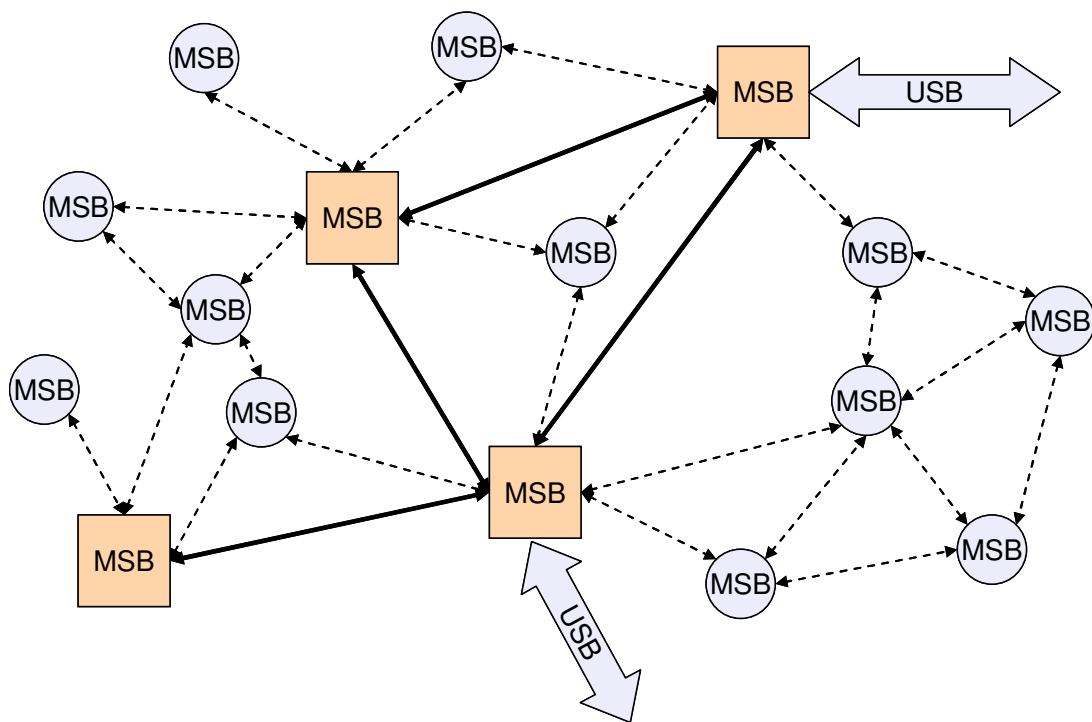


First Steps (for ScatterWeb²)

A platform for teaching & prototyping
wireless sensor networks

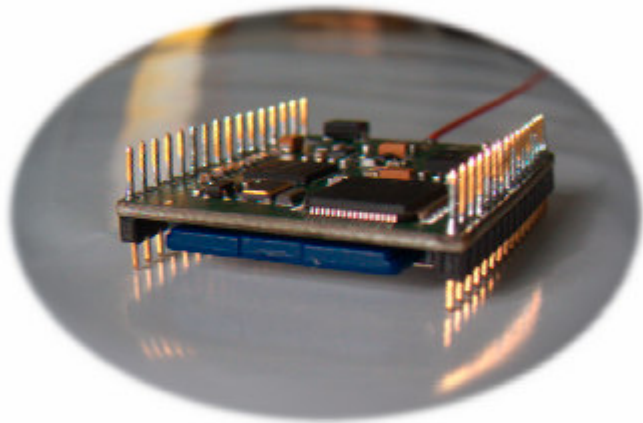


scatterweb.mi.fu-berlin.de

Computer Systems & Telematics
Freie Universität Berlin, Germany

Content

1	Introduction	5
2	Hardware	7
2.1	Modular Architecture	8
2.2	Core Module	9
2.3	Radio	9
2.4	Sensors	9
2.5	I/O Interfaces	10
3	Software	11
3.1	System	11
3.2	Application	12
4	Lessons	13
4.1	Switching LED on and off	13
4.2	Toggling on command	13
4.3	Using the Timer	14
4.4	Sending and receiving packets	15
4.5	Now it is your task	18



1 Introduction

This technical report provides an introduction into the first steps of working with the ScatterWeb sensor nodes. Sensor networks are currently a very intensive area of research, as they combine different requirements, to mention just a few:

