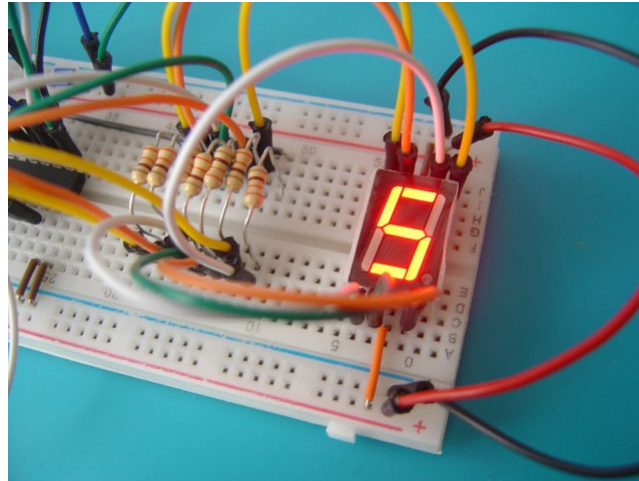


Arduino Tailored to Our Needs

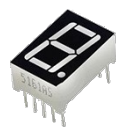
AVR Microcontroller Engineering with Pascal, Lesson 4

version 05/09/2026

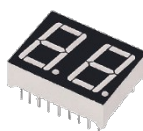


1. 7-Segment Displays

Remaining in the broadly understood LED theme, it's time for a slightly more ambitious topic. It is the handling of 7-segment LED displays¹. Will ATtiny13 meet this challenge?



1-digit



2-digit



3-digit



4-digit

¹ Excluding the decimal point (decimal separator). Popular digit-shaped displays with a decimal point actually have 8 segments.

7-segment displays are produced in various variants differing in the color of the displayed digit (red, blue, green) and size (from about 1.42 cm to 16.51 cm). In addition, two, three, or four displays are sometimes combined into sets enabling the display of a larger number of digits.

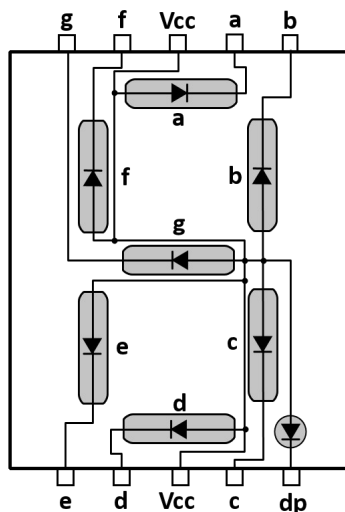
For the needs of our experiments, we will use only 1-digit displays and start using additional integrated circuits. We will need:

1 7-segment display with common anode	under 1\$
3 7-segment displays with common cathode	about 1\$
1 shift register 74HC595	from 0.5\$
Set of 330 Ω resistors	about 1\$
1 33 k Ω resistor	under 1\$
1 MAX7219 controller	from 2\$

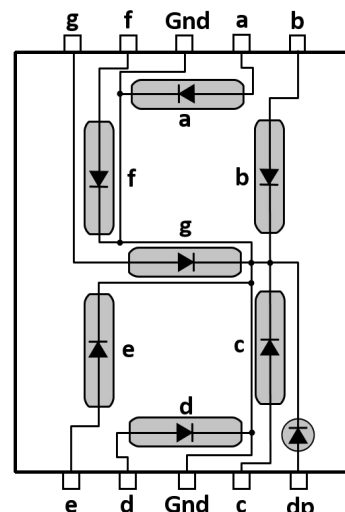
The prices are approximate, according to the cheapest offers on the popular online marketplace².

2. Display Operation Principle

In technical terms, LED displays are divided into displays with common anode and displays with common cathode. Each segment forming a digit is a separate LED, but for ease of control, one side of all segments is common. In the case of displays with common anode, all positive poles of the LEDs are connected to V_{cc} , while in the case of displays with common cathode, the negative poles of the LEDs are connected to ground. Segment control is done by connecting it to ground (displays with common anode) or applying voltage (displays with common cathode).



Display with common anode



Display with common cathode

An example display with common anode is FJ5161BH and with common cathode – 5611AH. Both have identical dimensions³ in principle and differ only in the way of

² <http://www.ebay.com/>

³ 19 × 12.6 × 8 mm.

controlling the segments. Both are also suitable for the experiment on the breadboard due to the appropriate pin spacing.

As in the case of any device consisting of LEDs, we should connect a current-limiting resistor to each segment, as omitting it may result in LED damage. It is worth using the manufacturer's datasheet, which should contain basic LED (segment) parameters and their behavior under different conditions.

The resistance value can be calculated using the formula $R=(V_{cc}-V_f)/I_f$, where V_{cc} is the supply voltage, V_f – the diode forward voltage (dependent on type and color), I_f – the forward current (usually dependent on the LED size). In our case, the supply voltage is 5 V, and the forward voltage and current fit in certain ranges; as safe values for both FJ5161BH and 5611AH, we can take $V_f=1.7-1.8$ V and $I_f=8-10$ mA⁴ respectively. After substituting into the formula, the resistance value ranges from 320 to 412.5 Ω , and since it is not possible to select a resistor of any value but one that occurs in the so-called series, a 330 Ω resistor should be sufficiently good. As mentioned above – we need to prepare as many resistors as there will be segments (LEDs) used, so for one display, there must be 7 (or 8, if the decimal separator indicator will be used). We cannot use one 330 Ω resistor connecting it in parallel with all segments, as then the brightness of the LEDs will depend on their number⁵.

3. Display with Common Anode. Shift Register.

Having theoretical knowledge, we can proceed to using it in practice. To start, let's deal with the display with common anode. The experiment will consist of connecting the display to ATtiny13 and controlling it in such a way that it displays digits from 0 to 9 changing the value every half second.

