

back to [sharewear](#)

Software

1. [Software](#)
2. [Getting the code](#)
3. [Building the firmware](#)
 1. [Compiling in new patterns](#)
 2. [Programming the AVR using the usbtiny ISP](#)
4. [Register layout description](#)
 1. [Pin directions](#)
 1. [PORT D](#)
 2. [PORT B](#)
 2. [Interrupt sources](#)
 1. [External Interrupts](#)
 2. [Pin Change Interrupts](#)
 3. [Timers](#)
 1. [Timer0 \(Soft Serial\)](#)
 2. [Timer1 \(LED Brightness Control\)](#)
 3. [Timer2 \(Animation Control\)](#)
 - 4.
5. [Watchdog Timer](#)
6. [RFID](#)
7. [Power-saving](#)
8. [Light animation editor](#)
 1. [Save](#)
 2. [Export](#)
 3. [Upload](#)

Getting the code

The sharewear repository lives in git. To get it use git clone:

```
git clone http://follicle.v2.nl/~simon/git/sharewear
```

Alternatively download the latest stable branch [without](#) or [with](#) AVR toolchain. The latter can be used with OSX systems without development tools.

The repository can also be viewed using [gitweb](#).

Building the firmware

To build the sharewear firmware the system needs to have a working **AVR toolchain**, see [avr-libc](#) for instructions. To communicate with the ATmega [avrdude](#) is used. If using the usbtiny for uploading, then see [here](#) for some guidelines. The sharewear image for OSX systems comes with a toolchain and no further software is needed if the [editor](#) application is used.

Inspect [make.conf](#) to see if the settings reflect the build system. Next issue a

```
make clean && make && make upload
```

This will make and upload the firmware to the ATmega. There is a script [upload_helper.sh](#) within the **scripts** directory which converts the animation files and builds the firmware, as well as uploads it to the ATmega. As an extra, it checks if **make** is present on your system. If not it uses [build.sh](#) in the **scripts** directory.

Compiling in new patterns

The recommended way to create patterns is by using the [editor](#). However it is possible to create patterns by hand. The numbers representing the pattern should be 10-bits (0-1023) and separated by newlines. These files can be converted by the **convert** program in the **wears** directory:

```
convert infile.dat outfile.binary
```

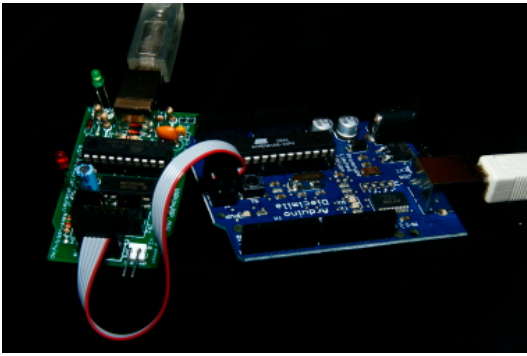
This converts the files into binary format.

The binary files need to be translated into a relocatable module format so that they can be linked in with the sharewear firmware. This can be done using **avr-objcopy**, distributed with the AVR toolchain. The input type is binary, the output type is elf32-avr. Also make sure to pass the following options `--rename-section .data=.progmem.data,contents,alloc,load,readonly,data`. It is also needed to rename the ***_start**, ***_end** and ***_size** symbols to **_binary_anim_mmodule[0-6]_data_start** using the `--redefine-sym` option. The created modules need to be placed in the **wears/moods**, the [wears/moods/moods.inc](#) file should reflect the module names.

