

Wireless technology bridging node

Per-Henrik Persson

Department of Information Technology
Lund University

Advisor: Mats Cedervall

September 28, 2005

Printed in Sweden
E-huset, Lund, 2005

Abstract

The objective of this master's thesis was to design a prototype of a node capable of bridging data between different wireless technologies. The node must support different communication technologies including IEEE 802.15.4/Zig-Bee, Bluetooth, and WLAN (IEEE 802.11x). Small physical size and low power consumption was of essence. Also, a software interface to utilize the communication hardware was to be defined. The work included investigation and comparison of different hardware and software components available on the market. The result is a hardware prototype, running proof of concept demonstration software.

Table of Contents

1	Introduction	1
1.1	Project description	1
1.2	Report outline	1
1.3	Background	2
1.3.1	LTH	2
1.3.2	connectBlue	2
1.3.3	RUNES	2
1.4	Acknowledgements	2
2	Wireless technologies	5
2.1	Bluetooth	5
2.2	IEEE 802.15.4/ZigBee	6
2.3	WLAN	9
2.4	Summary	10
3	Hardware design	11
3.1	Requirements	11
3.2	Development tools	13
3.3	Hardware components	13
3.3.1	Microcontroller	13
3.3.2	IEEE 802.15.4/ZigBee	15
3.3.3	Bluetooth	16
3.3.4	WLAN	17
3.4	Schematic	17
4	Software design	21
4.1	Requirements	21
4.2	Bridging	22

4.3	Development tools	25
4.4	Component design	26
4.4.1	Real-time operating system	27
4.4.2	UART driver	29
4.4.3	PPP implementation	30
4.4.4	SPI driver and WLAN interface	31
4.4.5	ZigBee interface	31
4.4.6	IP stack	34
4.4.7	Web server	35
4.4.8	ZigBee microcontroller software	36
5	Results and discussion _____	39
5.1	Results	39
5.2	Discussion	41
5.2.1	Improvements	41
5.2.2	Future	42
	References _____	45
A	Abbreviations and terminology _____	51
A.1	Abbreviations	51
A.2	Terminology	53
B	Tables _____	55
B.1	CPU comparison table	55
C	Hardware details _____	59
C.1	Schematics	59
C.2	PCB layout	64
C.3	Bill of materials	68

Introduction

1.1 Project description

As part of the European Union project RUNES, the Swedish company connectBlue is responsible for the development of a hardware platform capable of bridging data between several wireless technologies. The aim of this master's thesis was to develop a prototype of this platform and to implement proof-of-concept software, demonstrating wireless data bridging. Also, software interfaces to utilize the communication hardware was to be defined.

The final node developed by connectBlue will be used as a research and development platform for the RUNES project.

The RUNES project specifies the requirements for the node. Furthermore, some requirements were added by connectBlue. The wireless technologies to be supported were Bluetooth, IEEE 802.15.4/ZigBee and WLAN (Wireless Local Area Network).

The work of this thesis was split into two, parts. The first part was the hardware design, where different components were evaluated and then put together into a final design. The second part involved the design and implementation of the software.

1.2 Report outline

Chapter 2 describes the wireless technologies used in the project. Chapter 3 describes the hardware requirements, the evaluation of the different hardware components and the final hardware design. In chapter 4 the software requirements are summarized and the software model and the implementation of the components is described. Chapter 5 contains a summary of the work, the results and a discussion based on the experiences gained during the work. In the appendices a list of abbreviations, explanations, hardware comparison

tables, hardware schematics and PCB (Printed Circuit Board) layout can be found.

1.3 Background

1.3.1 LTH

The Lund Institute of Technology [1], founded 1961, is the third largest institute of technology in Sweden. LTH forms the Engineering Faculty of Lund University, a university with more than 40 000 students.

1.3.2 connectBlue

connectBlue AB [2] is a Swedish company focused on wireless communication solutions, primary Bluetooth, for industrial and commercial use. The company has great knowledge and experience in hardware and embedded software design. connectBlue offers different products ranging from simple serial cable replacement to complete custom designed wireless products. Currently connectBlue has 23 employees, all situated in Malmö.

1.3.3 RUNES

RUNES [3] is a research project within the European Union 6th Framework Programme. It started in late 2004 and will run for 32 months. The project brings together a consortium of industrial and academic partners from European countries, Australia and America.

The objective of the RUNES project is to “enable the creation of large-scale, widely distributed, heterogeneous networked embedded systems that interoperate and adapt to their environments” [4].

As a part of the RUNES project, three hardware platforms shall be developed. The project specifies the hardware capabilities of these platforms, that are to be used as development platforms for the software created in the project. connectBlue is responsible for developing the middle-class hardware platform, and to specify and implement software interfaces to utilize the hardware of the platform.

1.4 Acknowledgements

I would like to thank everyone who has contributed to this project, especially my advisor Magnus Johansson and all the co-workers at connectBlue.

Mats Cedervall at the Department of Information Technology at Lund University for introducing me to connectBlue and helping me out with the administrative part of this thesis.

Wireless technologies

Three wireless communication technologies are used in this project, namely Bluetooth, ZigBee and WLAN. These standards are designed to provide wireless communication in different areas and applications. They differ regarding complexity, cost, hardware requirements and power consumption. A brief introduction to the technologies follows below.

2.1 Bluetooth

Bluetooth [5] is an industrial standard for wireless radio communications. The standard defines a way to connect and exchange information between different devices in a PAN (Personal Area Network) using short range radio. The main goal is to provide an easy way to connect small electronic “gadgets” using a low-cost, secure standard [6]. Today the Bluetooth version 1.2 is the one most commonly used, even though the latest version is 2.0+EDR (Enhanced Data Rate). Probably, all new products will be supporting version 2.0 in a near future.

Bluetooth enables up to eight devices to communicate with each other. One of the devices always acts as master and controls the communication with the other devices acting as slaves. All data is exchanged using point-to-point communication between a master and one or more slaves. This means that all data is sent through the master.

Each device is identified by a unique 48-bit address. A device also has a more user friendly alphanumeric name which usually is user configurable. Even if this name changes, the unique 48-bit address remains the same.

Two devices can be bound together in a procedure called pairing. When two devices are paired, data can be exchanged. To simplify the pairing, a device may perform an inquiry to find other devices. A device can only be found during an inquiry if it is configured to be “discoverable”.

Pairs of devices can establish a trusted relationship. The user must then provide one of the devices with a passphrase of the other device. This passphrase combined with the unique Bluetooth addresses can be used to authenticate one of the paired devices to the other. Trusted devices may encrypt the data they exchange.

The globally available 2.4 GHz short range radio frequency band is used for the Bluetooth communication. Bluetooth uses frequency hopping to avoid radio interference with other devices using the same frequency band. The hopping mechanism uses 79 channels, each 1 MHz wide, switching channel up to 1600 times per second. By doing so, the sending frequency is changed continuously, thus minimizing the risk of collisions with other devices sending on the 2.4 GHz band. The maximum theoretical data speed is 723.2 kbit/s. The Bluetooth specification defines three different output power classes. Class 3 devices are rare due to their very limited output power. A Class 2 device has a practical range of approximately 20 meters, and a Class 1 device allows communications up to 100 meters, thus making Bluetooth useful not only in PANs.

To ensure compatibility between Bluetooth devices, a set of profiles are defined. The profiles can be seen as the three upper layers in the OSI (Open Systems Interconnection) communication model. For example, the Generic Access Profile is used to ensure that a device can establish connections and perform inquiries to discover other devices. The Serial Port Profile is used to provide a simple serial communication channel using Bluetooth.

The Bluetooth protocols are designed to enable low power consumption. The protocols implement different power saving modes where the transmitting interval is increased. Still the actual power consumption heavily depends on the application. Continuously data streaming of course consumes more power than sporadic data transfers.

Bluetooth version 2.0 supports up to three times higher data transfer speeds through EDR. In version 2.0 improvements regarding power consumption and bit error rate are made, still being fully backwards compatible with the 1.x versions.

2.2 IEEE 802.15.4/ZigBee

ZigBee [7] is a specification of high level communication protocols to be used with the IEEE 802.15.4 standard. Typical application areas are PANs and home and building automation.

The ZigBee specification is built upon the IEEE 802.15.4 standard, using it for the OSI physical and MAC (Media Access Control) layer [8]. The IEEE standard specifies how communication between two devices is done

on a low level, i.e. radio access and frequency bands, device addressing, connection establishing and data exchange. Upon this, the ZigBee protocol specifications are added, resulting in a standard for low complex, low cost, low power consumption and low data rate devices [9].

Many of the features inherited from the two bottom network layers provided by IEEE 802.15.4 are often seen as a part of the ZigBee protocols even though they are not. From here on, all properties and specifications are referred to as ZigBee even if it is a feature originating from IEEE 802.15.4.

ZigBee uses three different radio frequency bands at 2.4 GHz, 915 MHz and 868 MHz. Not all of these bands are available in all countries, making the 2.4 GHz band the only one globally available. The maximum speed at 2.4 GHz is 250 kbit/s. The maximum range is approximately 30-70 meters.

There are three types of ZigBee devices. All three types are able to communicate with other devices and exchange data. The first type is called a "coordinator" and is the master in a ZigBee network. In each ZigBee network there is exactly one coordinator, that coordinates the other devices in the network and stores information about the network structure. The second type of device is called a "router" and is less complex than the coordinator. It is capable of acting as a router, relaying data from other devices. The last device is called an "end device" and is just capable of talking to the network, with no relay functionality.

Sometimes the terms FFD (Full Function Device) and RFD (Reduced Function Device) are used to describe ZigBee devices. Only an end device can be an RFD. An FFD requires more powerful hardware than an RFD, thus being more expensive.

It is possible to create large ad-hoc networks, consisting of smaller network clusters. Because of the relay functionality, a coordinator device or a FFD can pass data from devices that are not in range of each other. This results in a mesh network. Unfortunately, the ZigBee version 1.0 standards can not handle a failure of a network coordinator, thus making the coordinator a single point of failure.

All devices can be identified using either a 64-bit unique address, or a 16-bit network address. The 16-bit network address is assigned to a device by the network coordinator when connecting to the network.

To minimize power consumption in coordinators and routers, the protocols support beaconing networks, where all the devices transmits beacons to indicate their presence in the network. These beacons are used to synchronize all the network devices. Devices can then be put in sleep mode between the beacons and only wake up to listen and send beacons at given times. Long beacon intervals lowers the radio duty cycle and minimizes the power consumption of the devices. The beaconing only improves the power consumption of the coordinators and routers. End devices must still wake

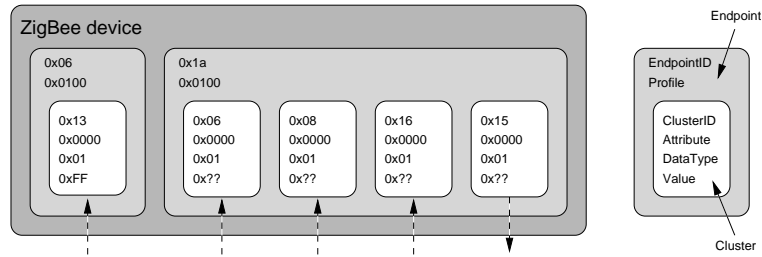


Figure 2.1: A ZigBee device with two endpoints supporting the “Home Control, Lighting” profile and clusters. The left endpoint includes the mandatory clusters, the right some of the optional ones.

up every beaconing interval and perform some processing. When not using a beaconing network, an end device can be put into sleep until an external event triggers it.

It is also possible to create non-beaconing networks giving the devices better response times. A device in a non-beaconing network usually has the receiver part of the radio enabled, listening for incoming events.

To enable data exchange between two devices in a standardized way, a concept with application profiles is used [10]. A ZigBee device implements one or more profiles, such as the “Home Control, Lighting” (HCL) [11] profile used to control building lighting. Each profile specifies mandatory and optional clusters. A cluster represents a function of the profile. For example, the HCL profile specifies that switching load controllers (lamp controllers) has one input cluster to turn on and off the load. Similarly, the switch remote control (lamp switch) has one output cluster representing the status of the switch. When a switch controls a lamp, the output cluster from the switch is associated with the input cluster of the lamp.

Clusters can be combined into endpoints. An endpoint represents a profile, and some or all of the clusters specified in the profile. A ZigBee device can have several endpoints supporting different profiles and clusters, as shown in figure 2.1.

The actual data to be exchanged between two devices can be sent using two methods. Either the message (MSG) method using raw, unformatted bytes is used, or the key value pair (KVP) method. A KVP consists of an identifier, a data type and a value. The KVP can be seen as a variable of a defined type holding a value. A ZigBee profile specifies the method to be used with every cluster in the profile. The HCL profile uses KVPs for all clusters.