It would seem that we can proceed to action, but a problem appears. ATtiny13 has 8 pins, of which one is V_{cc} and one GND, so 6 I-O pins⁶ are left to use. To control the display, we need 7 pins (we do not use the decimal separator in the display). So one is missing to be able to display any digit. Is there a solution?

Shift registers are used to expand the number of microcontroller pins. A shift register is a type of buffer that stores entered data (bits, i.e., high or low states), which can then be modified (shifted) and read, but in a specific way. Imagine that this buffer is an 8-bit byte, into which data can be entered serially, bit by bit, to ultimately read all 8 bits simultaneously (in parallel). This type of shift register is called *SIPO* (i.e., with serial input and parallel output)⁷ and most importantly – in such a system, we need only 3 control lines plus power and ground, gaining 8 output pins⁸. A *SIPO* type shift register is, e.g., the 74HC595 chip.

⁴ The values recommended by the manufacturers are as follows: for the display 1.8 V and 10 mA (5611AH) and 1.7 V and 8 mA (FJ5161BH).

⁵ Theoretically, the resistor value can be selected in such a way that in parallel connection it gives 330 Ω resistance on each LED.

⁶ From *Input-Output*.

⁷ From *Serial Input - Parallel Output*. Other types of shift registers are: *PIPO* – with parallel inputs and outputs; *SISO* – with serial output and input; *PISO* – with parallel input and serial output. There are also universal serial-parallel and parallel-serial shift registers.

⁸ We sacrifice 3 lines gaining 8, so the balance is 5 additional lines.

How does 74HC595 work? Data must be serially transmitted bit by bit to pin 14 (SER), but so that the chip can correctly sample the data state, it must be accompanied by a clock signal (SRCLK). The clock signal is inactive in low state and activated by high state. If we want to transmit logical 0, then on the SER input we must give low state and then activate SRCLK with high state. This operation simultaneously shifts the current cell of the register to the right (i.e., from QA sequentially to QH, hence shift register) and on the first (QA) flip-flop output, logical zero will appear⁹. If we then want to transmit logical 1, on the SER input we set high state and activate SRCLK. Then the logical zero from the first operation will be shifted to the second output (QB) and our logical 1 will be transmitted to the first output (QA). In this way, we can transmit 8 bits serially shifting subsequent cells and filling the buffer, but the data will not appear immediately on the output pins¹⁰. To latch them, use the latch (RCLK) by giving high state on pin 12. The chip has another important input – OE, which activates the ability to use outputs provided that pin 13 is low state. Usually, we connect it to ground so that the register outputs are active permanently. In turn, pin 10 (SRCLR) serves to clear the register contents and is activated by low state¹¹. If there is no need to clear the register contents during operation, pin 10 can be permanently connected to power (high state).

74HC595		Pin	Symbol	Description
QB	1	16	V _{CC}	Parallel outputs
QC	2	15	Q _A	
QD	3	14	SER	Serial output (next shift register)
QE	4	13	$\overline{\text{OE}}$	
QF	5	12	RCLK	Clock signal
QG	6	11	SRCLK	
QH	7	10	$\overline{\text{SRCLR}}$	Latch
GND	8	9	Q _H '	
		12	RCLK	Output enable
		13	OE	
		14	SER	Serial input
		16	V _{CC}	
				Power supply

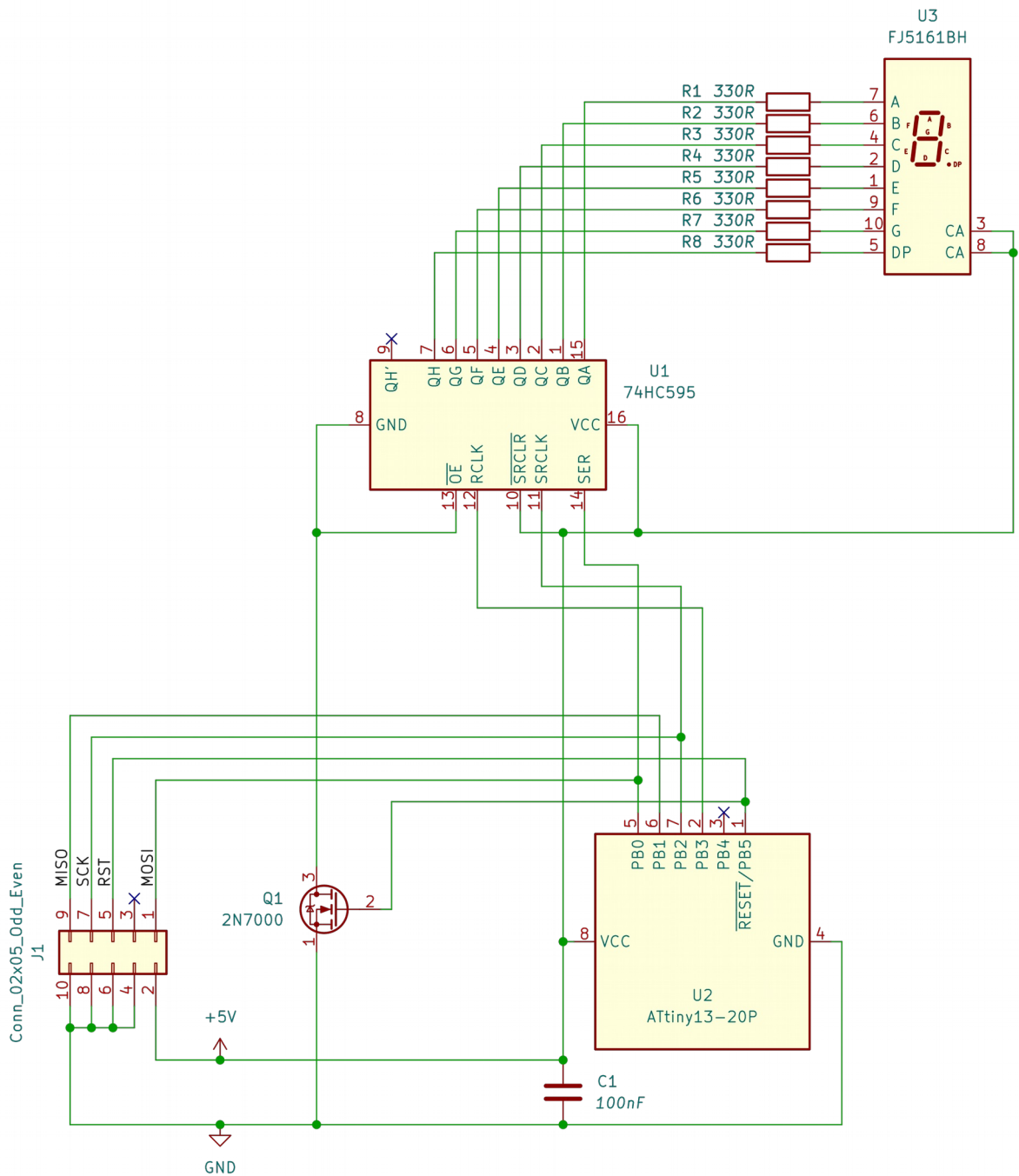
So let's connect the microcontroller to the 74HC595 chip and that to the display segments. We can use the breadboard with most elements from the previous part of the course, removing the button and LED with resistor. The register control inputs can be connected to any available IO pins, we will use PB0, PB2, and PB3 connecting them sequentially to SER, SRCLK, and RCLK¹². Then we will connect the register's parallel outputs to the display segments, remembering to include 330 Ω resistors in each connection. The connection order is also arbitrary, but to avoid complications, we will connect outputs QA-QH sequentially to segments A-DP of the display so that the index letters of the parallel outputs and display segments match (we treat segment DP as "H"). This will greatly facilitate programming the microcontroller. The schematic of our circuit can therefore look as below.

