Arduino SevenSeg v1.2

Sigvald Marholm

08.06.2015

Disclaimer

I do not take any responsibility for the usage of my code (or additional material such as this document). Although I've tested most of it, some parts may not be working, and some information herein may not be correct. Updates and backward/-forward compatibility are not guaranteed (but it may still happen). Use it at your own risk. I do provide it free of charge for non-commercial purposes.

Further on, I discovered after I started developing this library that there already was a similar library at the Arduino playground called `SevSeg` v.2.0. I decided to proceed, however, since that didn't quite have the flexibility I wanted. Nevertheless, `SevSeg` was first.

Contributions and Acknowledgements

I would like to thank Sascha Brüchert for contributing with a `keywords.txt` file to enable highlighting of library keywords. I would also like to thank those who have contacted me with potential improvements. Most of these are now implemented.

1 Introduction

`SevenSeg` is a flexible library for Arduino for outputting information to 7-segment displays. The main focus is to be a library which gets you started quickly while being flexible and cover most needs. That is, the most common 7-segment displays should be easily connected to an Arduino and information of various kinds should be easily output to it. This library is intended for beginners as well as more sophisticated users who just want something up and running. It is not intended to be an extremely lightweight library. Key functionality includes:

- Supports arbitrary number of digits and multiple displays
- Supports displays with decimal points, colon and apostrophe
- Supports common anode, common cathode and other hardware configurations
- High level printing functions for easily displaying:
  - Numbers (integers, fixed point and floating point)
  - Text strings
  - Time (`hh:mm`) or (`mm:ss`)
- Automatic multiplexing with adjustable refresh rate
- Adjustable brightness through duty cycle control
- Use of interrupt timers for multiplexing in order to release resources, allowing the MCU to execute other code
- Leading zero suppression (e.g. 123 is displayed as 123 rather than 0123 when using 4 digits)
- No shadow artifact

Future releases *may* happen, so feel free to contact me with suggested improvements and corrections (`marholm@marebakken.com`). In future releases, functionality and beauty of the code[1] will likely be prioritized above for instance backward compatibility.

How to Read This Document

Read the 2-page "Getting Started" section and you're good to go! Then use this as a look-up book for whenever you need to know how to do something.

[1] Today, parts of the code are clean an beautiful, while other parts are not. In the end I just had to finish up the project before getting tired of it. I hope to provide a cleaner code in future versions.

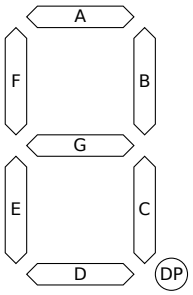

Figure 1: Labelling of a 7-segment digit including a decimal point (DP)

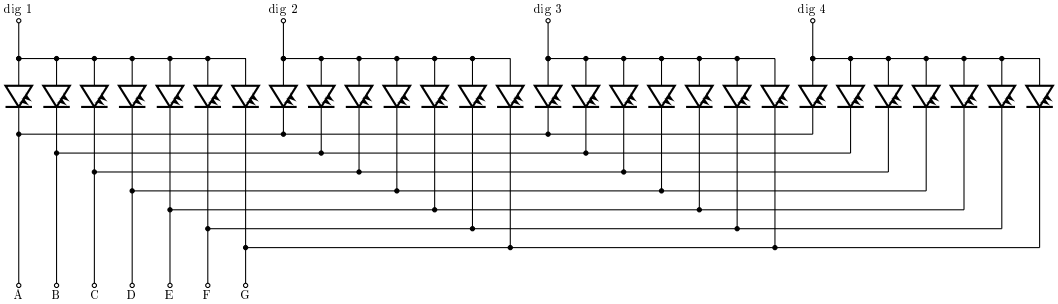

Figure 2: A 4-digit common anode display

2 Getting Started

Let's start with a quick example to get most users up and running. We will assume a 4-digit 7-segment common anode display. The segments in each digit are labelled A to G in the standard way as depicted in Fig. 1. Some displays also have an 8th segment for the decimal point (DP). This example assumes a display without that segment (see Sec. 3.3 for how to use `SevenSeg` with decimal points).

The schematics for a 4-digit common anode display is depicted in Fig. 2. Each segment houses a LED light. All anodes (positive terminal) on a digit are tied together into one pin which is hereinafter referred to as a *digit pin* (dig 1 to dig 4 on the schematic). The cathodes (negative terminals) of segments A to G are tied together across the digits, and their pins are hereinafter referred to as *segment pins* A to G.

For a common cathode display, it's the other way around: the cathodes are the digit pins, being connected together within each digit (hence the name "common cathode") while the anodes act as segment pins and are connected across the digits. Fig. 3 shows a 4-digit common cathode display.

Nevertheless, the digit pins should be connected directly to available output pins on the Arduino, while the segment pins should be connected to Arduino output pins through appropriately dimensioned resistors (see Fig. 10). Make sure the resistors are dimensioned such that neither the display nor the Arduino is damaged. For dimensioning of resistors, see App. A.