This process is automated with the [make_bin_data](#) script in the **scripts** directory. The script can use or skip the conversion step using the `-c` option:

```
scripts/make_bin_data -c -o wears/moods editor/data/anim_mmodule*.dat
```

Programming the AVR using the usbtiny ISP



The arduino bootloader on the diecimila takes up ~2K of flash. The only thing this bootloader does is emulate a stk500 programmer over the serial connection. So it is possible to upload your program using the USB connector on the arduino without additional hardware. To save this 2K of flash we use the [usbtiny](#) ISP programmer to program the AVR, directly discarding the need of the bootloader. The usbtiny is a small cheap USB ISP programmer that works on GNU/Linux, OSX and Windows. At least version 5.4 of [avrdude](#) is needed for it to work out-of-the-box. To get it to work on a 64-bit machine apply patch [#14754](#), which fixes some architecture dependent type size issues.

Uploading the firmware from within the git repository root:

```
avrdude -B1 -c usbtiny -p m168 -U flash:w:wears/wear.hex
```

Register layout description

Pin directions

name	arduino pin	function	port	direction	state
RFIDTX	2	INT0	PD2	in	x
RFIDGPIO	3	INT1	PD3	in	x
REED1	4	PCINT20	PD4	in	x
REED2	5	PCINT21	PD5	in	x
LED1FLASH	6	DOUT6	PD6	out	low
LED2FLASH	7	DOUT7	PD7	out	low
RFIDRS	8	DOUT8	PB0	out	low
LED1PWM	9	OC1A	PB1	out	low
LED2PWM	10	OC1B	PB2	out	low

PORT D

DDRD (C0h)

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	0	0	0	0	0

PORTD (00h)

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0

PORT B

DDRB (07h)

7	6	5	4	3	2	1	0
B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	1	1	1

PORTB (00h)

7	6	5	4	3	2	1	0
B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	0	1

Interrupt sources

External Interrupts

The external interrupt source (INT[01]) are used for the RFID reader. When the RFID reader finds a tag it sends the data blocks read over its TX line. The start bit of the first byte block sent by the RFID reader pulls INTO low, triggering INTO and starting the reader clock. INT1 is connected to one of the GPIO outputs of the RFID reader. This output is high when a scan has been performed but no actual read has taken place (no RFID tag present). Using INT1 we can discover the absence of an RFID tag after a successful read.

- **INT0** (RFIDRX) on falling edge
- **INT1** (RFIDGPIO) on rising edge

EICRA (0Eh)

7	6	5	4	3	2	1	0
-	-	-	-	ISC11	ISC10	ISC01	ISC00
0	0	0	0	1	1	1	0

EIMSK (03h)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	INT1	INT0
0	0	0	0	0	0	1	1

EIFR

Unused

Pin Change Interrupts

PD4 and PD5 are used as pin change interrupt sources. They are triggered by the reed switches and used for detection of a (surrogate)dress. Pin change interrupts are grouped together into 3 groups of which INT20(PD4) and INT21(PD5) are in group PCIE2 (group 3).

- **PCINT20**
- **PCINT21**

PCICR (04h)

7	6	5	4	3	2	1	0
-	-	-	-	-	PCIE2	PCIE1	PCIE0
0	0	0	0	0	1	0	0

PCMSK2 (30h)

7	6	5	4	3	2	1	0
23	22	21	20	19	18	17	16
0	0	1	1	0	0	0	0

Timers

We will be using timers for 3 purposes

- light brightness control
- light animation control
- serial clocking

Timer0 (Soft Serial)

Timer 0 is used for clocking in the serial bits originating from the RFID module at 19200 baud. The Timer is set CTC mode with the OCR0A register as top for the counter. The prescaler is set to 64.

TCCR0A (02h)

7	6	5	4	3	2	1	0
COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
0	0	0	0	0	0	1	0

TCCR0B (03h)

7	6	5	4	3	2	1	0
FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
0	0	0	0	0	0	1	1

Timer1 (LED Brightness Control)

Light brightness control will use Timer1 of the ATmega168. Using Timer1 we can generate 2 16-bit PWM signals. The signals will have a base frequency of 122Hz. Although the resolution can be 16-bits we will only be using 10-bits. This leaves an additional 6-bits for meta-data encoding (e.g. animation hints, transitions, etc) within the animation data without adding to size. The timer runs in 10-bit Phase-Correct mode, OCR1 is used for top.

