Implementation of a Bluetooth Stack for BTnodes and Nut/OS Version 0.9

Semester thesis by
Mathias Payer

Institute for Pervasive Computing,
Distributed Systems Group,
ETH Zurich
Prof. Dr. Friedemann Mattern
Assistant Matthias Ringwald

August 4, 2004

Abstract

The target of this semester thesis was to develop a modular Bluetooth stack for BTnodes. BTnodes are microcontrollers with very low computing power and memory and have a Bluetooth controller attached.
The main target of the stack is to ensure communication between different BTnodes. This leads to the implementation of some specific layers of the Bluetooth specification.
So the stack was implemented with BTnode constraints in mind, like speed and memory usage. The main differences to the old implementation are that this new stack uses threads, has a modular design and uses a sequential programming schema.

Contents

1 Introduction **7**

2 Preliminaries **9**

2.1 Bluetooth 9

2.1.1 Host Controller Interface (HCI) 10

2.1.2 Logical Link Control and Adaptation Protocol (L2CAP) 12

2.1.3 Higher layers 13

2.2 BTnode hardware 13

2.3 Nut/OS 14

3 Related work **15**

3.1 Old BTnode Bluetooth Protocol Stack 15

3.2 IAR Embedded Bluetooth Protocol Stack 15

3.3 BeeCon MicroBlue - A Bluetooth protocol stack for embedded systems 16

3.4 TinyBT - TinyOS meets BTnodes 16

3.5 lwBT - A light weight Bluetooth stack for lwIP 16

3.6 Implementations for PCs and PDAs 17

4 Concept **19**

4.1 Memory usage and organisation 19

4.2 Calling conventions 20

4.3 Functional range 20

4.4 Organisation of different layers 21

4.5 Connection handles 21

4.6 Buffer handling 21

5 Implementation **23**

5.1 Development 23

5.2 Layer implementation 24

5.3 Internal naming schema 24

5.4 Memory layout 25

5.5 Calling conventions 27

5.6 Semaphores 29
5.7 Transport layer 29
5.8 HCI layer 30
5.9 Packet handling 30
5.10 BTstack threads 31

6 Discussion 33

7 Appendix 35
7.1 Implemented HCI commands 35
7.1.1 Link Control 35
7.1.2 Link Policy 36
7.1.3 Host Control 36
7.1.4 Informational Parameters 38
7.1.5 Status Parameters 38
7.1.6 Vendor Specific 38
7.2 Dependencies of the OS 38
7.2.1 Threading 38
7.2.2 Thread queues and Nut/OS events 39
7.2.3 Low level functions 39
7.2.4 Association from functions to files 39
Bibliography 41

Chapter 1

Introduction

During the last years, the number of portable personal gadgets steadily increased. Many people use numerous autonomous devices like personal digital assistants (PDA), mobile phones and notebooks.
It would be much easier if these devices could communicate with each other. The PDA tells the mobile phone that there is no time for lunch with a colleague because of a meeting, the notebook downloads the actual emails via the mobile phone or synchronises itself with the PDA.
Another idea for sensor fields is that multiple devices are located close to each other and can communicate over radio.
The ETH Zurich is researching in this environment for quite a while and produced the BTnodes, small microcontrollers that communicate over a Bluetooth module. With these BTnodes experiments are made to study communication algorithms that arrange the nodes dynamically and establish a way to communicate with each device. The aim is that no human interaction is necessary.

The old implementation of the over 1200 pages lasting Bluetooth specification is rather inflexible and has speed penalties. It is also monolithic and therefore hard to maintain. Additionally the code provides only little documentation.
Another immense problem is that application programmers who want to work with this system cannot program in their usual way. They first have to get used to event based programming.

It was tried to write a modular stack that is easy understandable for the application programmer and can therefore easily be used for research projects. One of the main design goals was to keep the stack as portable as possible (to different host hardware/software platforms, transport links and Bluetooth controllers).
This report will give an overview over the Bluetooth specification, present

the microcontroller operating system upon which the stack has been implemented and talk about the BTnodes project at the ETH.
The next chapter explains the concept of the stack and presents the main features and decisions.
After the concept part, some details about the implementation will be shown and this thesis will end with the discussion about the work done and some indication about where future work could start.

