out,” or, in other words, it is overwritten.

On the rising edge of the RCLK input, illustrated in the right diagram of Figure 1-3, the data in the serial registers will be copied to the output registers. The shift registers will still retain the same data in the same locations after this operation is complete.

It is possible to operate this device with SRCLK and RCLK directly shorted together. In this mode of operation, the previous data in the shift registers is first sent to the outputs and then is shifted to the next positions. In this way, the outputs will always be one clock pulse behind the serial register values. To be clear, it will take nine pulses to send eight bits of data to the outputs, and those values will be read into the device with the first eight pulses.

The output $Q_{H'}$ comes directly from the last internal shift register, which allows it to be used for daisy-chaining devices together.

1.6 Daisy-Chain Two Shift Registers

Any shift register can be daisy-chained, so long as it has a serial data input and direct access to the last serial register’s contents. In order to connect two shift registers in series, connect the serial output of one device

(usually Q_H or $Q_{H'}$) to the serial input (usually SER) of the next device. Typically, all of the clock signals will be shorted together, although they can be separated for unique cases.

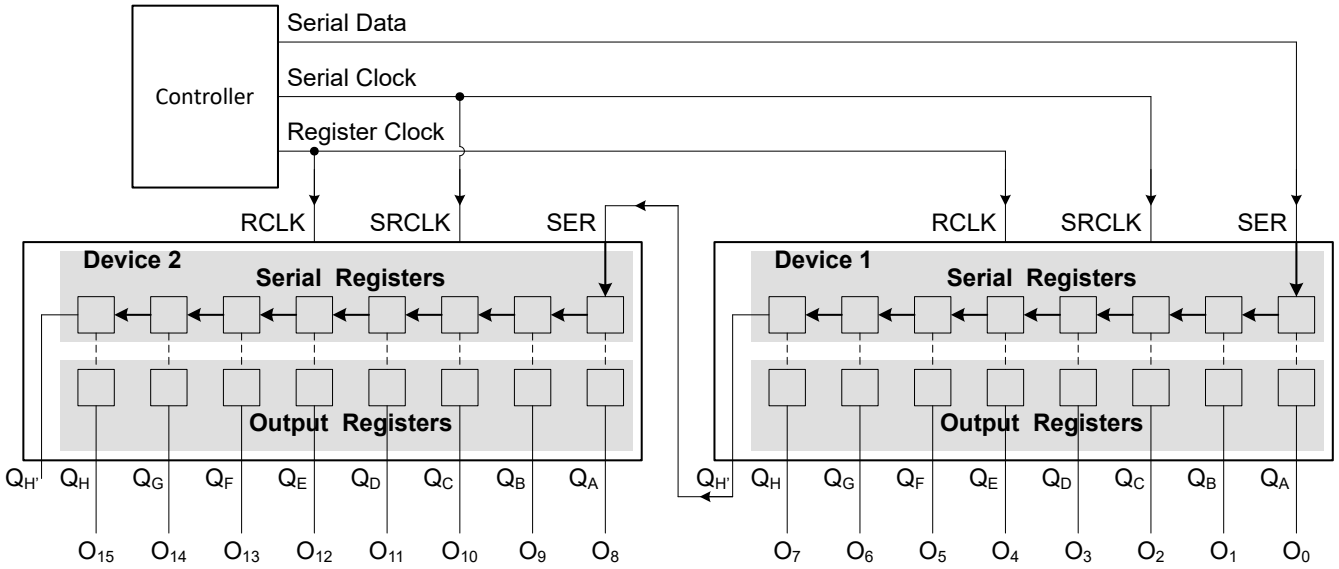


Figure 1-4. Daisy-Chaining two 595 Function Shift Registers Together

Figure 1-4 provides an example of connecting two 595 function shift registers together to convert 3 output pins into 16 output pins (O_0 to O_{15}). The first device connects to the system controller as usual, with the serial data and clocks coming directly from the controller. For the second device in the chain, the only difference is that the data input comes from the $Q_{H'}$ output of the previous device rather than the controller.

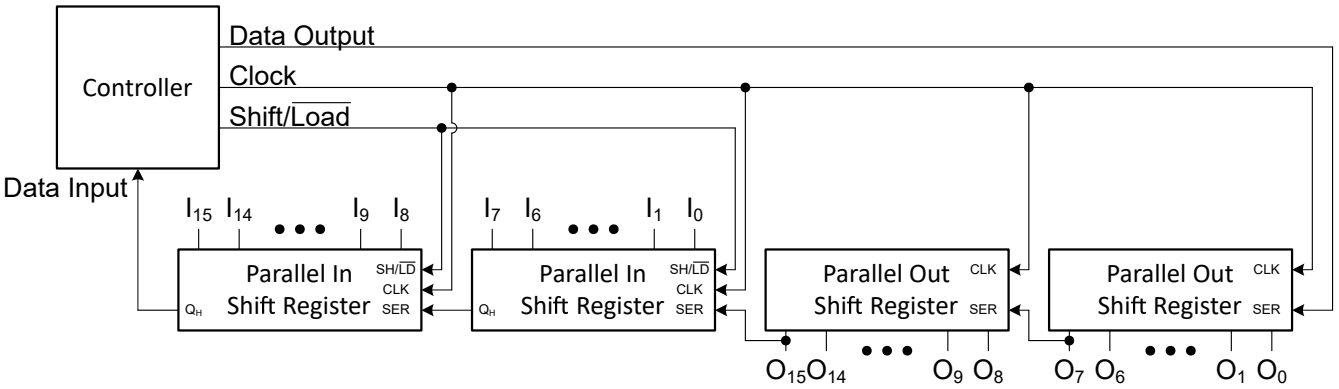


Figure 1-5. Block Diagram for Adding 16 Outputs and 16 Inputs to a Controller Using 4 I/Os

Figure 1-5 shows an example of connecting two parallel-in and two parallel-out shift registers to provide 16 new inputs and 16 new outputs. By using this configuration, the number of I/Os required by the controller remains only four, while the number of outputs (or inputs) can be increased almost indefinitely. There are multiple design challenges that need to be addressed to ensure proper operation. See Section 2 for details.

2 Design Challenges

2.1 Controller Loading Limits

Most CMOS logic circuits are designed to operate with relatively light loads. For example, the SN74LVC1G08, which has very strong output drivers, is designed to operate with a capacitive load of 50 pF or less (SN74LVC1G08, section 6.8). The SN74AUP1G08 has weaker output drivers and is specified for a load of 30 pF or less (SN74AUP1G08, section 6.9). For another example, the MSP430FR2311 has the output timing specified only up to 20 pF (MSP430FR2311, Table 5-11).

The output loading can usually be found in the data sheet timing specifications as a test condition. Most modern CMOS-based controllers will have relatively weak outputs, so it's a good idea to avoid loading them too heavily. It is recommended to avoid exceeding the tested load values provided in the data sheet. This limit can be bypassed, however, by adding a buffer to redrive the signal.