⁹ In practice, the 74HC595 contains two groups of flip-flops: the first forms the shift register itself, while the second forms the outputs and allows their state to be latched.

¹⁰ Random-like values will appear there due to transient states.

¹¹ A horizontal line above a symbol means that the signal is active low.

¹² The pin selection is dictated solely by the SPI interface naming convention (see the SPI description in Chapter 4).



The design and operating mechanism of an 8-output shift register somewhat dictate the programming method. The register is 8-bit (8 bits of serial data go to the parallel outputs), so our data portion will be a byte. In other words, each digit visible on the display can be encoded as 8 bits, where a given segment (bit) is turned on or off. In the case of the display we are using, i.e., common anode, a segment must have a low state (0) at the input to be turned on; for common cathode displays, it would be exactly the opposite, i.e., a high state (1) would be required. In our program, segments A-H/DP will correspond to powers of 2 in

increasing order (from 0 to 7)¹³, which will allow encoding a given digit as a number in the range 0-255.

Digit	A 2 ⁰ =1	B 2 ¹ =2	C 2 ² =4	D 2 ³ =8	E 2 ⁴ =16	F 2 ⁵ =32	G 2 ⁶ =64	H/DP 2 ⁷ =128	Com. anode (x=0; -=1)	Com. cathode (x=1; -=0)
0	x	x	x	x	x	x	-	-	192 (\$C0)	63 (\$3F)
1	-	x	x	-	-	-	-	-	249 (\$F9)	6 (\$06)
2	x	x	-	x	x	-	x	-	164 (\$A4)	91 (\$5B)
3	x	x	x	x	-	-	x	-	176 (\$B0)	79 (\$4F)
4	-	x	x	-	-	x	x	-	153 (\$99)	102 (\$66)
5	x	-	x	x	-	x	x	-	146 (\$92)	109 (\$6D)
6	x	-	x	x	x	x	x	-	130 (\$82)	125 (\$7D)
7	x	x	x	-	-	-	-	-	248 (\$F8)	7 (\$07)
8	x	x	x	x	x	x	x	-	128 (\$80)	127 (\$7F)
9	x	x	x	x	-	x	x	-	144 (\$90)	111 (\$6F)

With the above table at our disposal, we can proceed to write the program code. In the *uses* section, we will add the *attiny13_basics* module prepared in the previous lesson of the course to use the *DelayMs* procedure. Next, we will define constants PB0, PB2, and PB3 corresponding to the pins of the microcontroller's PB port. In the *const* section, we will add two more important arrays: *NumberMasks*, a ten-element array of encoded bytes corresponding to digits 0-9, and *PowersOfTwo*, an eight-element array of powers of 2, facilitating the decoding of bytes into bits.

At the beginning of the program, we will define the PB port pins as outputs. Since the task consists of displaying digits sequentially and continuously repeating this action, we will use an infinite loop, and in it, in an iterative loop, we will use the variable *num* assigning it values in the range 0 to 9. In each iteration of the iterative loop, first set low state on pin PB3 (connected to RCLK of the register) causing the release of latched data. Then in another, nested loop, we will use the variable *i* and the *PowersOfTwo* array to serially transmit bits to pin BP0 (connected to SER of the register). This is the key part of the program. The encoded value of the segments for a given digit taken from the *NumberMasks* array will be decoded by checking if successive bits are set or cleared. We will determine this using the logical operator *and* (conjunction) with successive powers of 2. The order of powers must be descending due to each shift of the current cell made by the register. Bits set to 1 should be transmitted by turning on pin PB0 (SER), while bits set to 0 should be transmitted by turning it off. Then we will inform the register that the value of the current cell has already been set and needs to be shifted. We will do this by setting high state and immediately low state on pin PB2 (SRCLK). After transmitting 8 bits, we can latch the data by setting high state on pin PB3 (RCLK). Thanks to this, appropriate signals will appear on the register outputs and the desired display segments will light up. Finally, we will use the *DelayMs* procedure for a software half-second delay so that the changes of digits on the display are readable¹⁴.