Security is provided in different ways. The IEEE 802.15.4 MAC layer supports access control lists, data encryption using up to 128-bit symmetric key encryption with authentication and mechanisms to ensure data integrity and freshness. In addition to this, the ZigBee protocols adds mechanisms to transfer encryption keys between devices in a network. Exactly what security features used is of course application specific, and the tradeoffs must be considered. Better security requires higher system processing power, limits the bandwidth and raises the power consumption.

2.3 WLAN

Today WLAN [12] usually refers to the family of IEEE 802.11x standards. Devices that are tested and verified to comply with the IEEE 802.11x standards are often referred to as Wi-Fi (Wireless Fidelity) devices.

The IEEE 802.11b [13] standard uses the globally available unlicensed 2.4 GHz frequency band. The data is sent using one of 14 overlapping 22 MHz wide channels. The maximum raw data speed is 11 Mbit/s. Due to protocol overhead the practical maximum speed is lower.

The common range is in the area of tens of meters. With high gain and/or directional antennas the range can be extended to kilometers.

The IEEE 802.11g standard uses the same frequencies as the b-standard but due to a different modulation technique the maximum raw data speed is raised to 54 Mbit/s, resulting in a practical speed of approximately half of that.

A version of the standard, IEEE 802.11a, using the 5 GHz band is also available. The main advantage of the higher frequency is that it is less used by other communication protocols, thus resulting in less interference.

All three standards are usually operating in a point-to-multipoint configuration where the central device is an access point. All data is routed through the access point. Other configurations include point-to-point communication and ad-hoc networks, where the devices are linked together without using an access point. Devices that are in the range of others form a network. By using broadcasts that are forwarded by each device, it is possible to exchange data between two devices that is out of range.

WLAN support different security mechanisms to ensure secure data exchange. Due to security flaws in the previously used WEP (Wired Equivalent Privacy) protocol, today's WLAN products uses WPA2 (Wi-Fi Protected Access) that features authentication and strong encryption.

2.4 Summary

Because Bluetooth, ZigBee and WLAN all operate in the same unlicensed 2.4 GHz frequency band, they all affect each others performance when used at the same time. Also, a lot of other sources causes interference, such as other communication protocols and microwave ovens. Hence all protocols using the 2.4 GHz band must be constructed so that they can withstand interference. WLAN and Bluetooth have proved to have good fault tolerance against interference. Recent tests indicate that ZigBee is less tolerant to interference in the 2.4 GHz band.

This chapter is summarized in table 2.1, which shows a comparison between the three wireless technologies.

	Bluetooth	ZigBee	WLAN
<i>Data rate</i>	≈720 kbit/s	250 kbit/s	54 Mbit/s
<i>Range</i>	20-100 m	30-70 m	10-100 m
<i>Power consumption</i>	Medium	Low	Medium
<i>Cost</i>	Medium	Low	High

Table 2.1: Table comparing the three wireless technologies used in the project.

Hardware design

To end up with a good hardware design, several steps must be taken. First the given requirements must be analyzed to get an idea of what components that are needed, and what the requirements on these are. It is also necessary to verify that the hardware chosen is sufficient to implement the desired functionality of the software.

The evaluation of what components to be used is an iterative process. Different components have different features that can only be combined with other components in certain ways. Therefore, the evaluation of components described below, were not performed one after the other but iteratively. When a decision regarding a component was made, the requirements on other components sometimes had to be changed.

3.1 Requirements

The RUNES project specifies the requirements for three different types of nodes, ranging from a simple node with small hardware resources to a node with the performance of a modern PDA (Personal Digital Assistant). The node implemented in this project is the middle class one. As mentioned in the introduction, requirements from RUNES and connectBlue were to be considered. The RUNES requirements [15] can be summarized as:

- 8-bit to low-end 32-bit microcontroller
- <20 kbytes RAM
- <500 kbytes program memory
- Small physical size, approximately the size of a match box
- Low power consumption, possible to run on battery power
- IEEE 802.15.4 (and if possible ZigBee) radio interface

- Minimal real-time operating system
- A few general purpose digital I/O lines
- A few analog I/O channels
- Total node cost of approximately 30 Euros, including IEEE 802.15.4 radio
- Software development can be done using free available tools

Added to this list were connectBlue's requirements:

- 16-bit to 32-bit microcontroller
- Bluetooth radio interface
- WLAN radio interface
- Only one of the radio interfaces mounted at the same time
- Fast wake-up time from low power mode

When investigating the hardware requirements to enable the use of Bluetooth and WLAN, it was clear that the absolute low limit of RAM (Random Access Memory) is 32 kbytes. The WLAN option also needs about 120 kbytes for static firmware data which sets the program memory low limit to 256 kbytes.

The low power consumption requirement is somewhat undefined. Ideally it would be possible to run the node for weeks on three to four AA batteries. The only way to do this is to put the node in some kind of sleep mode most of the time, only waking it up on specified intervals or on incoming communication events. Some microcontrollers have a built-in PLL (Phase-Locked Loop) that is used to generate a system clock of variable speed. Running a microcontroller at a lower speed reduces the power consumption.

When choosing peripheral components, such as the IEEE 802.15.4 radio, available power saving modes of the components were investigated.

When running the node as a communication gateway, low power consumption is somewhat unrealistic due to the sheer fact that transmitting data over radio requires a lot of power. Minimizing radio transmitting lowers the total power consumption.

The fast wake-up requirement is not just hardware dependent, but can also be optimized in software using clever power save schemes. One problem comparing different microcontrollers regarding wake-up time is that it is not always presented in the datasheets in a usable way.

When looking at the different peripherals that should be connected to the microcontroller it was clear that the WLAN module requires one SPI

(Serial Peripheral Interface) channel and the Bluetooth module requires one UART (Universal Asynchronous Receiver Transmitter) for serial communication. Furthermore, the best option to connect the ZigBee radio interface to the microcontroller was using serial communication (see section 3.3.2). The microcontroller interfaces needed can be summarized as: one UART and one SPI channel when configured for WLAN, and two UARTs when configured for Bluetooth.

For the node to be able to act as a sensor/actuator, a few digital input and output lines were needed, together with a few analog inputs and outputs. The number of analog inputs and outputs are not specified. With two inputs and one output, two analog sensors and one analog actuator can be used. Also digital input and output lines are needed to control the ZigBee, Bluetooth and WLAN radio interfaces. The ZigBee radio interface requires one output pin, the Bluetooth interface one output pin and the WLAN interface one output and one input with hardware interrupt capabilities.

The best way to minimize the total cost and the physical size of the node, was to minimize the amount of hardware components. Ideally one microcontroller would fit all the requirements eliminating the need for external UARTs and analog converters.

3.2 Development tools

The only development tools needed during the hardware design were CAD (Computer Aided Design) software for schematic drawing. For this, Cadence OrCAD Capture was used because this is the schematic drawing software used at connectBlue.

3.3 Hardware components

3.3.1 Microcontroller

Although there are many different CPU (Central Processing Unit) architectures used in microcontrollers, mainly controllers based on the ARM (Acorn RISC Machine) architecture were chosen. This because connectBlue has good experience and knowledge in the ARM architecture, and ARM development tools are available at the company. The ARM architecture is supported by free development tools such as the GNU toolchain [17] and the newlib C library implementation [18].

Taking all hardware requirements in consideration, microcontrollers from many of the large semiconductor companies were chosen for further comparison. Many companies have at least one controller that more or less fits the

requirements. The controllers are commented below, for further details see appendix B.1.

Philips LPC2136 [19] features enough RAM and program memory to fulfill the requirements. The controller is based on the ARM7 architecture. It has enough UARTs and SPI interfaces, sixteen 10-bit A/D converters, one 10-bit D/A converter and a real-time clock. The power consumption is at an average level.

Philips LPC2106 [20] is an older generation ARM7 based microcontroller, somewhat superseded by the LPC2136. The program memory is too small and the controller lacks analog inputs and outputs.

Texas Instruments MSP430F1611 [21] is the top-of-the-line model in Texas Instruments 16-bit low power microcontroller series. The controller seems very versatile accepting a supply voltage ranging from 1.8 to 3.6 V. The power consumption is outstanding low and the wake-up time from sleep is really low. Unfortunately the RAM and program memory is too small.

Atmel AT91SAM7S256 [22] seems like a good performer featuring an ARM7 core and fast SPI. It has two integrated UARTs and also eight 16-bit A/D converters, though it is lacking a D/A converter. It has a DMA (Direct Memory Access) controller that can be used to speed up memory operations. The power consumption is below average. The integrated USB (Universal Serial Bus) controller adds cost and the large 64 kbytes RAM is out of the RUNES requirements.

Atmel AT91SAM7A3 [23] is another controller from Atmel's ARM7 microcontroller series. It lacks a D/A converter and has unnecessary integrated I/O interfaces such as USB and CAN (Controller Area Network) controllers.

ADuC7020 [24] is the controller from Analog Devices that best fits the requirements. The controller is based on the ARM7 architecture and features a lot of analog inputs and outputs. Unfortunately it only has one UART and the RAM and program memory is too small.

ST Microelectronics STR711FR2T6 [25] is very similar to the AT91SAM7S256 and therefore has the same drawbacks, such as the lack of a D/A converter and a too large RAM.

Freescale MCF5212 [26] is based on the 32-bit ColdFire V2 architecture. It seems to be a very powerful controller, yet not consuming too much power. It has enough number of UARTs, one SPI channel and a DMA

controller, only lacking a D/A converter. The big drawback is the somewhat unusual architecture.

The conclusion of the microcontroller evaluation was that the only one fitting all the requirements was the Philips LPC2136, so this was chosen as the main microcontroller for the node. Later during the hardware design phase, it was suggested by connectBlue to actually use the LPC2138 in the node. The only thing differing between the controllers are the program memory size. The LPC2138 has 512 kbytes of program memory. The reason for the change was to make the node more versatile as a development platform in the RUNES research.

3.3.2 IEEE 802.15.4/ZigBee

As stated in the RUNES requirements, the node must be able to communicate using the IEEE 802.15.4 protocol. To make the node more interesting from connectBlue's perspective, communication using ZigBee must also be supported.

All available IEEE 802.15.4/ZigBee solutions are based on two parts, the radio hardware and a software communication stack. The software stack is supposed to run in a separate microcontroller, not in the radio chip. Ideally the stack would run in the main microcontroller. Unfortunately this was not as easy as expected due to different reasons explained below.

There are two stack solutions available, the Z-Stack [27] from Figure 8 and the EmberZNet [28] from Ember. Both stacks implement the upper layers in the OSI model. The physical and link layers are implemented in hardware in the radio chipsets. If no stack is used, it is still possible to support IEEE 802.15.4 communication. The two stacks can be combined with the available radio hardware chipsets available as follows.

The first radio chipset is the Freescale MC13192 [29], which can be used with the Z-Stack. Unfortunately, due to license reasons it seems, this combination can only be used with a Freescale microcontroller, and therefore is unusable in the non-Freescale based node.

The second radio chipset is the Chipcon CC2420 [30] which also can be used with the Z-Stack. This combination is bundled with Z-Stack port for Atmel 8-bit microcontrollers.

The last chipset is the Ember EM2420 [31] which is combined with the EmberZNet stack. The chipset is actually a re-branded CC2420. The stack is ported to 8-bit Atmel microcontrollers and the Texas Instrument MSP430 series, but no ARM7 port is available.

The lack of stack ports for the ARM7 architecture was a problem. This could be solved either by porting the a stack to the ARM7 architecture or

by using a separate 8-bit microcontroller to run the stack. None of the radio chipsets are bundled with the full source code for the stack, thus making a stack port impossible. Hence a separate 8-bit microcontroller to run the stack was required.

Freescale, Chipcon and Ember are planning a new generation of radio chipsets with a small integrated microcontroller that can be used to run the ZigBee stack. Such solution would be better, but the release of these chipsets are not scheduled within the time limit of this project. Both Chipcon and Ember are claiming that their next generation chipsets will be able to run the same code as their current solution with the separate 8-bit microcontroller, thus making these future proof solutions. For this project the Chipcon solution was chosen, based on the future proof solution and previous good experiences of Chipcon at connectBlue.

The CC2420 development kits and application notes from Chipcon are using the Atmel ATmega128 [32] 8-bit microcontroller to run the stack, so it seemed reasonable to use this in the node. The controller has 4 kbytes of RAM, 128 kbytes program memory and a lot of integrated I/O interfaces. It is supported by the GNU toolchain and avr-libc C library implementation [33]. It can be run at a maximum speed of 8 MHz. These resources are claimed to be enough to run the full Z-Stack and user applications. The user application in this case is an interface between the ZigBee stack and the UART to which the microcontroller is connected to the main microcontroller. This user application was to be written as a part of this project.