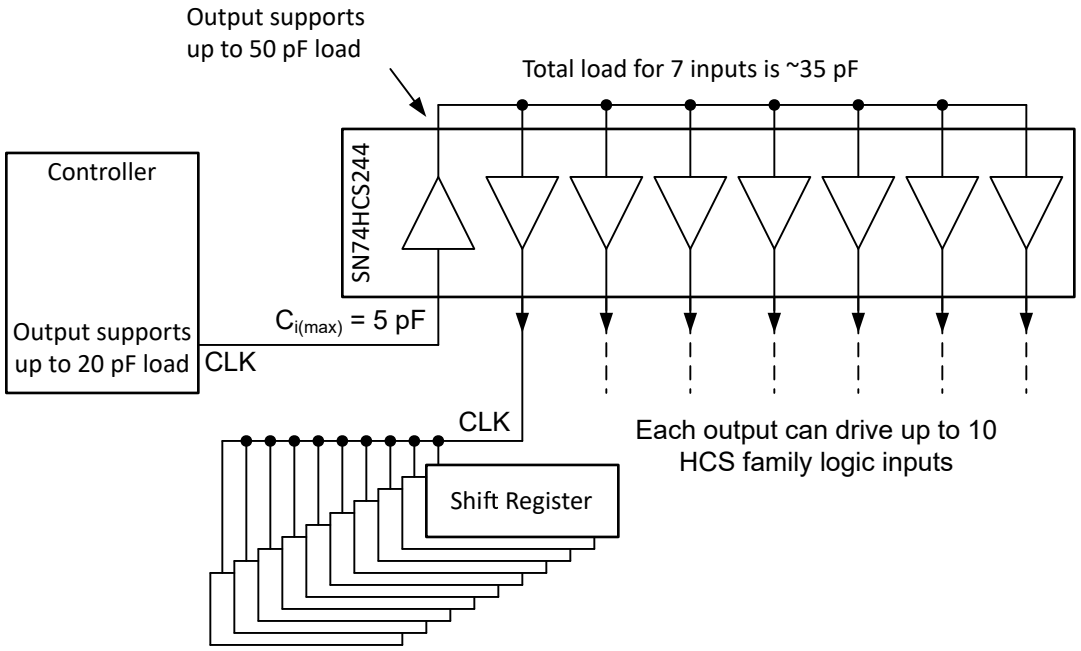


Figure 2-1. Example of Signal Fanout Using Only Logic Buffers

Discrete buffers can be used as many times as necessary to redrive the signal. Creating duplicate signals from a single input is known as *fanout* and can be accomplished using normal discrete logic buffers such as the SN74HCS244. See Figure 2-1 for an example of redriving a single weak output to multiple 50 pF loads. Each buffer will add some delay to the signal, so be sure to follow the guidelines in Section 2.3 to avoid issues.

2.2 Operating over Large Distances

When designing with small circuits, the time it takes electric signals to travel is typically negligible. As circuits get physically larger, delays increase and can become problematic. For example, in large digital billboard signs, the distance from the controller to the farthest corner of the display could be fifty feet or more. This large distance can cause problems for designing with shift registers.

In the ideal case (light speed in a vacuum), signals could propagate across 1 ft in approximately 1 ns, however we find in practice that signals in a transmission line (or on a PCB) tend to propagate at about half that speed, so 2 ns is a more realistic estimate. Additionally, this does not take into account the physical position on the line and other transmission line effects.

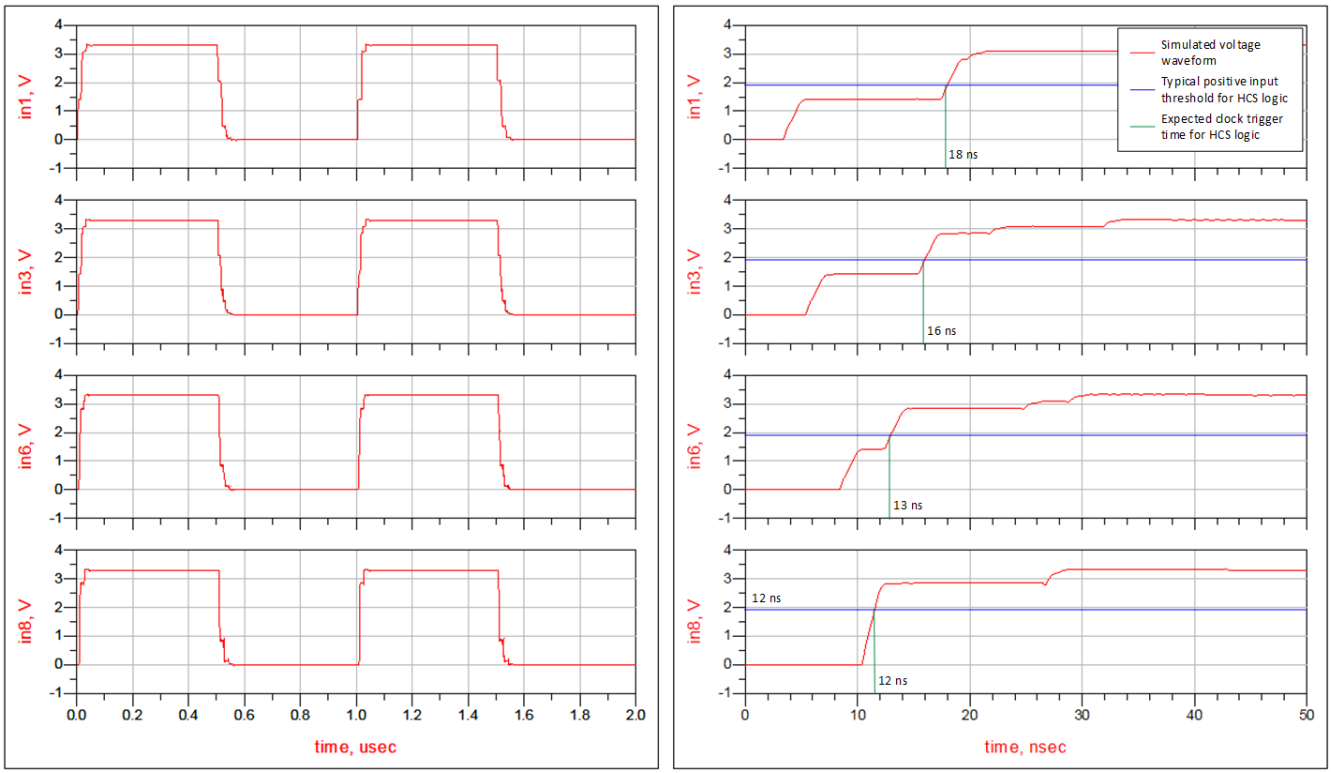


Figure 2-2. (left) Simulated Clock Waveforms Received by the First, Third, Sixth, and Eighth Shift Registers. (right) Zoomed Versions of the Same Four Waveforms to More Clearly Show the Rising Edge.

Utilizing the available IBIS models on TI.com and the PathWave ADS simulator, Figure 2-2 shows the same clock signal arriving at eight different shift registers spaced six inches (1 ns delay) apart from each other. The right-hand side is zoomed in on the rising edge to provide more details. The “stair step” waveform shown is caused by the transmission line effects (reflections), an explanation of which are beyond the scope of this application report.

Looking at the clock trigger times marked it green, it can be seen that there is a 6 ns delay from when the last shift register is triggered to when the first one is triggered. This seems counter-intuitive, but looking closer at the waveform shows that the wave does reach in1 first at approximately 4 ns, however the reflections cause the waveform to seemingly pause at approximately half the total voltage from approximately 5 ns until 18 ns.

There are two major issues that need to be considered from the input signals shown in Figure 2-2. First, the mismatch in clock trigger timing can cause data loss when shift registers are daisy-chained together. In the example shown, only one buffer is used to drive the line in order to better show the transmission line effects, however it is common practice to redrive signals using buffers throughout a system, which can help with signal integrity, but will add additional delays. For an explanation of how this can cause data loss, see the explanation in Section 2.3.