¹³ Of course, a descending order (an equally good solution) or any other arbitrary order could also be used, although this would introduce some complications in the code.

```

program display_7segca;

{$IFDEF attiny13}
  {$Fatal Invalid controller type, expected: attiny13}
{$ENDIF}

uses
  attiny13_basics;

const
  PB0 = %00000001; // PB0 = pin 5 of microcontroller -> SER
  PB2 = %00000100; // PB2 = pin 7 of microcontroller -> SRCLK (CLOCK)
  PB3 = %00001000; // PB3 = pin 2 of microcontroller -> RCLK (LATCH)

  //masks for digits 0-9
  NumberMasks: array[0..9] of UInt8 = ($C0, $F9, $A4, $B0, $99, $92, $82,
  $F8, $80, $90);
  //powers of 2, descending order due to shifts
  PowersOfTwo: array[0..7] of UInt8 = (128, 64, 32, 16, 8, 4, 2, 1);

var
  num, i: UInt8;
begin
  //direction register
  DDRB := DDRB or PB0; // PB0 as output
  DDRB := DDRB or PB2; // PB2 as output
  DDRB := DDRB or PB3; // PB3 as output

  while true do // infinite loop
  begin
    for num := 0 to 9 do
    begin
      PORTB := PORTB and not PB3; // low state on LATCH

      //serial transmission of data bit by bit
      for i := 0 to 7 do
      begin
        if (NumberMasks[num] and PowersOfTwo[i]) = PowersOfTwo[i] then
          PORTB := PORTB or PB0 // high state on SER
        else
          PORTB := PORTB and not PB0; // low state on SER

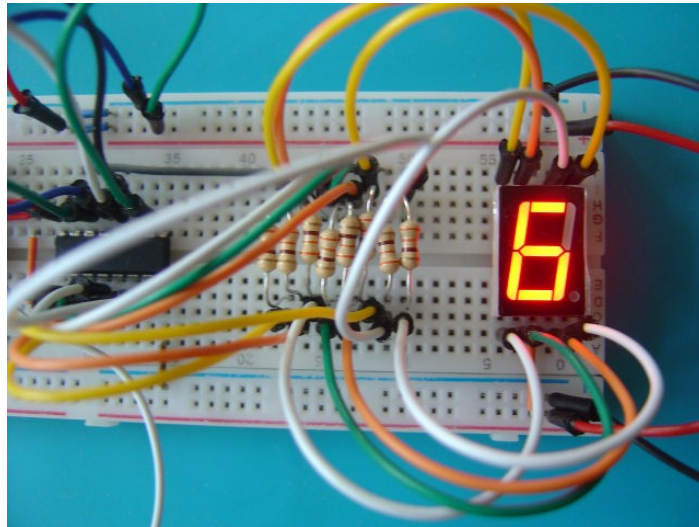
          PORTB := PORTB or PB2; // high state on CLOCK
          PORTB := PORTB and not PB2; // low state on CLOCK
        end;

        PORTB := PORTB or PB3; // high state on LATCH - latch data
        DelayMs(500); // delay
      end;
    end;
  end.

```

After compiling the above code and transferring it to the microcontroller's flash memory, the circuit should function as expected. A minor aesthetic drawback of the experiment is the tangled wires on the breadboard, which is difficult to avoid.

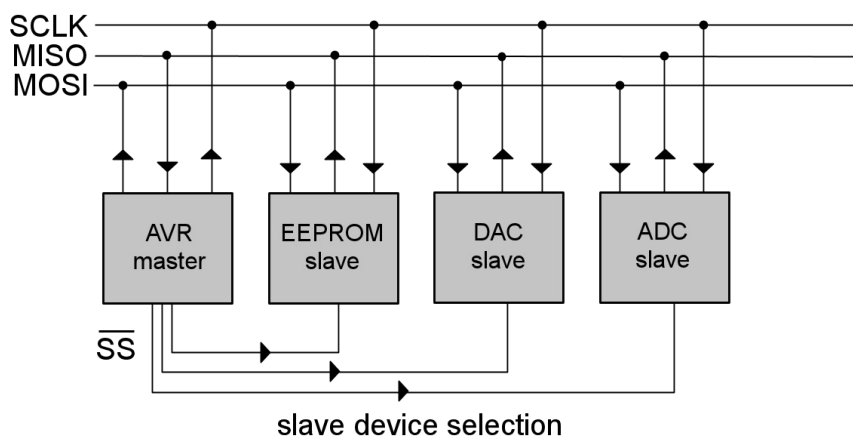
¹⁴ I chose a value of 500 milliseconds, which still guarantees correct operation of the *DelayMs* procedure. With higher values, the delay loop counter may overflow, resulting in incorrect timing.



4. SPI Interface

Continuing with the topic of serial transmission, let's consider whether we can simplify or improve its implementation. There are many ways to exchange data serially between devices. A widely used standard is SPI, considered relatively simple to implement and fast. SPI (*Serial Peripheral Interface*) is used to connect a master device, which is usually a microcontroller, with peripheral devices such as analog-to-digital converters, EEPROM memories, flash memories, SD cards, and others.

The SPI interface requires the presence of a master device (usually a microcontroller) and at least one slave device (usually a peripheral). It uses four lines for communication: MOSI, MISO, SCLK, and SS. The MOSI line (*Master Output Slave Input*) is used to send data from the master device to the slave. The MISO line (*Master Input Slave Output*) is responsible for data transmission in the opposite direction, from the slave to the master. The SCLK line (*Serial Clock*)¹⁵ is a clock (timing) signal that allows for the correct reception and sending of data between devices. In turn, the SS line (*Slave Select*)¹⁶ is used to choose the peripheral device that is currently active.