Finally, an example code for making this work is shown below:

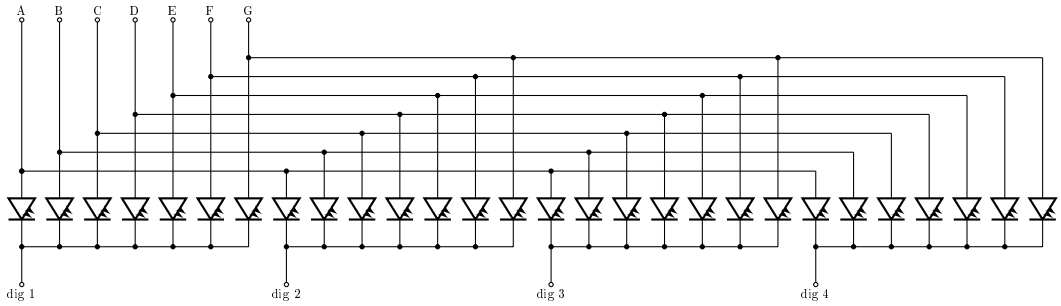

Figure 3: A 4-digit common cathode display

```
1  #include <SevenSeg.h>
2
3  SevenSeg disp(11,7,3,5,6,10,2);
4
5  const int numOfDigits=4;
6  int digitPins[numOfDigits]={12,9,8,13};
7
8  void setup() {
9
10   disp.setDigitPins(numOfDigits,digitPins);
11
12 }
13
14 void loop() {
15
16     disp.write(1358);
17
18 }
```

First, a 7-segment display object `disp` is initialized in line 3. The arguments of the constructor are simply the Arduino I/O pin numbers which the segment pins are connected to, i.e. segment A is connected to pin 11, segment B to pin 7 and so on.

On line 6 an array `digitPins` is created, holding the I/O pin numbers which the digit pins are connected to. I.e. the leftmost digit in the display is connected to pin 12 and the rightmost to pin 13. By sending this array (or actually the address to this array) to the member function `setDigitPins()` on line 10 you tell the display object to use these pins as digit pins. Note that the array `digitPins` must be kept in global scope[2]. The `SevenSeg` library assumes a common anode display is used unless otherwise stated. If a common cathode display was connected, it would suffice to insert `disp.setCommonCathode();` on line 11.

Finally, the member function `write()` is run repetitively on line 16 in order to write the number 1358. For other write-functions, to display clocks, and decimal numbers, see Sec. 4.1.

If other, time-consuming tasks must be performed within `loop()`, consider using interrupt timers to free resources. See Sec. 4.4 for more details.

[2]Sorry about that. It's global in order to allow an arbitrary number of digits while preventing dynamic memory allocation.

3 Hardware Setup

This section contain information about how to configure the SevenSeg-library to the hardware. These member-functions are typically called in `setup(){...}` except for the declaration of the `SevenSeg`-object which can be put in global scope to make it accessible in `loop(){..}`.

3.1 Set Digit and Segment Pins

These member functions are used to configure the segment pins (labelled A-G) and digit pins for the display (c.f. Sec. 2).

```
SevenSeg(int A,int B,int C,int D,int E,int F,int G);
```

Defines the segments A-G (c.f. Fig. 1) to be connected to the Arduino I/O pins given by the variables with the same name. See Sec. 2 for example.

```
void setDigitPins(int numOfDigits, int *pDigitPins);
```

Defines the number of digits to be `numOfDigits` and the digit pins to be the elements of the array `pDigitPins` (assumed to be of length `numOfDigits`). Keep in mind that `pDigitPins` must be stored outside the `SevenSeg`-object. See Sec. 2 for example.

If it is a one-digit display, it is possible to tie the digit pin directly to ground (in case of common cathode) or supply (in case of common anode) to spare one Arduino I/O-pin. In this case `setDigitPins()` should not be called.

3.2 Set Circuit Topology

```
void setCommonAnode();
```

Tells the library that the display connected is of common anode type. That is the default so I can't really think of a reason to call this function.

```
void setCommonCathode();
```

Tells the library that the display connected is of common cathode type.

```
void setActivePinState(int segActive, int digActive);
```

Configures whether the segment and digit pins should be active high or low. As an example, a common cathode display has segment pins that are active high, since the LEDs light up for high segment pins. The digit pins, however, are active low, since only digits with a low digit pin are on. Hence, calling `setCommonCathode();` is equivalent to calling `setActivePinState(HIGH,LOW);`.

The `setActivePinState()`-function, however, allows for a wider range of circuit topologies. Take for instance that you want to run more

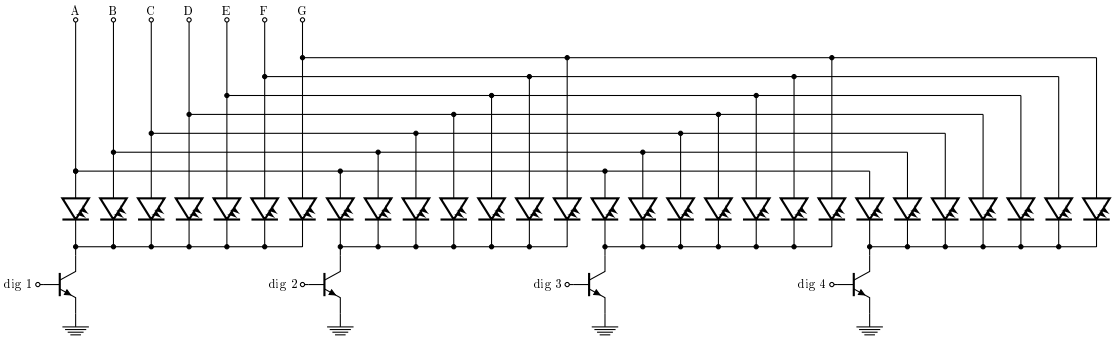

Figure 4: A 4-digit common cathode display with cathode transistors. (Remember to put resistors on the gates of the transistors to limit the base current. 10 kΩ usually works.)

current through the digit pins of a common cathode display than allowed per Arduino I/O pin. A classic solution in this case would be to add NPN-transistors at the digit pins as shown in Fig. 4. This, however, inverts the digit pins. For this topology, call `setActivePinState(HIGH,HIGH);`.