Second, although in this case the “stair step” in the input signals is only 6 ns, it does illustrate that long transmission lines can create slow and possibly non-monotonic edges. For many standard CMOS devices, slow edges and holding an input voltage near VCC/2 can cause internal oscillations, excessive power consumption, and reduction in reliability. Schmitt-trigger architecture inputs will prevent all of these issues, so the HCS logic family is ideal to handle this type of operation.

For a full explanation on how slow inputs affect standard CMOS devices, see the application report Implications of Slow or Floating CMOS Inputs. For a full explanation on the benefits of the Schmitt-trigger architecture, see the application report Understanding Schmitt Triggers.

2.3 Data Loss Due to Signal Timing

For the majority of systems involving shift registers, delays will be short and no added delays will be required. However, when building a system with many shift registers, especially in larger and more complex systems,

timing can become a critical concern. If one shift register in the chain is clocked much earlier than the next, then data can be lost.

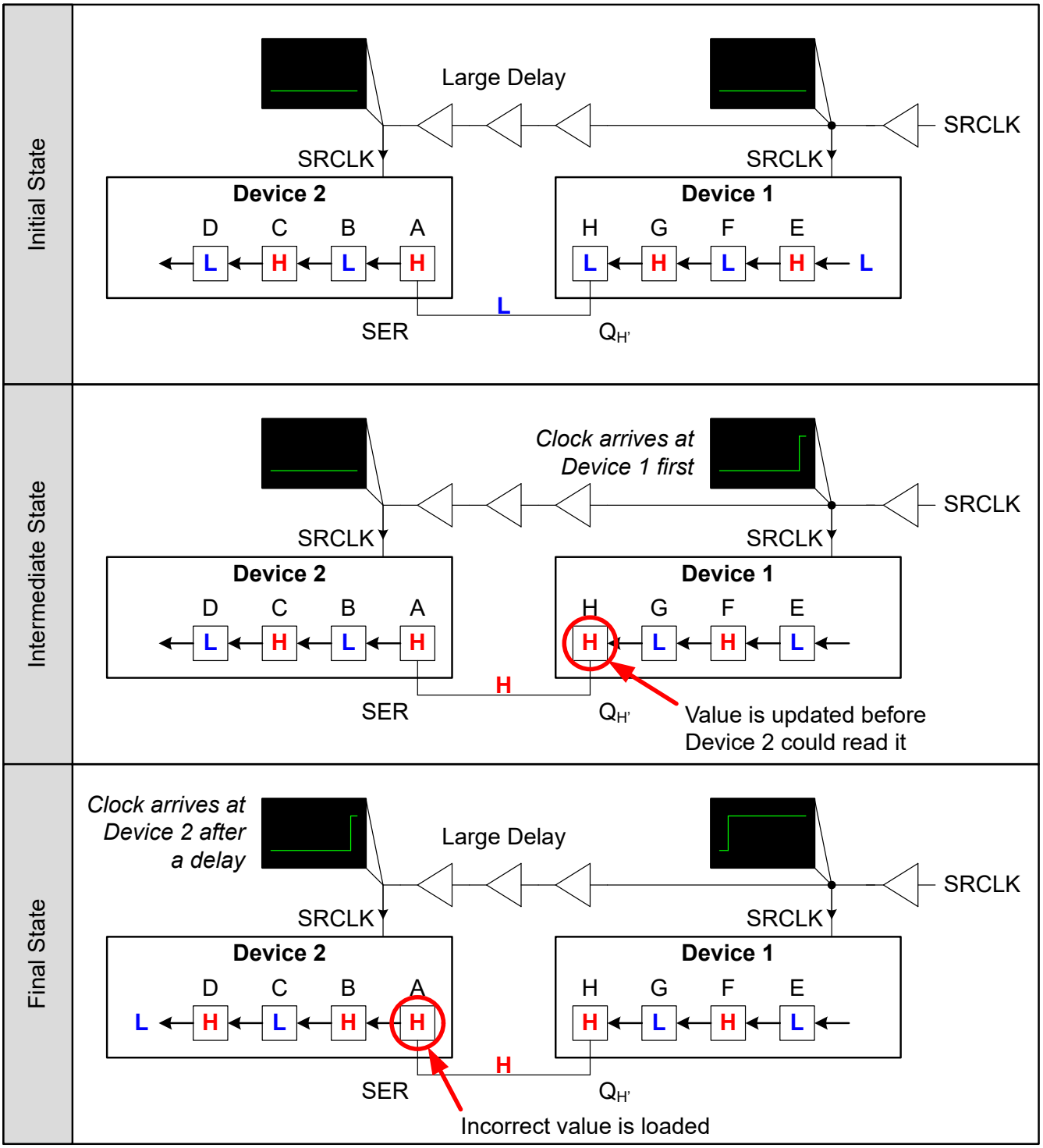


Figure 2-3. Example of Configuration That Can Result in Data Loss

Figure 2-3 shows an example scenario in which data can be lost. The initial configuration is shown in the top diagram, with the stored data shown as an alternating pattern, 01010101. The clock signal arrives first at Device 1, which causes all values in Device 1 to shift (to the left in the image). The value in register G is moved to register H (middle diagram). Device 2 does not receive the shift register clock pulse until after the value stored in register H has been overwritten, and thus the value loaded into register A of Device 2 is what was stored in register G of Device 1, and the value that was in register H of Device 1 is lost (bottom diagram). The data, which

has been shifted one place to the left, is now 01011101. The fourth bit (counting from the right) has flipped to an incorrect value only because of the timing of the clock inputs.