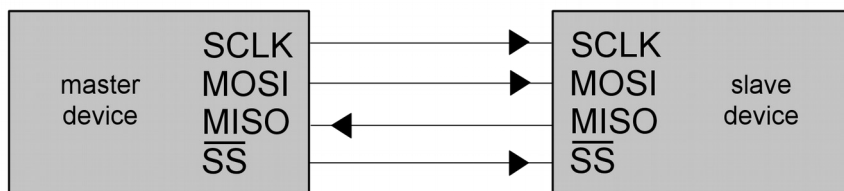


¹⁵ Another abbreviation is SCK.

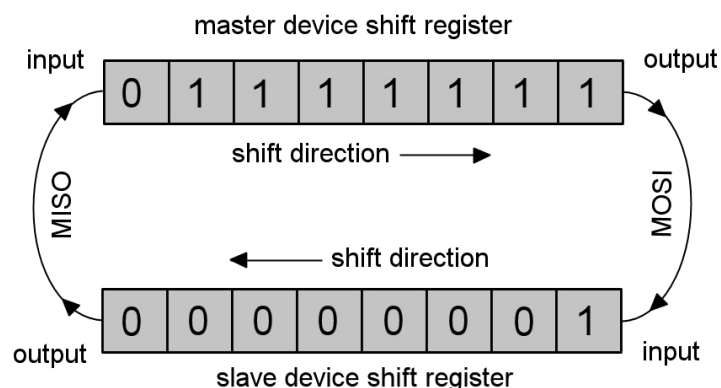
¹⁶ Another name is CS (*Chip Select*).

In connections with multiple devices, a so-called “bus” is used—a set of lines carrying data to devices connected to the bus. In the SPI standard, the MOSI, MISO, and SCLK lines form the bus, to which the master and slave devices are connected. On the other hand, the master device must have as many SS lines as there are slave devices connected to it.¹⁷

Let’s now look at the transmission directions in the SPI bus lines and the devices connected to it. The master device sends serial data to the MOSI line, while slave devices can only receive data from this line. Conversely, slave devices send serial data to the MISO line, while the master device can only receive data from this line. This means that the voltage on the MOSI line is controlled by the master device, while the slave device controls the MISO line. There is a common SCLK clock line for all devices, the signal of which is generated by the master device. Usually, timing occurs on the rising edge of the signal (i.e., changing from low to high state), but there are SPI communication modes where timing is signaled by the falling edge. Furthermore, the master device decides which slave device it currently wants to communicate with. Activating such communication is done by changing the state of its SS line from high to low, and ending it by returning to the high state.



In the simplest setup, data exchange takes place between only two devices: master and slave. The master device activates the slave device, provides the clock signal, and sends data on the MOSI line. The slave device, in turn, sends data on the MISO line. Sending and receiving data are performed simultaneously, meaning each device sends and receives data at the same time¹⁸.



¹⁷ This fact limits the number of slave devices to the physical capabilities (available pins) of the master device.

¹⁸ Not all data transmitted over SPI is meaningful. For example, while waiting for a command (i.e., a specific bit sequence), a slave device may send data that should simply be ignored. Usually, such data consists entirely of zeros (low states).

Devices that support the SPI protocol in hardware have a dedicated universal shift register that facilitates communication, connected to the MOSI and MISO lines. Along with the clock signal, a bit sent by the master device will appear in the last cell of the slave device's register, and simultaneously, a bit sent by the slave device will enter the last cell of the master device's register. The whole process can be imagined as a kind of data exchange ring. If the register in the master device originally contains all ones and the slave device contains all zeros, after a clock signal, the data will shift by one cell in the registers of both devices so that a one enters the end of the slave device's bit sequence, and a zero enters the master device's register.

Many AVR microcontrollers have hardware SPI support, including the appropriate registers; however, in the case of the ATtiny13, the matter is more complicated. Although the ATtiny13 has dedicated PB port pins for serial data transmission (MOSI - PB0, MISO - PB1, and SCLK - PB2; the RESET/PB5 pin serves the SS function¹⁹), they are handled by hardware only for the purpose of programming the flash memory in slave mode, serving the communication between the programmer and the microcontroller. The ATtiny13 does not have SPI registers available for general purposes (e.g., SPDR - data register). In this situation, the only solution is software-based SPI communication.

One might think that the entire digression about the SPI interface was pointless, but this knowledge will be useful to us soon. There are, in fact, driver circuits for 7-segment displays that utilize the SPI protocol.

5. Common Cathode Display Driver

One of the circuits designed to handle 7-segment displays and LED matrices is the MAX7219. It is a driver compatible with the SPI protocol. Similar to standard SPI, it requires serial data (DIN) and clock signal (CLK) lines in addition to power and ground. A notable difference is the LOAD line which, unlike the SS line in standard SPI, does not provide the start and end signal for serial transmission but serves as a latch for the internal 16-bit shift register built into the MAX7219.