- Scalability up to networks of thousand nodes
- Energy-efficiency for each single node
- Self-configuration, zero configuration, of all nodes in the network
- Graceful degradation in the case of single node failures

While these tasks still partly remain to be improved, especially for higher scalability, there are already many deployment scenarios where sensor network deployment is becoming reality soon:

- Home automation and surveillance
- Factory automation, process monitoring
- Disaster recovery, fire rescue
- Construction integrity surveillance
- Interactive environments

Research in wireless sensor networks needs a robust hardware platform that allows access to hardware parameters to achieve optimum solutions. In this document we present the new ScatterWeb MSB hardware platform which we believe soundly fits the needs of research and prototyping applications of the near future.

Nevertheless, main focus of this report is not the discussion of these scenarios and open research topics but to provide a small howto on developing own projects with the ScatterWeb nodes developed at the Freie Universität Berlin. The technical report provides a snapshot of ongoing work. Up-to-date information can be found at the project homepage <http://scatterweb.mi.fu-berlin.de>.

2 Hardware

The hardware provided as part of the ScatterWeb sensor network platform comes in two parts: The nodes and the gateways. This chapter describes the hardware and gives some hints about the installation of the software needed.

There are different ScatterWeb nodes, some more specialized and targeted for example only to the task of temperature monitoring, others universal platforms for testing and rapid prototyping. The nodes described here are referred to as MSB, modular sensor boards. Figure 1 gives an example of such a modular sensor board.

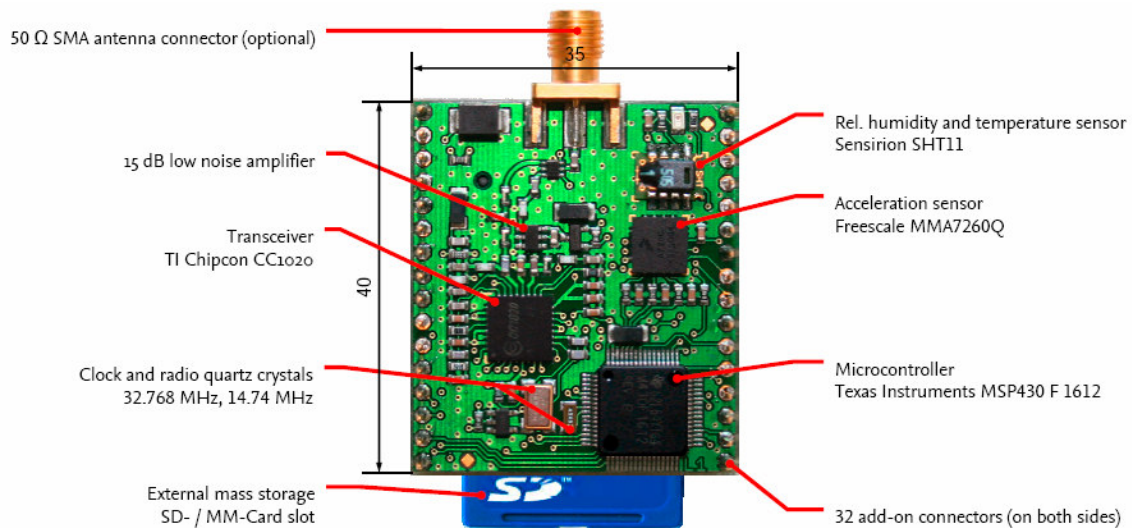


Figure 1: Modular Sensor Board (MSB)

Research in wireless sensor networks is currently focused on algorithms for data dissemination, self-configuration and energy management. For any real-world application, a robust hardware platform is required, that allows gearing into all important hardware parameters to achieve optimum results in software applications. For this reason we created the ScatterWeb hardware platform, called the “embedded sensor board” (ESB) in 2003. The success of the platform led to the spin-off of ScatterWeb GmbH in 2005. Based on the know-how and experiences gained, an industrial platform was designed and certified for radio spectrum and electromagnetic compatibility.

Since demands are manifold and rapidly changing, a new research platform, the “modular sensor board” (MSB) has been created. While still based on an MSP430 series microcontroller, the layout and peripherals have been

completely redesigned to better fit research needs of the near future. The new platform is available to the public since the beginning of 2007 through the ScatterWeb GmbH. In the following sections features and design aspects of the MSB platform will be presented.