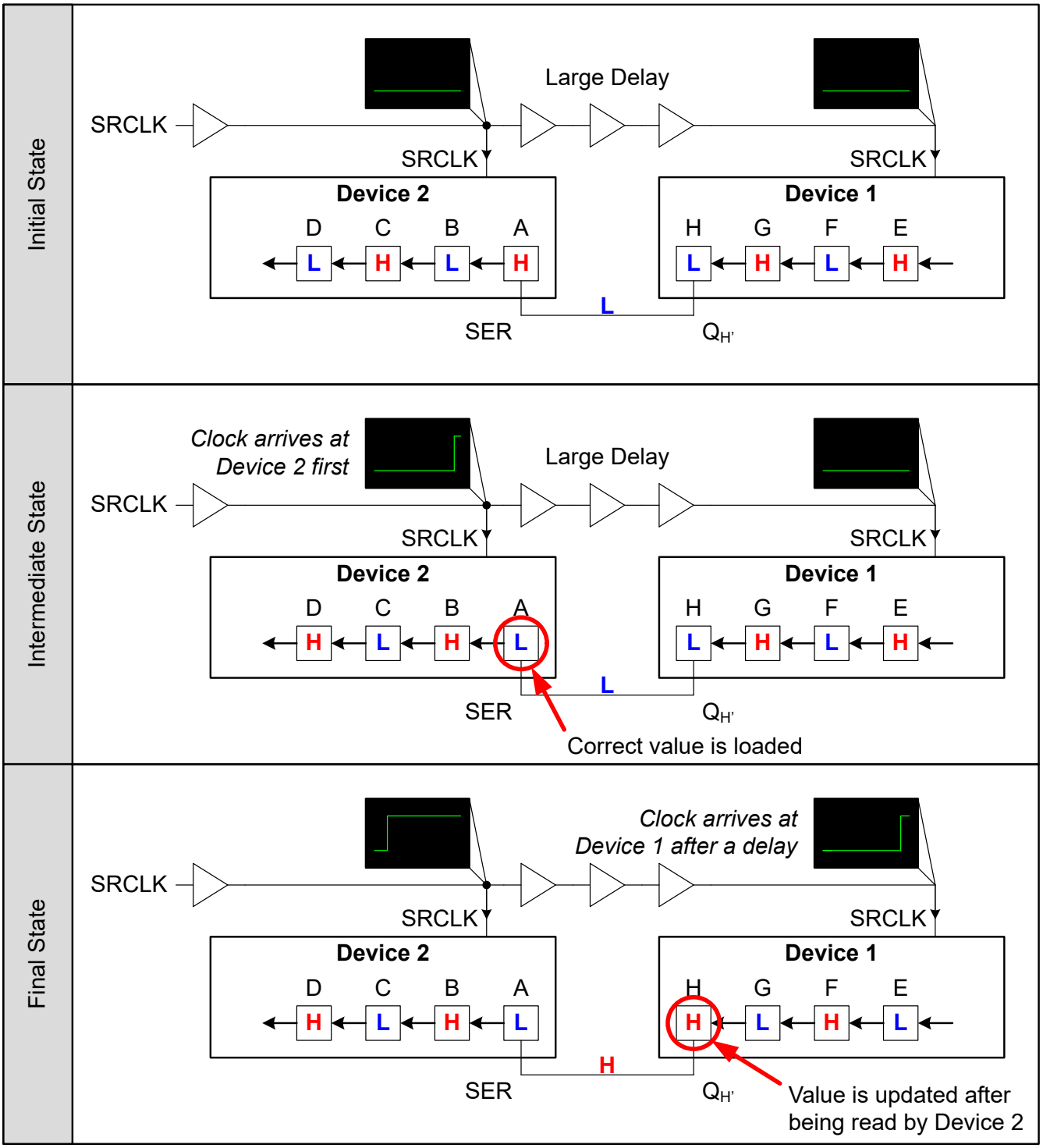


Figure 2-4. Example of Long Clock Delay That Does Not Result in Data Loss

The solution to this issue is to simply ensure that the clock reaches the devices in the reverse order that data is shifting. By sending the clock to the last device in the chain first, placing added delays in the reverse direction, it can be ensured that no data will be lost.

Figure 2-4 shows an example with this reversed order. The initial state is the same as the first example (top diagram), however the clock arrives to Device 2 first in this scenario. This allows Device 2 to read in the $Q_{H'}$

value from Device 1 before it changes (middle diagram). When the clock finally reaches Device 1, the data is shifted and overwrites the value in register H, however it has already been copied into Device 2 and no data is lost.

This configuration can be used with any number of shift registers. See Section 3 for a detailed design example.

2.4 Data Rate Limitations

Although the number of shift registers connected can be expanded indefinitely, the speed at which data can be loaded into the registers is finite. Each device has a limited maximum clock speed, and data is sent serially from one device to the next, so the total number of bits loaded and the clock frequency applied will affect the total time it takes to read or load the registers. The total time required to load a given number of serial shift registers (t_{load}) can be calculated using Equation 1 with N=Total registers to load and F_{clk}=Shift register clock frequency (Hz).

$$t_{load} = \frac{N}{F_{clk}} \quad (1)$$

For example, the SN74HCS595 has a maximum clock speed of 60 MHz. If 16 serial shift registers (128 bits total) are to be loaded at this speed, the fastest it can be completed is 2.133 µs. With a more typical clock speed of 1 MHz, the total time to load the same 128 registers would be 128 µs.

Similarly, when loading inputs from multiple parallel-in shift registers such as the SN74HCS165 the same equation can be used. For example, if reading 32 registers (4 devices) with a 100 kHz clock, the time to complete the read would be 320 µs.

2.5 Software Overview

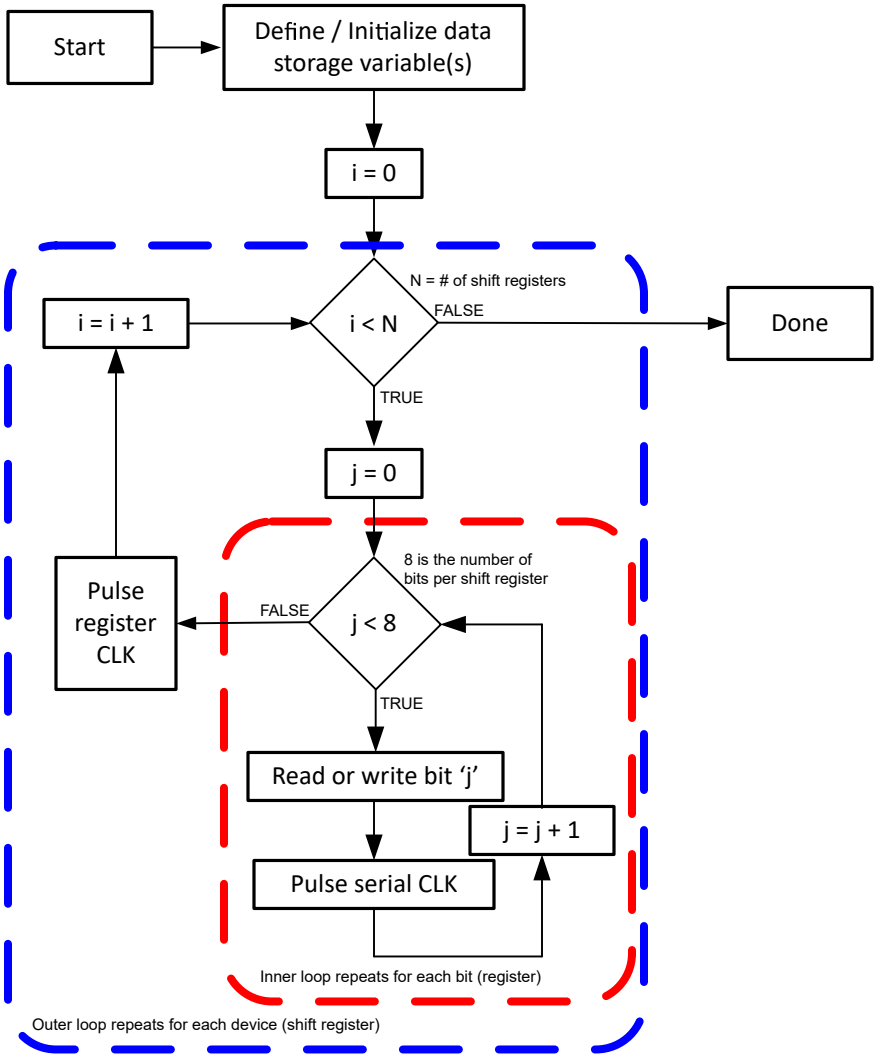


Figure 2-5. Generic Flowchart for Software Control of Shift Registers

The software required to control a set of shift registers is relatively simple. **Figure 5** provides a generic flowchart that can be used for producing code in any language and with any hardware for GPIO control of shift registers. In this flowchart, there are two loops. The outer loop, shown in blue, repeats for every shift register in the chain. The inner loop, shown in red, repeats for every bit within a shift register. Typically, the number of bits is eight, however it can vary.

