Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Topology Information Gathering with Scatterweb Sensor Boards

Frank Lyner

Term Project; December 14, 2004

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistant: Aaron Zollinger

Introduction

This is the documentation of the term project I carried out at ETH Zurich in the summer of 2004. The project consists of two parts. The goal of the first part is to document the installation procedure of the software needed to work with the Embedded Sensor Boards. This documentation is tailored to the IBM ThinkPads used by the institute.

In order to gain more practical experience with the ESBs, I implemented a program that extensively uses the radio transmission capability of the ESBs. This program collects neighborhood information in a distributed multi-hop way to display the network structure on a computer. Furthermore, it is possible to send commands to a specific node using source routing. The documentation of this program is the content of part two.

Part I
Embedded Sensor Boards

The installation instructions mentioned above can be found in the installation package available in the distributed computing group. This is a short description of the Embedded Sensor Boards and a documentation of the experiences gained through the work with the ESBs.

1 Introduction

The Embedded Sensor Boards (ESB) have been developed by Freie Universitaet Berlin. They are a part of the broader project Scatterweb [1] . This project wants to give students hands-on experience in the fields of embedded sensor networks, embedded systems and wireless networking. It is also used as test-bed for distributed computing algorithms and commercial applications are being developed, too.

Several components emerged from this project including the Embedded Web Server, the Embedded Sensor Board, the Scatterflasher (USB stick with the same controller as the ESBs) and the Embedded Chip Radio (an ESB without the sensors).

2 Hardware

The ESBs combine a multitude of sensors with radio transmission and very low power consumption.

At the heart lies the MSP430F149 controller. It has 60kb of flash memory and 2kb of RAM. It can be set to sleep mode with interrupt waking, to power saving mode with a frequency of 4kHz or to normal operating mode with a frequency of 1MHz. There is a JTAG interface to flash the controller. The JTAG interface allows to step through a program directly on the controller.

Of greatest importance for this work is the radio transceiver TR-1001. The transmission power is scalable from 0 to 99, which can be interpreted as percentage of its maximum send power. The Scatterweb documentation claims that the maximum range is 300m in open space. The receive power can be monitored and the data rate (bandwidth) is $19\frac{kb}{s}$. The greatest advantage of this transceiver is its very low power consumption.

Here is a short summary of the installed sensors:

Movement There is a Passive Infrared Sensor (PIR) that can detect movement in a range up to 8m. The signal can be measured in an analog or a digital way.

Light The Infrared Light Sensor allows to measure the light intensity. The light intensity is translated into a proportional frequency. Additionally, this sensor is capable of distinguishing natural from artificial light coming from 50Hz or 60Hz sources.

Temperature There is an module that measures the temperature and has a built in clock.

Tilt/Vibration There is also a sensor to measure tilt and vibration. This sensor is capable of sending an interrupt to the controller, even if it is in sleep mode.

Sound There is a microphone with adjustable thresholds (loudness).

Date and Time The integrated clock provides date and time information.

Buttons There are two buttons available. One is reserved to trigger a reset; the other can be used by an application.

The ESB comes with three different ways to produce output. First, there are three LEDs: red, green and yellow. Second, there is a beeper that can produce several tones that are quite loud. Third, there is a RS232 interface. This is the main means to communicate with a node. The FU Berlin developed a number of terminal commands that can be used to change every possible setting and to read all sensor and configuration values.

3 Software Architecture and Programming

The software is divided into two parts: the firmware and the user program. In the code and in the documentation the user program is called *userapp*, so I will use this term, too.

The firmware manages basic functions of the ESB and provides a simple API. Usually, the firmware is not overwritten during the flashing process.

The interaction between the firmware and the userapp is done through callback functions. The userapp has to provide a function called *userapp_init*(). This function is called when the ESB is initialized and registers the callback functions of the userapp with the firmware. The *userapp_init*() is always linked to the same address, so the firmware does not need to be re-linked with the userapp. The main function of the firmware defines an event loop that checks if an appropriate callback function needs to be called and calls it if necessary.

There are four possible callbacks:

Periodic Callback This callback function is called in each iteration of the event loop.

