

LED Fader 2

by

Mikael Ejberg Pedersen

Index

| | |
|--|----|
| 1 Introduction..... | 2 |
| 1.1 What is LED Fader 2?..... | 2 |
| 1.2 What isn't it?..... | 2 |
| 1.3 How do I use it?..... | 2 |
| 1.4 Hardware requirements..... | 2 |
| 1.4.1 Mega16..... | 3 |
| 1.4.2 Mega8..... | 3 |
| 2 Using the scripts..... | 4 |
| 2.1 Writing scripts..... | 4 |
| 2.2 The script commands..... | 5 |
| 2.2.1 S_Set..... | 5 |
| 2.2.2 S_Fade..... | 5 |
| 2.2.3 S_Increment..... | 6 |
| 2.2.4 S_Decrement..... | 6 |
| 2.2.5 S_Delay..... | 6 |
| 2.2.6 S_Goto..... | 6 |
| 2.2.7 S_Gosub..... | 7 |
| 2.2.8 S_Return..... | 7 |
| 2.2.9 S_Start..... | 7 |
| 2.2.10 S_Stop..... | 7 |
| 2.2.11 S_BranchIfBitSet..... | 8 |
| 2.2.12 S_BranchIfBitClear..... | 8 |
| 2.2.13 S_BranchIfEqual..... | 8 |
| 2.3 Future planned commands..... | 9 |
| 3 Making your own script commands..... | 10 |
| 3.1 Parameters to the command..... | 10 |
| 3.2 The script records..... | 10 |
| 3.3 Accessing the script record..... | 11 |
| 3.4 The LED records..... | 11 |
| 3.5 AVR register usage..... | 11 |
| 3.6 Example script command..... | 12 |
| 4 The PWM generator..... | 13 |
| 4.1 Timer1 interrupts..... | 13 |
| 4.2 The Timer1 reload table..... | 13 |
| 4.3 The PWM tables..... | 14 |
| 5 What the future may bring..... | 15 |
| 6 Software release notes..... | 16 |

1 Introduction

1.1 What is LED Fader 2?

The LED Fader 2 is an application for the ATmega8 and ATmega16 controllers, that can control up to 16 independent LED's. The 16 outputs are pulse width modulated to be able to make nice color transition effects.

The outputs are controlled by one or more scripts. The scripts have a number of commands available to control the PWM outputs and manipulating script execution. New commands can easily be added.

1.2 What isn't it?

LED Fader 2 is specifically made for dimming LED's. Some special techniques are used for this very purpose, which makes the PWM unsuitable for driving other things, like lightbulbs or electric motors.

1.3 How do I use it?

If you just want to use the program to make some LED's light up in nice patterns, you'll only need to read chapter 2 Using the scripts. Almost no AVR assembler skills are required to do this. The only steps involved are to modify the scripts, compile the code and program the AVR with it.

If you want to develop your own commands, that can be used in the scripts, then read chapter 3 Making your own script commands. Some AVR assembly language knowledge is required.

If you want to use the PWM part of the program, without the script system, then this is possible too. The PWM control is independent of the scripts. However, you'll need to know how the PWM output is generated in order to use it directly. This is explained in detail in chapter 4 The PWM generator.

1.4 Hardware requirements

The program is written for the Atmega8 and ATmega16 AVR's running at 16 MHz. The code should easily port to other mega AVR's capable of running 16 MHz.

The program can run with the outputs inverted or non-inverted. The LED's on the STK500 kit are inverted.

Settings for controller type, output inversion and which ports are used, are changeable at the beginning of the `Ledfade2.asm` file.

1.4.1 Mega16

The Mega16 implementation is straightforward. It uses two full ports for the 16 outputs. Nothing special to consider here, which also makes the Mega16 implementation the obvious choice for modifications or porting to other AVR's.

1.4.2 Mega8

The Mega8 code is a little bit odd, which doesn't make it as good a starting point for modifications as the Mega16 code. The Mega8 does not have two full 8-bit ports available, and that complicates things a little bit.

The 16 outputs are as follows:

Output 0 to 3 are at PortD pin 4 to 7.

Output 4 to 9 are at PortB pin 0 to 5.