2.1 Modular Architecture

The ESB node included a number of common sensors and some actuators which were thought of being useful for teaching and research but made it quite heavy. In addition to that, an USB gateway and a LAN gateway were built sharing the same core components but different peripherals. This platform was well suited for teaching and research at Freie Universität Berlin and other groups within the past years, but progress in technology and new projects with divers requirements for sensors, connectivity and energy sources demanded a new platform. It currently comprises a core module, shown in Figure 2, which is only 13.5 cm². Add-on boards can be attached sandwich-like to both sides. These usually would be a carrier module providing power and I/O connectivity and, if needed, a sensor board.

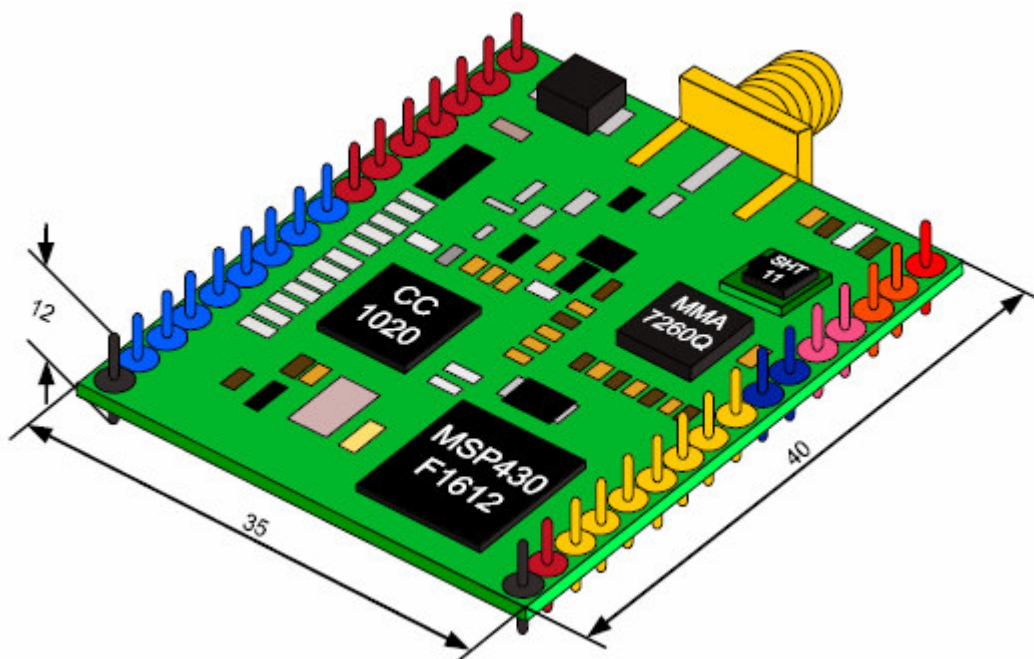


Figure 2: Scheme of a MSB

2.2 Core Module

The core module is a complete sensor node, consisting of microcontroller (MCU), radio, external storage and two sensors. Power is provided externally through a carrier board. For maximum compatibility and because of positive experience, again an MSP430x1xx-series MCU was chosen. Since the 2 KB of RAM available on the ESB, were a very tight limitation, the MSP430F1612 with 5 KB was selected.

This MCU offers 60 KB of memory divided into 5 KB RAM and 55 KB Flash-ROM. It is clocked by a digital controlled oscillator (DCO) which can be configured from software between 100 kHz and 8 MHz. For synchronization the external 32.768 kHz quartz is used. New features over the ESB's MCU are a DMA-controller, I2C-bus support and two digital-analog-converters (DAC). Instead of EEPROM an SD-/MM-card slot is included for secondary storage of up to 4 GB. It is connected to an UART and accessed using the SPI protocol.

