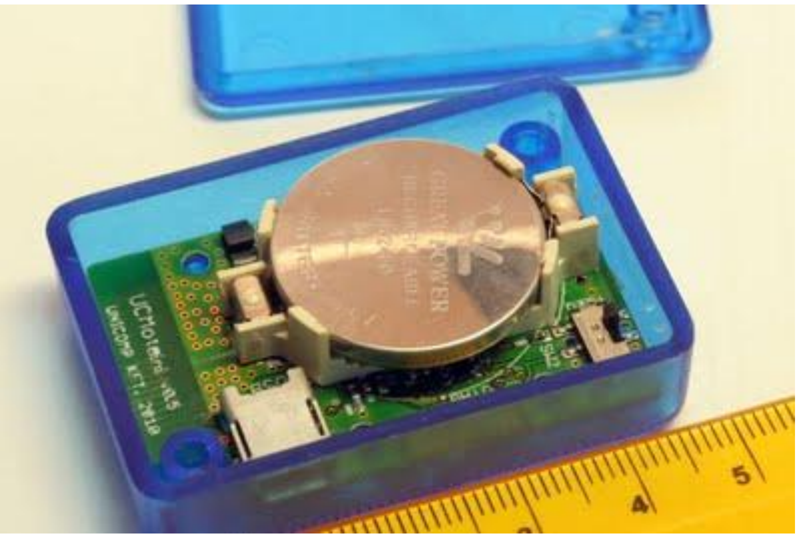
# User Manual

## UCMote **mini**



# Table of Content

## Datasheet

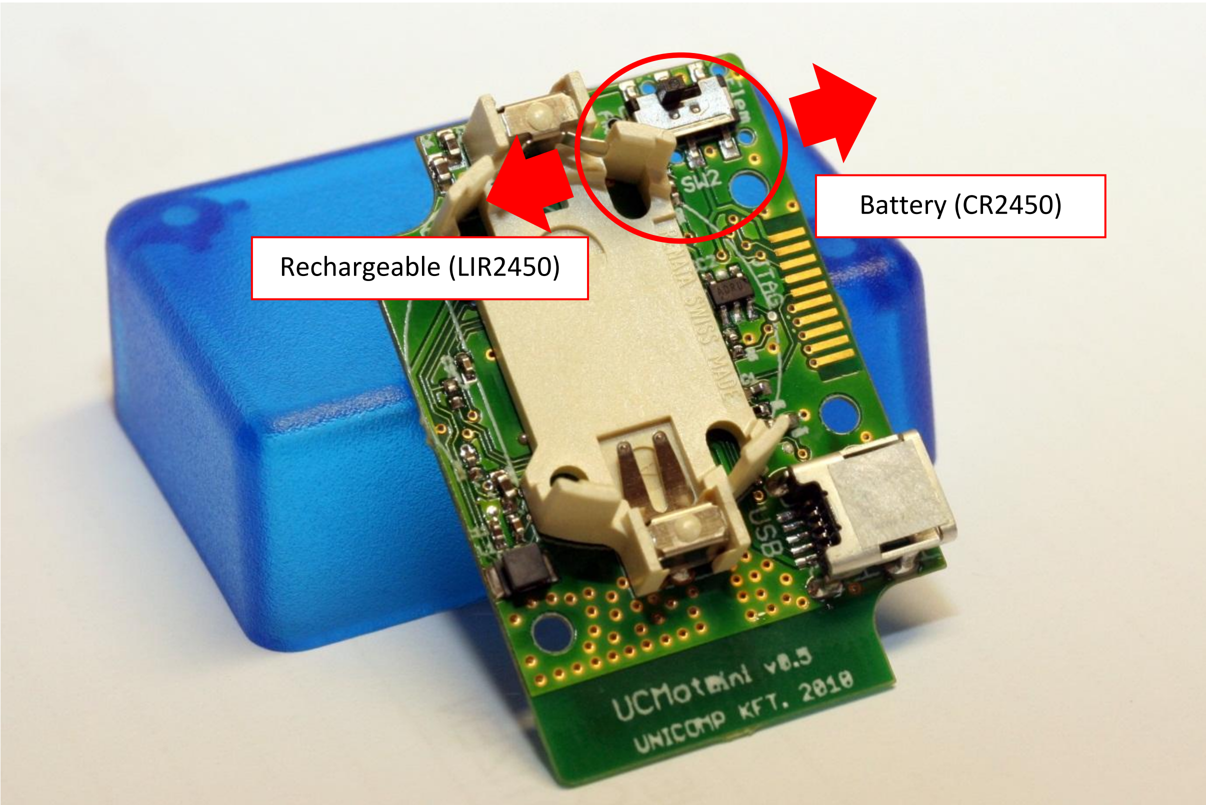
| | |
|---|---|
| MCU: | Atmel ATmega128RFA1, 8-bit, radio 2,4GHz |
| Antenna: | Chip antenna |
| Flash: | 16Mbit, SPI |
| Sensors: | |
| *Ambient light* | 16bit, $I^2C$ |
| **Humidity, Temp* | 12bit/14bit, $I^2C$ |
| **3D accelerometer* | 14bit, SPI |
| **Barometric sensor* | 24bit, $I^2C$ |
| JTAG: | for debugging and communication |
| micro USB: | for external power and communication |
| User interface: | 4 LEDs (+2 LEDs for external power presence and charging indicator) |
| Battery: | Both CR2450 non-rechargeable and LIR2450 rechargeable coin battery |
| Battery charger: | charging LIR2450 battery in case of present external power |
| Software: | TinyOS 2.x and NesC compatibility |
| Enclosure: | Hammond 1551F, outside dimensions: 50mm X 35mm X 15mm |

## ATTENTION!

For the error free operation high temprature (above 85° Celsius) and the direct contact with water should be avoided! The temperature affects the battery capacity.