Serial Callback This callback function is called when a complete line was received through the RS232 interface. The line has to be terminated with \r\n.

Radio Callback The firmware defines a packet format to use with radio communication. When such a packet is received, then the radio callback function is called with a pointer to the packet as parameter.

Sensor Callback In each iteration the sensor values are read and if any sensor delivers some data, then this callback is called.

3.1 Terminal Commands

As mentioned above, the main way to communicate with an ESB node is through its RS232 serial interface. In order to configure the node and read different values from it, the FU Berlin developed a series of terminal commands. There are commands to read each possible sensor value, to set or read the send power and so on. Unfortunately, these commands are not part of the firmware. In order to be used they need to be included in the userapp.

To do this, the terminal.c and terminal.h files need to be included in the current project. Then either the `term_execute()` function is directly registered as a callback in the `userapp_init()` function or another callback function that calls `term_execute()` is defined.

4 Problems and Solutions

In this section I give some hints and report on my experience concerning software development for the ESBs. In several cases the system did not work as I expected and I hope that this information will help others to avoid these problems.

The Clock

The clock is very powerful. It does not only return the current time, but also the current date. This makes it very useful for creating timestamps for logging. The problem I encountered was that when the clock is read very frequently, it returns inconsistent values. Tests showed that the value of the field of the time record—returned by the `createTimeRecord()` function of the firmware—that indicates the seconds did not increase continuously and/or was over 60(!).

I realized later, when I had already implemented a workaround, that the current time is being written at each event loop cycle into `sensorData.timeStamp`. I did not check, but maybe reading this value instead of creating a new timeRecord could lead to more reliable time information.

Dynamic Memory

The use of dynamic memory is very problematic. There is a `malloc()` function available that works only under very specific but not defined circumstances. The pointers returned by this function are not valid, from time to time. The people at FU Berlin are aware of this problem but have no solution but a work-around. The solution is to define a global variable that is certainly big enough to hold the dynamic data. This is not very elegant. However, we implemented a function `myalloc()` based on this solution. We globally defined three two-dimensional arrays. Their sizes are defined by the constants `BIG-ARRAY-NUM, BIG-ARRAY-LEN, MIDDLE-ARRAY-NUM, MIDDLE-ARRAY-LEN, SMALL-ARRAY-NUM, SMALL-ARRAY-LEN`. The `myalloc()` function returns an array of the smallest size that is larger than the requested amount of memory passed as an argument. It keeps track of the reserved arrays. Of course, we also implemented a `myfree()` function that allows to free a no longer needed array.

Data Types

When programming an embedded system, it is important to keep in mind that memory is a scarce resource. Therefore, the choice of the correct data type is very important. Since the smallest amount of memory that can be used is a byte and most numbers that occur in a program are positive and below 255, it is common to use `unsigned char` as standard type. Another reason is that `unsigned char` can be used as a universal type, holding numbers, characters or booleans. The type `u_int8_t` is internally mapped to `unsigned char` but it is much shorter to write. This also increases the readability of the code.

Using Terminal Commands from within a program

There are several cases where it is handy to directly access the terminal commands. One case would be to send the commands over radio. Most functions that handle the commands use a pointer, called `command`, that normally points to the line received through the serial interface. In order to use these functions, even when no line was received, I implemented a function `set_command()` that allows to first set the command and then execute it with `exec_command()`. `exec_command()` is also used by `term_execute()`, so the behavior will be exactly the same.

Deployment

The result of a compilation is a hexadecimal file called `out.hex`. This file contains every byte that needs to be flashed to the ESB, including the firmware. There are two different ways to do this:

JTAG is a well-known hardware interface to flash embedded systems. It additionally allows to debug the code that runs directly on the device. It is important to remember that when an ESB is flashed with JTAG, the firmware is also overwritten.
JTAG is faster than using the Scatterflasher. However, I frequently experienced write errors and had to retry several times before succeeding.