3.3 Decimal Point (Comma)

```
void setDPPin(int DPPin);
```

Specifies that the digits have a decimal point (DP) as depicted in Fig. 1, and that its segment is connected to `DPPin`. See Fig. 10 for an example of a display with a decimal point.

3.4 Colon and Apostrophe

Colons and apostrophes are realized in many different ways in 7-segment displays. The `SevenSeg` library will try to cover the most frequently used implementations. To do this, three main categories of displays are identified, all assuming that no more than one colon and one apostrophe are present:

Additional segment pin for colon. Two separate LEDs are used for the upper and lower dot in the colon (UC and LC) as shown in Fig. 5. They share an additional segment pin (labelled "colon") for turning on/off colon. They share digit pins with other digits. On some variants only one LED (UC) is present (at least apparently). This is indicated by the dashed line. These kind of displays are configured by `setColonPin()`.

Additional digit pin for colon and apostrophe. On these kind of displays a separate digit pin "symb" is used for symbols: apostrophe (AP) and colon (UC and LC if split in two). The symbols share segment pins with the usual segments A-G. See Fig. 6. These displays are configured by `setSymbPins()`.

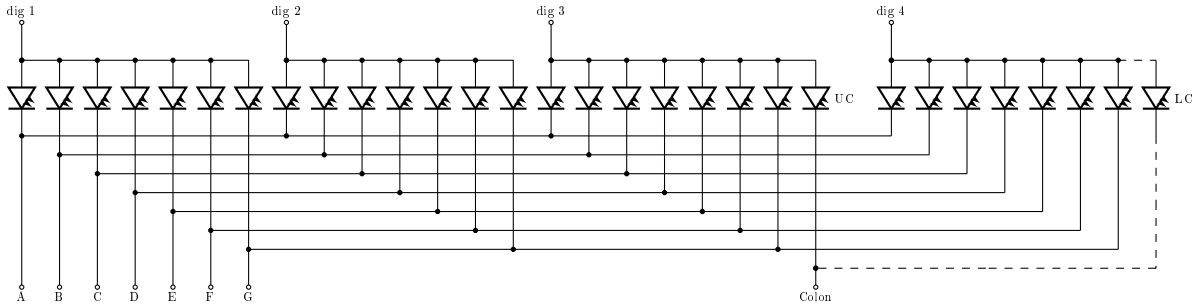

Figure 5: A 4-digit common anode display with a separate segment pin for colon

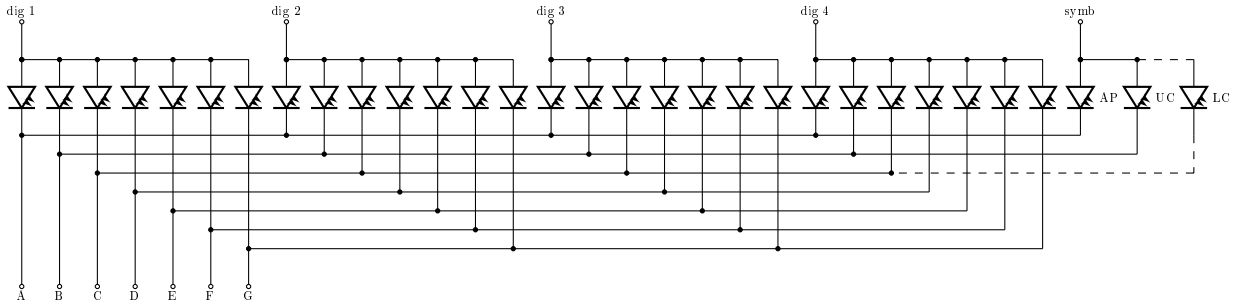

Figure 6: A 4-digit common anode display with a separate symbol digit pin for colon and apostrophe

Unterminated LEDs for colon and apostrophe. A third category of displays have both anodes and cathodes freely available without being tied to anything. These displays must be hardwired into a configuration that fits with one of the other two configurations.

```
void setColonPin(int colonPin);
```

Tells the `SevenSeg`-object that it has a colon available as an additional segment on pin `colonPin`. Whether it is split in two (UC and LC) or not is non-relevant. It is also non-relevant with which digits they share the digit pin.

```
void setSymbPins(int digPin, int segUCPin, int segLCPin, int segAPin);
```

Tells the `SevenSeg`-object that it has a colon and an apostrophe available. The symbol pin is set to `digPin` and the apostrophe segment pin is set to `SegAPin` (which will be the same as one of the segment pins for segments A-G). In case of split colon, `segUCPin` and `segLCPin` are the segment pins for the upper and lower colon LED, respectively. In case of only one LED for colon, let `segUCPin` be equal to `segLCPin`.

For high-level printing functions, `SevenSeg` will automatically take care of multiplexing properly through all digits, including the symbol pin.

3.5 Example: 4-digit Display with Symbols

Finally, a more complex example. How to setup a 4-digit common cathode display with decimal points, one unterminated LED for apostrophe, and one unterminated LED for colon.