3.3.3 Bluetooth

One of the main prerequisites for this project was that connectBlue's Bluetooth and WLAN modules were used in the node. Because of this, no real evaluation or comparison was needed.

The Bluetooth modules from connectBlue are available in several different configurations. All the modules provide the same basic functionality. The modules are hooked up to the microcontroller using a UART providing a serial communication link. The module acts as a Bluetooth serial adapter, a Bluetooth profile providing serial cable replacement. Using AT commands the module can be configured to connect to other serial adapters. Data can then be exchanged over the link, just as if the peers were connected using a RS232 serial cable. The more feature-rich modules can be used to establish several simultaneous data links to different serial port adapters. These can also be configured to support direct control of the lower layers of the Bluetooth protocol, through the use of a serial control protocol.

The most basic module was used in the node, namely connectBlue's third generation Bluetooth module OEMSPA311i-04 with internal antenna [34].

Using that module it will be possible to establish a serial data link to other nodes or Bluetooth serial adapters. This link can then be used to exchange raw data.

3.3.4 WLAN

Compared to the Bluetooth module, the WLAN module requires more from the software. It is connected to the microcontroller using SPI. Every time the module is powered up, the firmware for the WLAN chipset on the module must be fed to the module.

The module also requires a software driver library to be run in the main microcontroller. This library then provides the interface between the WLAN module and the IP (Internet Protocol) stack run in the main microcontroller.

The WLAN module used was connectBlue's first generation WLAN module OEMWLAN211bi-04. The module supports IEEE 802.11b/g but is not yet commercially available, and only preliminary specifications exists [35].

3.4 Schematic

The four main hardware components have now been presented and evaluated. The hardware design is built upon the main microcontroller, the IEEE 802.15.4/ZigBee radio, the WLAN and the Bluetooth modules. These are wrapped together into the final hardware design. Some glue logic and helper circuits were also needed to get everything to work as desired. A block diagram of the design is shown in figure 3.1.

First, several connectors were added. The connectors are used to supply power to the node, microcontroller program memory programming, microcontroller debugging, connecting the Bluetooth and WLAN modules and to access to the general purpose inputs and outputs. All connectors except the Bluetooth/WLAN connector are simple pin headers because these are common, cheap and limited in size. The Bluetooth/WLAN connector that is used is a 40 pin connector to which a connectBlue module can be mounted with screws.

The pinout used for the program memory programming connectors of the two microcontrollers is more or less standardized. It is possible to program the flash memory of the two microcontrollers using different serial protocols. The Philips controller has an integrated bootloader that can be used to program the flash memory using one of the integrated UARTs. Only a signal level converter to convert between RS232 and logical levels was needed. This converter is located on a separate programmer PCB that can be connected to a PC serial port and to one of the previously mentioned connectors on the

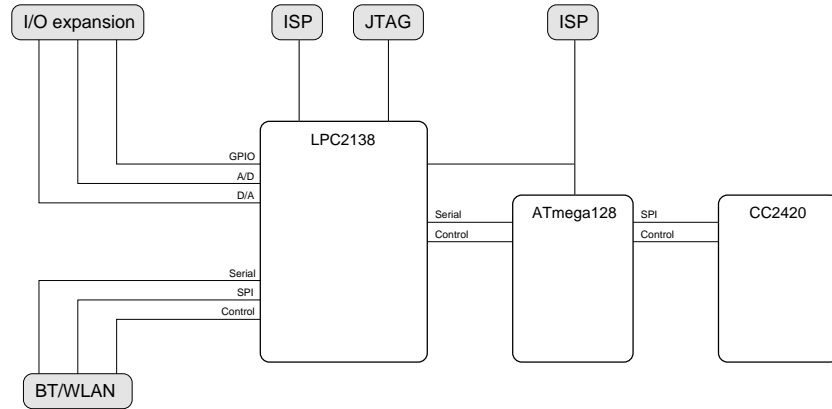


Figure 3.1: Block diagram showing the hardware design.

node. By not integrating the converter in the node, the physical size can be kept down. The pinout for the connector were taken from [36].

The 8-bit Atmel controller used to run the ZigBee stack is programmed using a serial protocol similar to SPI. The programmer that was used is very common and can be built for a few Euros. The programming signals are also routed to the main microcontroller. By doing so, it is possible to program the flash memory in the Atmel controller using special written software in the main microcontroller. No programmer for the Atmel controller is then needed.

In the power supply connector two extra pins were added which can be used to power the real-time clock in the main microcontroller from a separate battery. This makes it possible to disconnect main power and still keep track of time.

The JTAG (Joint Test Action Group) connector used to access the debugging port on the main microcontroller, uses a standardized pinout common in most hardware designs. By doing so, any JTAG debugger can be connected to the node.

The main power to the node is regulated by an on-board voltage regulator. The regulator ensures that the different components in the node have a stable 3.3 VDC supply voltage. Also two thermal fuses and a transient voltage protection are used, working as a polarization, over-current and over-voltage protection. The minimum input voltage to the voltage regulator is 3.8 V which makes it possible to power the node by four 1.2 V AA rechargeable batteries.

To indicate the status of the Bluetooth or WLAN module, an RGB LED (Red-Green-Blue Light-Emitting Diode) was added. The LED needs a supply voltage greater than the 3.3V regulated supply, and is thus connected directly to the input voltage. The LED is current controlled to get an even light emission independent of input voltage.

The design of the balun connected to the CC2420 radio chip was taken from the datasheet. The antenna output was then connected to a surface mounted 2.4 GHz antenna through an impedance matching network.

The reset signals of the two controllers are connected in a way that makes it possible to keep one of the controllers in a reset state during the programming of the other one. This is necessary because some of the signals between the controllers are also used for the serial programming, and would otherwise result in the programming signals being corrupt.

The node prototype PCB has four-layers to ensure simple and efficient signal routing. The PCB layout was made by an external CAD company. All the components are surface mounted, except for the pin header connectors.

Most of the components chosen, such as resistors, capacitors, crystal oscillators, the antenna and the voltage regulator, are components that is common in connectBlues own hardware designs.

Software design

The software design involved several steps. First an analysis of the functional requirements for the software design was performed. Based on this, a basic block based model was made, describing the different software components and how they are related to each other.

These blocks were then further split into pieces to determine a detailed model of how they are supposed to work. When this was done, all the blocks were implemented and then integrated with each other. During the implementation, necessary testing was performed.

4.1 Requirements

The main functionality of the node is to provide a data bridge between Bluetooth/WLAN and ZigBee protocols. Exactly how this can be done is discussed in section 4.2.

The RUNES requirements states that a minimal RTOS (Real Time Operating System) supporting interrupts, threads and basic messaging and scheduling should be used. Software written can then take advantage of the OS functionality in the RTOS, such as threaded execution and exchanging messages between processes.

The communication with the ZigBee microcontroller and the Bluetooth module requires serial communication using the UART hardware in the main microcontroller. Therefore, a driver to interface the hardware in a useful way was necessary.

To interface the WLAN hardware, the SPI port of the main microcontroller must be used. Because the WLAN part of the project is low prioritized, no time was spent on designing and implementing this driver.

Furthermore, the software for the ZigBee microcontroller needed to be implemented. The idea was to minimize the need to write software for this controller, instead using available application examples.

All this should, as part of project, be combined into a proof-of-concept demonstration application, showing how data can be exchanged using the different supported wireless technologies. A good way to demonstrate a working ZigBee network is to set up a number of devices running the ZigBee HCL profile acting as switching load controllers (lamps) and switch remote controls (switches).

4.2 Bridging

As an extension of the ZigBee standards, work is being done on how to bridge data between the IP protocol and the ZigBee protocol. That work aims to create a generic way to route data from an IP network to ZigBee devices using a gateway. Unfortunately this work is far from finished and can at best be used as inspiration. Based on this work and own ideas, two bridging techniques were considered. They are describe below.

One way to perform the bridging is to implement a custom designed protocol for sending and receiving commands to the node. This works as follows.

The bridging node listens to incoming requests on a specified TCP (Transport Control Protocol) port. When an incoming request arrives, it is processed and eventually an answer is generated and sent back to the IP host. The exact design of the protocol to use is not obvious. One way is to use a protocol similar to the serial debug protocol used in congestion with the ZigBee stack (see section 4.4.5).

A few other things must also be taken in consideration, such as how to handle non-instant reply data from the bridging node. For example, a request from an IP host results in a data message being sent to the ZigBee network. The bridging node expects an answer to that message, but the answer never arrives due to network problems. Then some kind of timeout message must be generate and sent back to the IP host instead of the requested message reply from the ZigBee network. During this operation, the TCP connection between the IP host and the node must be kept open. This can be implemented as a “session”. The IP host connects to the bridging node and a session is created. This session is then active until either the IP host or the node requests it to be closed. This is depicted in figure 4.1.

This method, using a custom designed protocol, is probably the most versatile and generic one. It can be implemented in a clever way, supporting exactly the features needed. Unfortunately this method requires the IP hosts and the bridging node to have a complete interface for sending and receiving commands using the custom designed protocol. Things like IP network problems must also be handled by the protocol.

Another approach is to let the the bridging node be aware of what data

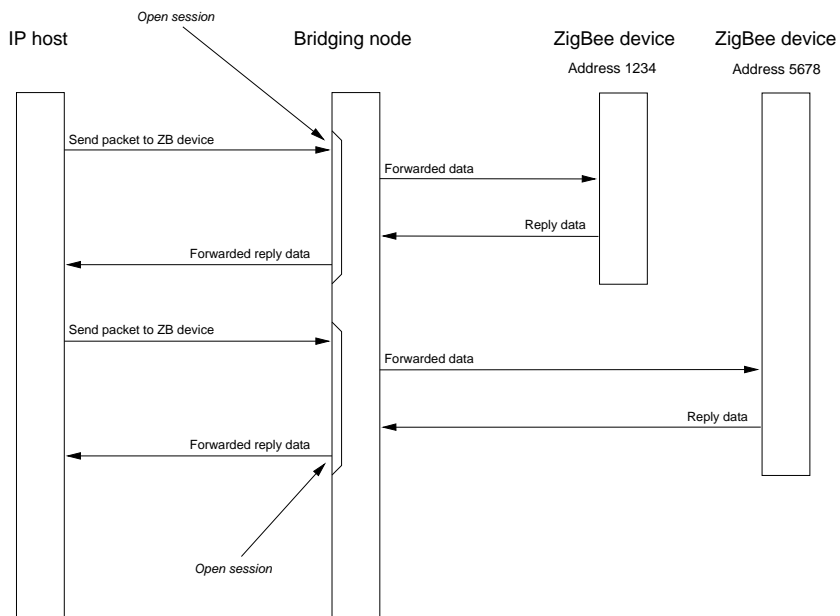


Figure 4.1: Bridging between an IP host and ZigBee devices using sessions and a custom designed protocol.

it sends and gets from the ZigBee network. IP hosts send data to the node, which translates it to appropriate ZigBee data packets and sends them to the ZigBee network. If an IP host wants to read data from a ZigBee device, a request to the bridging node is done. The node then requests and acquires the data from the ZigBee network and saves this data locally. The IP host then fetches the data from the bridging node.

One way to implement this is to let the bridging node act as a web server, and use a web browser to interact with the node. The web server implements CGI (Common Gateway Interface) functionality to generate dynamic web pages. By accessing different files on the web server, the IP host can get and set data on ZigBee devices. For example, one file returns a list of available devices in the ZigBee network, and another can be used to send specific data to a specific ZigBee device. This is shown in figure 4.2.

This method offers less flexibility. The web server running on the bridging node must parse incoming web requests and translate them to appropriate ZigBee packets. For each type of command that is supported, some kind of translation must be implemented.

Also, because of the way the HTTP (HyperText Transfer Protocol) protocol is working, there is no easy way to push data from a web server to an IP host. If an IP host requests data from a ZigBee device, and the actual request of data from the ZigBee device is not instant, there is no good way to notify the IP host when the requested data is available. The IP host must poll the web server to know this.

One of the large benefits using this web server approach is that the IP host does not need to be aware of the ZigBee protocol at all. By only using a web browser, it is possible to fully interact with ZigBee devices as long as all the requested functionality is implemented in the web server.