Scatterflasher The Scatterflasher is a USB-stick. It has exactly the same controller and transceiver as the ESBs. The drivers, that are packed in the installation package, simulate a new COM port. So, it is possible to communicate with the Scatterflasher through the normal terminal. However, the main use of this device is to flash the ESBs without cable connection. This is done with the Scatterflasher software. The Scatterflasher only updates the userapp and leaves the firmware unchanged.
The Scatterflasher is quite reliable and much more convenient than the JTAG-interface when working with several nodes. In order for program flashing to work properly, the transmission power of the node to be flashed should be at 99.

Part II
Multi-hop

1 Goal

The goal of the second part of the project was to gain practical experiences with the ESBs by developing an application. This application should use radio transmission and be of help for future projects if possible.

With these requirements in mind we decided to implement a distributed algorithm that collects the topology of the network. This application could then be used to help set up a network for other applications. The topology information could also be used to find routes with corresponding algorithms.

The user interface to this application is a Java client that displays the network topology and gives access to all features of the application.

The experiences gained were already discussed in Section 4 of Part 1. What follows is the description and documentation of the developed application and the Java client.

2 Distributed Collection of Information - Echo

To collect information from every node in a network in a distributed way it is best to implement an echo algorithm. This algorithm consists of two phases. In the first phase a request message is flooded through the whole network. This establishes a tree structure on the network nodes with the originator as root. The leafs of the tree start the second phase by sending back a reply message to their parent nodes. Such an inner node collects all messages from its children, appends its own message and sends its reply to its own parent node. The algorithm terminates when the root receives a reply from all of its children.

How do we define the root? There are two different possibilities to interact with a network of ESBs. First, one can trigger an event through the sensors or buttons on an ESB. Second, one can choose one node, connect a computer to its serial interface and use the terminal commands. Since the second possibility is the only way to receive information from the nodes, this possibility will preferably be chosen. This automatically defines the root node.

In our case the information that we want to collect is commonly known as neighborhood information. A neighbor of a node is another node with which it can directly communicate.

2.1 First Try

At first, we tried to generate the neighborhood information at the same time as the echo algorithm is flooding the network. Since every node in the network is broadcasting the request message during the flooding, it is possible for a node to detect its neighbors. Since both phases of the flooding can overlap, it was necessary to delay the start of the second phase at the leafs.

This approach assumes that the links are bidirectional and more or less constant. These assumptions made sense in the beginning since there is no node movement. But unfortunately, they do not hold. Tests showed that the links are neither bidirectional nor constant. It is common that a node can send several packets to another node over a short time and for a long time after that this is not possible anymore. And if a node receives the request packet from its parent node during the first phase of the flooding, there is a high probability that it will not be able to send its reply back in just one try.

In short one can say that the transmission capability of the ESBs is too unreliable to be used without error handling.

2.2 Solution: Make Transmission Reliable

In order to make the transmission capability of the ESBs more reliable we had to implement a low level communication layer. The basic idea is that the communication layer retries to send a packet to a known neighbor as long as it does not succeed. If the packet could not be transmitted after 30 retries, the send is aborted without any notification. The successful reception of the packet is confirmed by the receiver with an acknowledgment packet.

Since the send operation should not be blocking, it was necessary to implement a send buffer. The send operation places the packet to be sent in that buffer. The ESB periodically retries to send the packets that are in the buffer. This is done one packet at a time in order to allow for reception of other packets. In order to prevent several nodes to send simultaneously and therefore constantly interfering with each other, the time between two retries is varied. When an acknowledgment packet is received, the corresponding packet is deleted from the send buffer.

2.3 Neighborhood Detection

The send operation of a packet now relies upon a bidirectional link. To avoid long retries that do not succeed, it is important that only these links are used for communication that have already proved to be bidirectional with reasonable quality. This can be achieved by separating the neighbor detection from the collection of this information.

We changed the neighborhood detection as follows. The central node of the network floods the start command through the network. When a node receives the start command, it starts broadcasting a search packet. The start command contains the number of search packets that each node has to send. When a node receives a search packet, it sends back an answer packet. Each node counts the number of answers it receives. If the number is higher than a given threshold, the link is considered to be reliable. The threshold is also included in the start command. Of course, at this point in time the whole communication does not use the reliable communication layer.