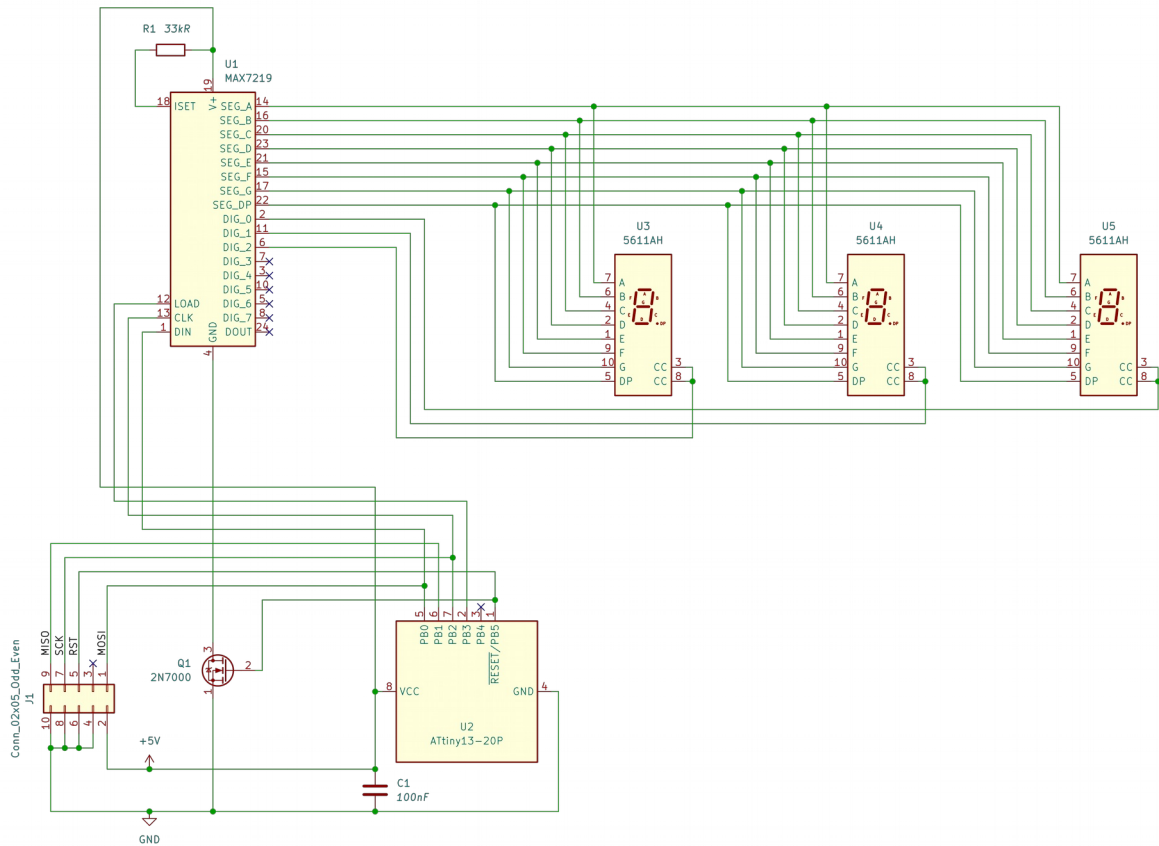
The MAX7219 features 8 parallel output lines (SEG A-G and SEG DP) to handle display segments and 8 DIG control lines. This means a single chip can control up to eight 7-segment displays. Additionally, it has an ISET pin for a current-limiting resistor to control brightness and a DOUT serial output to allow daisy-chaining multiple MAX7219 chips for larger projects.

Simple arithmetic suggests that for a circuit to separately support eight displays, each consisting of eight segments (counting the decimal point), it should allocate 64 lines for this purpose. However, the MAX7219 operates differently. Taking advantage of the inertia of human vision, it sequentially switches the connected displays displaying a given digit, doing so quickly enough that the overall display appears as a stable image. Therefore, it only needs a single line to turn a given display on and off (DIG), while the lines handling the segments (SEG) are shared.

¹⁹ For this reason, the RESET pin is held low during microcontroller programming.

MAX7219		Pin	Symbol	Description		
DIN	1	24	DOUT	Serial output		
DIG 0	2	23	SEG D	Display control lines		
DIG 4	3	22	SEG DP			
GND	4	21	SEG E			
DIG 6	5	20	SEG C			
DIG 2	6	19	V+			
DIG 3	7	18	ISET	12	LOAD	Latch
DIG 7	8	17	SEG G	13	CLK	Clock signal
GND	9	16	SEG B	14–17, 20–23	SEG A-G, DP	Parallel outputs for display segments
DIG 5	10	15	SEG F	18	ISET	Current-limiting resistor output
DIG 1	11	14	SEG A	19	V+	Power supply
LOAD	12	13	CLK	24	DOUT	Serial output

In this new experiment, we'll use the MAX7219 to control three common-cathode displays. The display cathodes will be connected to the DIG 0 - DIG 2 outputs, while the segments will be connected to the common SEG A - SEG DP line. The active display will be selected by connecting its cathode to the ground line, and the segments will be controlled by high states on the SEG outputs.



The selection of the display current-limiting resistor (ISET) requires some consideration. Its value depends on the current that can flow through the display segments. According to the 5611AH display datasheet, a safe value is between 10 and 20 mA. The MAX7219 documentation, in turn, specifies the segment current as approximately 100 times greater than the current flowing through the ISET resistor. Therefore, assuming an input voltage of 5 V, the resistor value should be between 25 (for 20 mA) and 50 (for 10 mA) kΩ²⁰. A 33 kΩ resistor seems to be a safe solution.

Let's now look at the communication protocol with the MAX7219. We'll connect the DIN serial data line to pin PB0 (MOSI), the CLK clock signal line to pin PB2, and the LOAD line to pin PB3 of microcontroller²¹. We don't need a controller-to-microcontroller (MISO) transmission line, as the controller doesn't send any data via SPI. As mentioned earlier, the MAX7219 has a built-in 16-bit data register (D0-D15) that can be used to send commands to the controller. It also has 14 other registers for configuring and controlling the displays.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
*	*	*	*	Address				Data							

The values of bits marked with an asterisk (*) have no significance.

The structure of a data register determines the format of the data sent to the system. To send commands correctly, two bytes must be used. The first byte should contain the address of the configuration or control register, and the second byte should contain the data intended for that register. The four most significant bits of the address are irrelevant and can have any value²². Let's examine the functions of the system's configuration and control registers.

Register	Address					Value
	D15-D12	D11	D10	D9	D8	
No-Op (No-Operation)	0*	0	0	0	0	0 (\$00)
Digit0	0*	0	0	0	1	1 (\$01)
Digit1	0*	0	0	1	0	2 (\$02)
Digit2	0*	0	0	1	1	3 (\$03)
Digit3	0*	0	1	0	0	4 (\$04)
Digit4	0*	0	1	0	1	5 (\$05)
Digit5	0*	0	1	1	0	6 (\$06)
Digit6	0*	0	1	1	1	7 (\$07)
Digit7	0*	1	0	0	0	8 (\$08)
Decode Mode	0*	1	0	0	1	9 (\$09)
Intensity	0*	1	0	1	0	10 (\$0A)

²⁰ The values were calculated using the formula $R_{ISET} \approx (100 \times V_{CC}) / I_{SEG}$, where R_{ISET} is the required resistance of the ISET, V_{CC} is the supply voltage, and I_{SEG} is the allowable segment current.