2.3 Radio

The MSB-430 has a new radio design using the licensefree 868 MHz ISM band. It has been designed for much larger coverage than the ESB nodes and provides up to 8.6 dBm (= 7.2 mW) output power instead of 0 dBm. A Chipcon CC1020 transceiver is used in combination with an additional low-noise amplifier on the receiver. With appropriate configuration ESB compatibility is possible. The radio frequency can be selected separately for receiver and transmitter by software. This allows usage of multiple radio channels for advanced routing schemes. Transmission power can also be adjusted to reduce power consumption. While the maximum raw bit rate is 153.6 kbit/s the board is optimized for channel conformity which allows a typical data rate of 19.2 kbit/s when using Manchester encoding. The CC1020 also includes a digital received signal strength indicator (RSSI) which is used for carrier detection and can be used for simple ranging and quality measurements. Radio emissions have been measured between 9 kHz and 4 GHz operating at 869.525 MHz with 8.6 dBm and 19.2 kbit/s.

2.4 Sensors

Despite the modular concept of the MSB series the core board is equipped with two sensors. For temperature compensation, that is essential for accurate clock frequencies and measurements, the temperature and relative humidity sensor Sensirion SHT11 is used. It provides calibrated digital output with good

accuracy at room temperature. The second sensor is the micro machined accelerometer MMA7260Q from Freescale. It measures acceleration in three axes with 1.5 to 6 g sensitivity and can be used in movement detection and to support positioning. Since it is used in many of our research applications, it is mounted on the core module to reduce overall production costs.

For teaching use a general purpose sensor board (MSB-430-S) has been designed. It offers enhanced sensing capabilities to that of the ESB nodes. A serial EEPROM is included for identification or driver code.

2.5 I/O Interfaces

All I/O ports that are available for custom additions are externally accessible. For analog in- and output two 12-Bit DAC and ADC can be used. 18 digital I/O pins (8 with IRQ support) are freely disposable. One UART is available for serial communication. For programming and debugging the JTAG interface is used. The standard carrier board MSB-430-T together with a FTDI converter-cable offers USB connectivity and a JTAG connector. It powers the core board via USB or from three AAA batteries.

Apart from programming the sensor boards from scratch embedded operating systems provide libraries for the board's hardware and configuration as well as high-level functionality. All ScatterWeb research platforms are supported by the open-source ScatterWeb operating system, which includes sample code for all the peripherals. The sources are compatible to the GNU compiler for the MSP430. Alternatively ScatterWeb nodes can be programmed graphically without writing a single line of C code using our QMSB software. Our ScatterWeb SDK for Microsoft .NET and a driver for National Instrument's LabVIEW provide convenient abstractions for using ScatterWeb sensor networks. The open-source operating system Contiki, featuring protothreads and the μ P-stack was ported to both the ESB and MSB platforms.

3 Software

First of all, the software is divided into two parts:

- The system
- The so-called applications

While the system handles all interrupts, packet sending and receiving and access to the hardware, the application resembles a user level application in full-featured operating systems. The application code is strongly dependent on the tasks that you want to realize while the firmware should be kept as it is.

3.1 System

The system first has to initialize the hardware (setting port directions, baud rates etc.). Then it enters an infinite loop:

```
...\ScatterWeb.System.h:
interrupt(NOVECTOR) superloop() {
    [...]
    if(runModule & MF_TIMER) Timers_eventHandler();
    if(runModule & MF_RADIO_RX) Net_rxHandler();
    if(runModule & MF_RADIO_TX) Net_txHandler();
    if(runModule & MF_SERIAL) {
        extern volatile UINT8* serial_line;
        if(serial_line != 0) {
            if(callbacks[C_SERIAL]) callbacks[C_SERIAL]((void*)serial_line);
        }
    }
    if(runModule & MF_SENSOR) {
        if(callbacks[C_SENSOR])
            callbacks[C_SENSOR](&Data_sensorEventValue);
        runModule &= ~MF_SENSOR;
    }
    [...]
}
```

As can be seen in this code snippet, the system software continuously checks whether sensor events were noticed, whether data arrived at the serial port and data in the outgoing buffers has to be handled. But if you look closer at it you will notice that all these events and actions are only handled if the respective flag in the RunModule is set.

The interaction between system and applications is realized via callbacks, as explained in the next section.

3.2 Application

The application is responsible for registering callback functions with the system if the application is interested in:

- Sensor events
- Incoming packets
- Being called at regular intervals

The application therefore registers its own handler functions as callbacks with the system during the initialization phase:

```
...\ScatterWeb.Process.h:  
  
void Process_init() {  
    System_registerCallback(C_RADIO, Process_radioHandler);  
    System_registerCallback(C_SENSOR, Process_sensorHandler);  
}
```

These functions have to be implemented in the application and are linked with the system after compilation.