Several searches can be done consecutively. If an already known neighbor answers in a new search then the link quality is recounted. But if a known neighbor does not answer in a new search the link information is *not* deleted. We used this to build an artificial topology during development, to observe the behavior of the algorithm. To get the physical topology at hand it is necessary to perform a multihop reset or a global reset.

2.4 Echo Algorithm

The echo algorithm can now take full advantage of the reliable communication layer. Each node sends a request packet to all of its neighbors except its parent. When a node receives a request packet for the first time, it sends back a confirmation that it is the senders child and that the sender should wait for its reply. When it hears a request a second or further time it sends back a packet indicating that it is a neighbor but not a child of the sending node.

The rest of the algorithm works the same way as the normal flooding described above. The only exception is that the more extended neighborhood detection allows to gather more information about the link quality. This information is also included in the reply.

The reliable transmission layer and the need to store the replies of the children of a node were the reason why we implemented the `myalloc()` and `myfree()` functions (see section 4 of Part 1). Of course, the numbers and the sizes of the globally defined arrays used by these functions represent a limitation on the number and size of replies that can be stored. The total memory reserved for the the global arrays is also very limited because there are only two kilobytes of RAM. Because of these limitations it will be necessary to check and possibly adapt the structure of the reserved memory for each concrete network setting.

2.5 Sending Commands with Flooding

It is often important to change the configuration of every node in the network. Examples are: resetting all nodes to the start state, changing the transmission power to reflash the nodes with the scatterflasher et cetera. We implemented a new terminal command which takes another terminal command as an argument and sends it through the whole network with flooding. A flooding algorithm relies upon message identifiers to check if the message was already received or not. For this reason a command counter was used that is included in the flood message. If a node receives a flood message with a command counter that is below its own command counter, then the message is ignored. If the command counter is higher than its own counter, then the node sets its command counter to the value received in the message. The node then rebroadcasts the message and executes the terminal command. This flooding does *not* use the reliable communication layer. This is done deliberately because it is often necessary to broadcast a command before the reliable communication layer is ready to be used.

2.6 Sending Commands with Multi-hop

Additionally, to broadcast a command through the whole network we need the possibility to send a command to a specific node even if this node is out of range of the node that is connected to the computer. We decided to use a simple multi-hop source routing algorithm. We implemented a terminal command that takes as arguments the command to send and the route to use. It then sends a packet that includes the command and the route. Upon reception of such a packet a node checks if it is the destination or a part of the route. If it is the destination, it executes the terminal command and sends back a confirmation message using the inverted route. If it is a part of the route, it forwards the whole packet toward the next node in the route. All communication is going through the reliable communication layer. At the moment no check is done if the next hop is a neighbor of the current node. It is assumed that the topology of the network is known to the entity that defined the route.

2.7 Further Improvements

Link Quality

As described above, the link quality is determined with the number of answers a node receives from a neighbor. This means that only these times are counted in which the link worked in both directions. Since every node does the same, each link is checked twice for its bidirectional quality. It would be an improvement to determine the unidirectional quality for both directions of a link.

Next Hop Check

As mentioned above our source routing algorithm does not check if the next node in the route can be reached from the current node. This check should be implemented. Otherwise, a node will retry very long to send a packet.

Retry Threshold

A retry threshold is already implemented. But in the current version when a node is not able to forward a packet, it does not notify the original sender. Some kind of error reporting and error handling should be implemented.

Store Replies in EEPROM

In section 2.4 we described that the number and size of replies is limited because of the small memory. A solution could be to store the replies in the EEPROM.

This may be slow and power consuming, but the size of sixty kilobytes should resolve the problem.

3 Java Client

The user interface to the ESB application is a Java Client. It is thought to run on a laptop or desktop computer that is connected to an ESB with a serial cable. The client enables to use all functionalities and algorithms described above in a user-friendly way. The most important feature is the translation of the topology information into a graphic representation.

The following is a more detailed description of the features and how to use them. Subsequently we give a short documentation of the program which is followed by some thoughts on further development of the client.

3.1 Usage

3.1.1 Selecting COM Port

First, it is necessary to build the connection between the client and a selected node with a serial cable. After the client is started go to `ESB→COM port...`. The system prompts to select the appropriate COM port.