The [*Read or write bit 'j'*] block is the most important and the most difficult to implement. Please see the example code shown in code example 1 for an approach using an MCU's built-in SPI module, and the example shown in code example 2 for a direct GPIO control approach.

There is also code provided for the *Example Design – Daisy Chain 128 Shift Registers*.

3 Example Design - Daisy Chain 72 Shift Registers

3.1 System Overview

For this example design, we will be driving 576 total indicator LEDs only using three GPIO pins from our selected MCU. We will also be reading in eight digital values from a set of DIP switches using only one additional GPIO pin.

This design is intentionally overly complex in order to show how to handle the “worst case” for driving multiple shift registers. In the majority of systems, none of the complexity shown here will be required, however all of the issues that are solved below should be considered.

For any system, the first step is to define the requirements.

- Voltage supply is 3.3 V
  - Since this document is not concerned with the power supply design, it is assumed that the supply is stable and all boards receive the appropriate amount of power required
- 8-channel DIP switch on controller board provides configuration
- Six individual remote boards contain 96 LEDs each
- LED forward voltages range from 2 V to 2.4 V, operate at 1 to 2 mA each
- Update LEDs and read configuration 30 times per second (33 ms max refresh time)
- Controller maximum output frequency is 1 MHz at 20 pF load
- Controller board is physically separated from LED boards, see Figure 3-1 for the system diagram
- Boards are connected via 50 cm long 120 ohm transmission lines (ribbon cables)

Based on the system requirements, it would be very difficult and costly to get a single controller that could individually handle all the required LEDs (96 per board, 576 total), not to mention the large number of wires that would be required to connect them. Instead, we will be using shift registers to reduce the required number of GPIOs and signal wires to only five.

The SN74HCS595 is used to allow for the LEDs to change simultaneously. This capability does require an extra control signal (RCLK), however it is very beneficial for LED applications since flickering can be observed while loading simpler shift registers like the SN74HCS164, which do not include output storage registers.

For the controller board, the DIP switch states will be read by a parallel-in, serial-out shift register. Each switch is debounced using a simple RC circuit and the SN74HCS165 is used to support the slow input transition times caused by the debounce circuits. See Debounce a Switch for more details regarding debounce circuits.

3.2 System Design

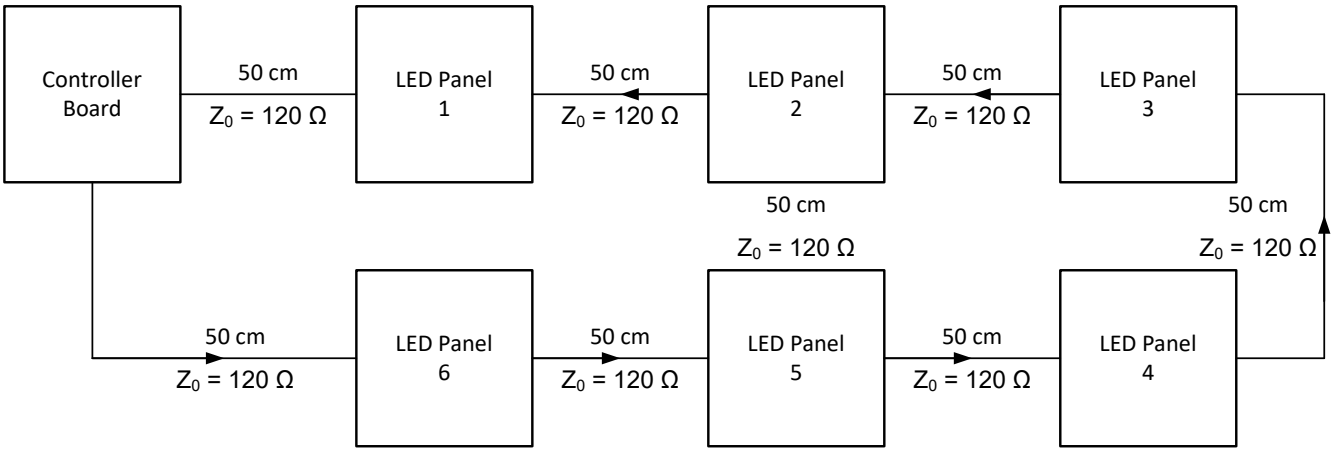


Figure 3-1. System Block Diagram With Arrows Indicating the Direction of Clock Propagation From the Controller Board

A hierarchical design is being used to more easily show this relatively large-scale system design. See Figure 3-1 for the top-level system block diagram. The LED Panel boards are labelled numerically in the order that the data

is propagating through the system. The clock propagates in the reverse order, arriving at board six first, then travelling back to board five and so on. The order of the clock arrival is indicated by arrows in the block diagram.

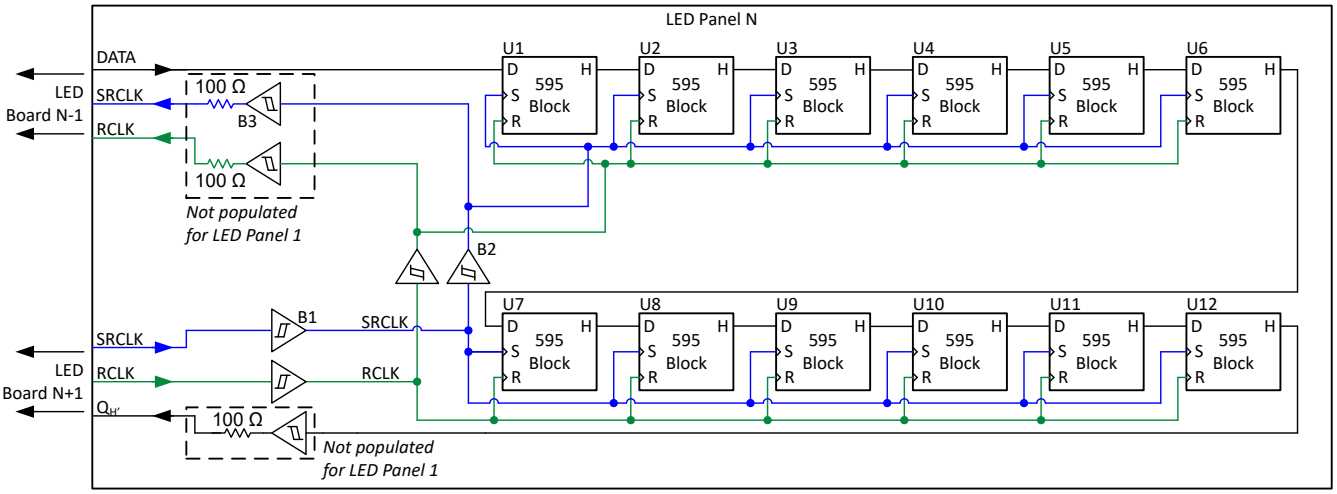


Figure 3-2. Block Diagram for LED Panels.