First, the cathodes of the apostrophe LED and the colon LED are tied together and connected to Arduino I/O pin 1. This is the symbol pin. The anode of the apostrophe LED is tied together with segment pin A and connected to Arduino I/O pin 11 through a resistor. The anode of the colon LED is tied together with segment pin B and connected to the Arduino I/O pin 7 through a resistor. The other segment pins C-G are connected (through resistors) to I/O pins 3, 5, 6, 10 and 2, respectively. The DP segment pin is connected to pin 4. At last, the digit pins 1 to 4 (leftmost to rightmost) are connected to pins 12, 9, 8 and 13, respectively. Then, the code to initialize it would be:

```
#include <SevenSeg.h>

SevenSeg disp(11,7,3,5,6,10,2);

const int numOfDigits=4;
int digitPins[numOfDigits]={12,9,8,13};

void setup() {

  disp.setDigitPins(numOfDigits,digitPins);
  disp.setCommonCathode();
  disp.setDPPin(4);
  disp.setSymbPins(1,7,7,11);

}

void loop() {

// Printing functions here

}
```

3.6 Example: Multiple Displays

It is possible to have multiple displays controlled by different `SevenSeg` objects although this requires particular attention. Consider for example a 4-digit and a 2-digit common anode display. The displays should share segment pins, e.g. segment A of both displays are connected to the same pin on the Arduino and likewise for segments B to G. Remember that if you're out of digital pins you can also use analog pins, as is done in this example. The digit pins, of course, are separate.

In order to make the library multiplex all $n = 6$ digits at once, the write functions for both displays are executed consecutively in `loop(){...}`. However, after writing to one display, all segments must be set inactive by calling `clearDisp()`. Otherwise the last digit written will influence the first digit written to the next display. See the example below:

```
#include <SevenSeg.h>

SevenSeg disp1(10,11,13,A2,A1,12,A0);
SevenSeg disp2(10,11,13,A2,A1,12,A0);
```

```
5
6  const int numOfDigits1 = 4;
7  const int numOfDigits2 = 2;
8  int digitPins1[numOfDigits1]={9,8,7,A4};
9  int digitPins2[numOfDigits2]={6,5};
10
11 void setup() {
12
13     disp1.setDigitPins(numOfDigits1,digitPins1);
14     disp1.setDigitDelay(1667);
15
16     disp2.setDigitPins(numOfDigits2,digitPins2);
17     disp2.setDigitDelay(1667);
18
19 }
20
21 void loop() {
22
23   disp1.write(1234);
24   disp1.clearDisp();
25
26   disp2.write(56);
27   disp2.clearDisp();
28
29 }
```

Moreover, since the two objects are unaware of each other both running in `loop(){...}`, the refresh rate will not be 100 Hz which is default. The function `setRefreshRate()` also do not work. When multiplexing several displays the time to spend (in microseconds) on each digit must be manually calculated and specified using `setDigitDelay()`. For this example the time is calculated as follows (assuming a refresh rate of $f = 100\,\text{Hz}$ which is usually a good number):

$$T_{\text{digit}} = \frac{1}{nf} = \frac{1}{6 \cdot 100\,\text{Hz}} \approx 1667\,\mu\text{s} \tag{1}$$

Unfortunately, it is not possible for several displays to share segment pins while using interrupt timers.

4 Writing to the Display

The printing functions usually goes into `loop(){...}` and are divided into two categories; low-level functions which are rather basic but gives the programmer full control of the display, and high-level functions which are easier, more intelligent and should be suitable for most needs. For the low-level functions the programmer will have to handle multiplexing and parsing himself, whereas the high-level functions does this for you. The example in Sec. 2 uses the high-level printing function `write()`.

4.1 High-level Printing Functions

The high-level printing functions takes care of parsing and displaying numbers, text strings, etc. for you as long as they are run in an endless loop (i.e. `loop(){...}`). If you cannot afford to run them in an endless loop, consider using the built-in support for interrupt timers as described in Sec. 4.4. `void write(long num);`

Writes the number `num` on the display. See Sec. 2 for an example.

Supports signed integers. If the numbers are out of range they are trimmed to the largest positive or negative number the display can show. I.e. `write(1234)` will output 1234 on a 4-digit display but 99 on a 2-digit display. `write(-50)` will print -9 on a 2-digit display and -50 on displays with at least three digits.

```
void write(long num, int point);
```

Similar to `write(long num)` except that this one writes a fixed point decimal. The integer `point` tells how many digit should be treated as decimals. Example: `write(1234,2)` outputs "12.34". As for the above function `num` will be trimmed if outside the range of what the display can handle.

```
void write(double num);
```

Writes the (positive or negative) floating point number `num`. Example: `writeFloat(123.45)` yields "123.5" on a 4-digit display (note the round-off[3]). Numbers are trimmed if out-of-range. Uses as many decimals as can fit the display.

```
void write(double num, int point);
```

Same as above, but number of decimals to print are specified by `point`.

```
void write(char *str);
```

Writes the null-terminated text string pointed to by `str`. Example: `write("open")` displays the text "open".

Valid characters are a-z, A-Z, 0-9, minus (-), space ( ), decimal point (.), and degree (°). Small and capital letters are displayed equally. The degree-symbol should probably be written as the escaped character '\370'. I.e. to print "24°C" you should write `write("24\370C")`. Note that this `"24\370C"` is actually a four character string.

```
void write(String str);
```

Same as above but uses the `String` object rather than null-terminated string.

```
void writeClock(int mm,int ss,char c);
```

Writes the time in the format `mm:ss` if `c==':'`, `mm.ss` if `c=='.'` or simply `mmss` if `c=='_'`. `mm` and `ss` suggests using it for minutes and seconds but it could, of course, also be used for hours and minutes.

```
void writeClock(int mm,int ss);
```

[3] Most people would agree that 0.5 is rounded up to 1, but it is not so clear which way -0.5 is rounded. To 0 or to -1? The thing is that "rounding" is not a well-defined operation so both could be correct. This library uses the convention of rounding -0.5 to -1 for no particular reason.