Besides USB power, **mini** can be operated from both rechargeable (LIR2450) and non-rechargeable (CR2450) coin battery. Since the voltage level of the two types of battery differs. (CR2450 ~3V, LIR2450 ~4,2-3,6V) and in *battery* mode there is no regulator between the power source and the MCU:

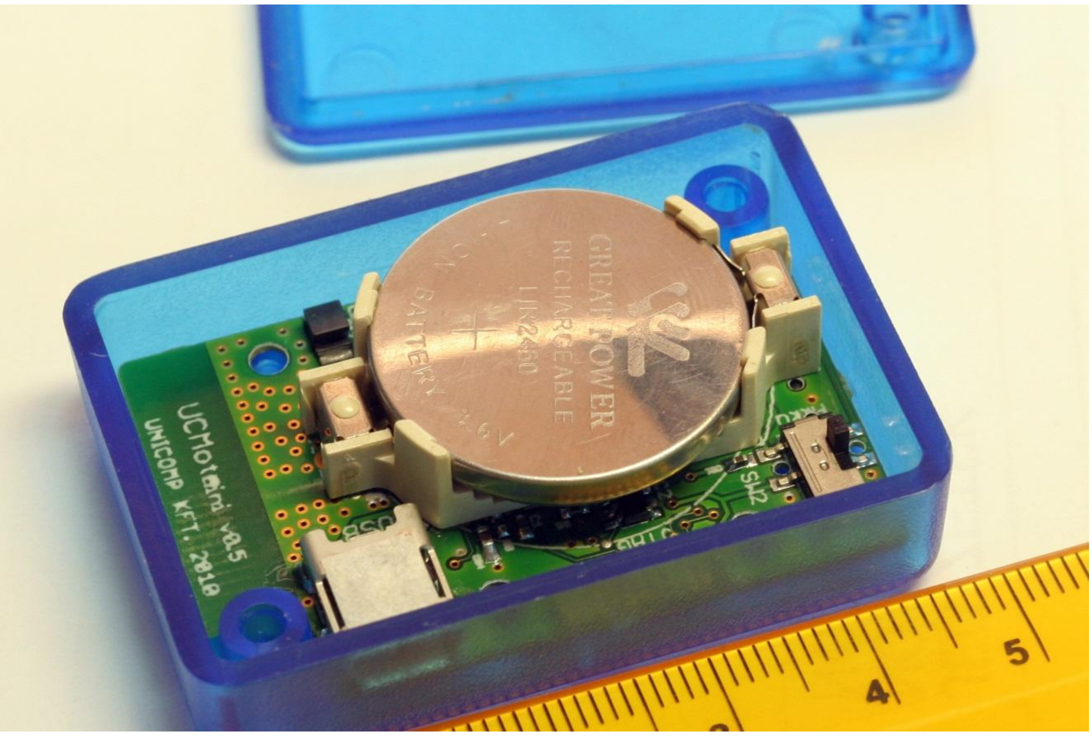
It is important to check the power source type switch, before inserting the power source (accumulator or battery), since the incorrect selection can cause damage!




Picture 1: Battery type selection

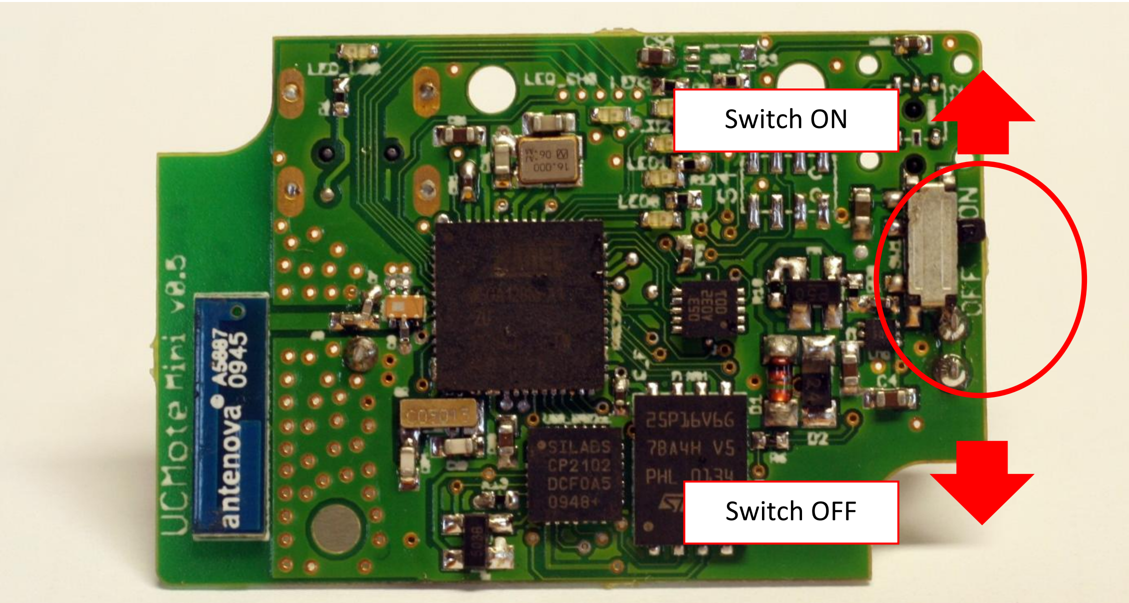
## First steps

The battery should be inserted with the (+) side up. Incorrect polarity can damage the device.