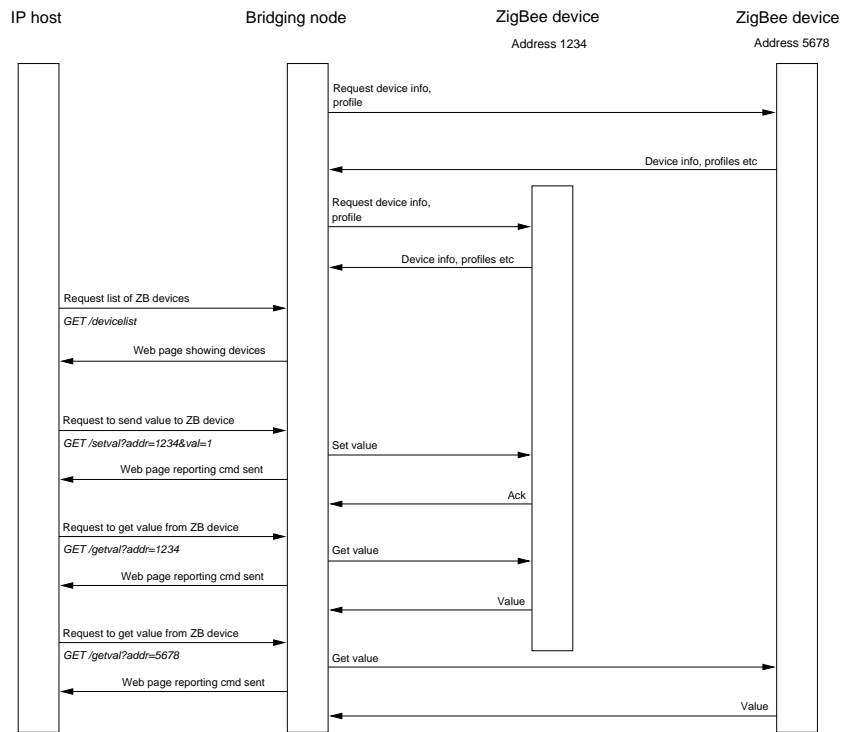


Figure 4.2: Bridging between an IP host and ZigBee devices using a web interface.

The two methods suggested above only works when bridging between ZigBee and IP based networks. An easy way to provide the same functionality using Bluetooth is to use a Bluetooth serial communication channel to exchange IP data using PPP (Point to Point Protocol). This requires the two

hosts to be able to communicate using IP over PPP. In one end of the Bluetooth serial link is the bridging node, in the other a networked host able to bridge between IP and PPP.

The easiest way to achieve this is to use a PC running Linux. Linux has built-in support for IP networking and PPP. The bridging node connects to the Linux host using PPP over a Bluetooth serial link. The Linux PC acts as a bridge between the IP network it is connected to, and the bridging node.

Considering the limited time available to study and fully understand the ZigBee hardware and software, the bridging method using HTTP seemed the most feasible. As described in the hardware section, the ZigBee stack was be run in an Atmel 8-bit microcontroller. By choosing the HTTP method, the protocol used between the Atmel microcontroller and the main microcontroller could be kept simple, supporting only a few ZigBee operations such as querying available devices for their addresses and what profiles they support. This decreased the time needed to be spent on developing software for the Atmel microcontroller.

4.3 Development tools

To implement and test the software, two development kits were used. For the main microcontroller software, the IAR KickStart Kit for Philips LPC2138 [37] was used. The kit contains a lab board featuring a Philips LPC2138 microcontroller, two serial ports routed to the UARTs of the microcontroller and some peripheral connectors. The program memory of the microcontroller can be programmed using either the JTAG debug connector or one of the serial ports. During the development, the serial port was used.

In the hardware design, the two UARTs of the microcontroller are used to connect it to the Bluetooth module and the ZigBee microcontroller. The connection to the Bluetooth module requires hardware flow control, which none of the serial ports on the lab board supported. Thus the board had to be modified to support this. The modification was a simple matter of adding another RS232 line driver for the extra signals needed for the flow control.

Software development was done using the GNU toolchain run in Cygwin [38] under Microsoft Windows 2000. The newlib version used was slightly modified by connectBlue to handle target specific things such as standard input and output.

Real-time debugging was done using a Lauterbach JTAG Debugger [39] in combination with the Lauterbach TRACE32 debugging environment. This was connected to the microcontroller using the JTAG debug connector. When connected, it enables real-time debugging of the code executing in the microcontroller. All standard debugging features are supported, such as break-

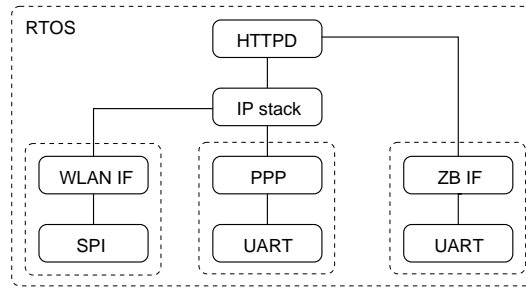


Figure 4.3: Block diagram showing the software design.

points, line by line code stepping and writing and reading to and from the different memories and I/O ports of the microcontroller.

For ZigBee development, the Chipcon CC2420 ZigBee DK Development Kit [40] was used. The kit contains five lab boards, a ZigBee radio sniffer board, the Figure 8 Z-Stack and tools for developing software for the microcontroller on the lab boards. The lab boards consist of the same 8-bit microcontroller and radio chipset used in this project’s hardware design. The boards feature a serial port that can be used to connect them directly to a PC or, as in this case, to the main microcontroller lab board.

The ZigBee development kit uses WinAVR [41], a collection of open source software development tools for the Atmel AVR microcontrollers run on the Windows platform. WinAVR includes GNU `avr-gcc` [42] and the `avr-libc` library. Real-time debugging was done using Atmel AVR Studio 4 [43]. AVR Studio is a complete IDE (Integrated Development Environment) for software development for the AVR microcontrollers.

4.4 Component design

The block model describing the suggested software is shown in figure 4.3. The model can be divided into three main blocks; The Bluetooth block, the WLAN block and the ZigBee block.

The Bluetooth and WLAN blocks consist of drivers for Bluetooth and WLAN respectively. The driver provides the necessary interface between the communications hardware and the IP stack. If Bluetooth is used, a UART driver is necessary together with a PPP interpreter that enables PPP communication over the serial Bluetooth link. If WLAN is used, an SPI driver and a driver for the WLAN module is needed.

The Bluetooth block depicted, uses the Bluetooth link as a serial cable

replacement. The Bluetooth link could instead be replaced with a cable. A better way to do this, is for example to use the PAN Bluetooth profile for the IP communication. The PPP interpreter would then be replaced by a Bluetooth interface which controls the Bluetooth module by accessing the lower layers of the Bluetooth protocols (see section 5.2.1).

An IP stack is plugged into either the Bluetooth or the WLAN driver depending on what module used. The IP stack enables communication using the TCP/IP protocol. For each application specific protocol encapsulated in the TCP/IP protocol, a support application must be written that implements the protocol used. For example, to use the HTTP protocol, the supporting application acting as a web server must be written. Its purpose is to interpret and generate valid HTTP protocol packets. These packets are then encapsulated in TCP packets.

Similar to the Bluetooth/WLAN drivers, a driver to interface the ZigBee hardware is necessary. A UART driver enables serial communication with the ZigBee microcontroller. The ZigBee stack running in the ZigBee microcontroller is controlled by sending commands over the serial link. The control protocol to use was not specified, and needed to be defined and implemented.

The bridging functionality is provided by the web server application which communicates with the ZigBee stack via the ZigBee interface in the main microcontroller.

The software run in the ZigBee microcontroller is not shown in the figure. This software is described in section 4.4.8.

The different blocks described are implemented as several separate tasks running in parallel. The RTOS is responsible for scheduling and execution of these tasks, as well as passing messages between them. Before the different blocks are described more in deep, the choice of what RTOS to use will be described.

4.4.1 Real-time operating system

There are a lots of RTOSes available on the market, ranging from complex commercial ones to tiny open source variants. A few different RTOSes were compared to find the one best fitting the requirements. Because of the academic nature of the RUNES project, a free RTOS with source code available was preferred. The different RTOSes are commented below.

eCos is a royalty-free, open source RTOS that fits the requirements. A working port for the ARM7 architecture is available [44].

uC/OS-II is a small RTOS provided by Micrium. It fits the requirements and a port for the LPC2136 controller is available. It is only free for

non-commercial, educational use, thus making it less interesting to use in the node [45].

XMK is an open source RTOS designed to run on small microcontrollers.

Because of this, the RTOS is a bit too simple for this application [46].

Contiki is a small operating system designed specifically for small resource constrained microcontrollers. This is too minimalistic for the application [47].

FreeRTOS is an open source, free to use RTOS that fits the OS requirements. The documentation is straight forward and a port for the Philips controller and lots of demos are available [48].

There are several more free RTOSes available on the Internet. The ones listed above were the most interesting ones. eCos and FreeRTOS seems very similar regarding functionality and documentation.

For eCos, there are a lot of third-party components available, providing useful extra functionality. FreeRTOS lacks these third-party components, only providing basic RTOS functionality. However, FreeRTOS seems to be the more minimalistic of the two, therefore the one best fitting the RUNES requirement for a minimal RTOS.

The FreeRTOS source code includes ports for different CPU architectures and compilers. This includes a port for the ARM7 architecture that can be compiled using the GNU toolchain.

Applications can be written directly for FreeRTOS, using the basic operating system API (Application Programming Interface) provided by FreeRTOS. The drawback is that all applications then written only works on FreeRTOS.

One way to get better portability is to write the applications for a virtual operating system. A virtual OS can be described as a middle layer between an application and an OS. The virtual OS provides the same basic OS API as a real OS. The API calls from the application are translated by the virtual OS to API calls supported by the underlying OS. By doing this, only the virtual OS needs to be ported when the underlying OS is changed. All applications written can be reused without modifications.

connectBlue has its own virtual OS that is ported to several different OSes, including the FreeRTOS ARM7 port. There was no good reason not to use the companys virtual OS and FreeRTOS port when implementing this project's software. By doing this, it is easy to reuse the code within the company. The API of connectBlue's virtual OS is very similar to the API provided by FreeRTOS, making it easy to modify the project's software to native FreeRTOS applications if necessary.

The most recent version of FreeRTOS is 3.2.1, but connectBlue's virtual OS port is for version 2.5.3. When comparing this version with the latest

version, the major differences found were API modifications that does not affect the functionality. Therefore, no time was spent on modifying the virtual OS to use the latest version of FreeRTOS.

The behavior of FreeRTOS can be configured in several ways. Things like overall behavior, memory management and stack sizes can be controlled. The default settings were used except for memory management, which was configured to use the standard memory allocation functions provided by newlib. The stack sizes were first set to be large. The sizes were then trimmed down when the test applications were running. By monitoring the writes to the different stacks, it was possible to set the size just above what is necessary.

4.4.2 UART driver

The virtual OS API provides a software interface to utilize the UART independent of what underlying OS and hardware the virtual OS is running on. Because the UART is a hardware component, a driver needed to be written to support the specific UART hardware in the microcontroller used. Exactly how this driver should work could be derived from the different UART drivers already implemented for other virtual OS ports.

The general functionality of the driver is to provide an easy way to send and receive data over a serial link. The virtual OS API specifies a number of functions which can be used to initialize and configure the UART, read a certain number of bytes from the UART and send a number of bytes to it.

The driver was implemented as a separate process in the virtual OS. When an application process wants to send or receive data, it notifies the UART process about this. The UART process processes the request and notifies the calling application when the request is finished.

To ensure data integrity, it is essential that the incoming UART data is buffered. In this project the UARTs was used to exchange PPP data. Because IP and therefore PPP is packet based, data arrives in bursts of one or more packets. A complete packet must arrive before it can be processed. The maximum packet size used is approximately 1700 bytes. The receive buffer was implemented as a 2000 byte circular buffer. The driver continuously reads incoming data and writes it to the buffer. If no application has read any data from the UART driver and the buffer is full, an overflow occurs. Incoming data is then discarded until there is enough room in the buffer to fit the incoming data.

The UART driver implements hardware flow control using the RS232 CTS/RTS lines. When the incoming buffer starts to fill up, data flow off is signalled to the sender which stops the transmission until the buffer data is consumed. Similarly, the driver pauses sending data if the other end signals flow off.

The dedicated UART hardware in the microcontroller is used to speed up data transmissions. The UART hardware consists of a 16-byte transmission FIFO (First In First Out) and a 16-byte receive FIFO. The status of the FIFOs is signalled using hardware interrupts. The UART hardware is configured to send/receive data to/from these FIFOs at the desired serial line speed.

Unfortunately the 16-byte FIFOs somewhat limits the efficiency of the UART driver implementation. When receiving data at high speed, a lot of interrupts are generated. Each interrupt needs pre-emption, generating a lot of code overhead. Also the code for transmitting small chunks of data is inefficient because it is impossible to read out FIFO fill rate.