Same as above but automatically uses colon if it exists, decimal point if not or simply nothing if the display has neither colon nor decimal points. See Sec. 3 for configuration of decimal points and colons.

```
void writeClock(int ss,char c);
```

This writes the time in the format `mm:ss`, `mm.ss` or `mmss` depending on whether `c` is ':', '.' or '_'. The minutes are derived from the seconds. Example: `writeClock(72,':')` outputs `01:12` since 72 seconds is 1 minute and 12 seconds.

```
void writeClock(int ss);
```

Same as above but automatically uses colon if it exists, decimal point if not or simply nothing if the display has neither colon nor decimal points. See Sec. 3 for configuration of decimal points and colons.

4.2 Low-level Printing Functions

```
void clearDisp();
```

Clears the display.

```
void changeDigit(int digit);
```

Activates the digit given by `digit` (and deactivates the others). I.e. `changeDigit(1)` makes the leftmost digit the active one. Each time `changeDigit()` is called all the segments are cleared. Hence, the following (erroneous) code will leave digit 2 empty:

```
writeDigit(4);     // Activates segments in '4'
changeDigit(2);    // Error: clears all segments
delay(5);
```

while this code will print the number 4 on digit 2:

```
changeDigit(2);
writeDigit(4);
delay(5);
```

The reason for this behaviour is to prevent shadows[4]. The purpose of the delay is to leave the segment on for some milliseconds before changing digit again. Immediately changing digits will make the digits be on for only a brief moment (as long as it takes to process the code) before the `changeDigit()` clears the digit again resulting in a very dim light.

```
void changeDigit(char digit);
```

`changeDigit('')` (with a white space as argument) deactivates all digits (same as `clearDisp()`) while `changeDigit('s')` activates the symbol digit. The symbol digit is a separate digit used for representing colon and apostrophe. See Sec. 3.4 for more about the symbol digit. When the symbol digit is activated, the SevenSeg-library remembers which segments should be active and not. See `setColon()`, `clearColon()`, `setApos()` and `clearApos()`.

[4] The shadow effect: If the digit 1 displays "4" and digit 2 is activated without clearing the segments first, the digit 4 would show up for a short moment until the segments are cleared, leaving a weakly visible shadow of the number "4" on digit 2.

```
void writeDigit(int digit);
```

Writes the number `digit` to the active digit. See `changeDigit()` for an example of how to use it.

```
void writeDigit(char digit);
```

Outputs the character `digit` to the active digit. The following example outputs the string "-3F":

```
changeDigit(1);
writeDigit('-');
delay(5);
changeDigit(2);
writeDigit('3');
delay(5);
changeDigit(3);
writeDigit('F');
delay(5);
```

Valid characters are a-z, A-Z, 0-9, minus (-), space ( ), and degree (°). Small and capital letters are displayed equally. The degree-symbol should probably be written as an escaped character, i.e `writeDigit('\370')`.

```
void setDP();
```

Activates the decimal point (comma) on the active digit. Example of writing "3." on digit 2:

```
changeDigit(2);
writeDigi(3);
setDP();
delay(5);
```

The decimal point is automatically reset when calling `changeDigit()` in order to prevent shadowing.

```
void clearDP();
```

Deactivates the decimal point on the active digit.

```
void setColon();
```

Turns on the colon segment(s). See Sec. 3.4 for how colons are implemented in hardware. If colon utilizes an additional segment pin, this function behaves similar to `setDP()` in that it is cleared on each `changeDigit()`. If a separate digit pin for symbols is used instead, `setColon()` means that colon segment should be automatically turned on each time the symbol pin is activated using `changeDigit('s')`. To clear it, you must call `clearColon()`.

```
void clearColon();
```

Clears colon. This happens automatically if colon is implemented in hardware by using an additional segment pin. If a symbol digit pin is used, however, this must be called manually or the library will remember it each time `changeDigit('s')` is called upon. See `setColon()`.

```
void setApos();
```

Sets the apostrophe. Behaves in the same way as `setColon()`.

```
void clearApos();
```

Clears the apostrophe. Behaves in the same way as `clearColon()`.

4.3 Multiplexing

The high-level printing functions (c.f. Sec. 4.1) automatically parses and multiplexes[5] the data to be displayed. The functions in this subsection allows the user to tweak parameters of the multiplexing.

```
void setRefreshRate(int freq);
```

Sets the refresh rate used for the display for high-level printing functions. I.e. `setRefreshRate(150)` means that the whole display (all digits) updates 150 times each second.

If you have n digits and a refresh rate of f (in Hz) the display will spend $T_{digit} = 1/(nf)$ seconds per digit[6]. The limit for when flickering becomes visible lies at under 50 Hz[7] (or perhaps somewhat higher if the display is vibrating or moving with respect to the observer). The `SevenSeg`-library has a default refresh rate of 100 Hz to ensure smooth operation by default.

```
void setDigitDelay(long int delay)
```

Rather than setting the refresh rate you can also set the quantity T_{digit} to `delay` (in microseconds) directly. See `setRefresRate()`.

```
void setDutyCycle(int dc);
```

The brightness of the display can be controlled by adjusting the duty cycle. As mentioned for `setRefreshRate()`, the time spent per digit is $T_{digit} = 1/(nf)$. The duty cycle determines for how large part of this time the digits will actually be on. I.e. `setDutyCycle(40)` means that each digit will be on only 40% of its assigned time T_{digit} and off for the rest of it[8]. `dc` should be a number (in percent) in the range $[0, 100]$. The default value is 100% (max brightness). See App. A for details about calculating the brightness.