Picture 2: Correct battery placement

The device can be turned ON or OFF by the switch circled on the photo below.



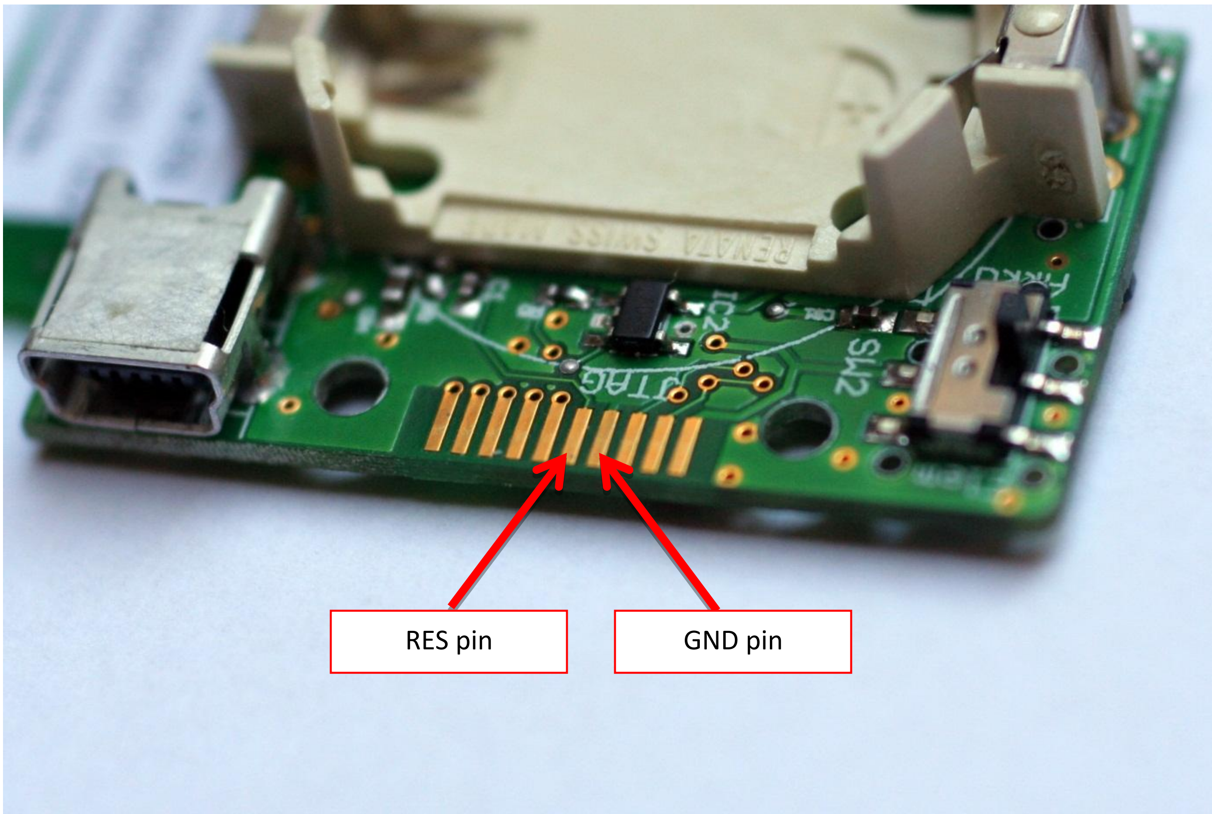

Picture 3: Switching ON/OFF mini

## USB power

When USB power presents, **mini** automatically switches to USB power (it's not necessary to turn the device off)

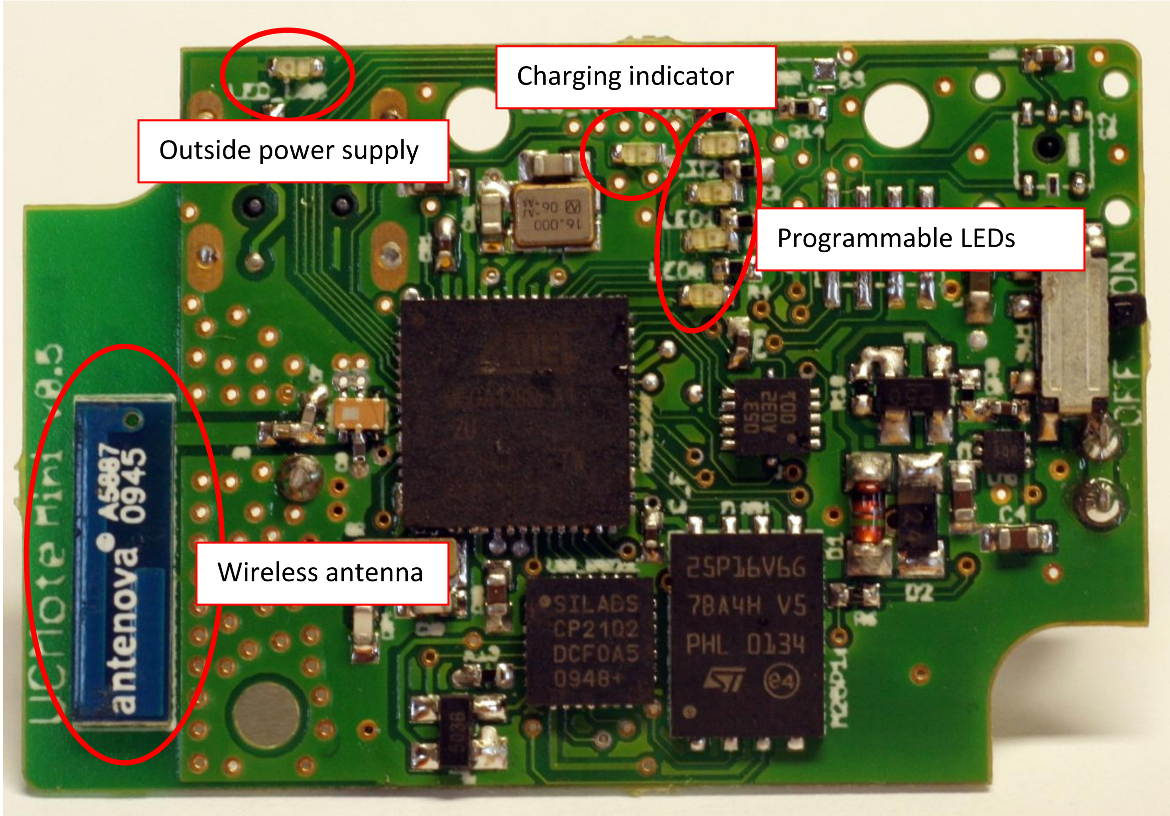
When you are using rechargeable battery, the device starts to recharge it (charging indicator LED lights up [Picture 5]). When the charging has been completed, the indicator blacks out. During the charging process, the temperature of the battery should not go over 50° Celsius. If you observe overheating, or any physical deformation on the battery, immediately pull out the USB cable, because the exploding battery can cause serious injury or fire. If the output voltage of the battery falls below 3.5V, it should be recharged, because the low voltage reduces the battery life. The fully charged battery produces 4.2V output voltage.