The UART driver is tested at a speed of 115200 kbit/s and works pretty well at this speed even though one byte sometimes is lost when receiving really large chunks of data. This should not be a problem in this project.

4.4.3 PPP implementation

The PPP implementation is a modified version of an implementation for Nintendo GameBoy Advanced made by Adrian O'Grady [49]. It supports the PPP protocols LCP (Link Control Protocol), CCP (Compression Control Protocol), IPCP (Internet Protocol Control Protocol) and encapsulated IP. No authentication is supported, though it would not be hard to implement support for PAP (Password Authentication Protocol) or CHAP (Challenge-Handshake Authentication Protocol). During the development, the lab instructions for the Computer Communication course at the Department of Communication Systems at LTH [50] and the book TCP/IP Illustrated, Volume 1 [51] were of great help.

A separate operating system process takes care of the PPP communication. Because of the close interrogation between PPP data and the IP stack, the stack is also a part of this process.

The implementation was developed and tested against the Linux PPP implementation. Some modifications are probably necessary to get it to work with other PPP implementations.

The connection establishing is driven by the remote peer (in this case the Linux machine). It initiates the connection by sending the appropriate LCP packets to the PPP process, which decodes the packets and generates a response. Because of the limited support for compression and other fancy PPP features, a few packets are exchanged during the connection parameter negotiation. When the PPP link is up, the remote peer initiate the negotiation for the IP protocol. During this phase, the PPP process receives the IP address to be used with the IP stack.

When this is done, the PPP link is set up and it is possible to exchange IP data over it. Incoming IP data packets are decoded and are fed directly to the

IP stack. The decoding involves identifying the packet length and removing of the escape characters and the PPP packet header. Outgoing data from the IP stack is encoded in the reversed manner.

One thing worth mentioning about the PPP implementation is how it uses one of the two UARTs. During UART writes, i.e. when sending a PPP packet, the PPP driver is put in a wait state until all the data is written. This is not a problem, because incoming PPP packets are still buffered by the UART driver. Also, more data is sent than received, due to the nature of the web server. A HTTP request is small compared to the web page sent as a reply.

There are a few known issues with the PPP implementation. It is somewhat sensitive to in what order the initial PPP packets are fed to it. It works fine with the Linux PPP implementation though. Also, packet fragmentation is not very thoroughly tested. This should not be a problem because the maximum IP packet size is limited to 1500 bytes by the IP stack, which is the same as the default maximum data payload size of a PPP packet.

4.4.4 SPI driver and WLAN interface

As previously mentioned, WLAN support was not prioritized. Still, a few things can be mentioned about the design of the software interface needed to get WLAN to work.

A driver that enables communication using the SPI interface of the microcontroller is needed. The design of such driver can probably be similar to that of the UART driver with the exception of the large buffer for incoming data. This because all transfers on the SPI bus are managed by the microcontroller, acting as a master. No incoming data arrives without the master asking for it.

The WLAN module uses an external interrupt line to notify the microcontroller that it requires attention. The WLAN interface must be written to utilize this external interrupt.

The WLAN module firmware needs to be downloaded to the module before it can operate properly. The firmware consists of a binary file that is stored in the program memory of the main microcontroller. At startup, the firmware is sent to the WLAN module over the SPI bus.

4.4.5 ZigBee interface

During the hardware design, it was clear that the ZigBee hardware needed a separate microcontroller to run the ZigBee stack. This stack and the software running in the ZigBee microcontroller is described in section 4.4.8.

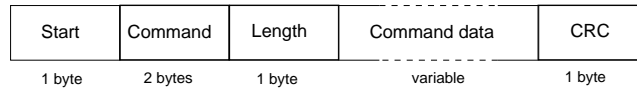


Figure 4.4: Packet format used in the communication between the main microcontroller and the ZigBee microcontroller.

The idea was to come up with a simple serial protocol to use for communication between the two microcontrollers. Because of the choice to use the web server approach for the bridging functionality, the protocol only needed to support limited command sending and data exchange between the microcontrollers.

When studying the documentation of the ZigBee stack it became clear that a serial debug protocol already was implemented. The debugging protocol [52] was supposed to be used to control different operations of the ZigBee stack from a PC during development. This debug protocol fully supports all necessary ZigBee operations needed to implement the desired bridging functionality of the node. In this application the main microcontroller replaces the PC otherwise used with the debug protocol.

Commands are sent as packets over a serial line at 38.4 kbits/s. No flow control is used, but that should not be a problem at this low speed. Furthermore, the protocol itself does not require much data to be sent over the line.

Each packet consists of a start byte, two command bytes, a byte indicating the length of the packet, the packet data and a checksum byte. The packet structure is shown in figure 4.4. The checksum is calculated as an XOR of all the bytes in the packet except the start byte.

An example of the data exchange between the main microcontroller and the stack in the ZigBee controller is shown in figure 4.5.

The ZigBee interface implemented consists of a UART driver for the serial communication and a state machine sending and receiving all necessary commands to and from the ZigBee stack.

Before initialization of the ZigBee interface, the ZigBee microcontroller is reseted. The state machine then waits for a reply from the ZigBee stack indicating that the reset was successful and that the stack is running properly. A number of commands are then executed to set up the stack.

Then device and service discovery is performed. A list of available devices in the network is created, holding information about device addresses, the state of the devices and what profiles they support. The discovery is done once at startup, and when requested by a user via the web interface.

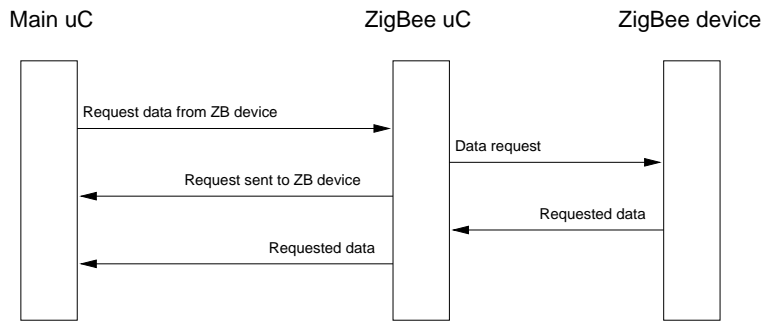


Figure 4.5: Data exchange between the main microcontroller and the ZigBee microcontroller.

The device and service discovery is performed as described in the ZigBee standard [10]. Pseudo code describing the discovery is shown below:

```

/* Get the IEEE address of the coordinator and the network
   address of all the associated devices */
(ieeeAddr, @assoc) = IEEEAddressRequest(0x0000, 1)

/* Add the addresses to the list of devices. The coordinator
   is neither light load controller nor switch */
devList.add(0x0000, ieeeAddr, NONE);

/* Iterate through the devices associated with the coordinator */
foreach (@assoc as dev)
{
  /* Only get the IEEE address of the device */
  address = IEEEAddressRequest(dev, 0)

  /* Add the addresses to the list of devices */
  devList.add(dev, address, NONE);

  /* Does the device have any endpoints using the Home Control,
     Lightning profile (ProfileID=0x0100)? */
  @eps = MatchDescriptorRequest(dev, 0x0100);

  /* Iterate through the endpoints found */
  foreach (@eps as ep)
  {
    /* Get a list of clusters associated with the end point */
    (nbrIn, @inc, nbrOut, @outc) = SimpleDescriptorRequest(dev, ep);
    /* If the endpoint has exactly one in cluster with id == 0x13
       the endpoint is a Switching Load Controller (a lamp) */
    if (nbrIn == 1 && @inc[0] == 0x13)
  
```

```
    {
        devList.update(dev, SLC);
    }
    /* Else if only one out cluster with id == 0x013, the endpoint
       is a Switch Remote Control (a button) */
    else if (nbrOut == 1 && @outc[0] == 0x13)
    {
        devList.add(dev, SRC);
    }
}
}
```

Every ten seconds all discovered devices are contacted to verify that they still are available. If a device does not respond, this is noted in the device list.

From the web interface, it is possible to set and get the status KVP value from a ZigBee device that supports the HCL profile. To do so, the web server calls the set or get ZigBee interface function, which in turn sends the appropriate control packet to the ZigBee stack. The only two operations implemented is reading the status value from lamps and buttons, and writing a value to a lamp. When a read is requested, the returned value from the ZigBee device is saved in the device list.

4.4.6 IP stack

There are several free open source IP stacks available, such as the ones shipped with FreeBSD [53] and OpenBSD [54] and the lwIP [55], requiring more or less hardware resources. The uIP [56] stack written by Adam Dunkels at SICS (Swedish Institute of Computer Science) is an extremely slimmed down IP stack requiring less than hundred bytes of RAM excluding buffers, and runs on 8-bit CPUs. Even though there are hardware resources to run a less slimmed down stack, the author has previous experience with uIP and therefore it was chosen for this project.

The uIP version 0.9 was used in this project. The only modifications made to the stack was that the buffer used when sending data is shared between the PPP driver and uIP. This minimizes the amount of data copied between different buffers and also lowers the amount of RAM needed for buffers.

Because the stack is fully event driven, it does not need to be run as a separately OS process, but can be called only when there is new incoming data. Decoded incoming IP data is sent to uIP that processes the data and eventually generates reply data. This data is encapsulated by the PPP driver and sent back over the PPP link. Depending on what kind of protocol an incoming IP packet contains, the packet is handled in different ways.

Incoming ICMP (Internet Control Message Protocol) data packets are processed directly by the uIP stack which is capable of generating ICMP echo replies.

If an incoming IP packet contains TCP data addressed to a TCP port number that has an application associated to it, the data is sent to that application. In our case, the web server application is associated with TCP port 80. The incoming HTTP requests are processed and replys are generated according to the HTTP protocol standards [57].

To handle certain time dependent functions of the IP stack, such as refreshing caches, a timer in the PPP process is used to call the appropriate functions of the IP stack every half second.

4.4.7 Web server

The web server application used is a slightly modified version of the web server demo application distributed with uIP. It supports a rudimentary static file system and simple CGI functionality to generate dynamic web content. The following modifications have been done to the web server:

- Necessary static HTML (HyperText Markup Language) files were added to the filesystem.
- A few CGI functions were added to generate dynamic web page content for listing ZigBee devices and to read and write values to specific devices.
- The CGI support in uIP was extended to support passing of data variables from a web browser using the file request string, also known as the GET method [57].

The result is a web server that presents custom web pages to a client accessing the node using a web browser. The content on some of the pages are dynamically generated. The ZigBee devices accessible from the node, together with their type and status are listed. It is possible to set and get values from the devices supporting the HCL ZigBee profile, see also section 2.2 and [11].

The integration between the web server and the ZigBee interface is far from perfect. The device list in the ZigBee interface process is used to exchange data between the two processes. Because these are two separately running processes that shares the same data, collisions can occur when accessing the data. The web server only reads data from the device list, thus limiting the problem to reading wrong values when these are updated. If this happens, it is a matter of reloading the web page to get the right values. This concurrency problem can be solved in several ways, but no time was spent on it.

4.4.8 ZigBee microcontroller software

The intention was to spend as little time as possible on implementing the software for the ZigBee microcontroller. This actually proved to be easy.

Several application examples are bundled with the Z-Stack, including fully functional implementations of the SRC (Switch Remote Control and SLC (Switching Load Controller) ZigBee devices. This combined with the debug interfaced described in section 4.4.5 actually provides all the necessary functions needed to complete the demonstration application for the node.

Both the SRC and SLC application can be configured to be built as an end device, router or coordinator. The SRC application example, configured as a coordinator was used in the ZigBee microcontroller without any modifications. Because there is no need for the node to implement any ZigBee profile functionality, it was possible to totally strip the SRC functionality from the application example.

The SRC and SLC application examples were configured to use no encryption and simple device binding with indirect addressing. By using indirect device addressing [10], the ZigBee interface in the node can be kept simple. Simple binding and indirect addressing can be

described in the following way.

When a ZigBee device wants to send a message to another device, the message is sent to the network coordinator device. The coordinator is then responsible for passing the message to the destination device. To know what device the message should be passed to, the coordinator keeps a binding table. In this table, the network address and endpoints to bind are saved in pairs. All messages originating from a certain endpoint at a device with a certain address are checked against the binding table, and passed to the address and endpoint of the other device in the pair.

The binding table is updated by setting the two ZigBee devices that are to be bound, in binding mode. By doing so, the two devices notify the coordinator to update its binding table.

The contrary to indirect addressing is direct addressing. When direct addressing is used, a device which wishes to send a message to another device, sends the message directly to the destination device. This requires the sending device to know the network address of the destination address, which is not always the case for RFDs.

Results and discussion

In this chapter, the results of the project is presented and future improvements discussed.