[5] Multiplexing is the process of showing one digit at a brief time before showing the next digit and so on. Doing this repetitively and at a sufficiently fast refresh rate makes it appear as if all digits light up at the same time.

[6] If you have for instance a 4-digit display with a separate symbol digit for apostrophe and colon then n also includes the symbol pin; $n = 5$. See Sec. 3.4.

[7] That's why old CRT TVs has a refresh rate of 50Hz (in Europe at least).

[8] Technically, it might be more correct to say that dc/n is the duty cycle rather than dc, since that's the percentage of the time each digit is on. I.e. if you set $\mathrm{dc} = 100\%$ and have $n = 4$ digits then, technically, each digit is on only 25% of the time, not 100%. Nevertheless, I find the definition used herein more convenient, since its easier and maps directly to brightness without depending on the number of digits.

4.4 Using Interrupt Timers

Running the printing functions in an endless loop to perform multiplexing is not always an ideal way to do things. Outputting information to a 7-segment display is not a computationally intensive task, but due to the delay used for multiplexing, the microcontroller just sits there and waits for most of the time. Sure, you can insert commands taking little time in a loop together with the printing functions, but if they are slightly time-consuming, the display will halt or flicker. For this purpose it is possible to use SevenSegalong with interrupt timers. Then, you can do whatever you want inside `loop(){...}`, and simply run a high-level printing function only when you want to change what's on the display. The microcontroller will automatically be interrupted just briefly to update the display as needed. You can still change the refresh rate and the duty cycle like normal, the SevenSeg-library will take care of configuring the timers for you.

Let's begin with an example of how to get started with timers:

```
1  #include <SevenSeg.h>
2
3  SevenSeg disp(11,7,3,5,6,10,2);
4
5  const int numOfDigits=4;
6  int digitPins[numOfDigits]={12,9,8,13};
7
8  void setup() {
9
10   disp.setDigitPins(numOfDigits,digitPins);
11
12   disp.setTimer(2);
13   disp.startTimer();
14
15 }
16
17 void loop() {
18
19   for(int i=1;i<=10;i++){
20     disp.write(i);
21     delay(1000); // Or other time-consuming tasks
22   }
23
24 }
25
26 ISR(TIMER2_COMPA_vect){
27   disp.interruptAction();
28 }
```

Timer 2 is initiated in the `setup(){...}` section. At the bottom of the file is an Interrupt Service Routine (ISR) which is called whenever timer 2 interrupts the controller. What needs to be done when this happens is taken care of by the function `interruptAction()`. Finally, run the high-level printing functions such as `write()` like before. But notice how there's a delay of one second after it's called. This, however, will not disturb the display. The delay could be as long as you wish, or other time-consuming code could be executed. `write()` only needs to be executed when you need to change the value on the display.

Beware that there's a big caveat with using interrupt timers in Arduino; the Arduino platform uses the interrupt timers internally for its built-in functions such as `delay()`, `tone()`, for serial communications, etc. `delay()` for instance uses

timer 0. Hence in order for `delay()` to work, you can not use timer 0 for SevenSeg (or other purposes).

```
void setTimer(int timerID);
```

Tells the library that timer number `timerID` is to be used for multiplexing. `timerID` can be '0', '1' or '2'. Timers 3, 4 and 5 are not supported yet.

```
void clearTimer();
```

Clears the timer settings from the `SevenSeg`-object such that the object can again multiplex in the default way.

```
void interruptAction();
```

This function is to be put in `ISR(TIMER0_COMPA_vect){...}`, `ISR(TIMER1_COMPA_vect){...}` or `ISR(TIMER2_COMPA_vect){...}` for using SevenSeg together with timer 0, 1 or 2, respectively.

```
void startTimer();
```

This function is called to start the timer (automatically configures the timer for correct use with SevenSeg).

```
void stopTimer();
```

This function is called to stop the timer from running.

Version History

v1.0 (12.07.2013) Initial version

v1.1 (02.06.2015)

- `writeFloat(float)` changed to `write(double)`.
- `write(String)` created to support `String` objects (previously only character arrays was supported).
- Maximum number of digits increased from 4 to 9 (changed `int` to `long int` several places).
- User guide now includes example of how to control multiple display objects (also possible with v1.0 library).
- Leading zero suppression implemented (e.g. 123 is displayed as 123 and not 0123 on a 4-digit display).
- `write(double,int)` implemented.

v1.2 (08.06.2015) – Bug fix: Error in leading zero suppression. Numbers '0.02' would show as ' . 2' (and similar).

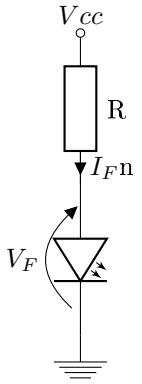

Figure 7: Forward biased diode with current limiting resistor. The order of the diode and the resistor is insignificant.

A Current Calculations

A.1 Basic LED Current Calculation

Consider first a simple forward biased[9] Light Emitting Diode (LED) as shown in Fig. 7. LEDs are current controlled devices, and the easiest way to control the current through an LED is by limiting it with a resistor. The way to dimension the resistor is to first choose the forward current I_F through the LED, then, determine the voltage drop V_F over the LED, and finally you compute the resistance R of the resistor.