Output 10 to 15 are at PortC pin 0 to 5.

This way, the four lower PortD pins are free for other purposes. I have future plans involving some RS-485 communication, which will require the UART and a control pin or two.

Do not change PortD in the code. The outputs on PortB and PortC can be interchanged, as they both use the same 6 bits for output.

2 Using the scripts

2.1 Writing scripts

At startup, the program will start a single script, called `ScriptStart`. This script can be the only script, controlling all needed outputs. Or it can start other scripts. Up to 10 scripts can run simultaneously. The scripts are located in the file *scripts.inc*.

A script is in essence nothing but a table in the program memory. A line in a script always start with a command. This command is then executed. Some commands require one or more parameters. An example is the `S_Delay` command: It needs a single parameter that determines the duration of the delay. All parameters are given after the command.

WARNING: Be sure to write the correct number of parameters, or the script can easily crash the program. There is no syntax checking.

Also, be sure to end the script by either stopping it or make it jump to another location.

Example of a small script:

```
ScriptStart:
.dw S_Fade, 0, 16, 68
.dw S_Delay, 68
.dw S_Fade, 0, 240, 224
.dw S_Delay, 224
.dw S_Fade, 0, 255, 68
.dw S_Delay, 68
.dw S_Goto, ScriptStart
```

2.2 The script commands

2.2.1 S_Set

```
Syntax:  
.dw S_Set, <channel>, <PWMvalue>
```

This command set the output of one PWM channel to a given value. It does not do any fading, just sets the new value directly.

The channel is offset by the value given for the script, if it isn't run from the master script. See S_Start for further information.

<channel> can be 0-15.

<PWMvalue> can be 0-255, where 0 is completely off and 255 is completely on.

2.2.2 S_Fade

```
Syntax:  
.dw S_Fade, <channel>, <final>, <duration>
```

This is the main LED controlling command. It is used to control all fading on the PWM channels. The script will NOT wait for the fading to come to an end. It just starts the fading, and then immediately moves on to the next command in the script.

The channel is offset by the value given for the script, if it isn't run from the master script. See S_Start for further information.

Final defines the end value that, when reached, the fading will stop and the output will remain steady at this level.

Duration dictates how long it shall take to reach the end value. It is given in units of 10 ms. To make the fading happen in 2 seconds, you would have to set the duration to 200.

<channel> can be 0-15.

<final> can be 0-255.

<duration> can be 1-65535. 0 is not valid, and may result in unpredictable behavior. Use the command S_Set to instantly set an output to a desired value instead.

2.2.3 S_Increment

```
Syntax:  
.dw S_Increment, <channel>
```

The S_Increment command increments the PWM output value of a single channel. If this channel is fading, the increment will stop it.

The channel is offset by the value given for the script, if it isn't run from the master script. See S_Start for further information.

<channel> can be 0-15.

2.2.4 S_Decrement

```
Syntax:  
.dw S_Decrement, <channel>
```

Just as S_Increment increments an output, so will S_Decrement decrement it.

The channel is offset by the value given for the script, if it isn't run from the master script. See S_Start for further information.

<channel> can be 0-15.

2.2.5 S_Delay

```
Syntax:  
.dw S_Delay, <time>
```

The delay command stops script execution for <time> * 10 ms.

Fading of outputs does not stop. Other parallel running scripts are not stopped either.

<time> can be 1-65535. 0 can be used, but will not result in any delay.

2.2.6 S_Goto

```
Syntax:  
.dw S_Goto, <label>
```

The goto command move the script execution to another location. Usually used at the end to make the script start over from the top.

<label> is the address where the script should continue execution.

2.2.7 S_Gosub

```
Syntax:  
.dw S_Gosub, <label>
```

Gosub also jumps to another location to continue the execution, but the current location is remembered. Later, when the S_Return script is executed, the script will return to this location and continue. Nested gosub's are possible, but only 3 levels deep. If more gosub's are attempted, they will be ignored.

<label> is the address where the script should continue execution.

2.2.8 S_Return

```
Syntax:  
.dw S_Return
```

Returns execution to a previously called S_Gosub.
If no gosub has been called, the script will terminate.