While `Process_sensorHandler` and `Process_radioHandler` are directly related to the respective event (sensor event and radio event), the regular intervals are different and realized by using software timers.

4 Lessons

In this section a simple userapp example is described: Switching LED on and off, toggling if requested so via the serial interface, using the timer, and sending and receiving packets.

4.1 Switching LED on and off

Have a look at the following lines of code:

```
..\ScatterWeb.Process.h:
UINT8 counter;

void toggleLED() {
    if (counter == 1) {
        Data_enable(0, 0, NULL); // enable the red LED
        counter = 0;
    }
    else {
        counter = 1;
        Data_disable(0); // disable the red LED
    }
    Timers_add(1024, toggleLED, 0xFFFF);
}

void Process_init() {
    [...]
    counter = 0;
    Timers_add(1024, toggleLED, 0xFFFF);
}
```

This code is easy to understand: It toggles the red LED; i.e. switches it on and off in regular intervals. Try it out!

4.2 Toggling on command

Next, what you will need quite often are new terminal commands for testing. For now, try to write a terminal command called “tog” that has as the only argument the number of times the LED should toggle. So if you enter “tog 5” it toggles five times and so on.

Therefore, you first of all have to add the “tog” command to the list of known and supported commands. This can be done quite easily using a macro:

```

..\ScatterWeb.Process.h:
COMMAND (tog, 0) {
    UINT16 iterations = 0;
    iterations = String_parseInt(&str[4], NULL); // parse an integer
    Comm_print("%i iterations\r\n", iterations); // print over serial
    term_toggle(iterations);
}

```

The function `String_parseInt` reads from the command-array (`str`) where all the input of the serial line is buffered. In this case, it reads beginning from position 5 (C begins counting at zero, so at positions 0, 1 and 2 the command `tog` is stored as ASCII, at position 3 the space).

The next thing you need is to implement the toggling itself. Look at the straightforward solution here:

```

..\ScatterWeb.Process.h:
void term_toggle(UINT16 iterations) {
    UINT16 i = 0;
    for (i = 0; i < iterations; i++) {
        Data_enable(0, 0, NULL);
        System_wait(10000); // block for 10,000 nops
        Data_disable(0);
        System_wait(10000); // block for 10,000 nops
    }
}

```

Try it out, it works! But: It works only for small numbers, so if you type in “tog 2” you will see it toggling twice, but “tog 19” will most likely not bring the expected results. The serial output of the node will give you a hint on what happens instead (also lookup the implementation of `System_wait()` in `System/ScatterWeb.System.c`). A better idea is to use the timer as described in the next section.

4.3 Using the Timer

The better way to implement this kind of long-term, returning action is to use timers. Using timers, you would once switch on the LED, switch it off immediately and then set the timer for the next time it should toggle again.

The only function you need to know for this is the function `bool Timers_add(UINT16 t, fp_timer fp, UINT16 data)` that is defined in `System/src/ScatterWeb.Timers.c`. If you are not used to C-style function

pointers, this might be a bit tricky, but can be copied from the example given in the following.

Imagine a revised toggle-command that calls `term_toggle2()` implemented as follows:

```
...\ScatterWeb.Process.h:
void term_toggle2(iterations) {
    if (iterations <= 0) return;
    Data_enable(0, 0, NULL);
    System_wait(10000);
    Timers_add(500, term_toggle2, --iterations); // call the function
    Data_disable(0);
}

COMMAND (tog, 0) {
    UINT16 iterations = 0;
    iterations = String_parseInt(&str[4], NULL);
    Comm_print("%i iterations\r\n", iterations);
    term_toggle2(iterations);
}
```

The function `Timers_add` is passed as first argument the value 500. That means after ca. 500 ticks (1024 ticks = 1 second) the timer calls the function that is passed as second argument. This function, `term_toggle2` in our example, needs an argument. This argument is the third parameter of the `Timers_add`-function. So in this example if `term_toggle2(4)` is called, after a time the function `term_toggle2(3)` will be called and so on, until the iterations variable is zero or less. Hint for the C programmers: Think (before testing) if `iteration--` would work as well! Play a little bit with these values; you could for example easily implement a countdown that ticks faster as nearer it comes to its end.