As an example, we'll assume the diode 17-21USRC from Everlight. Some selections from the datasheet are included in Fig. 8 and Fig. 9. We see that the diode has a maximum forward current of 25mA, but according to the curve of luminous intensity I_V versus forward current, the diode should still light up relatively well at smaller currents (for more about luminous intensity, see Avago Application Brief D-004). Besides, we should have some margin to account for component tolerances and round-off errors in selection of components. For now we'll just choose the forward current somewhere in the mid-range: $I_F = 10\,\mathrm{mA}$. Next, the I_F vs. V_F curve shows that the voltage will be approx. $V_F = 1.8V$. If V_{cc} is the output of an Arduino then $V_{cc} = 5V$ (when the output is high) and the voltage across the resistor is

$$V_R = V_{cc} - V_F = 3.2\,\mathrm{V} \tag{2}$$

The current through the resistor must be I_F and using Ohm's law the resistance must be

$$R = \frac{V_R}{I_F} = \frac{3.2\,\mathrm{V}}{10\,\mathrm{mA}} = 320\,\Omega \approx 330\,\Omega \quad \text{(E12)} \tag{3}$$

The resistance was approximated to $330\,\Omega$ since not all values of resistors are commonly available. $330\,\Omega$ is a standard value in the E12-series of resistors.

[9] Forward biased simply means that plus is connected to the diodes anode and minus/-ground to its cathode such that the current flows in the forward direction. Diodes prevent currents from flowing in the reverse direction. For 7-segment displays this is utilized for multiplexing by letting only one digit be forward biased at a time.

■ **Absolute Maximum Ratings at Ta = 25℃:**

Parameter	Symbol	Rating	Unit
Reverse Voltage	V_R	5	V
Forward Current	I_F	25	mA
Operating Temperature	Topr	-40 ~ +85	℃
Storage Temperature	Tstg	-40 ~ +90	℃
Soldering Temperature	Tsol	260 (for 5 second)	℃
Electrostatic Discharge	ESD	2000	V
Power Dissipation	Pd	60	mW
Peak Forward Current(Duty 1/10 @ 1KHz)	I_F (Peak)	160	mA

■ **Electronic Optical Characteristics :**

Parameter	Symbol	*Chip Rank	Min.	Typ.	Max.	Unit	Condition
Luminous Intensity	Iv	A2	-----	2	-----	mcd	I_F=2mA
			16	34	-----		I_F=20mA
		A3	-----	3	-----		I_F=2mA
			29	46	-----		I_F=20mA
		A4	-----	3	-----		I_F=2mA
			35	58	-----		I_F=20mA
		A5	-----	5	-----		I_F=2mA
			46	75	-----		I_F=20mA
		A6	-----	6	-----		I_F=2mA
			58	93	-----		I_F=20mA
Viewing Angle	2θ 1/2	-----	-----	140	-----	deg	I_F=20mA
Peak Wavelength	λ p	-----	-----	639	-----	nm	I_F=20mA
Dominant Wavelength	λ d	-----	-----	631	-----	nm	I_F=20mA
Spectrum Radiation Bandwidth	△λ	-----	-----	20	-----	nm	I_F=20mA
Forward Voltage	V_F	-----	-----	2.0	2.4	V	I_F=20mA
Reverse Current	I_R	-----	-----	-----	10	μA	V_R=5V

Figure 8: Absolute maximum characteristics for Everlight 17-21USRC. Note that they may deviate from these values at temperatures other than $T = 25 \deg C$.

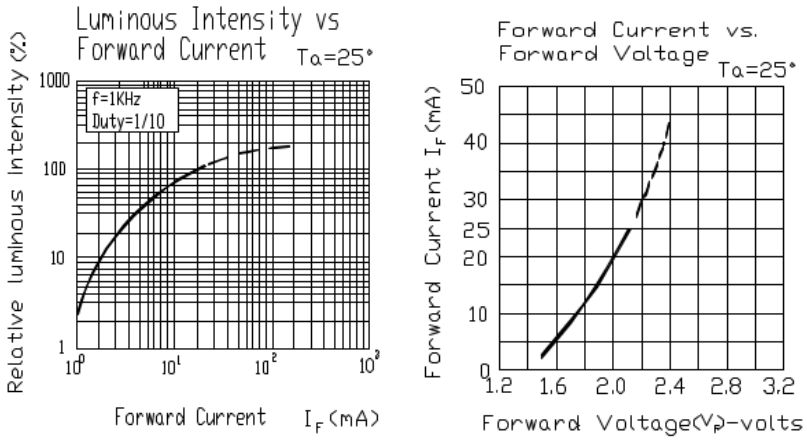

Figure 9: I_V vs. I_F (left) and I_F vs. V_F (right) for Everlight 17-21USRC. Notice how the dashed line is for higher currents than maximum DC-rated I_F. This region can only be utilized if multiplexing sufficiently fast.

This increased resistance will make the forward current slightly smaller, but visually not notably different.

Sometimes datasheet lack information. A typical dirty way to do it is to simply assume I_F to 5 or 10 mA and $V_F = 2V$, but you should make sure you're not overriding the absolute maximum characteristics which always should be in the datasheet.

A.2 The Current of Multiplexed Segments

For an n-digit multiplexed 7-segment display, you must connect the limiting resistors as shown in Fig. 10 (the example circuit is only 2 digits). The digit pins are multiplexed, such that only one digit is on at a time, before switching to the next digit. This happens fast enough to not be seen.