Each LED panel board contains 12 shift registers, split into two banks of six for the clock signals. See Figure 3-2 for an illustration of how these are connected. The incoming clock signals are first buffered by B1 and B2, each of which drives six total CMOS inputs. This equates to approximately 35 pF of total load capacitance. The trace capacitance also needs to be estimated, which is approximately 22 pF.1 This gives a total capacitive load of 57 pF, which is a bit higher than the load at which the datasheet specifications are measured, but not so much that it would be considered an area of concern.2

Table 3-1. Clock Timing for one LED Panel

Time (ns)	Event
0.0	Clock edge (CLK) is generated
3.3	CLK arrives at LED Panel 6, triggering input buffer B1
13.2	CLK is generated at the output of buffer B1
13.3	CLK arrives at input of buffer B2
15.2	CLK has reached all 6 of the shift registers U7 to U12, loading all SER inputs
23.2	CLK is generated at the output of buffer B2
23.3	CLK arrives at the input of buffer B3
25.0	Shifted values appear at QH' outputs for registers U7 to U12
25.2	CLK has reached all 6 of the shift registers U1 to U6, loading all SER inputs
33.2	CLK is generated to send to the next board
35.0	Shifted values appear at QH' outputs for registers U1 to U6

To give an idea of the clock timing for each board in the system, see Table 3-1. It takes approximately 33.2 ns from the time the signal is generated at the controller to the time the clock signal is leaving LED Panel 6 to go towards LED Panel 5. In total, the time for a clock pulse to reach all boards is approximately 200 ns. Given this information, we would set our clock to only pulse once per 200 ns, however that is not really necessary since the signal doesn't need to propagate through all shift registers in the chain for data to be preserved using the configuration we have chosen, as explained in Section 2.3.

$$F_{max} = \frac{1}{21.7\ ns} = 46\ MHz \quad (2)$$

It takes approximately 21.7 ns from the time a clock signal is received at a given set of shift registers to the time the shifted data is available to be read by the following board in the chain. This value gives a good idea of the maximum operating frequency, as calculated in Equation 2.

Table 3-2. System-level Timing of Clock and Data

Time (ns)	Event
0.0	Serial clock edge (CLK) is generated Serial data edge (DATA) is generated
3.3	CLK arrives at LED Panel 6, triggering input buffer B1 DATA arrives at LED Panel 1, now waiting to be loaded into U1
33.2	CLK is generated from B3 to send to LED Panel 5
36.5	CLK arrives at LED Panel 5, triggering input buffer B1
66.4	CLK is generated from B3 to send to LED Panel 4
69.7	CLK arrives at LED Panel 4, triggering input buffer B1
99.6	CLK is generated from B3 to send to LED Panel 3
102.9	CLK arrives at LED Panel 3, triggering input buffer B1
132.8	CLK is generated from B3 to send to LED Panel 2
136.1	CLK arrives at LED Panel 2, triggering input buffer B1
166.0	CLK is generated from B3 to send to LED Panel 1
169.3	CLK arrives at LED Panel 1, triggering input buffer B1
191.2	CLK arrives at the last set of shift registers (U1 to U6), loading DATA into SER

In theory, this system of shift registers could be driven at 46 MHz without any issues, however our system has a maximum operating speed of 1 MHz (from the system requirements), and our desired refresh rate won't require such a high speed. Additionally, operating beyond 5 MHz (200 ns per clock pulse) would complicate loading of data as the serial data signal would have to be delayed based on the total time it takes the clock to propagate through the system to reach the first set of shift registers (U1 to U6) on LED Panel 1.

To load all 576 registers in under 33.3 ms, the clock speed must be at least 17.3 kHz ($F = \frac{1}{33.3\ ms} \approx 30\ kHz$). This works out very well for our system, as we can load in data much faster than this and have RCLK control the output timing precisely.

The clock speed of 100 kHz is selected for SRCLK, loading all 576 registers in approximately 5.76 ms, which allows plenty of time for additional processing, if necessary, and RCLK can be separately timed to send the loaded data to the outputs every 33.3 ms. This speed also allows time for data to be set for each clock pulse without concern for matching propagation delays. In other words, the clock will completely propagate through the system in under 200 ns, as shown in Table 3-2, while the period of the clock is 10 µs, providing over 9.8 µs of buffer time to set the serial data input value. Table 3-2 shows the data arriving at the same time as the clock is generated, which provides a setup time of 187.9 ns – far more than is necessary for correct operation of the device. According to the datasheet, only 71 ns is necessary for operation at 3.3 V.3

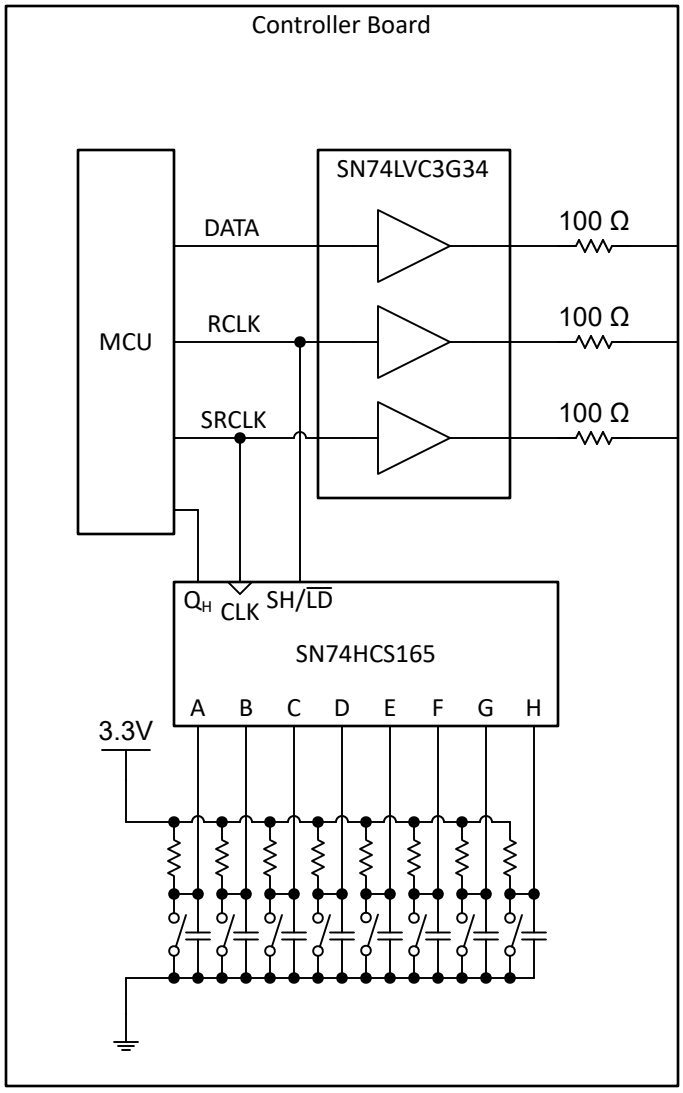


Figure 3-3. Controller Board Block Diagram. Unlabeled Resistors and Capacitors are 22 kΩ and 1 µF, Respectively

The last area of concern is reading the configuration data in from the SN74HCS165 parallel-in shift register. Figure 3-3 shows the selected configuration for the serial-in shift register to reduce the total number of pins required at the controller. RCLK and SRCLK are shared with the SH/$\overline{LD}$ and CLK pins, respectively.

Each switch includes a simple RC debounce circuit with values of 22 kΩ and 1 µF for each respective component. The slow edge produced by this debounce circuit would normally need to be put through signal conditioning prior to reaching a standard CMOS input, however the SN74HCS165 includes Schmitt-trigger inputs that support slow signals, so no additional circuitry is required.

In this configuration, the switch states can either be read in during the write operation of the other shift registers, or it can be read in separately. For the sake of this imagined system, the switch states will be read in at the same time that the first eight outputs bits are being written to the parallel-out shift registers.