4.4 Sending and receiving packets

Let us consider the case that you want to carry out simple connectivity tests: You send a command to a sensor node, this triggers a broadcast packet sent, and all nodes that receive this packet switch on their red LED.

```
...\ScatterWeb.Process.h:
COMMAND (scp, 0) { sendConnectivityTestPacket(); }
```

Then you could for example implement the function `sendConnectivityTestPacket()` directly above the command as follows:

```
...\ScatterWeb.Process.h:
void sendConnectivityTestPacket () {
    // define a payload
    struct ConnectivityTestPacketStruct ConnectivityTestPacket;
    // fill the payload
    ConnectivityTestPacket.type = 0;
    ConnectivityTestPacket.src = Configuration.id;
    //define a packet
    packet_t p;
    // fill the packet
    p.to = BROADCAST;
    p.type = CONNECTIVITY_TEST_PACKET;
    p.header = 0;
    p.header_length = 0;
    p.data = (UINT8*) &ConnectivityTestPacket;
    p.data_length = sizeof(ConnectivityTestPacket);
    // sends a packet
    Net_send(&p, NULL);
}
```

You basically have to define a structure of the data part (payload) of your packet, define your own packet type (in this example `CONNECTIVITY_TEST_PACKET`, define this in `System/ScatterWeb.Net.h`) and send it over the air. The structure of the data part could be defined as follows in `ScatterWeb.Process.h`:

```
...\ScatterWeb.Process.h:
struct ConnectivityTestPacketStruct{
    UINT8 type;
    UINT16 src;
};
```

When receiving packets, all you have to do is very simple: Look for the packet type you just defined (`CONNECTIVITY_TEST_PACKET` in our example), interpret the payload of the packet as structured as you defined it at the sender side (in our example, as defined in `struct ConnectivityTestPacketStruct` in `ScatterWeb.Process.h`) and act accordingly.

Packets that the system layer does not filter are passed to the application, where the appropriate action according to the packet type can be chosen in the switch (rxPacket.type)-construction in the packet handler function void Process_radioHandler () in ScatterWeb.Process.c.

Insert a new case-statement for example as follows:

```
...\ScatterWeb.Process.h:
void Process_radioHandler(void* receivedPacket) {
    [...]
    else {
        switch(rxPacket.type) {
            [...]
            case CONNECTIVITY_TEST_PACKET:
                Comm_print("Connectivity test packet received\r\n");
                handleConnectivityTestPacket(rxPacket.from, rxPacket.data);
                break;
            [...]
        }
    }
}
```

Well, this only filters out the case that a CONNECTIVITY_TEST_PACKET was received. Everything further is done in the handler function:

```
...\ScatterWeb.Process.h:
void handleConnectivityTestPacket(UINT16 from, UINT8* payload) {
    struct ConnectivityTestPacketStruct* ConnectivityTestPacket;
    ConnectivityTestPacket =
        (struct ConnectivityTestPacketStruct*) payload;
    Comm_print("Conn.test packet from: ");
    Comm_print("%i", from);
    Comm_print(",from field in payload: ");
    Comm_print("%i\r\n",ConnectivityTestPacket->src);
    Data_enable(0, 0, NULL);
}
```

The casting of the payload to the structure is a bit tricky here. For the beginning you should keep strictly to this scheme, especially if you are a beginner with the C programming language.

4.5 Now it is your task

Make a real two-way connectivity test by sending back a `CONNECTIVITY_TEST_RESPONSE` packet to the sender of a `CONNECTIVITY_TEST_PACKET`. But be careful: Doing it that simple straightforward way will result in a lot of simultaneous answers of different nodes, resulting in packet collisions and traffic jam. Therefore you should send packets to a specific node, no broadcast. Therefore you need to implement:

- A new terminal command “cnt” (cnt stands for connectivity test) that takes the destination node ID as argument
- A modified handler of the `CONNECTIVITY_TEST_PACKET` that sends back the `CONNECTIVITY_TEST_RESPONSE` packet
- A new handler that handles the reception of a `CONNECTIVITY_TEST_RESPONSE` packet and writes to the serial port which node answered the test packet

Basically, that’s all. Feel free to play around with this and find new combinations. For the rest you will have to read the documentation or the source code. Avoid changing the firmware, there is definitely no need for that and it makes portability of implementations much easier if the firmware is everywhere the same. For further information take a look at <http://scatterweb.mi.fu-berlin.de>.

Have Fun!