²¹ Of course, other free ATtiny pins could be selected, but this choice makes it easier to reuse part of the code from the previous experiment.

²² This follows from the number of supported registers (14), whose binary-encoded addresses occupy only 4 bits, leaving the remaining 4 bits unused.

Scan Limit	0*	1	0	1	1	11 (\$0B)
Shutdown	0*	1	1	0	0	12 (\$0C)
Display Test	0*	1	1	1	1	15 (\$0F)

The bits marked with an asterisk (*) have no significance, to simplify the calculation I gave them the value 0 here.

The *No-Op* register (address \$00) does not perform any operation, but is used when cascading multiple MAX7219 devices. The DOUT pin then transmits data to the DIN input of the next device. Since we are using only one MAX7219 device, we will not use this register.

Registers *Digit0 - Digit7* (addresses \$01-\$08) are responsible for display selection. Data (D0-D7) then indicates which segments of the display should be lit. This information can be transmitted directly as an encoded byte or as BCD²³ numbers. We will use a format that uses a whole byte, directly indicating which segments should be lit.

The *Decode Mode* register (address \$09) determines the encoding method for the display segments. If this register is selected, the data bits (D0-D7) determine whether the numbers displayed as digits on a given display are to be encoded in BCD format (bit set, i.e., 1) or as an encoded byte specifying the lit segments (bit unset, i.e., 0).

The *Intensity* register (address \$0A) controls the intensity of the LED segments, ranging from 0 to 15 (\$00-\$0F). The maximum allowable numerical value will cause the segments to light at maximum intensity and the highest current to flow through them, but not greater than the value of the resistor connected to ISET.

The *Scan Limit* register (address \$0B) controls the number of displays connected to the system, so its value should be specified in the data byte in the range 0-7. Remembering that only one display is on at a time, the switching frequency (multiplexing) of the displays is 800 Hz if eight displays are connected. With fewer displays, the frequency increases²⁴. This also means that as the number of connected displays decreases, the intensity of their segments will increase.

The *Shutdown* register (address \$0C) specifies whether the system should operate in normal mode (\$01 in the data register) or in shutdown mode (\$00 in the data register). In shutdown mode, the displays are turned off, the internal oscillator is suspended, the segment lines are pulled to ground, and the control lines are pulled to the positive supply voltage. This shuts down the system and reduces current consumption to approximately 150 μ A.

The *Display Test* register (address \$0F) allows us to enable test mode by writing the value \$01 to the data register, which causes all segments to light up at maximum intensity. It does not change the contents of other registers.

Now that we know how to configure the MAX7219 controller, we can start writing the code. SPI transmission is key, as the ATtiny does not support it. So how can we program it? The good news is that we've almost done it, as software SPI support isn't much different from programming a shift register. Handling the data and clock lines will be identical, with a

²³ BCD (*Binary-Coded Decimal*) is a decimal representation encoded in binary form, where each decimal digit of a number is encoded separately using four bits (i.e., half a byte).

²⁴ It can be calculated using the formula $8 \times f_{osc} / N$, where f_{osc} is the typical refresh frequency for 8 connected displays and N is the number of connected displays.

slight difference in the latch call. Because the MAX7219 shift register is 16-bit, the data must be latched after the second byte is sent.

There's another significant difference compared to the code we prepared for the shift register. It concerns the encoding of enabled segments into bytes. The MAX7219 requires that the bit value (power of 2) decrease for each segment, counting from A to G of a given digit, which is the opposite of what we did for the shift register. Therefore, we need to prepare a new digit encoding table.

Digit	H/DP $2^7=128$	A $2^6=64$	B $2^5=32$	C $2^4=16$	D $2^3=8$	E $2^2=4$	F $2^1=2$	G $2^0=1$	Common cathode (x=1; -=0)
0	-	x	x	x	x	x	x	-	126 (\$7E)
1	-	-	x	x	-	-	-	-	48 (\$30)
2	-	x	x	-	x	x	-	x	109 (\$6D)
3	-	x	x	x	x	-	-	x	121 (\$79)
4	-	-	x	x	-	-	x	x	51 (\$33)
5	-	x	-	x	x	-	x	x	91 (\$5B)
6	-	x	-	x	x	x	x	x	95 (\$5F)
7	-	x	x	x	-	-	-	-	112 (\$70)
8	-	x	x	x	x	x	x	x	127 (\$7F)
9	-	x	x	x	x	-	x	x	123 (\$7B)

In our code, we'll use the pin definitions prepared for the shift register. We'll also define several new constants (REG_TEST, REG_SCAN_LIMIT, REG_DECODE, REG_SHUTDOWN, REG_INTENSITY) useful for controller configuration. We'll also update the *NumberMasks* array with the new values. Since we'll be sending serial data multiple times, we'll separate the *Max7219_SendData* procedure for this purpose. This procedure will use the code (with minor modifications) prepared for the shift register. This procedure will be responsible for sending two bytes: the configuration (address) and the data to the MAX7219.

We'll leave the PB port pin definitions in the main part of the program. Next, we'll configure the controller. To ensure all connections are made correctly, we'll test our device, which should result in all segments of the three displays lighting up for approximately half a second²⁵. Next, we'll specify the number of displays to use (3), the digit decoding mode (0, i.e., manual), set the maximum segment brightness, and clear the data for the connected displays. We'll perform these operations in shutdown mode to prevent any accidental values from being displayed during our device configuration.