In case you need reset condition, use the on/off switch, or use a conductive tool to make contact between the ground GND) and reset (RST) pins on the JTAG connector surface.




Picture 4: Reseting mini

## Periferies




Picture 5. LEDs and the antenna

## Programming the mote.

The UCMini devices can be programmed through two interfaces.

- Direct programming by JTAG interface (optional accessory needed)
- Bootloader programming by USB interface

The microcontroller unit (MCU) of the device shipped with a pre-programmed Bootloader firmware, which makes possible to program the device simply via an USB connection. After startup, the Bootloader waits a few seconds for the connection of the PC side bootloader handling software. If the Bootloader finishes waiting for PC connection (USB), it switches ON all leds two times, and starts the main application.

## USB programming (Linux)

To program the **mini** via USB cable, you have to follow these steps:

- Compile your application to SREC
- CD to the folder of your SREC
- Connect **mini** to the PC with the USB cable

- Execute following command within 10 secunds:

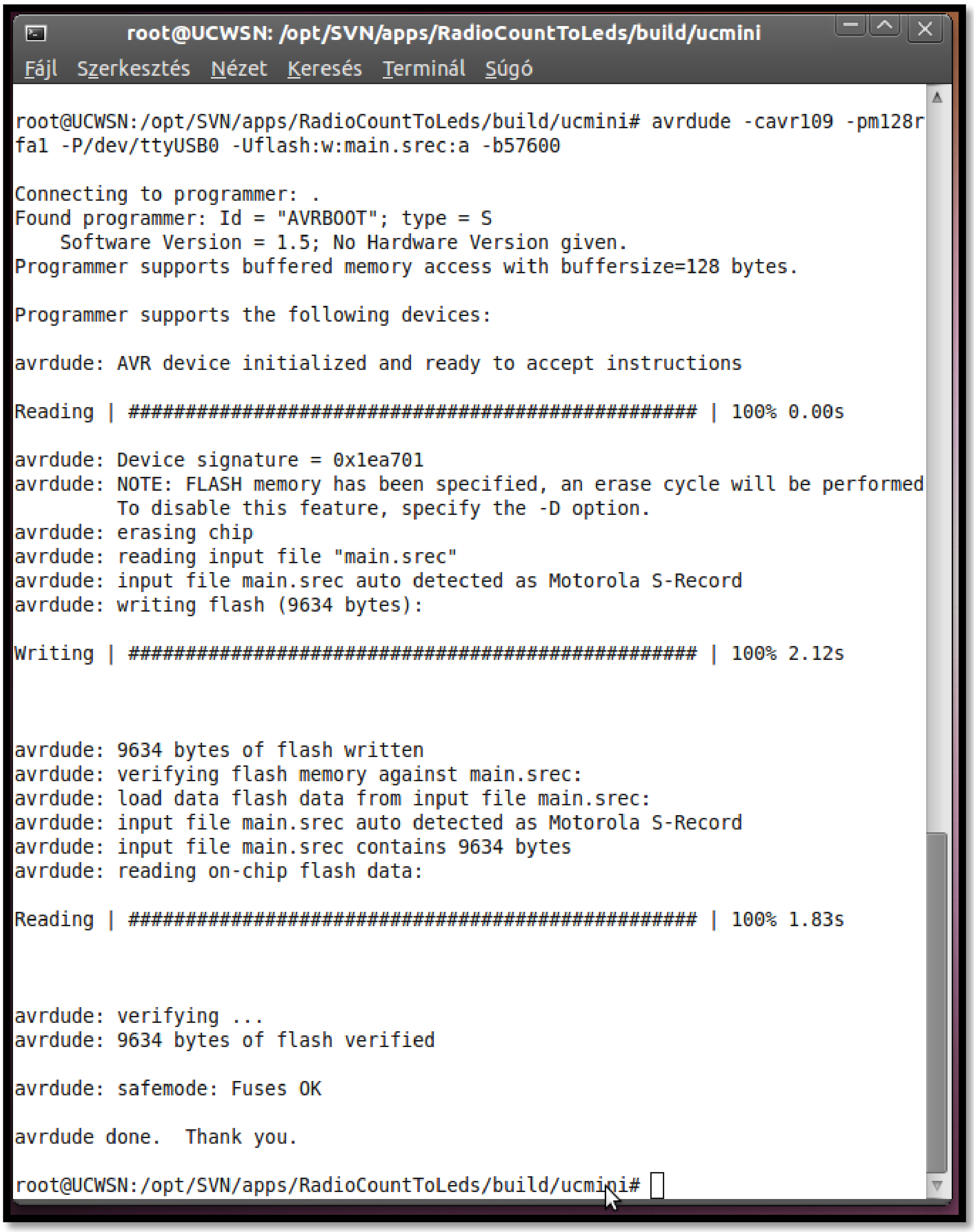
  *avrdude -cavr109 -pm128rfa1 -P/dev/ttyUSB0 -Uflash:w:main.srec:a -b57600*