By keeping RCLK in the normally high state, the data in the SN74HCS165 can be loaded every time RCLK is pulsed, and the data can be read in with the first eight clock pulses of SRCLK. Although this is a very effective and efficient use of GPIO pins, there is one drawback to this configuration. The software required to read in and write out data simultaneously is slightly more complex than that required to do these tasks separately. In some systems, it may be beneficial to perform these tasks separately. See Section 2.5 and Section 3.3 for details regarding the software required for this system.

Since each board is separated by 50-cm long transmission lines, source-terminating resistors are used to prevent reflections from causing issues. The output of the LVC logic family buffers used is expected to provide approximately 15 ohms of source impedance. A perfect match is not possible or necessary4, so the common value of 100 ohms is selected to match this to the 120 ohm characteristic impedance transmission lines.

Note

1. The calculator at https://technick.net/tools/impedance-calculator/microstrip/ was used to estimate the capacitance using the following values: W = 0.000203 m, H = 0.00025 m, T = 0.0000348 m, Er = 4.2
2. All capacitors will cause a large transient output current initially, which will taper off after a short time. Since the device is characterized with a particular load, that value can definitely be used safely, however there is an unspecified value that can cause damage. It is recommended to add a series resistor to limit current and avoid damage if the capacitive load is too large. For a good rule of thumb, don't go past 150% of the maximum load in the switching characteristics table without adding a series limiting resistor. For most devices, this means 75 pF is the maximum direct capacitive load.
3. Timing requirement for 3.3-V supply is estimated using linear interpolation from the existing 2-V and 4.5-V supply data.
4. The output of a CMOS device is not a constant value of resistance. It can vary with temperature, process, supply voltage, and output voltage.

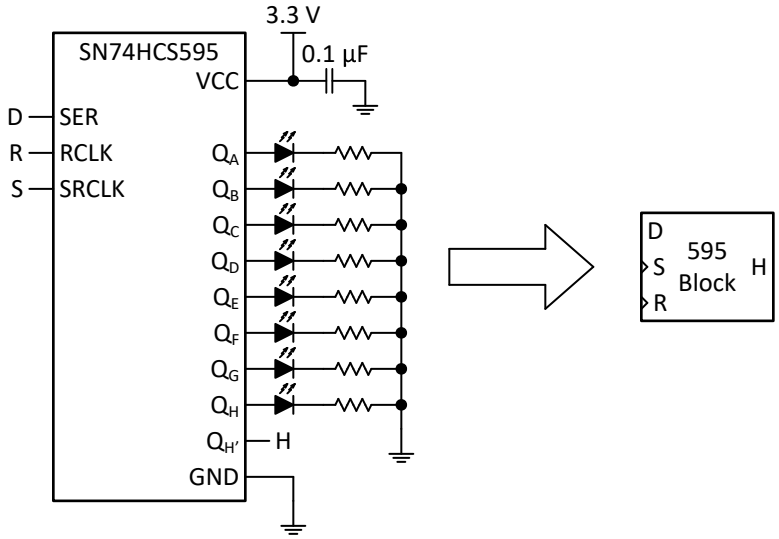


Figure 3-4. Detailed Schematic for 595 Block

Each 595 Block contains eight LEDs as shown in Figure 3-4, each with an individually selected resistor to limit the current through that LED to the appropriate value. For example, with a 2.2-V forward voltage, a resistor value of 732 Ω is selected to get 1.5 mA of current.

The eight LEDs for each SN74HCS595 device will draw less than 16 mA total, which is safely below the total maximum current for the device (70 mA). Shift registers can be used to control LEDs that require more current, however additional LED driver circuitry may be necessary if more than 8.75 mA per channel (70 mA total) is required.

Each LED board is small enough that the delays are negligible between the shift registers on that board.

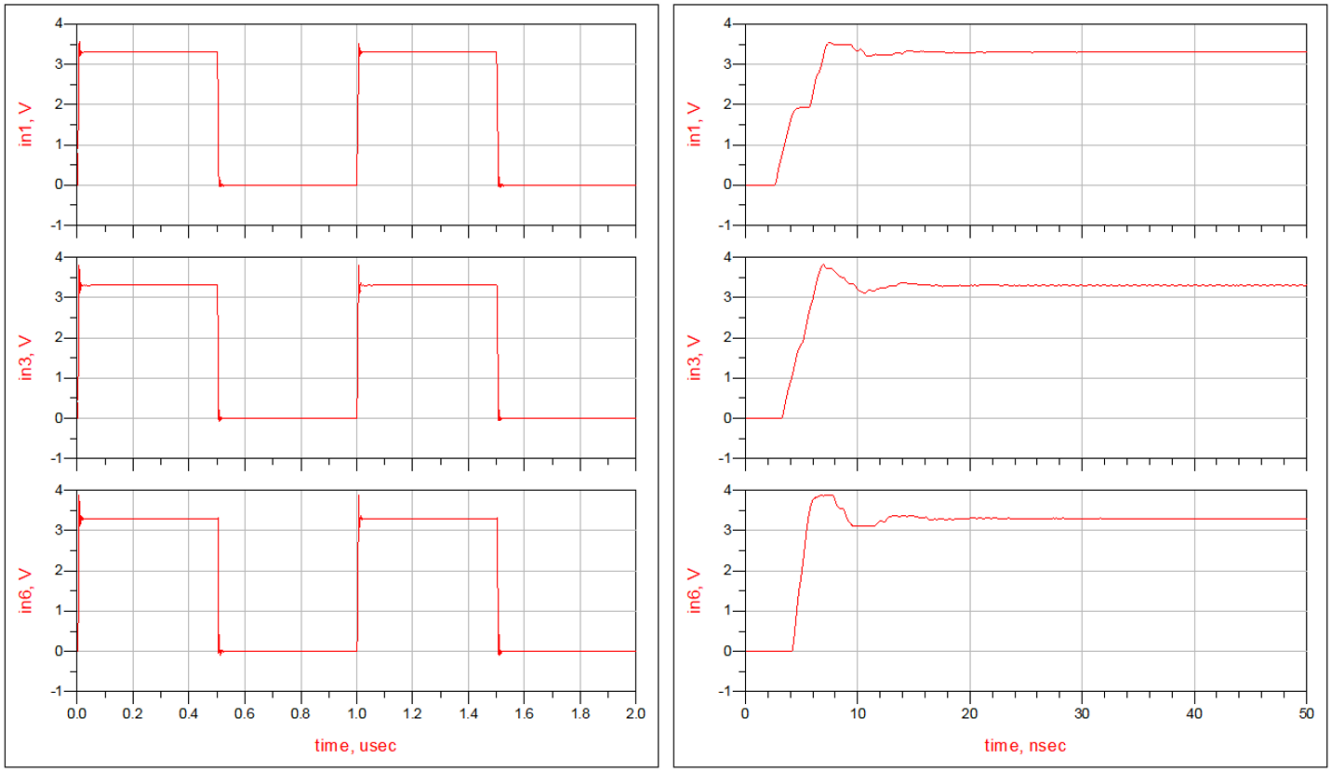


Figure 3-5. (left) Simulated Clock Waveforms Received by the First, Third, and Sixth Shift Registers. (right) Zoomed in Versions of the Same Three Waveforms to More Clearly Show the Rising Edges.

Figure 3-2 shows how the signals are routed on the LED boards.

Figure 3-5 shows the expected signal integrity at each clock input. Due to the short traces involved, there is little distortion or delay and the devices would be expected to operate normally.