2.2.9 S_Start

```
Syntax:  
.dw S_Start, <label>, <offset>
```

The system can run up to 10 different scripts in parallel. At startup only one script is started, but this script can start new scripts with the S_Start command. The current script will continue to run unaffected.

<label> indicates the start address of the new script.

<offset> can offset all channel references in S_Set and S_Fade. This is useful for starting the same script more than once. If a script is written to control output 0, 1 and 2, then starting the script with an offset of 5 will make the script use output 5, 6 and 7 instead. This way, the same script can be used several times to control different outputs. If this feature is not needed, set the offset to 0.

2.2.10 S_Stop

```
Syntax:  
.dw S_Stop
```

The stop command terminates the current script.
Other scripts running will continue unaffected.

2.2.11 S_BranchIfBitSet

```
Syntax:  
.dw S_BranchIfBitSet, <memaddr>, <bit>, <label>
```

This command will test a certain memory location for the status of a single bit. If the bit is set, the script will continue execution somewhere else (dictated by <label>). If the bit is cleared, the script will continue unaffected.

<memaddr> is the address of the byte to be tested. This can be a register (0x00 – 0x1f), an I/O-port (0x20 – 0x5F) or a RAM location. If an I/O-port is to be used, then remember to add 0x20 to its normal address.

<bit> can be 0-7.

<label> is the address where the script should continue execution if the bit was set.

2.2.12 S_BranchIfBitClear

```
Syntax:  
.dw S_BranchIfBitClear, <memaddr>, <bit>, <label>
```

This command will test a certain memory location for the status of a single bit. If the bit is cleared, the script will continue execution somewhere else (dictated by <label>). If the bit is set, the script will continue unaffected.

<memaddr> is the address of the byte to be tested. This can be a register (0x00 – 0x1f), an I/O-port (0x20 – 0x5F) or a RAM location. If an I/O-port is to be used, then remember to add 0x20 to its normal address.

<bit> can be 0-7.

<label> is the address where the script should continue execution if the bit was cleared.

2.2.13 S_BranchIfEqual

```
Syntax:  
.dw S_BranchIfEqual, <memaddr>, <value>, <label>
```

This command will test a certain memory location for a specific value. If the content are equal to <value>, the script will continue execution somewhere else (dictated by <label>).

<memaddr> is the address of the byte to be tested. This can be a register (0x00 – 0x1f), an I/O-port (0x20 – 0x5F) or a RAM location. If an I/O-port is to be used, then remember to add 0x20 to its normal address.

<value> is the value to test for. Can be 0-255.

<label> is the address where the script should continue execution if the content matches.

2.3 Future planned commands

All planned commands have now been implemented.

If you have other suggestions for commands, then you can contact me at mikael@ejberg.dk

3 Making your own script commands

If the existing script commands doesn't fit your needs, you can write your own. Read the rest of this chapter, and have a look at the existing commands for inspiration. Script commands are found in the file *scr_cmd.inc*.

3.1 Parameters to the command

The first thing every script command must do, is to move the content of the X register (XL and XH) to the Z register (ZL and ZH). The Z register is then pointing at the program memory location where the first parameter for the command is stored. Parameters can then be loaded using the `lpm` instruction.

Parameters are usually given in words, even if a byte would be enough. The main reason for that, is that it's easier to write the script using only `.dw` directives, thereby keeping each command and its parameters on one line.

The parameters are usually read with the Z+ variant of the `lpm` instruction. The goal is for the Z register to point at the first address past the last parameter. This is used to find the next command that should be run after the current one. Unless the command is going to force execution to continue elsewhere, or maybe stop the script completely, the Z register must point to the next command when the current command has been executed.

3.2 The script records

Every running script has a record of data that hold all relevant information for being able to run the script. Currently the record contains the following variables:

| Name | Size | Description |
|--------------------------------|---------|---------------------------------------|
| <code>ScriptRecPtr</code> | word | Pointer to the command being executed |
| <code>ScriptRecDelay</code> | word | Counter for delays |
| <code>ScriptRecOffset</code> | byte | Channel offset |
| <code>ScriptRecStackPtr</code> | byte | Local stackpointer |
| <code>ScriptRecStack</code> | 3*words | Local stack |

If `ScriptRecPtr` is 0, there is no active script, and no commands will be executed.