5.1 Results

Most of this project has been performed during the summer holiday period, thus a few things have been delayed. One of the goals was to manufacture a working prototype of the designed hardware. Unfortunately, the start of the manufacturing of the prototype was one of the things delayed. The final prototype was finished the day before the presentation. After the presentation, the board has been tested to verify the functionality.

The result of the project is a working hardware prototype of the bridging node, partly running software implemented as part of the project. The node can be equipped with either a WLAN or a Bluetooth module. Because of the time limit of the project, and the fact that the WLAN module is not available yet, no software has been written to test WLAN communication.

The physical size of the node is 67x51 mm and the height is approximately 18 mm. It is possible to mount the PCB in a plastic housing with the dimensions 96x60x36 mm. This is a bit larger than the wanted size of a match box. The PCB could have been made smaller if some of the debugging and expansion pin header connectors were removed.

Only limited measurements of the power consumption of the prototype has been performed. During the measurements the supply voltage

were 5.8 VDC. The average power consumption with the Bluetooth module mounted was 107-113 mA. During radio transmissions the consumption went up but no accurate peak value were measured. Because the design uses a linear voltage regulator, the excessive voltage is turned into heat. If the input voltage is lowered, less heat is generated. If the node is powered by four AA batteries, this would result in a battery lifetime in the range of ten hours. This is lower than the wanted battery lifetime, but still acceptable. The lifetime heavily depends on what tasks the node is performing. If the microcontrollers in the node can be put into sleep mode, much better lifetime can be expected.

The node has thirteen digital I/O pins. Of these, four pins can be configured as analog inputs and one as analog output. Two of the digital I/O pins can also be configured to generate external interrupts, and two as an I²C (Inter-Integrated Circuit) bus.

It is possible to connect the real-time clock in the main microcontroller to separate power source such as a backup battery. By doing so, it is possible to keep track of time even if the node loses its main power.

The main microcontroller runs the FreeRTOS real-time operating system. On the OS, the demonstration application is run. The application is used to demonstrate bridging of data between different wireless communication technologies, in this case Bluetooth and ZigBee.

The bridging is done by using the Bluetooth as a serial link running IP over PPP. The node acts as a web server that is connectable via the PPP link. The node gathers information about available ZigBee devices and presents the information on dynamically generated web pages. A client connecting to the web server using a web browser can request certain data values to be sent to or fetched from a ZigBee device.

The Bluetooth support is somewhat limited when seen from RUNES perspective. As of now it only acts as a serial cable replacement. To be useful in the RUNES project, the lower layers of the Bluetooth protocols must be accessible.

The ZigBee devices used in the demonstration act as lamps and lamp switches that can be controlled wirelessly.

No work has been done to specify and implement software interfaces to interface the expansion ports and the communication hardware of the node.

As the node is still in the stage of a prototype, it is hard to tell anything about the final cost. A good guess is that the required cost of 30 Euros (excluding Bluetooth and WLAN modules) can be reached if the manufacture volume is high enough.

5.2 Discussion

5.2.1 Improvements

Several improvements of the design of the node are possible. The only reason for these not being implemented during the project is the lack of time. Possible improvements are listed below.

- The UART driver could be more optimized. As of now, it is split in two parts where one part implements the software interface specified by the virtual OS, and the other part interfaces the actual hardware. By combining these two, the driver would be less complex and more resource efficient.
- The ZigBee interface could be modified for better flexibility. One way to do this is to modify the serial protocol used between the main microcontroller and the ZigBee microcontroller. Also, the UART speed could be raised if higher throughput is necessary.
- Support for the WLAN module could be added. This is a matter of implementing SPI communication and modifying the WLAN driver provided with the WLAN chipset.
- Specify and implement generic software interfaces to utilize the communication hardware. This interface could provide functions to perform device searches, managing devices and sending and receiving data.
- Modify the ZigBee interface to manage binding. By doing so, it would be possible to manage a large network of ZigBee devices. Also, by enabling the security service functionality of the ZigBee stack, it is possible to tighten the security of the ZigBee network.

- Improve the Bluetooth support. The serial cable replacement profile could be changed to a more appropriate profile, such as the PAN profile. By doing so, IP data could be exchanged directly over the Bluetooth link. Unfortunately, the Bluetooth module used does not directly support these more complex profiles. Either the current module is reconfigured to run the Bluetooth stack in the main microcontroller, or the module is exchanged to a more feature rich one. If the Bluetooth stack is run in the main microcontroller, limited hardware resources may be an issue.
- Authentication could be added to the PPP driver. By supporting authentication, it would not be hard to modify it to communicate with a wider range of different remote PPP peers, such as the ones used in dialed-up connections. The node could then be used to set up a Bluetooth serial link to a GSM cell phone, use the modem in the phone to call an ISP (Internet Service Provider) and get a direct connection to the Internet.
- Modify the ZigBee microcontroller software to enable raw IEEE 802.15.4 communication. This would be a matter of removing the Z-stack and writing software capable of bridging data between the ZigBee chipset and the UART that connects the ZigBee microcontroller to the main microcontroller.
- Better ZigBee radio chipsets are now available from both Chipcon and Ember. The new chipsets feature a built in microcontroller, thus making the ZigBee microcontroller used in the project unnecessary. Still the ZigBee stack can not be run in the main microcontroller. Ideally, a ZigBee stack would be ported to the ARM7 architecture.

5.2.2 Future

It is hard to speculate in the future use of the node. The main purpose of the node, is to work as a development platform in the RUNES research. The prototype developed fulfills the hardware requirements stated in the RUNES specification. If the appropriate software interfaces to utilize the communication hardware is implemented, the node would be even more useful.

The ZigBee part of the project opens up a lot of possibilities. As of now, very few commercially available products uses ZigBee. Also, the exact future of ZigBee is unknown. Competing technologies are UWB (Ultra WideBand) for the radio part and IPv6 (Internet Protocol version 6) over IEEE 802.15.4. One of the reasons to potential success of ZigBee is that it is an open standard with a great number of large companies supporting it.

References

- [1] Lund Institute of Technology, <http://www.lth.se/> (September 2005).
- [2] connectBlue AB, <http://www.connectblue.se/> (September 2005).
- [3] RUNES, <http://www.ist-runes.org/> (September 2005).
- [4] RUNES brochure, http://www.ist-runes.org/docs/brochures/RUNES_brochure.pdf/ (September 2005).
- [5] Bluetooth.org - The Official Bluetooth Membership Site, <http://www.bluetooth.org/> (June 2005).
- [6] Bluetooth - Wikipedia, <http://en.wikipedia.org/wiki/Bluetooth> (June 2005).
- [7] Zigbee Alliance, <http://www.zigbee.org/> (June 2005).
- [8] ZigBee - Wikipedia, <http://en.wikipedia.org/wiki/ZigBee> (June 2005).
- [9] IEEE Standard for Information Technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific Requirements—, “IEEE Std 802.15.4-2003”, 2003.
- [10] ZigBee Alliance, “03525r6 ZigBee Application Framework Specification Revision 6, Version 1.00”, 2004.

-
- [11] ZigBee Alliance, “03540r6 ZigBee Application Profile Home Control, Lighting, Version 1.00”, 2004.
 - [12] Wireless LAN - Wikipedia, <http://en.wikipedia.org/wiki/WLAN> (June 2005).
 - [13] IEEE 802.11 - Wikipedia, http://en.wikipedia.org/wiki/IEEE_802.11 (June 2005).
 - [14] Universal asynchronous receiver transmitter - Wikipedia, <http://en.wikipedia.org/wiki/UART> (June 2005).
 - [15] RUNES, “D3.1 Requirement Analysis Report”, 2005.
 - [16] Universal asynchronous receiver transmitter - Wikipedia, <http://en.wikipedia.org/wiki/Microcontroller> (June 2005).
 - [17] GNU toolchain - Wikipedia, http://en.wikipedia.org/wiki/GNU_toolchain (September 2005).
 - [18] The Newlib Homepage, <http://sources.redhat.com/newlib/> (September 2005).
 - [19] Philips LPC2136 product information, <http://www.semiconductors.philips.com/pip/LPC2136FBD64.html> (July 2005).
 - [20] Philips LPC2106 product information, <http://www.semiconductors.philips.com/pip/LPC2106.html> (July 2005).
 - [21] Texas Instruments MSP430F1611 product information, <http://focus.ti.com/docs/prod/folders/print/msp430f1611.html> (July 2005).
 - [22] Atmel AT91SAM7S256 product information, http://www.atmel.com/dyn/products/product_card.asp?part_id=3524 (July 2005).
 - [23] Atmel AT91SAM7A3 product information, http://www.atmel.com/dyn/products/product_card.asp?part_id=3578 (July 2005).
 - [24] Analog Devices ADuC7020 product information, http://www.analog.com/en/prod/0,,762_0_ADUC7020%2C00.html (July 2005).

-
- [25] ST Microelectronic STR711FR1T6 product information, <http://www.st.com/stonline/products/literature/ds/10350/str711f.htm> (July 2005).
- [26] Freescale MCF5212 product information, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCF5212&nodeId=0162468rH3YTLC00M91436 (July 2005).
- [27] Figure 8 Wireless Z-Stack, <http://www.figure8wireless.com/Zstack.html> (July 2005).
- [28] Ember EmberZNet, <http://www.ember.com/products/software/zigbee.html> (July 2005).
- [29] Freescale MC13192 Product Summary Page, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13192&nodeId=0162468rH3YTLCFqnN (July 2005).
- [30] Chipcon CC2420 product information, http://www.chipcon.com/index.cfm?kat_id=2&subkat_id=12&dok_id=115 (July 2005).
- [31] Ember EM2420 product information, <http://www.ember.com/products/chips/em2420.html> (July 2005).
- [32] Atmel ATmega128 product information, http://www.atmel.com/dyn/products/product_card.asp?part_id=2018 (July 2005).
- [33] AVR Libc Home Page, <http://www.nongnu.org/avr-libc/> (September 2005).
- [34] connectBlue, E M Datasheet OEMSPA cB-0901, 2005
- [35] connectBlue, E M Datasheet OEMWLAN cB-0904, 2005
- [36] ARMuC Wiki: Standard ISP Header/Discussion, http://www.open-research.org.uk/ARMuC/index.cgi?Standard_ISP_Header/Discussion (July 2005).
- [37] IAR Systems KickStart Kit for Philips LPC2138, http://www.iar.com/index.php?show=35390_ENG (September 2005).

-
- [38] Cygwin Information and Installation, <http://www.cygwin.com/> (August 2005).
 - [39] Lauterbach BDM/JTAG Debugger, <http://www.lauterbach.com/bdm.html> (September 2005).
 - [40] Chipcon CC2420 ZigBee DK Development Kit, <http://www.chipcon.com/b2b/index.cfm?action=detail&id=1125&category=8> (September 2005).
 - [41] WinAVR, <http://winavr.sourceforge.net/> (September 2005).
 - [42] GCC Home Page, <http://gcc.gnu.org/> (September 2005).
 - [43] Atmel AVR Studio 4, http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725 (September 2005).
 - [44] eCos Home Page, <http://sources.redhat.com/ecos/> (July 2005).
 - [45] Micrium Products: uC/OS-II RTOS, <http://www.ucos-ii.com/contents/products/uc-os-ii-RTOS.html> (July 2005).
 - [46] SourceForge.net: Project Info - e(X)treme (M)inimal (K)ernel, <http://sourceforge.net/projects/xmk> (July 2005).
 - [47] The Contiki Operating System, <http://www.sics.se/~adam/contiki/> (July 2005).
 - [48] FreeRTOS - A Free RTOS for small embedded real time systems, <http://www.freertos.org/> (July 2005).
 - [49] GBA Web Server, <http://www.fivemouse.com/gba/> (August 2005).
 - [50] Lund Institute of Technology, Department of Communication Systems, Computer Communication (ETS052), PPP Lab Preparations, <http://www.telecom.lth.se/Kurser/ppp/preparations.htm> (August 2005).
 - [51] W. Richard Stevens, "TCP/IP Illustrated, Volume 1", pp. 33-51, 1994.

-
- [52] Figure 8 Wireless, “F8W-2003-00001 Z-Stack/Z-Tool Serial Port Interface”, 2005.
 - [53] The FreeBSD Project, <http://www.freebsd.org/>, (August 2005).
 - [54] OpenBSD, <http://www.openbsd.org/>, (August 2005).
 - [55] lwIP - A Lightweight TCP/IP Stack, <http://www.sics.se/~adam/lwip/>, (July 2005).
 - [56] The uIP TCP/IP Stack for Embedded Microcontrollers, <http://www.sics.se/~adam/uip/>, (August 2005).
 - [57] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1, <http://www.faqs.org/rfcs/rfc2616.html>, 1999.