  where the ttyUSB0 must be overwritten to the correct tty of your connected device.
- The AVRDUDE writes to the console its success [Picture 7] or fail.
- If the Bootloader timed out [Picture 6], **mini** must be restarted and try again to execute the command above.


Picture 6: Bootloader timed out

The „not responding" error message means, that **mini** did not answer its programming request. There are two common reasons can be in this case:

- the Bootloader timed out
- or incorrect tty given to the AVRDUDE.




Picture 7: AVRDUDE normal execution

## JTAG programming (AVR Studio)

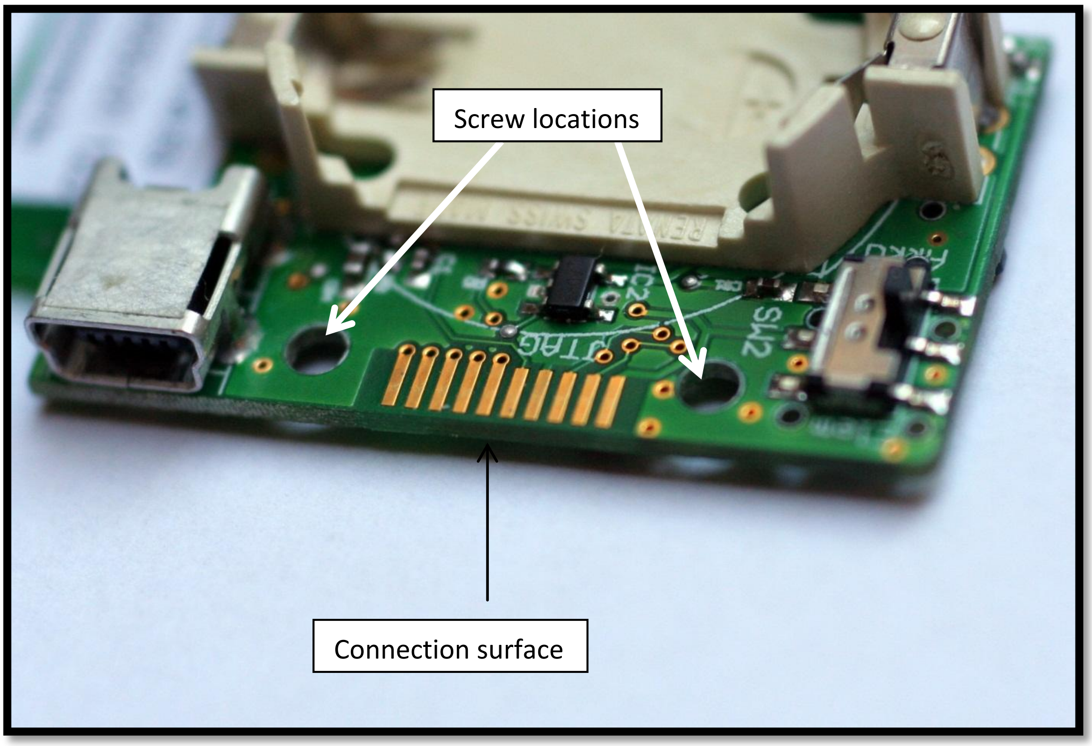
The direct programming is possible only if there is an AVR programming board, like AVR Dragon and you need the JTAG connector (https://sites.google.com/a/unicomp.hu/ucmote/kiegeszitok)

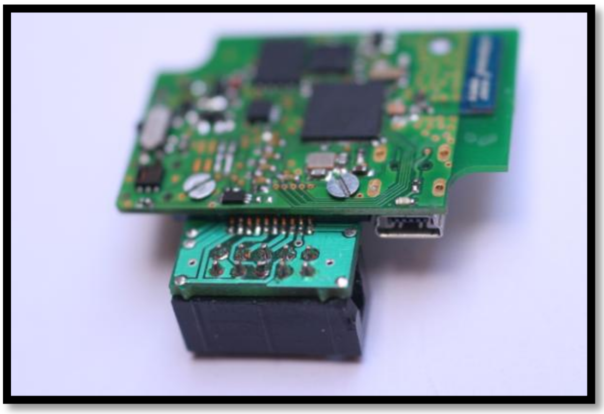
CAUTION! The direct programming overwrites the Bootloader section of the program memory, after that no way to program the device via USB cable. (the Bootloader firmware can be uploaded by JTAG and AVR Dragon, but it requests special configuration, to place it correctly to the bootloader section of the MCU-s program memory)