3.3 Software Examples

For this design, the software is relatively simple. For simplicity, we are using Energia and the MSP430G2553 as the controller, which is a simple and widely available microcontroller (MCU).

By using the built-in SPI module for the MCU, much of the code is simplified. The SPI.transfer() function provides both read and write simultaneously, as shown in the provided code.

Example Code for using the SPI module of MSP430G2553 in Energia to load and read shift registers.

```
/*
 * This is a demo file for using SPI to control 72 parallel-out (595) shift registers
 * and one parallel-in shift register (165)
 *
 * The hardware used for verification was the MSP-EXP430G2 launchpad with the MSP430G2553 MCU
 *
 * The USCI_B0 SPI interface is used (which is the default)
 *
 * P1.5 --> SPI Clock
 * P1.6 --> SPI Data Input (to QH of SN74HCS165)
 * P1.7 --> SPI Data Output (to SER of first SN74HCS595)
 *
 * P1.2 --> Register Clock and Shift / Load(active low) Output
 */

#include <SPI.h>

byte data[72];   // Data to be sent to the shift registers (1 byte per shift register)
byte sw_val = 0; // DIP switch contents

void setup() {
  // Configure RCLK (also SH/LDn) as output and set default LOW
  pinMode(P1_2, OUTPUT);
  digitalWrite(P1_2, HIGH);

  // Configure SPI module
  // Energia sets MCLK to 16 MHz by default
  // SPI clock is selected at MCLK / 128 = 125 kHz
  SPI.setModule(0);
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE0);
  SPI.setClockDivider(SPI_CLOCK_DIV128);
  SPI.begin();

  for(int i = 0; i < 72; i++){
    // This data is generic just for example purposes
    data[i] = i;     // Initialize data contents
    SPI.transfer(0); // Initialize all shift registers to contain '0'
  }

  // Load DIP switch values into the SN74HCS165
  // Also loads register values (0) to all output registers
  digitalWrite(P1_2, LOW);
  delayMicroseconds(10); // Wait 10 us (100 kHz rate)
  digitalWrite(P1_2, HIGH);
}

void loop() {
  // Load values into the shift register

  // The first byte of data needs to be retained. This is coming
  // in from the SN74HCS165.
  // The 'SPI.transfer()' function sends and receives simultaneously
  sw_val = SPI.transfer(data[0]);

  // For the remaining data, the received value is discarded
  for(int i = 1; i < 72; i++) {
    SPI.transfer(data[i]);
  }

  // The preceeding 128 data transfers should take ~1.024 ms to complete

  // Pulse RCLK and SH/LDn to complete
  digitalWrite(P1_2, LOW);
  delayMicroseconds(10); // Wait 10 us (100 kHz rate)
  digitalWrite(P1_2, HIGH);
```

```
  // 32 ms is added to have a 'frame length' of ~33ms (30 fps)
  delay(32);

  // In a real system, other operations would replace the above delay.
  // For example, the 'data' will likely need to be created for each
  // frame, or loaded from some memory to provide the desired effect.

}
```

Example Code for using GPIOs of MSP430G2553 in Energia to load and read shift registers.

```
/*
 * This is a demo file for using GPIOs to control shift registers.
 *
 * This code loads values into four shift registers then reads back the four values.
 * The shift registers are loaded as though they are driving seven-segment displays.
 *
 * The hardware used was the MSP-EXP430G2 launchpad with the MSP430G2553 MCU
 * Four 595-type shift registers are connected in series
 *
 * P1.2 --> Register Clock
 * P1.5 --> Serial Clock
 * P1.6 --> Data Input
 * P1.7 --> Data Output
 */

// The following pin definitions are chosen arbitrarily.
// Any GPIO can be mapped to any of the following signals.
static const uint8_t RCLK   = P1_2; // Output -- Register Clock (rising edge)
static const uint8_t SRCLK  = P1_5; // Output -- Serial Clock (rising edge)
static const uint8_t DI     = P1_6; // Input  -- Serial data from shift registers
static const uint8_t DO     = P1_7; // Output -- Serial data to shift registers

#include <SPI.h>

// Seven segment interface data
// digit[#] will display # on the seven segment display
byte digit[] =  {0b00111111,
                 0b00000110,
                 0b01011011,
                 0b01001111,
                 0b01100110,
                 0b01101101,
                 0b01111101,
                 0b00000111,
                 0b01111111,
                 0b01101111};
// dp only lights up the decimal point for the seven segment
// this is intended to be used with digit[] and a logical OR
byte dp = 0b10000000;

// Global counter
int i = 0;
int value = 0;

void pulseSRCLK() {
  // Advance the shift registers when called
  digitalWrite(SRCLK, HIGH);
  delayMicroseconds(1); // Wait 1 us
  digitalWrite(SRCLK, LOW);
}

void pulseRCLK() {
  // Loads serial register data into the output registers when called
  digitalWrite(RCLK, HIGH);
  delayMicroseconds(1); // Wait 1 us
  digitalWrite(RCLK, LOW);
}

int SR_transfer_byte(byte tx_val) {
  // This function writes 'tx_val' to the first shift register
  // while simultaneously reading 'rx_val' in from the last shift register
  int rx_val = 0;   // Initialize receive value
  int read_val = 0; // Initialize read value

  for( int i = 0; i < 8; i++ ) {
    // Loop over eight bits
```

```
    if( (0b00000001 << (7-i)) & tx_val )
      // When the 'i'th bit in tx_val is 'high'
      digitalWrite( DO, HIGH );
    else
      // When the 'i'th bit in tx_val is 'low'
      digitalWrite( DO, LOW );

  read_val = digitalRead( DI );
  rx_val = (read_val << (7-i)) | rx_val;

  // Pulse the serial clock
  pulseSRCLK();

  } // byte send/receive complete

  return rx_val;

}

// The setup() function is called once at startup and is used for configuration.
void setup() {
  // Configure pin modes
  pinMode(SRCLK, OUTPUT);
  pinMode(RCLK, OUTPUT);
  pinMode(DO, OUTPUT);
  pinMode(DI, INPUT);

  // Initialize output values
  digitalWrite(SRCLK, LOW);
  digitalWrite(RCLK, LOW);
  digitalWrite(DO, LOW);

  // Load in values to all 4 shift registers
  SR_transfer_byte(digit[1]);
  SR_transfer_byte(digit[2]);
  SR_transfer_byte(digit[3]);
  SR_transfer_byte(digit[4]);
  pulseRCLK();

  // Initialize 'value' for first loop
  value = digit[5];
  delay(500);
}

// The loop() function is called after the setup() function and loops forever
void loop() {
  value = SR_transfer_byte(value);
  pulseRCLK();
  delay(500);
  // For this example, the values are looped and the imaginary
  // 4 digit 7-segment display would show:
  // 1 2 3 4
  // 2 3 4 5
  // 3 4 5 1
  // 4 5 1 2
  // 5 1 2 3 -- this would loop forever
}
```

4 References

- Appliances
  - Dishwasher
  - Refrigerator and Freezer
  - Washer and Dryer
  - Air Conditioner Indoor Unit
  - Air Conditioner Outdoor Unit
- Medical
  - Oxygen Concentrator
- Motor Drives
  - Servo Drive Control Panel
  - Servo Drive Power Stage Module
  - AC Drive Power Stage Module
- Building Automation
  - Fire Alarm Control Panel (FACP)
  - HVAC Motor Control
  - Calling Buttons Operating Panel

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265