If `ScriptRecDelay` is anything but 0, the next command will not be executed. Instead this variable will be decremented by 1 for each 10 ms until 0 is reached.

`ScriptRecOffset` contains the channel offset that should be added to all output channel related operations. See chapter 2.2.9 for details of how it is used.

The stack and stackpointer are used for the `S_Gosub` and `S_Return` commands.

3.3 Accessing the script record

Certain commands may need access to some of the data in the script record for the running script. When a command is called, the Y register is pointing to the beginning of the script record. All data in the script record can then be accessed with the `ldd` and `std` instructions.

3.4 The LED records

Controlling the PWM outputs are also done by records. Each of the 16 outputs has a LED record. The content of a LED record is:

| Name | Size | Description |
|----------------|------|-----------------------------|
| LedRecCurrent | byte | Current output value |
| LedRecFinal | byte | End value |
| LedRecDuration | word | Time to complete the fading |
| LedRecCounter | word | Internal working variable |
| LedRecSlope | byte | Internal working variable |
| LedRecSign | byte | Internal working variable |

LedRecCurrent contains the current value of the output channel. Its range is 0-255. This value is directly used to generate the PWM output, and will be updated automatically.

LedRecFinal and LedRecDuration are equal to the parameters used in the `S_Fade` script command (chapter 2.2.2).

The remaining variables (LedRecCounter, LedRecSlope and LedRecSign) are used internally to generate the fading at the desired speed. When fading should start, LedRecFinal and LedRecDuration should be set to whatever is needed. Then the rest of the variables need to be initialized. This is done by calling the function `CalculateLedFadeParams`.

3.5 AVR register usage

The following registers are available to new script commands:

| | |
|-----------|---|
| r0 and r1 | May be used freely. Usually only used for the result after a <code>mul</code> instruction. |
| w1 to w6 | General purpose working registers. May be used freely. |
| u1 to u8 | General purpose working registers (low regs. No LDI etc). May be used freely. |
| X | Contains the parameter pointer. After copying it to the Z register, X may be used freely. |
| Y | Contains the script record pointer. If access to the script record is not needed, Y may be used freely. |
| Z | Contains the address of the called script. Should be overwritten with X as soon as possible to be able to read command parameters. At the end of the command, it must contain a pointer to the next script command to be run. |

Any other register should not be used.

3.6 Example script command

An example of a new script command could be one that makes a delay in seconds (instead of the default S_Delay, that operates in units of 1/100 of a second):

```
Example of a new script command to delay execution in seconds:

Syntax:
.dw S_DelaySeconds, <time>           ; <time> can be 1-255

S_DelaySeconds:
    movw ZH:ZL,XH:XL                 ; Restore Z (points at first parameter)
    lpm w1,Z+                         ; Read <time> parameter into w1
    adiw ZH:ZL,1                      ; Word alignment
                                       ; Z now points to the next command
    ldi w2,100                        ; It takes 100 * 10 ms to give a second
    mul w1,w2                          ; Multiply parameter by 100
    std Y+ScriptRecDelay,r0           ; Store the result in the Delay-
    std Y+ScriptRecDelay+1,r1        ; counter in the script record
    ret
```

4 The PWM generator

4.1 Timer1 interrupts

The heart of the PWM generation is the Timer1 interrupt. It is called somewhere between 400 and 1600 times per second. The timeslots for the interrupts are changed dynamically, according to the desired brightness of the LED's.

The timeslots for a PWM cycle are not evenly distributed over time. When pulsing LED's, the human eye "sees" the light as being brighter than it really is. The relation is roughly equivalent to x^2 . Note that this does not apply to ordinary lightbulbs, only LED's.

Therefore, the time between two following timeslots is varied according to this formula:

$t_{Interrupt} = k \cdot x^2 - k \cdot (x-1)^2$ where x is the PWM counter going from 1 to 255. k is a constant used to match the timing so we end up with an 100 Hz output.

The PWM outputs may have 256 different brightness levels, but as there are only 16 output pins, a maximum of 16 different output levels can be active at any given time. Therefore a dynamic timeslot allocation has been implemented to minimize the interrupt load on the AVR.