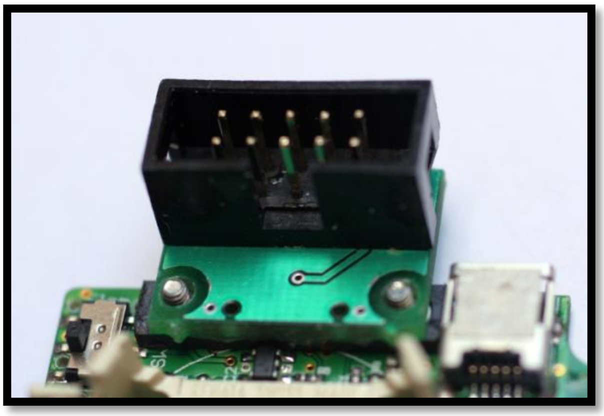
The direct programming makes debugging and the usage of AVR Studio development environment possible .

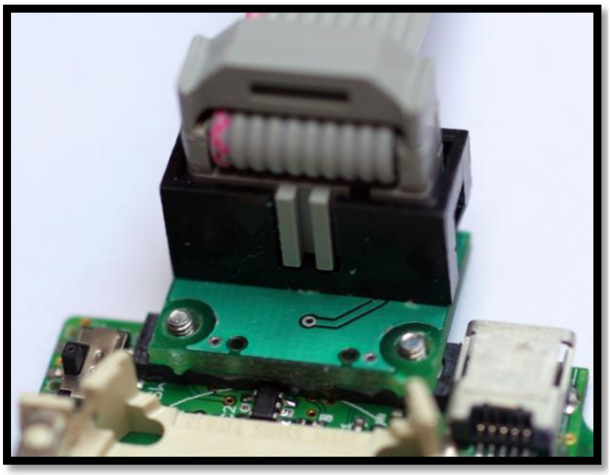
For direct programming, you should follow these steps:

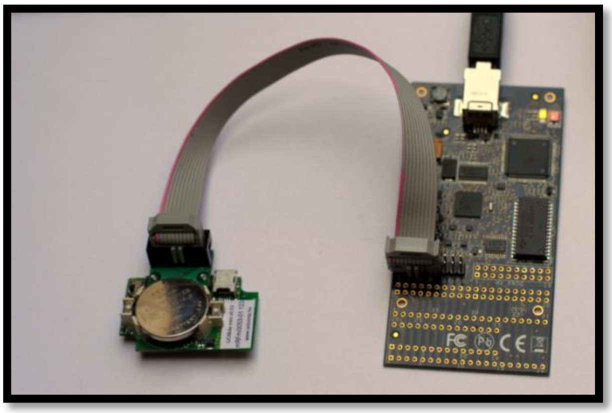
- Connect the AVR Dragon (or other compatible JTAG programmer)
- Run AVR Studio
- Plug in the screws from the opposite side of the battery holder ( only countersunk screws can be used)
- Contact the PCB connect the PCB connector to the JTAG labeled surface (with gold-plated pins)of the device
- Carefully screw the screws
- Connect the JTAG cable to **mini**, and the AVR Dragon.



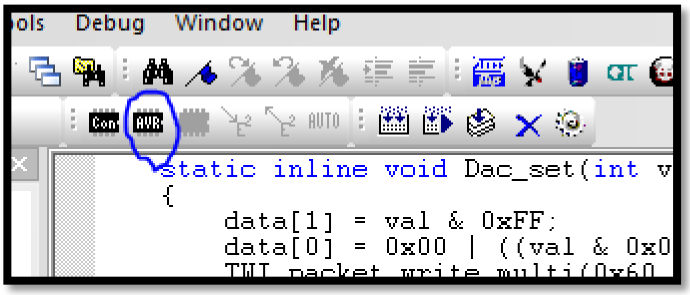




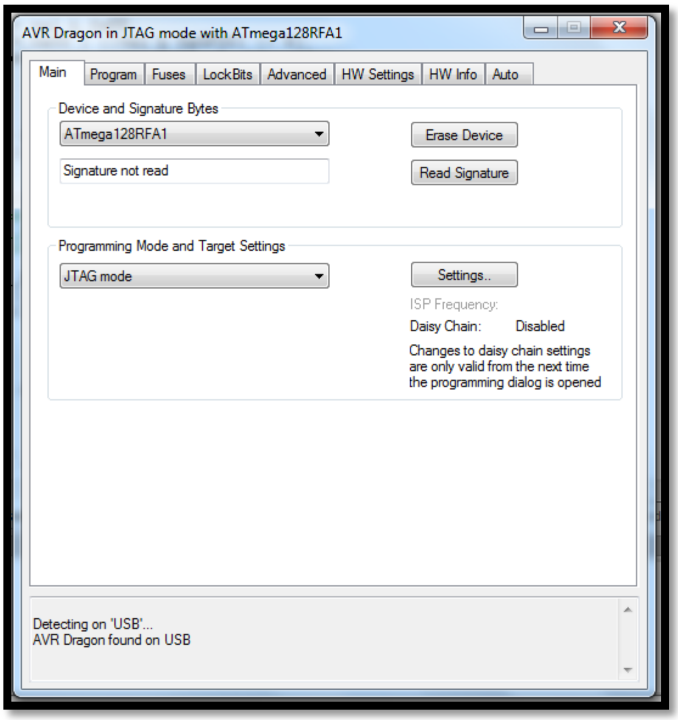




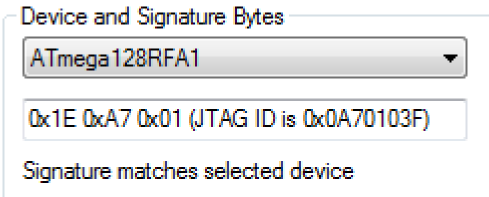
After connecting the cables, the connection can be tested. Click the AVR button on the toolbar.