Due to this multiplexing, the *average* current through each segment will be

$$I_{\text{avg}} = (dc/n) \cdot I_F \tag{4}$$

where dc is the duty cycle as defined for the function `setDutyCycle()`. If `setDutyCycle()` is not used $dc = 1\,(= 100\%)$. Hence to have the same average current in the case of multiplexing a seven-segment display as for a constantly forward biased LED like above you should dimension R like before but for n times higher I_F. Even though the average forward current is equal, it may be that the luminous intensity is not, due to what is called the relative efficiency of the LED (Avago Application Brief D-004 is highly recommended for more information about this topic). However, it usually is not that big an error to say that the peak current I_F through an LED should be n times higher that it would've been for a single non-multiplexed LED since our eyes are not sensitive to an error of a couple of tens of percent. When choosing I_F, however, you should make sure the following criteria are met such that nothing is damaged:

1. I_F should not override the maximum value in the datasheet, typically 20-30 mA.

2. The maximum reverse voltage V_R for the segment LEDs should be higher than Arduinos I/O pin voltage $V_{cc} = 5V$ since multiplexing 7-segment displays imply reverse biasing some segments LEDs.

3. The current I_F will flow through the segment pins. Hence I_F should be lower than the maximum current handled by the Arduino I/O pin, which is 40 mA. You should probably stay well below this due to tolerances, etc., say, not more than 20-30 mA.

4. $7I_F$ will flow through the digit pin while displaying the number '8'. Hence $7I_F$ should also be lower than what can handled by the Arduino I_F pin; 40 mA or preferrably not more than 20-30 mA. Note that some displays have more than 7 segments per digit, i.e. if there is a decimal point (see Fig. 10) or colon (see Fig. 5). In that case, you need to multiply I_F by 8 or whatever number of segments are present per digit.

As a design procedure, you could start by assuming that you want $I_{\text{avg}} = 3\,\text{mA}$ of average current through each segment and calculate the resistors accordingly. You make sure all the above criteria are met, and test it (without using `setDutyCycle()`). From there on, experiment with different values of resistors until you are satisfied. If you want to utilize the adjustable brightness feature this will now be the maximum brightness. This is it! If you have troubles overriding the above listed criteria, go on reading the next two subsections.

A.2.1 Higher Currents Than Rated for the Display

Sometimes, you may want to put more current through the segments than the display can handle. For instance, you may want $I_{\text{avg}} = 10\,\text{mA}$ of average current through each digit, but since you have 4 digits and the maximum rating is $I_F = 20\,\text{mA}$ you can't get more than $I_{\text{avg}} = 5\,\text{mA}$. If you increase the I_F beyond the maximum, the display is damaged by the excess heat generated internally in the semiconductor. This only takes a few milliseconds, so even if the *average* current is below the maximum I_F there is a danger of damaging the display if you don't know what you're doing. If the display is switched on and off fast enough, however, the semiconductor does not heat up quick enough to be damaged. To do this, you typically need to increase the refresh rate of the multiplexing to 1000 Hz or more by calling `setRefreshRate()`. The datasheet often contain information about this; in Fig. 8 you see that the (peak) forward current of our example LED can be increased to 160 mA if the refresh rate is 1000 Hz, and the *true* duty cycle dc/n is not more than 10%. This yields an average current of maximally $I_F = 16\,\text{mA}$. You must, of course, set dc with `setDutyCycle()` to not override the 10% requirement. Let's take an example, you run the following code in `setup(){...}`:

```
1  setRefreshRate(1000);
2  setDutyCycle(40);  // dc = 10% * 4 digits = 40%
```

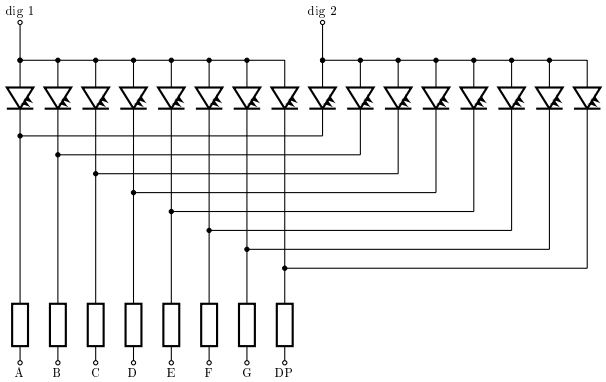

Figure 10: A 2-digit common anode digit with resistors (and a decimal point)

A.2.2 Higher Currents Than Rated for Arduino I/O Pins

On the other hand, you might need to put more current through a digit or segment pin to get the brightness you want than allowed by the Arduino. In that case you can add a transistor to the pin which you need to put more current through and use the function `setActivePinState()` to adapt the `SevenSeg`-library to your circuit. How to design such circuitry, however, is outside the scope of this text, but you can see an example in Fig. 4.

A.3 Dimming Through Duty Cycle Control

The power dissipated in an LED is proportional til *dc*. I.e. if the duty cycle is reduced to 50% the power consumed by the LED is also reduced to 50% and the luminous intensity is accordingly reduced to approximately 50%. This does not, however, imply that the LED will look half as bright. According to Weber-Fechner's law, our sight is a logarithmic function. A doubling (or halving) of luminous intensity is barely distinguishable except, perhaps, by direct comparison. Thence to create what appears to be a linear increase in brightness you should step up the duty cycle in accordance with either Weber-Fechner's law or Stevens' power law (which is slightly less accepted than Weber-Fechner's law but seems to be easier to implement).

Resources about this topic:
`http://forum.arduino.cc/index.php?topic=147818.10;wap2`
Avago Application Brief D-004
Avago Application Note 1005
`http://en.wikipedia.org/wiki/Weber%E2%80%93Fechner_law`
`http://en.wikipedia.org/wiki/Stevens%27_power_law`

Maybe I'll add more about this later.