veritas
iustitia
libertas

Freie Universität Berlin

First steps with the ScatterWeb sensor nodes

Technical Report

April 2005

Freie Universität Berlin

Institute of Computer Systems & Telematics

A. Liers, H. Ritter, J. Schiller

http://www.inf.fu-berlin.de/inst/ag-tech

Content

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 config*, 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

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.

Achim Liers, Hartmut Ritter, Jochen Schiller, April 2005

2 Hardware and Software

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 give some hints about the installation of the software needed.

2.1 ESB: The ScatterWeb Nodes

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 ESB, embedded sensor boards. Figure 1 gives an example of such an embedded sensor board.

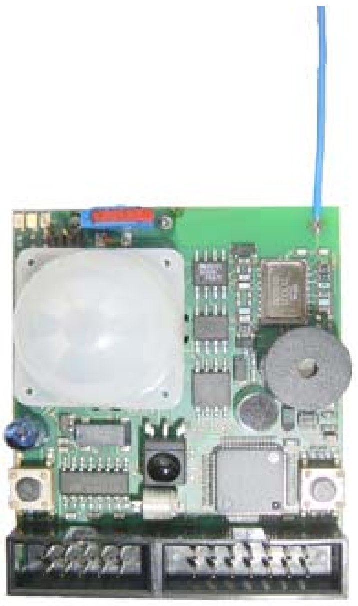

Figure 1: Embedded Sensor Board

It provides the following sensors and actuators:

- Light sensor
- Microphone
- Vibration sensor
- PIR movement detection
- IR transmitter/receiver
- Beeper, LEDs
- Real-time clock
- 868 MHz radio transceiver

2.2 ScatterFlasher: The ScatterWeb Gateways

The ScatterWeb gateways serve as interfaces between the sensor network and a monitoring station or the Internet. As with the nodes, different gateways exist that interface the sensor network via USB to a PC, via Bluetooth to any other Bluetooth device, or via Ethernet to a LAN and the Internet. As all these devices allow not only reading access to the sensor network, but also over-the-air reprogramming of the nodes, the gateways are also called ScatterFlasher.

In this report, the ScatterWeb gateway between the sensor network and a PC via USB is described. This gateway is also referred to as ScatterFlasher/USB. A view of a ScatterFlasher/USB is given in Figure 2.

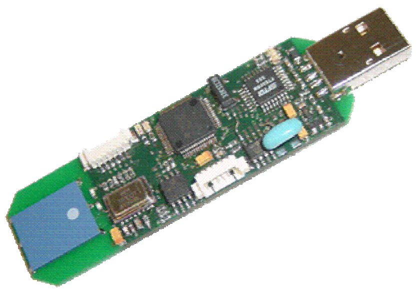

Figure 2: ScatterFlasher/USB

2.3 Software Installation

The development environment consists of

- An editor (PSPad in this case)
- The Gnu C-Compiler
- A tool for flashing the complete image down to the Controller

These tools are all preinstalled on the machines in the PC cluster.

2.4 Software architecture

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 userlevel 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.

2.4.1 System

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

```
<System/src/ScatterWeb.System.c>
[…]int main(void){ […]
for(;;){
   System_startWatchdog(); // starts&resets the watchdog
    if(callbacks[C_PERIODIC]) callbacks[C_PERIODIC]((void*)NULL);
    if(runModule & MF_SCOS) Threading_eventHandler();//look for runnable tasks
    if(runModule & MF_RADIO_RX) Net_rxHandler();//handle received radio packets
    if(runModule & MF_RADIO_TX) Net_txHandler();//handle radio packets to be sent
    if(runModule & MF_TIMER) Timers_eventHandler();//check for expired timers and overflow
    if(runModule & MF_SERIAL_RX) { // Callback for serial line
      extern volatile UINT8* serial_line;
      if(serial_line != 0){if(callbacks[C_SERIAL]) callbacks[C_SERIAL]((void*)serial_line); }
    }
    if(runModule & MF_SENSORS) if(sensorFlags != 0x00) Data_sensorHandler();//handle sensors
    if(runModule & MF_RC5) Data_RC5ReceiveHandler();//handle IR RC5 code
//now: go to sleep (LPM1) if all runModules have been handled (i.e. if runModule == 0)
    dint(); nop();
   if(runModule==0) { // enter lpm3 and enable interrupts at once to not loose an interrupt
     System_stopWatchdog();eint(); LPM1;
   }
   else eint();
}   }
```

code snippet 1: main loop of the system

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.

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

2.4.2 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 it own handler functions as callbacks with the system during the initialization phase:

```
<Applications/.[…]/ScatterWeb.Process.c>
bool Process_init(){
  System_registerCallback(C_RADIO, Process_radioHandler);/*function Process_radioHandler gets called if a radio packet is received*/
  System_registerCallback(C_PERIODIC, periodic_function);/*the function periodic_function is called every 1/1024 seconds*/
  Event_init();/*defined in ScatterWeb.Event.c, calls besides others the function System_registerCallback(C_SENSOR, handleEvent); thus the function handleEvent is called if a local sensor event detected */
  return True;
}
```

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

While handleEvent and Process_radioHandler are directly related to the respective event (sensor event or radio event), the periodic_function() is different. You might think of it as the main function of the application. periodic_function() is called regularly from the firmware.