A window pops up, where the programming mode JTAG connection can be handled. Check if the correct device is selected (ATmega128RFA1) and the JTAG mode is active at the Programming mode combo box.




The read signature button reads the device signature, which must be like this:




At the Program tab we can write the AVR Studio output files to the device, the program code (.hex), the EEPROM content (.eep) and the .elf files.

At the fuses tab we can give the fuse bits, they are the configuration settings of the MCU.

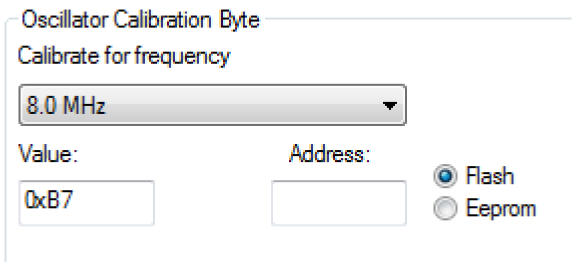
- Brown-out detection
- On-Chip debug
- JTAG program
- SPI program
- Watchdog timer
- Preserve EEPROM
- Boot size
- Boot reset
- Clock divider
- Clock out
- Clock source

DO NOT change these settings, if you do not know exactly what you are doing. Incorrect settings can damage the device, or make it totally unusable. The default settings with the devices are shipped: 0xFF 0x98 0xE2

The *LockBits* tab manages the lock of the memory spaces, the change of these settings can make **mini** unusable too, be careful.

The *Advanced* tab can be used to calibrate the internal RC oscillator of the MCU. The calibration for 8Mhz descibed int he followings:

Measure the speed of the MCU by pressing the Read button. The AVR Studio measures the speed, and offers a calibration value which can be written to the OSCCAL register, to exactly set the frequency of the oscillator.




The OSCCAL register can tune the RC oscillator to run faster or slower. The value can be given to the register with the following simple expression:

```
OSCCAL = 0xB7;
```

The AVR Studio can write this value to a given address of the EEPROM, to manage this setting from the firmware with same code. (The calibration value usually differ at devices)

# Sensors

Sensors can be accessed via SPI and $I^2C$ interfaces.

The $I^2C$ is an address based master-slave multi access bus implementation, where the master (MCU) starts the transmission, and it manages which slave device may use the bus, by „calling" them with the address.

The SPI bus doesn't carry device addresses, the channel access are controlled by chip select pins. The MCU can select a device by pulling its chip select pin low. Only one device may be selected at once.

## SHT21 – Temperature and Humidity sensor


The measurable data

- Temperature
- Humidity

Interface: I2C (Address: 0x40)

The most important command bytes

```
#define SHT_TEMP_HOLD           0xE3
#define SHT_HUMID_HOLD          0xE5
#define SHT_TEMP                0xF3
#define SHT_HUMID               0xF5
#define SHT_W_REG               0xE6
#define SHT_R_REG               0xE7
#define SHT_RESET               0xFE
```

Power supply: PORTF 1

The measurement process

- Power supply on (pull up PORTF 1)
- Sending the desired measure command byte
- Collect the data from the sensor

### Example code:

The TWI (the Atmel calls $I^2C$ as Two Wire Interface, TWI) functions must be implemented.

```
char SHT_MeasureTemp(unsigned char *data)
{
    if(TWI_packet_write(SHT_ADDRESS, SHT_TEMP) == TWI_FAIL) return TWI_FAIL;
    _delay_ms(100);
    if(TWI_packet_read(SHT_ADDRESS, 3, data) == TWI_FAIL) return TWI_FAIL;

    return TWI_SUCCESS;
}
```

The received data must to be converted with the manufacturer given coefficients.

$$T = -46.85 + 175.72 \cdot \frac{S_T}{2^{16}}$$

Example code for the conversion:

```
int32_t SHT_MeasureTempC()
{
    unsigned char rxBuf[3] = {0,0,0};
    uint16_t adc = 0;
    int32_t ret = 0;

    if(SHT_MeasureTemp(&rxBuf[0]) == TWI_SUCCESS)
    {
        adc = (rxBuf[0] << 8) | (rxBuf[1]);

        ret = (17572 * (int32_t)adc)/65535;
        ret -= 4685;
    }

    return ret;
}
```

The result is a fixed point signed integer (the temperature in Celsius), with two decimals. (The coefficients were multiplied by 100, to avoid using floating point arithmetic in an MCU, because its slow and hardly manageable at byte level transfer)

## BH1750fvi – Ambient light sensor

Measureable data

- The visible light quantity

Interface: $I^2C$ (Address: 0x23)

The most important command bytes

```
#define BH_POWER_DOWN           0x00
#define BH_POWER_ON             0x01
#define BH_RESET                0x07
#define BH_CONT_H_RES           0x10
#define BH_CONT_H2_RES          0x11
#define BH_CONT_L_RES           0x13
#define BH_ONE_SHOT_H_RES       0x20
#define BH_ONE_SHOT_H2_RES      0x21
#define BH_ONE_SHOT_L_RES       0x23
```

Power supply: PORTF 1

The measurement process

- Power supply on (pull up PORTF 1)
- Sending the turn on command byte
- Sending the desired measure command byte
- Collect the data from the sensor