The main code will be placed in an infinite loop and will consist of displaying successive numbers from 0 to 999 approximately every 100 milliseconds on our displays. To determine the correct values for the ones, tens, and hundreds digits of a given number, we'll use the temporary variable *tmp*, which we'll divide by ten (*div*) and assign the remainder (*mod*) to the subsequent digits. It's worth noting that the *tmp* variable is of type UInt16 to fit within the required numerical range.

²⁵ Except for the decimal points, which we intentionally left unconnected.

```

program display_7segcc;

{$IFDEF attiny13}
  {$Fatal Invalid controller type, expected: attiny13}
{$ENDIF}

uses
  attiny13_basics;

const
  PB0 = %00000001; // PB0 = pin 5 of microcontroller -> DIN
  PB2 = %00000100; // PB2 = pin 7 of microcontroller -> CLK
  PB3 = %00001000; // PB3 = pin 2 of microcontroller -> LOAD

  REG_TEST      = $0F;
  REG_SCAN_LIMIT = $0B;
  REG_DECODE     = $09;
  REG_SHUTDOWN   = $0C;
  REG_INTENSITY  = $0A;

  INTENSITY_MAX = $0F;

  // masks for digits 0-9
  NumberMasks: array[0..9] of UInt8 = ($7E, $30, $6D, $79, $33, $5B, $5F,
  $70, $7F, $7B);
  // powers of 2, descending order due to shifts
  PowersOfTwo: array[0..7] of UInt8 = (128, 64, 32, 16, 8, 4, 2, 1);

procedure Max7219_SendData(const aCtrl, aData: UInt8);
var
  i: UInt8;
begin
  PORTB := PORTB and not PB3; // low state on LOAD

  // control byte - serial data transmission bit by bit
  for i:=0 to 7 do
  begin
    if (aCtrl and PowersOfTwo[i]) = PowersOfTwo[i] then
      PORTB := PORTB or PB0 // high state on DIN
    else
      PORTB := PORTB and not PB0; // low state on DIN

    PORTB := PORTB and not PB2; // low state on CLK
    PORTB := PORTB or PB2; // high state on CLK
  end;

  // data byte - serial data transmission bit by bit
  for i:=0 to 7 do
  begin
    if (aData and PowersOfTwo[i]) = PowersOfTwo[i] then
      PORTB := PORTB or PB0 // high state on DIN
    else
      PORTB := PORTB and not PB0; // low state on DIN

    PORTB := PORTB and not PB2; // low state on CLK
    PORTB := PORTB or PB2; // high state on CLK
  end;

```

```

PORTB := PORTB or PB3;           // high state on LOAD - latch data

DelayMs(10);
end;

var
  i, dig: UInt8;
  num, tmp: UInt16;
begin
  // direction register
  DDRB := DDRB or PB0;           // PB0 as output
  DDRB := DDRB or PB2;           // PB2 as output
  DDRB := DDRB or PB3;           // PB3 as output

  PORTB := PORTB or PB3;         // high state on LOAD - latch data

  // display test
  Max7219_SendData(REG_SHUTDOWN, 1); // normal display operating mode
  Max7219_SendData(REG_TEST, 1);     // enable test mode
  DelayMs(500);
  Max7219_SendData(REG_TEST, 0);     // disable test mode
  Max7219_SendData(REG_SHUTDOWN, 0); // display shutdown mode

  // display configuration
  Max7219_SendData(REG_SCAN_LIMIT, 2); // number of displays: 3
  Max7219_SendData(REG_DECODE, 0);    // manual decode mode
  Max7219_SendData(REG_INTENSITY, INTENSITY_MAX); // maximum brightness

  // clear display data
  for i:=0 to 2 do
    Max7219_SendData(i + 1, 0);

  Max7219_SendData(REG_SHUTDOWN, 1); // normal display operating mode
  DelayMs(500);

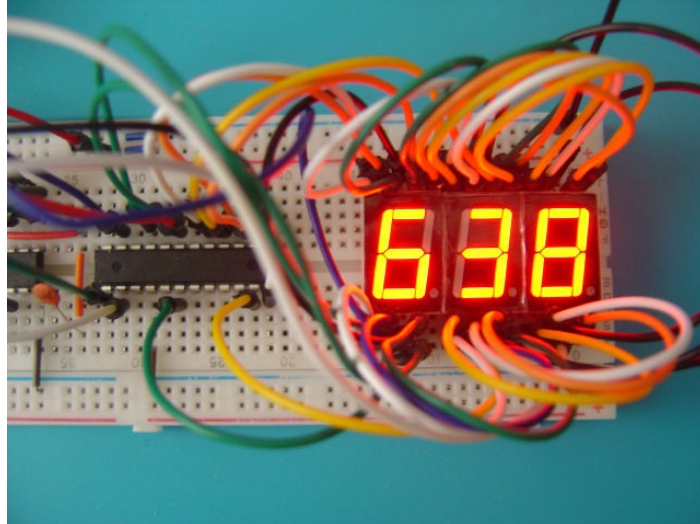
  while true do // infinite loop
  begin
    for num := 0 to 999 do
    begin
      tmp := num;

      for i := 0 to 2 do // ones-tens-hundreds iteration
      begin
        dig := tmp mod 10;
        Max7219_SendData(i + 1, NumberMasks[dig]);
        tmp := tmp div 10;
      end;

      DelayMs(100);
    end;
  end;
end.

```

If we haven't made any mistakes, the above code will work as expected. Naturally, we can modify it to create more interesting visual effects, for example, using the previously mentioned 8x8 LED matrices. However, it's clear that the small ATTiny13 microcontroller, especially when combined with other integrated circuits, can achieve quite a lot.



A cost that was difficult to avoid was the large amount of wiring we had to use. This didn't positively impact the aesthetics of our project, but we have to take this inconvenience into account when creating more complex projects.

Have fun!

Andrzej Karwowski

Nobody is perfect – found an error or inaccuracy, have a question or suggestion – write to ackarwow@gmail.com