2.5 A simple application example

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

2.5.1 Switching LEDs on and off

Have a look at the following lines of code:

```
<ScatterWeb.Process.c>
void periodic_function(){
  Comm_print(" %i",counter);//debug print over the serial line
  if (counter>=250){//don’t forget to define and initialize the static variable somewhere
    Data_greenToggle();
    counter=0;
  }
  else
    counter++;
}
```

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

2.5.2 Beeping on command

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

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

```
<Applications/[…]/ScatterWeb.Event.c>
COMMAND (bip, 0){
  int iterations = 0;
  iterations = String_parseInt(&str[4], NULL);
  Comm_print(" %i",iterations);
  term_beep(iterations);
}
```

The function readUnsignedInt reads from the command-array 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 bip is stored as ASCII, at position 3 the space).

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

```
<Applications/[…]/ScatterWeb.Event.c>
void term_beep(int iterations){
  int i = 0;
  for (i=0;i<iterations;i++){
    Data_beeperOn();
    System_wait(10000);
    Data_beeperOff();
    System_wait(10000);
  }
}
```

Try it out, it works! But: It works only for small numbers, so if you type in “bip 2” you will hear it beeping twice, but “bip 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.

2.5.3 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 beeper, switch it off immediately and then set the timer for the next time it should beep 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 beep-command that calls term_beep2() implemented as follows:

```
<Applications/[…]/ScatterWeb.Event.c>

void term_beep2 (iterations){

  if (iterations <= 0)

    return;

  Data_beeperOn();

  System_wait(10000);

  Timers_add(500, term_beep2, --iterations);

  Data_beeperOff();

}
```

The function Timers_add is passed as first argument the value 500. That means after ca. 500*5 ms (= 2500 ms = 2.5 s) the timer calls the function that is passed as second argument. This function, term_beep2 in our example, needs an argument. This argument is the third parameter of the Timers_add-function. So in this example if term_beep2 (4) is called, after 2.5 s the function term_beep2(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.

2.5.4 Sending and receiving packets

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

Pressing a sensor button is handled similar like any other sensor event: The function handleEvent in ScatterWeb.Event.c is called. There it must be filtered out, which event has happened. In case of a button press this can be done as following:

```
<Applications/[…]/ScatterWeb.Event.c>

void handleEvent(sdata_t* data) {

   if (data->sensor == SENSOR_BUTTON && data->value == 1){

      Comm_print("%s", "button pressed");
```

```
    sendConnectivityTestPacket();
  }
[…]
}
```

Then you could for example implement the function sendConnectivityTestPacket() directly above the function handleEvent(sdata_t* data) as follows:

```
<Applications/[…]/ScatterWeb.Event.c>
void sendConnectivityTestPacket(){
    struct ConnectivityTestPacketStruct ConnectivityTestPacket;//define a payload
    ConnectivityTestPacket.type = 0;//fill the payload
    ConnectivityTestPacket.src = Configuration.id;//fill the payload

    packet_t p;//define a packet p
    p.to = BROADCAST;//set the destination address to broadcast
    p.type = CONNECTIVITY_TEST_PACKET;//set the packet type to test packet
    p.header = 0;//don't use an optional second header
    p.header_length = 0;//don't use an optional second header
    p.data = (UINT8*) &ConnectivityTestPacket;;//pass the beginning of the paylod
    p.data_length = sizeof(ConnectivityTestPacket);//give the length of the payload
    Net_send(&p, NULL);//use the function Net_send to send the packet
}
```

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 System/ScatterWeb.Event.h:

```
<Applications/[…]/ScatterWeb.Event.h>
struct ConnectivityTestPacketStruct{
  char type;
  unsigned int 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.Event.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 Applications/ScatterWeb.Process.c.

Insert a new case-statement for example as follows:

```
case CONNECTIVITY_TEST_PACKET:
        Comm_print("%s","Connectivity test packet received ");
        handle_connectivity_test_packet((unsigned int*)&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:

```
int handle_connectivity_test_packet(unsigned int* from, char* payload){
  struct ConnectivityTestPacketStruct* ConnectivityTestPacket;
  ConnectivityTestPacket = (struct ConnectivityTestPacketStruct*) payload;//cast to the payload
      Comm_print("%s","Conn.test packet from: ");
      Comm_print("%i",*from);
      Comm_print(",from field in payload: ");
      Comm_print("%i\r\n",ConnectivityTestPacket->src);
      Data_yellowToggle();
  return 1;}
```

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.

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 have a look at http://scatterweb.mi.fu-berlin.de.

Have Fun!