### Example code:

```
char BH_Measure(unsigned char mode, unsigned char *data)
{
    if(BH_On() == TWI_FAIL) return TWI_FAIL;

    if(TWI_packet_write(BH_ADDRESS, mode) == TWI_FAIL) return TWI_FAIL;
    _delay_ms(BH_TIMEOUT_H_RES);
    if(TWI_packet_read(BH_ADDRESS, 2, data) == TWI_FAIL) return TWI_FAIL;

    //if(BH_Off() == TWI_FAIL) return TWI_FAIL;

    return TWI_SUCCESS;
}
```

Converting

```
int32_t BH_MeasureC()
{
    int32_t ret = 0;
    uint8_t rxBuf[2] = {0,0};

    if(BH_Measure(BH_ONE_SHOT_H_RES, &rxBuf[0]) == TWI_SUCCESS)
    {
        ret += rxBuf[0] << 8;
        ret += rxBuf[1];

        ret *= 83;
    }

    return ret;
}
```

The result value is a fixed point integer with two decimals, the value is the measured light quantity expressed in Lux.

## BMA180 – 3D Acceleration sensor


The measureable data

- Acceleration (X,Y and Z axes)
- Temperature

Interface: SPI (USART0) (Chip Select: PORTB 6)

The most important command bytes (in fact they are register and EEPROM addresses, which controls the sensor's internal logic)

```
#define BMA_CHIP_ID         0x00
#define BMA_VERSION         0x01
#define BMA_ACC_X_LSB       0x02
#define BMA_ACC_X_MSB       0x03
#define BMA_ACC_Y_LSB       0x04
#define BMA_ACC_Y_MSB       0x05
#define BMA_ACC_Z_LSB       0x06
#define BMA_ACC_Z_MSB       0x07
#define BMA_TEMP            0x08
```

Power supply: PORTF 0

The measurement process

- Power supply on (pull up PORTF 0)
- Select device (pull down chip select. The device needs this falling edge to operate correctly, so it must be driven to high before operate the sensor)
- Read the data registers
- Deselect device (pull up chip select)

The sensor is the most complex of all available sensors on **mini**, the operation of the sensor needs to read the manufacturer provided datasheet.

CAUTION: DO NOT WRITE the address space from 0x3B, because writing to reserved registers, or overwriting the eeprom image can damage the sensor logic, make it fully unusable. The address space before 0x3B is volatile, at boot the content of them are written from eeprom image space.

The sensor is connected to the USART0 interface of the MCU, which can be operate in SPI mode.

### Example code:

Reading registers:

```
uint8_t BMA_GetReg(uint8_t address)
{
    uint8_t txBuf[2] = {address | 0x80, 0};
    uint8_t rxBuf[2] = {0,0};

    BMA_Select();
    SPIUS0_command(&txBuf[0], &rxBuf[0], 2);
    BMA_Deselect();

    return rxBuf[1];
}
```

Writing registers:

```
void BMA_SetReg(uint8_t address, uint8_t data)
{
    uint8_t txBuf[2] = {address & 0x80, data};
    uint8_t rxBuf[2] = {0,0};

    BMA_Select();
    SPIUS0_command(&txBuf[0], &rxBuf[0], 2);
    BMA_Deselect();
}
```

## M25P16 – 16Mbit flash memory


Interface: SPI (Chip Select: PORTB 4)

The most important command bytes:

```
#define M25_WREN        0x06
#define M25_WRDI        0x04
#define M25_RDID        0x9F
#define M25_RDSR        0x05
#define M25_WRSR        0x01
#define M25_READ        0x03
#define M25_FAST_READ   0x06
#define M25_PP          0x02
#define M25_SE          0xD8
#define M25_BE          0xC7
#define M25_DP          0xB9
#define M25_RES         0xAB
```

### Example code:

Initializing:

```
void M25_Init()
{
    unsigned char sr = 0;

    CS_DDR |= _BV(CS_PIN);
    CS_PORT |= _BV(CS_PIN);

    _delay_ms(100);

    SPI_MasterInit();

    M25_WriteEnable();
    M25_ReadStatus(&sr);

    while(sr != 0x02)
    {
        sr = 0x02;
        M25_WriteStatus(sr);
        _delay_ms(1);
        M25_ReadStatus(&sr);
    }

}
```

Writing with page program function:

```
void M25_PageProgram(unsigned long address, unsigned char *buffer, unsigned char len)
{
    unsigned char i = 0;
    unsigned char j = 0;
    unsigned char txBuf[4 + len];

    txBuf[0] = M25_PP;
    txBuf[1] = (unsigned char)((address & 0xFF0000) >> 16);
    txBuf[2] = (unsigned char)((address & 0x00FF00) >> 8);
    txBuf[3] = (unsigned char)(address & 0x0000FF);

    for(i = 4; ((i < len + 4) && (i < 256)); i++)
    {
        txBuf[i] = *(buffer + j);
        j++;
    }

    M25_Select();
    SPI_command(&txBuf[0], buffer, len+4);
    M25_DeSelect();
}
```

Reading:

```
void M25_Read(unsigned long address, unsigned char *buffer, unsigned char len)
{
    unsigned char i;
    unsigned char txBuf[4 + len];

    txBuf[0] = M25_READ;
    txBuf[1] = (unsigned char)((address & 0xFF0000) >> 16);
    txBuf[2] = (unsigned char)((address & 0x00FF00) >> 8);
    txBuf[3] = (unsigned char)(address & 0x0000FF);

    for(i = 4; ((i < len + 4) && (i < 256)); i++)
    {
        txBuf[i] = 0;
    }

    M25_Select();
    SPI_command(&txBuf[0], buffer, len+4);
    M25_DeSelect();

}
```