Abbreviations and terminology

A.1 Abbreviations

API	Application Programming Interface
ARM	Acorn RISC Machine
CAD	Computer Aided Design
CAN	Controller Area Network
CCP	Compression Control Protocol
CGI	Common Gateway Interface
CHAP	Challenge-Handshake Authentication Protocol
CPU	Central Processing Unit
DMA	Direct Memory Access
EDR	Enhanced Data Rate
FFD	Full Function Device
FIFO	First In First Out
GNU	GNU is Not Unix
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
I ² C	Inter-Integrated Circuit
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol

– *list continued on next page* –

– list continued from previous page –

IPCP	Internet Protocol Control Protocol
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
JTAG	Joint Test Action Group
KVP	Key Value Pair
LAN	Local Area Network
LCP	Link Control Protocol
LED	Light-Emitting Diode
MAC	Media Access Control
OS	Operating System
OSI model	Open Systems Interconnection Reference Model
PAN	Personal Area Network
PAP	Password Authentication Protocol
PCB	Printed Circuit Board
PDA	Personal Digital Assistant
PLL	Phase-Locked Loop
PPP	Point to Point Protocol
RAM	Random Access Memory
RFD	Reduced Function Device
RGB	Red Green Blue
RISC	Reduced Instruction Set Computing
RTOS	Real Time Operating System
SICS	Swedish Institute of Computer Science
SLC	Switching Load Controller
SPI	Serial Peripherals Interface
SRC	Switch Remote Control
TCP/IP	Transport Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
UWB	Ultra WideBand
WEP	Wired Equivalent Privacy
Wi-Fi	Wireless Fidelity

– list continued on next page –

– list continued from previous page –

WLAN		Wireless Local Area Network
WPA2		Wi-Fi Protected Access

A.2 Terminology

Below a few concepts and abbreviations used in this report are explained.

FIFO *First In First Out* Can be seen as a queue holding data elements. Data elements can be added and removed. Elements added first is always removed first.

Hardware flow control A way to control the data flow over a serial communication channel. The receiving part can, using extra signalling pins, pause the transmission from the sender when the receive buffer in the receiver starts to fill up.

I²C *Inter-Integrated Circuit* A simple two-wire serial bus often used to connect different low-speed peripherals to a microcontroller.

IP *Internet Protocol* The most commonly used protocol in computer networks. Provides addressing functionality, making it possible to efficiently route data packets in large networks. Can be used on different hardware links, such as Ethernet.

IP stack A set of software components that implements the IP communication protocol and usually also the TCP protocol. The stack is used to send and receive data according to the IP protocol.

Microcontroller A microcontroller is a complete computer in one chip. It consists of a CPU, RAM, program memory and different I/O interfaces integrated in one chip. A microcontroller only needs a few or none external components to work [16].

PPP *Point to Point Protocol* A protocol used to set up a communication channel between two peers over a serial line. Different communication parameters can be negotiated. The channel can

then be used to exchange data using a variety of other protocols, such as IP. PPP supports several authentication protocols, such as PAP and CHAP.

RTOS *Real Time Operating System* An operating system specifically designed to efficiently schedule application execution to meet the timing requirements of the application.

SPI *Serial Peripherals Interface* A serial master-slave protocol/interface mainly used for communication between a microprocessor and a peripheral.

TCP/IP *Transport Control Protocol/Internet Protocol* A protocol used in combination with the IP protocol. TCP packets are encapsulated in IP packets. Provides functionality to ensure data integrity, resend lost packets and addressing of data to different application on the same IP host. Usually application specific protocols are encapsulated in TCP/IP packets.

UART *Universal Asynchronous Receiver Transmitter* A hardware device that translates between parallel and serial bits of data [14].

B.1 CPU comparison table

The *Power save* field describes the power save modes available on the microcontroller. The value given in the *Power consumption* field depends heavily on the frequency and power save mode used.

Due to page size restrictions, the CPU comparison table follows on the next page.

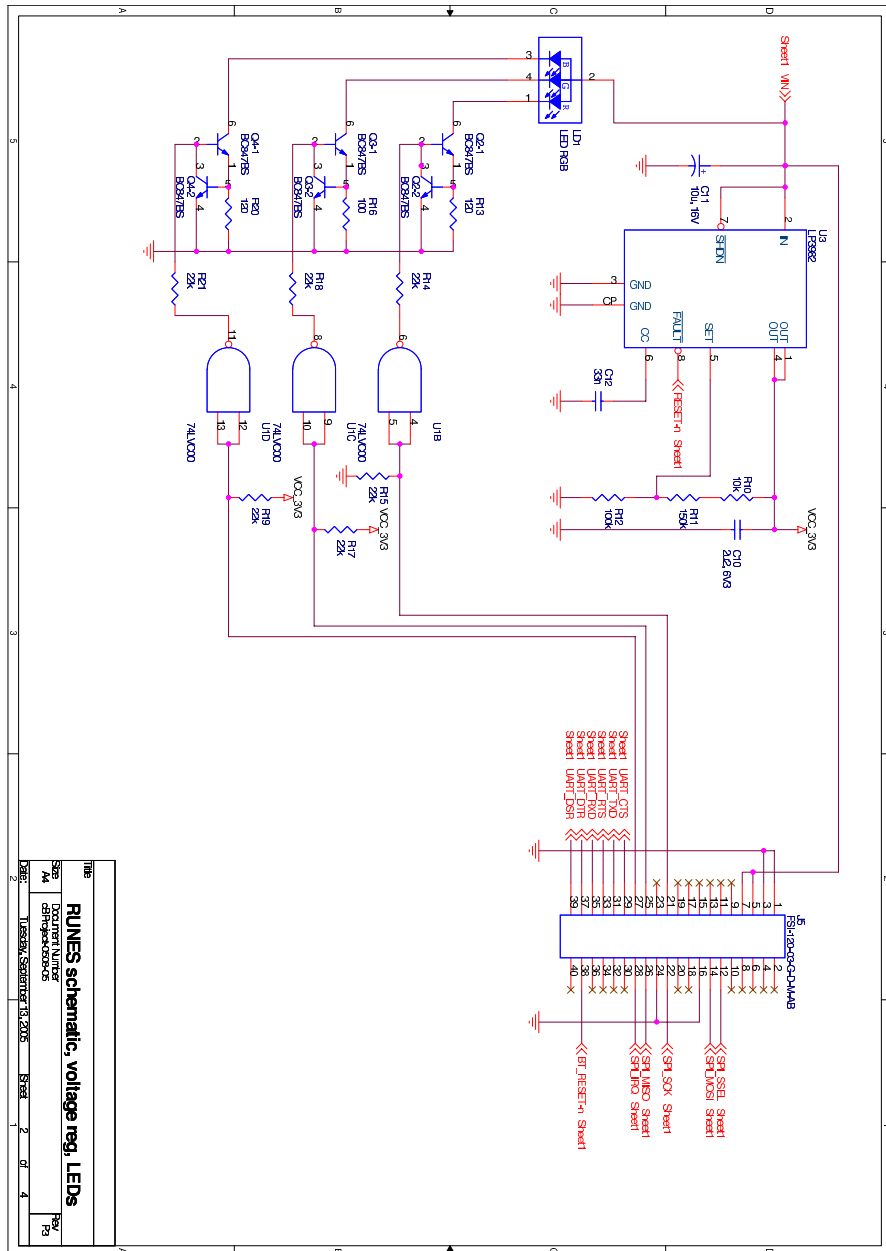
Model	LPC2136	LPC2106	MSP430F1611	AT91SAM7S256
Architecture	ARM7 (32-bit)	ARM7 (32-bit)	16-bit	ARM7 (32-bit)
Program memory	256 kB	128 kB	48 kB	256 kB
RAM	32 kB	64 kB	10 kB	64 kB
Max frequency	60 MHz	60 MHz	8 MHz	55 MHz
SPI / speed	2 / mclk/8	1 / mclk/8	2 / /mclk/2	2 / mclk/2
UARTs	2	2	2 (shared SPI)	2
GPIO	47	32	48	32
ADCs	16x10-bit	None	8x12-bit	8x10-bit
DACs	1x10-bit	None	2x12-bit	None
Timers	2x32-bit	2x32-bit	2x16-bit	3x16-bit
Other peripherals	RTC	RTC	None	USB, DMA
Supply voltage	3.3 V	3.3 V and 1.8 V	1.8-3.6 V	3.3 V
Power save	AC, ID, PD	AC, ID, PD	Several modes	Several modes
GCC	Yes	Yes	Yes	Yes
Power consumption	2.5-40 mA	2.5-40 mA	0.5-4 mA	5-35 mA

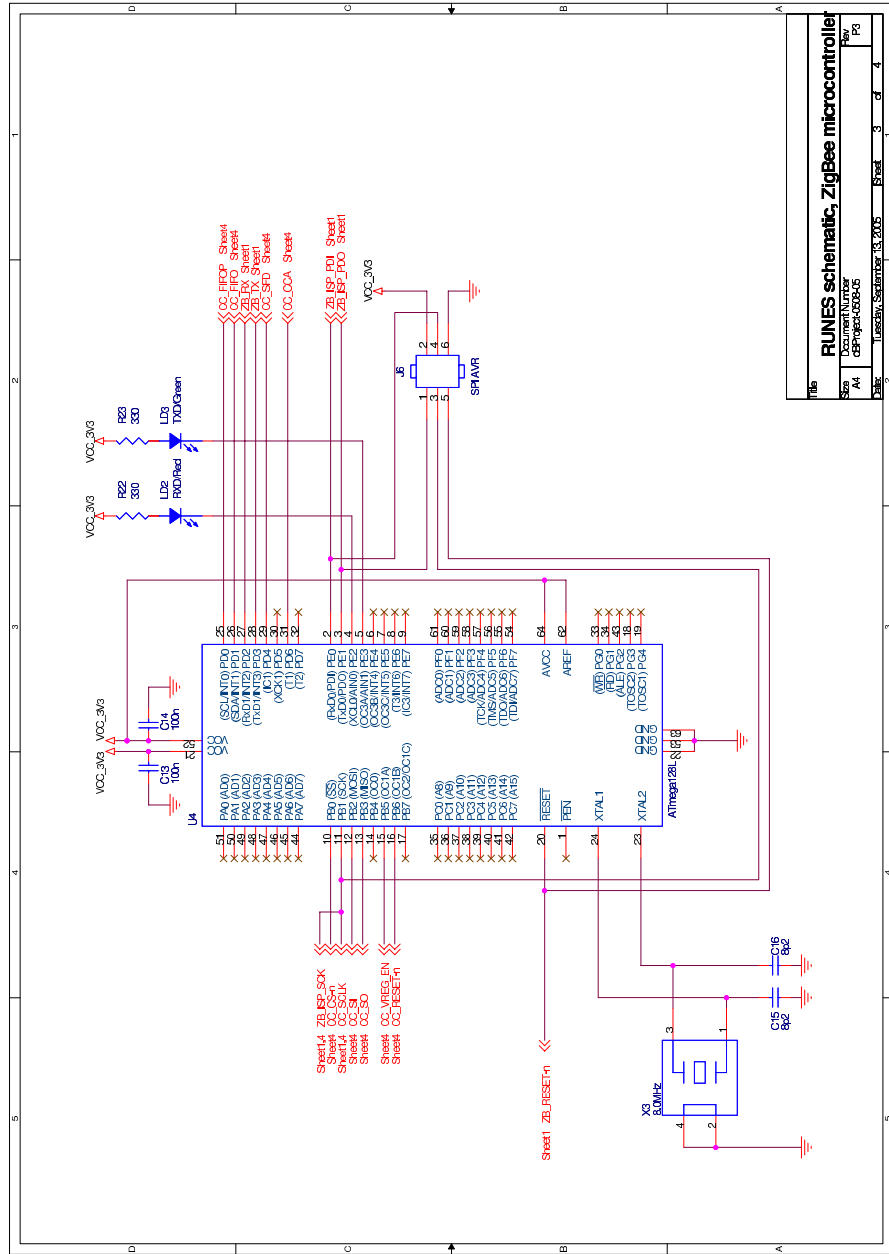
Model	AT91SAM7A3	ADuC7020	STR711FR2T6	MCF5212
Architecture	ARM7 (32-bit)	ARM7 (32-bit)	ARM7 (32-bit)	ColdFire V2 (32-bit)
Program memory	256 kB	62 kB	256 kB + 16 kB	256 kB
RAM	32 kB	8 kB	64 kB	32 kB
Max frequency	60 MHz	45 MHz	50 MHz	80 MHz
SPI / speed	2 / mclk/2	1 / mclk/8	2 / mclk/6	1
UARTs	4	1	4 (1 shared SPI)	3
GPIO	62	14	30	56
ADCs	16x10-bit	5x10-bit	4x12-bit	8x12-bit
DACs	None	4x10-bit	None	None
Timers	9	2x16-bit, 2x32-bit	5x16-bit	2x16-bit, 4x32-bit
Other peripherals	USB, CAN	Analog interfaces	RTC, USB	DMA
Supply voltage	3.3 V	3.3 V	3.3 V	3.3 V
Power save	Several modes	Unknown	AC, ID, PD	Several modes
GCC	Yes	Yes	Yes	Yes
Power consumption	<100 mA	<40 mA	3-55 mA	2-65 mA