TCCR1A (A3h)

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
1	0	1	0	0	0	1	1

TCCR1B (03h)

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
0	0	0	0	0	0	1	1

Timer2 (Animation Control)

This timer is used for animation control and overflows every ~0.017ms. Every overflow an interrupt is generated at which a next brightness level is read. Timer 1 is adjusted accordingly changing the LED brightness. The Timer is set CTC mode, the prescaler is set to 1024. OCR2A is used to set the animation speed.

TCCR2A (02h)

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
0	0	0	0	0	0	1	0

TCCR2B (07h)

7	6	5	4	3	2	1	0
FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
0	0	0	0	0	1	1	1

Watchdog Timer

The ATmega168 has a watchdog timer (WDT) which can be used in 3 different modes

- System Reset
- Interrupt
- Interrupt & System Reset

The WDT is used in Interrupt mode, triggering an interrupt every second. This interrupt is used to reset the RFID module and to put the ATmega168 into sleep mode if no switches (PCINT2[01]) are enabled. The WDT interrupt is disabled before entering sleep mode, as not to wake up the ATmega168 when it triggers. The only way to wake up is by an interrupt of PCINT2[01] or a system reset. After waking up the WDT is re-enabled.

The WDT is reset and disabled on startup as early as possible (right after stack initialization). This prevents possible hiccups caused by the WDT being setup wrongly due to low power resets etc.

RFID

See [sharewear/hardware#RFIDModule](#) for a description of the RFID hardware. The RFID module is kept in reset and is woken up by the WDT every second if any of the switches are on. If none of the switches are on, then the RFID is kept in reset mode. The reset/wake sequence is done to make sure the RFID module wakes up out of a temporary power drop, e.g. if the battery power drops below a certain threshold the RFID module halts but the arduino keeps working. Experiments have shown that when bringing back the power to normal operating levels the RFID module is not able to wake up. Only a reset pulse brings the RFID module back to life.

The RFID module reads Manchester-64 encoded RFID tags with a magic starting block of **52588B45**. If this block is absent the tag is ignored, otherwise the data is sent over a serial line to the ATmega. The sharewear firmware is only interested in the first block following the magic block. This block holds the Migrating Mood Module identification number. The encoding is a simple sequence of **01010101** for module 1, **02020202** for module 2 and so on. 6 Modules have been defined, any unknown sequence will be treated as if no module is present.

Power-saving

The ATmega has 5 different sleep modes. For the most aggressive one, Power-Down, it is only possible to wakeup from an external interrupt (either INTO, INT1 or a pin change), the Two-Wire interface, or the WDT. The sharewear firmware puts the ATmega in Power-Down mode if no switches are active, e.g. no dress interaction is happening. To prevent the WDT waking up the ATmega the WDT is disabled before entering the sleep mode. However, the sleep mode is entered from within the WDT interrupt service routine, so interrupts must be explicitly be re-enabled before going to sleep. The RFID module is held in reset during sleep, preventing interrupts being generated on the INT[01] pins. Activating one or both of the switches wakes up the ATmega, restoring the SREG register (reset the interrupt status) and re-enabling the WDT. The RFID module is taken out of reset to check if an RFID tag appeared during sleep.

Because the time before the discovery of an RFID tag can take up to a second, the lights switch to the default pattern and switch to a new pattern as soon as an RFID tag is found. The default functionality can be changed by #defining **WAIT_FOR_RFID** while compiling. This starts the animation only after an RFID search has taken place and an RFID tag has been found or not.

The following modules have been disabled to reduce power consumption even more:

- TWI (Two Wire Interface)
- SPI (Serial Peripheral Interface)
- ADC (Analog to Digital Converter)
- USART (except when debugging is enabled)

Light animation editor

To easily create light patterns the sharewear package comes with a pattern editor. This is a simple dedicated curve editor with which it is possible to edit six patterns plus a default pattern, each up to six seconds. The patterns can be assembled using bezier curves and/or line segments. The pattern editor is written using processing/java and runs as a standalone application for Linux, OSX (ppc and intel) and windows. Only the OSX implementation is currently in use and tested.