4.2 The Timer1 reload table

The Timer1 interrupt uses a dynamic table to lookup the duration until the next timer interrupt should occur, and then use that to program the Timer1 TOP value. This table is generated from the constant table "TC1TopTable", which contains all the durations between any two possible interrupts.

The table is made with the above mentioned formula. But in order to use that, we need to determine the value of k . We want 100 PWM cycles per second. And as the AVR is running at 16 MHz, one PWM cycle should last 160000 clock cycles. This gives us:

$$t_{PwmCycle} = \sum_{x=1}^{256} k \cdot x^2 - k \cdot (x-1)^2 = 160000 \quad \Rightarrow \quad k = \frac{160000}{256^2} = 2.4414$$

You need to recalculate k and the table, if you want to change the frequency of the AVR, the number of PWM levels in use or the PWM output frequency.

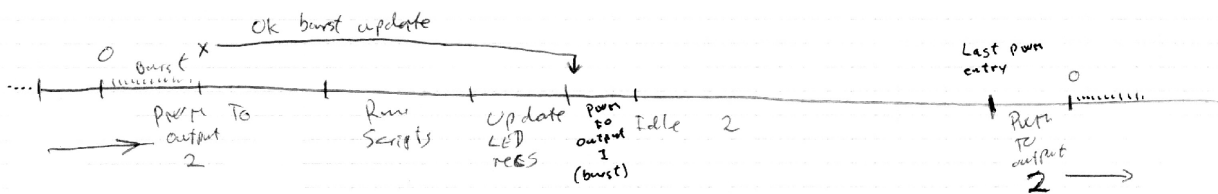
When k is known, the table almost makes itself. The first 16 entries aren't used, as the available time is too short for an interrupt to start, execute the code and finish in time for the next interrupt. So the first 16 PWM levels are generated from the first interrupt in the PWM cycle.

4.3 The PWM tables

The Timer1 interrupt also control the PWM outputs. To save time within the interrupt, all output values for an entire PWM cycle have been precalculated and stored in the two tables "BurstOutputTable" and "OutputTable". The first table is for the first 16 PWM values, that are very close together. The table contains only the bitmasks for the output ports. The second table contains the data for the remaining PWM values. In addition to the output data, it also contains the timer reload values, used to determine the time for the next interrupt.

The functions that prepares these tables, are "PWMTtoOutput1" and "PWMTtoOutput2". The first one makes "BurstOutputTable" and the second one makes "OutputTable". It has been split up into two separate functions due to timing issues. This way, "BurstOutputTable" is ready when it is needed, while the values for "OutputTable" are still being calculated.

Here should be something about the timing between the interrupts and the rest of the program, but I'm not quite sure of how to describe it. But for now, I'll let my handwritten design notes be here:



5 What the future may bring

My future plans for this software is to make some extension modules for it.

I am currently having two ideas:

1. To make a pulse width reader. This is to be able to read the output from a RC-receiver. The plan is to use the scripts to simulate the lights of an airplane, and use this on one of my model airplanes. The plan is to control the lights from my RC-transmitter.
2. I also have a plan of connecting several AVR's together, using RS-485. One AVR should be master, and be able to control the scripts of the other connected AVR's.

6 Software release notes

| Version | Date | Notes |
|----------|-------------|--|
| Ver 2.00 | 10 Oct 2003 | Initial release. Total rewrite from version 1. |
| Ver 2.01 | 12 Oct 2003 | Added S_Increment and S_Decrement script commands. Divided the ledfade2.asm file into several .inc files. |
| Ver 2.02 | 21 Oct 2003 | Added support for Mega8: Ledfade2_mega8.asm and pwm_mega8.inc added. |
| Ver 2.03 | 27 Oct 2003 | Interrupt speed optimized. TC1TOP table fixed. |
| Ver 2.50 | 6 Dec 2003 | Interrupt system rewritten to use dynamic time allocation. Support for 256 brightness levels instead of 64. |
| Ver 2.51 | 23 Dec 2003 | Updated the conditional assembly directives to the new AVR Assembler 1.74 format. |