Hardware details

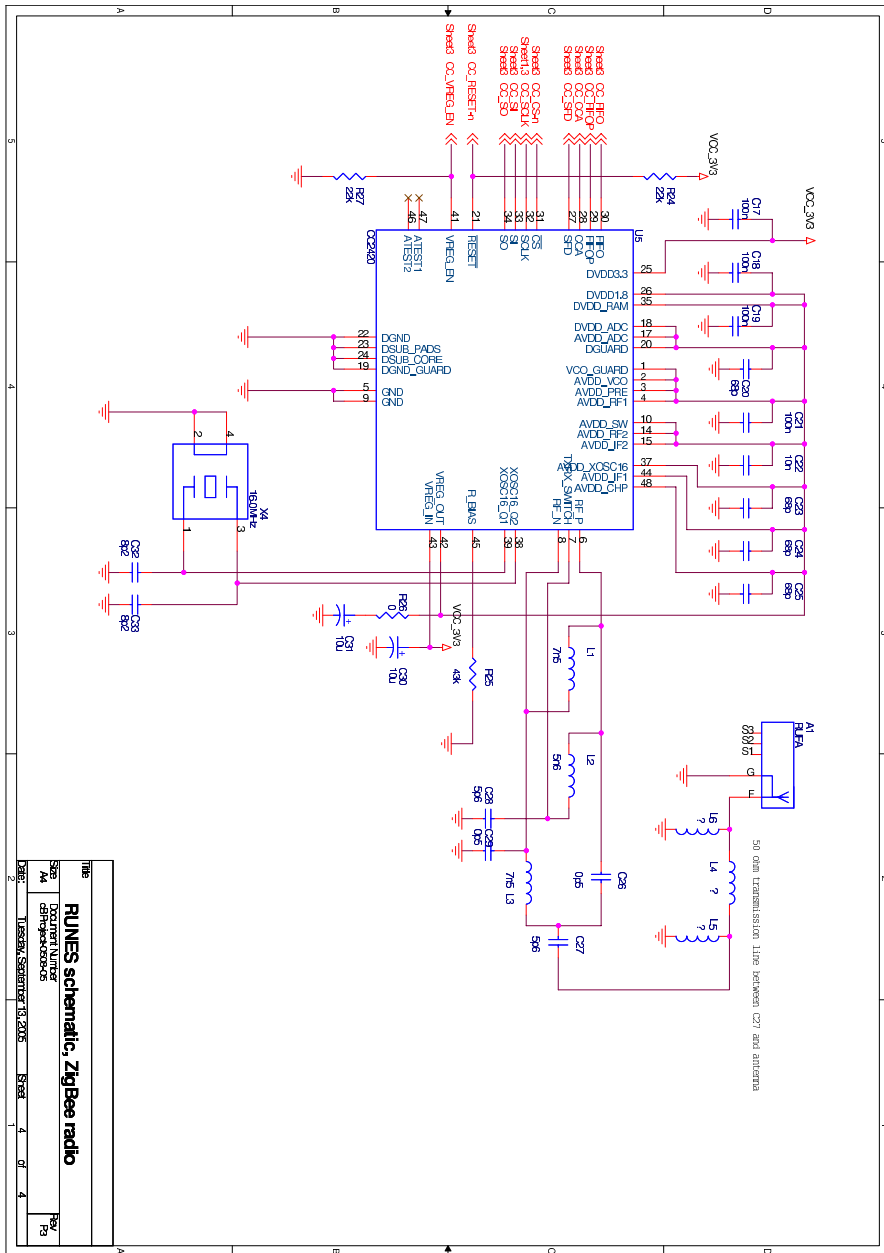
C.1 Schematics

Due to page size restrictions, the schematics follows on the next page.





RUNES schematic, ZigBee microcontroller			
Sheet	U4	U5	P3
DBZ	U4	U5	P3
DBZ	U4	U5	P3



C.2 PCB layout

The PCB layouts shown on the following pages are in actual size.

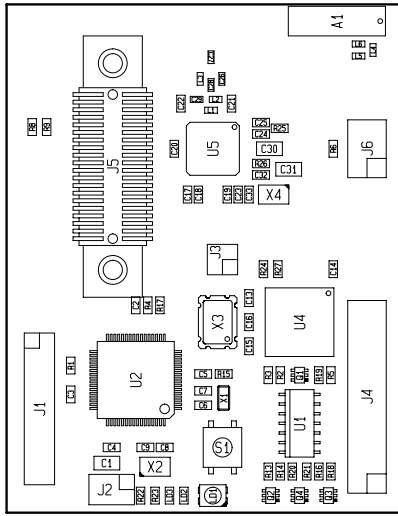


Figure C.1: Component placement, PCB top side.

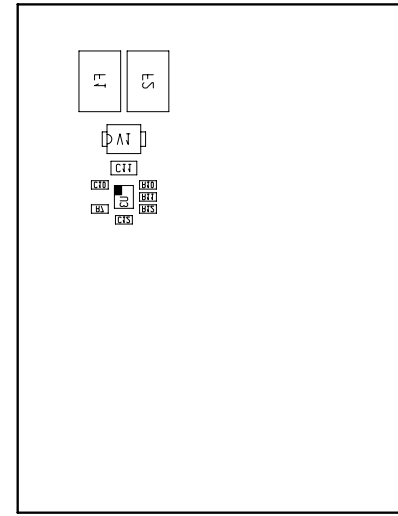


Figure C.2: Component placement, PCB bottom side.

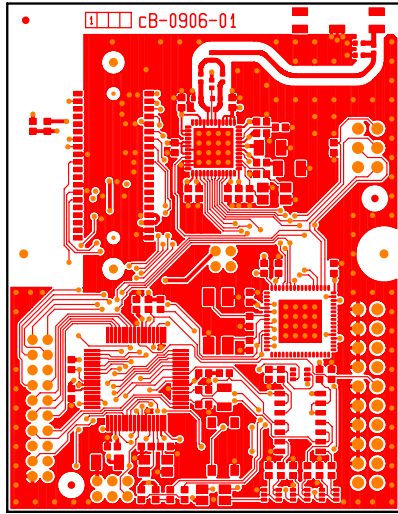


Figure C.3: PCB layout, layer 1 (top side).

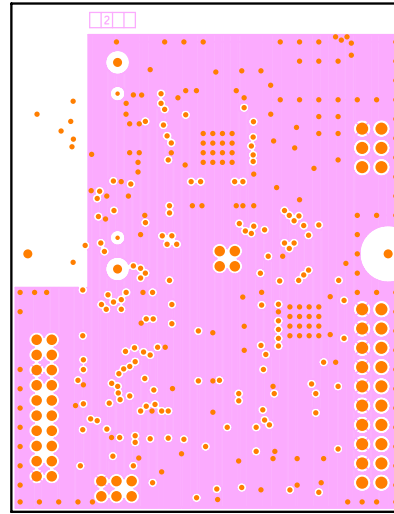


Figure C.4: PCB layout, layer 2.

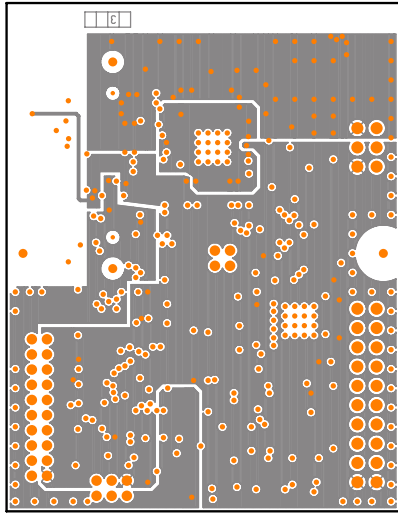


Figure C.5: PCB layout, layer 3.

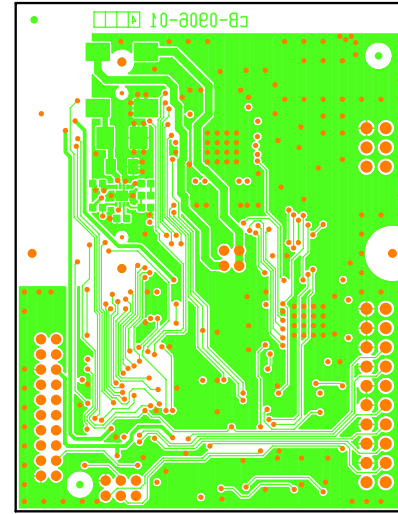


Figure C.6: PCB layout, layer 4 (bottom side).

C.3 Bill of materials

RUNES schematic Revised: Tuesday, September 13, 2005
 cBProject-0508-05 Revision: P3

Bill Of Materials September 13,2005 8:45:30 Page1

Item	Quantity	Reference	Part	PCB Footprint
1	1	A1	RUFA	rufa_12.8x3.9mm
2	3	C1	10u	1206
		C30	10u	1206
		C31	10u	1206
3	10	C2	100n	0603
		C3	100n	0603
		C4	100n	0603
		C5	100n	0603
		C13	100n	0603
		C14	100n	0603
		C17	100n	0603
		C18	100n	0603
		C19	100n	0603
		C21	100n	0603
4	8	C6	8p2	0603
		C7	8p2	0603
		C8	8p2	0603
		C9	8p2	0603
		C15	8p2	0603
		C16	8p2	0603
		C32	8p2	0603
		C33	8p2	0603
5	1	C10	2u2, 6V3	0603
6	1	C11	10u, 16V	1206
7	1	C12	33n	0603
8	4	C20	68p	0603
		C23	68p	0603
		C24	68p	0603
		C25	68p	0603
9	1	C22	10n	0603
10	2	C26	0p5	0402
		C29	0p5	0402
11	2	C27	5p6	0402
		C28	5p6	0402
12	2	F1	SMD075	smd030
		F2	SMD075	smd030
13	1	J1	Expansion	pinheader_p2.00_2x10
14	1	J2	Prgm. LPC2138	pinheader_p2.00_2x3
15	1	J3	Power connector	pinheader_p2.00_2x2

16	1	J4	JTAG	pinheader_p2.54_2x10
17	1	J5	FSI-1203	conn-samtec-fsi-40
18	1	J6	SPI AVR	pinheader_p2.54_2x3
19	1	LD1	LED RGB	led-osram-latbt686
20	1	LD2	RXD/Red	0603
21	1	LD3	TXD/Green	0603
22	2	L1	7n5	0402
		L3	7n5	0402
23	1	L2	5n6	0402
24	3	L4	?	0402
		L5	?	0402
		L6	?	0402
25	4	Q1	BC847BS	sot363-6
		Q2	BC847BS	sot363-6
		Q3	BC847BS	sot363-6
		Q4	BC847BS	sot363-6
26	14	R1	22k	0603
		R2	22k	0603
		R3	22k	0603
		R4	22k	0603
		R6	22k	0603
		R7	22k	0603
		R14	22k	0603
		R15	22k	0603
		R17	22k	0603
		R18	22k	0603
		R19	22k	0603
		R21	22k	0603
		R24	22k	0603
		R27	22k	0603
27	2	R5	10k	0603
		R10	10k	0603
28	2	R8	0	0603
		R9	0	0603
29	1	R11	150k	0603
30	1	R12	100k	0603
31	2	R13	120	0603
		R20	120	0603
32	1	R16	100	0603
33	2	R22	330	0603
		R23	330	0603
34	1	R25	43k	0603
35	1	R26	0	0603
36	1	S1	Reset	switch_5.3x5.1mm
37	1	U1	74LVC00	so14
38	1	U2	LPC2138	lqfp064-p05_10.0mm
39	1	U3	LP3982	llp8_p05_3x2.5mm
40	1	U4	ATmega128L	mlf64-p0.5_9.0x9.0mm
41	1	U5	CC2420	qlp048-p05_7.0mm
42	1	V1	SM6T6V8A	sod6

43	1	X1	32.768kHz	fc-135_3.2x1.5mm
44	1	X2	14.7456MHz	xtal_fa248
45	1	X3	8.0MHz	xtal_7.0x5.0mm
46	1	X4	16.0MHz	xtal_fa248