The editor has three buttons

- save
- export
- upload

Save

This saves the curves to **data/curves.dat**. On re-launch of the editor the last saved curves will be opened.

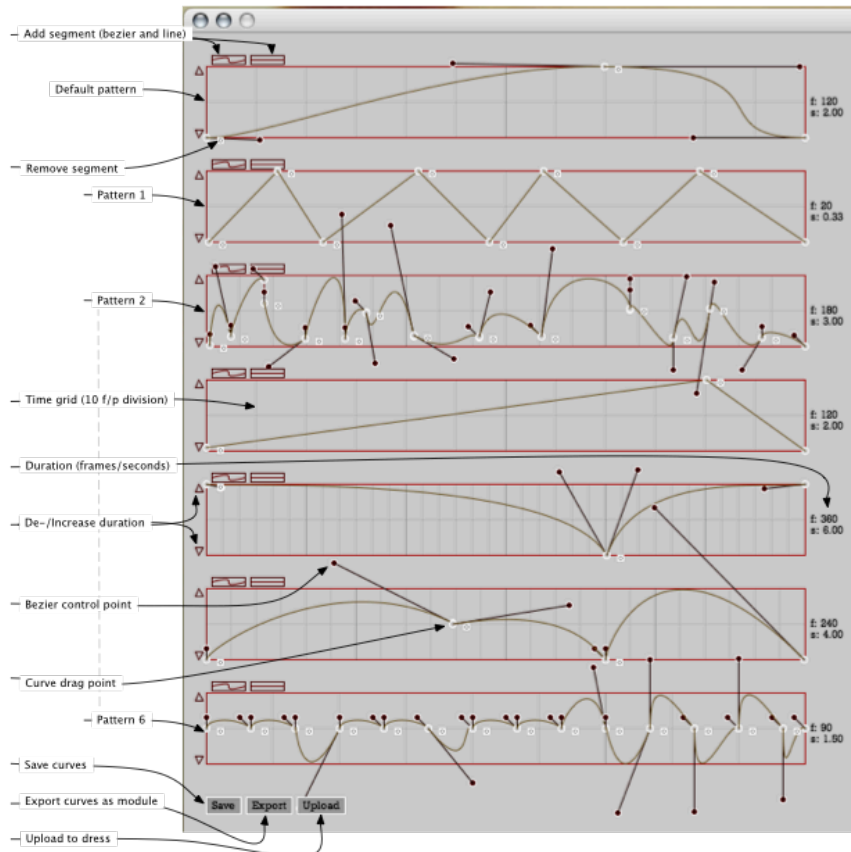
Export

The export button exports the curves to a readable format, one file per curve. These curves are placed in the data directory and are named **anim_mmodule[0-6]_data.dat**. Before these curves can be used by the sharewear firmware, they need to be converted to loadable modules. This is a 2 step process explained in The editor does these steps before uploading the firmware.

Upload

To upload the curves: press this button. The button will start up a Terminal and converts the exported curves. After conversion, a new firmware hex file is compiled and uploaded to the ATmega over a serial line. Edit [make.conf](#) to change the default programmer and/or serial port.

Before uploading the curves, export them, otherwise the previously exported curves will be used.
Starting the terminal can take a while on some computers, be patient.



Attachments

- [usbtiny_avr_s.jpg](#) (83.7 kB) -"Usbtiny ISP & Arduino Diecimila", added by simon on 2008-03-05 14:24:13.
- [Screenshot.png](#) (85.0 kB) -"softserial timings", added by simon on 2008-03-07 16:23:24.
- [editor_graph.png](#) (142.4 kB) -"Animation pattern editor", added by simon on 2008-05-08 12:22:39.