3.1.2 Reset the Nodes

With the current firmware a node is reset when the button located on the same side as the antenna is pressed. This only resets the firmware and has no influence on the values of global variables used in the userapp. Since our application uses a number of global variables—such as the command counter described in section 2.5—it can be necessary to reset them separately. We implemented three different levels of resetting:

Global Reset The global reset is the most thorough one. All global variables used are reset. This also includes the send buffer, the command counter, the echo counter (used as identifier for the echo flooding), the stored link information, et cetera. It is recommended to do a global reset when new nodes are added to the network. This will ensure that no node will ignore a broadcasted command.

Multi-hop Reset Here nearly all global variables are reset. The command counter will not be reset but all link information is destroyed.

Echo Reset The echo reset only removes the tree structure that is built by the echo algorithm. The link information remains intact.

Every reset level is implemented as a terminal command. When they are triggered from the Java client with `ESB→reset...` the appropriate command will be flooded through the whole network using the command broadcast described in section 2.5. The global reset command cannot be ignored by any node even if its command counter is higher than the one in the command message.

3.1.3 Start Neighborhood Search

The neighborhood search can be started with `ESB`→`Start neighbor search...` . The client first prompts for the number of search tries and then for the minimal number of successful tries that determine the link threshold. The root node then floods the start-search packet through the network. All nodes of the network react by the procedure described in section 2.3. The root node resets its transmission power to the old value and starts the same procedure as all other nodes.

3.1.4 Collecting Neighborhood/Topology Information

With the menu item `Network`→`Collect neighborhood infos` the echo algorithm is started that collects the topology information. If no neighborhood search has been done before, this will not work. The root node sends the topology information to the client which translates it into the graphical representation.

3.1.5 Work with Graphical Representation

Once the topology is collected, it is displayed. The nodes are arranged in a circle and their order is random. They can be moved with drag and drop. A right-click context menu allows to delete or rename them. A node is selected by clicking on it. It is possible to draw a new connection between two nodes. If one node is selected, then the connection can be dragged to the other node.

3.1.6 Broadcast a Command

The menu item `ESB`→`Broadcast command...` leads to this feature. The user is prompted to enter the terminal command. After clicking `OK` the client sends the command to the root node which floods it through the whole network and then executes it itself.

3.1.7 Send a Command through Multi-Hop

To send a command to a specific node with multi-hop, not only the command must be supplied but also the route to take. This is started with the menu item `ESB`→`Multihop command...`. If no topology information was collected, the client aborts the operation with an error message. Then the client prompts for the command to send. After that the route has to be specified. This is done by clicking on each node that needs to be added in the desired order. The first node to add always has to be the root node. When the route is completed, the command is sent with a double-click.

3.1.8 Send a Command to the Root Node

As described above, the root node is defined as being the ESB connected directly to the Computer on which the Java client is running. When the `Enter` key is pressed, a dialog opens. It allows to enter the command to send. This command is then sent to the root ESB and the output can be viewed in the lower part of the window.

3.2 Further Improvements

- The only automatic arrangement available now is a circle layout. This is good for small networks. For large networks it would be better to implement a spring system.
- When sending a command with multi-hop no check is done concerning the validity of the route. Only neighbors of the node that was last added to the route should be enabled to be added to. The other nodes should be disabled.
- Each time the Java client is started it is necessary to select the COM port. This information could be saved and automatically applied for more convenience.

4 Conclusions

In this paper we described the Embedded Sensor Boards and an application that collects topology information and displays it in a Java client. All parts are working. Tests showed that the small memory limits the size of the network. With ten nodes the application still worked well although not every time. Depending on the structure of the tree that is built with the echo algorithm it can happen that the number of reserved arrays for the dynamic memory allocation is not sufficient. On several occasions the application behaved unexpectedly because a node ran out of memory and some parts of its RAM were overwritten.

The ESBs have a very low power consumption and very good sensor capabilities but they have limitations to run elaborate distributed algorithms.

References

[1] Freie Universitaet Berlin. Homepage scatterweb. http://www.scatterweb.de.