Chapter 2

Preliminaries

In this chapter some information will be provided that are heavily used in later chapters.
From now on I will use the terms *host* for the host controller that has the Bluetooth controller attached and *controller* for the Bluetooth controller itself. As a shorthand for Bluetooth stack for BTnodes the term btstack will be used.
First some information about Bluetooth in general will be given. The asynchronous design of the specification and the different layers are mentioned. Then some information about the BTnode hardware is mentioned and features of the Nut operating system will be presented.

2.1 Bluetooth

Bluetooth [1] is an international standardised wireless technology that transponds in the 2.4 GHz band with frequency hopping. The intended use is as a cable replacement and in newer versions as Personal Area Networks and for near-distance transmission.
There are three different power-classes of devices, resulting in a range from 10 m to about 100 m. The total bandwidth is about 1 Mb/sec.
A piconet consists of up to 8 active devices. Exactly one device is the master of a piconet. The other devices are called slaves and are synchronised to the hopping sequence of the master. Up to 255 slaves can be in standby, this means synchronised to the hopping sequence, but unable to send.
As every piconet has its own hopping sequence the maximum transfer rate per piconet is the total bandwidth.
Multiple piconets can be linked together when a device is master in a piconet and slave in another piconet. These spanned nets are called scatternets.
The Bluetooth protocol stack itself consists of many different layers that are built on top of each other. See figure 2.1 for an example of a scatternet with piconets. There the master of one piconet is a slave for the master of the

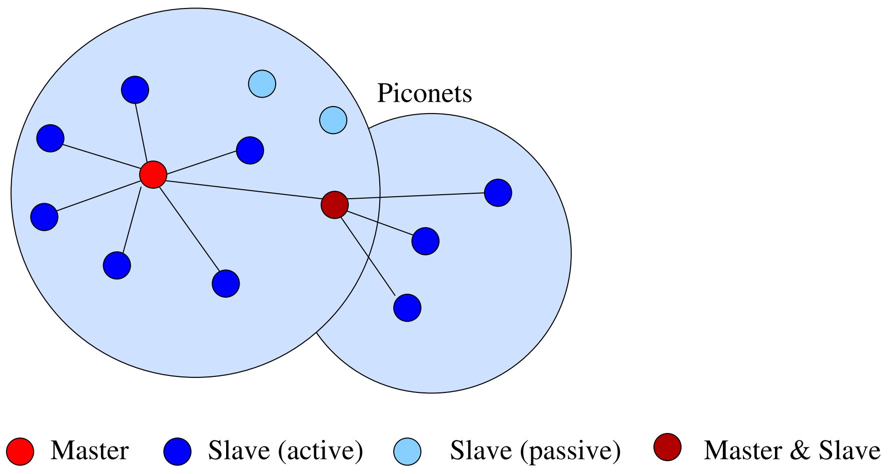

Figure 2.1: A scatternet consisting of two piconets with multiple active and passive slaves.

other piconet.
The baseband layer, link manager and the link control layer exist on the controller chip. The controller offers the host the HCI interface to control all the low level functions.
The host implements the HCI layer, L2CAP layer and on top the higher protocols like SDP and RFCOMM. These layers are implemented in a software stack (Figure 2.2).
The Bluetooth specification also defines profiles. To satisfy a profile, some given layers have to be implemented. Profiles define exactly what parts of the protocol are needed to fulfil a given function.
Now the host layers will be introduced in detail.

2.1.1 Host Controller Interface (HCI)

This is the lowest software level on the host side. Actually it is not a layer by itself, but it offers an interface to all the layers from the Bluetooth controller. To communicate between host and controller a serial or USB transport layer is used.
Access from host to controller is done asynchronously on the basis of events. The host sends a command and receives events as status or results. After a connection is initialised data can be sent to the controller (who forwards the data to the other device) and received from the controller.
The HCI layer is capable to look for other devices near by (inquiry), to exchange parameters, to open connections and to send data.
If a specific command does not complete immediately, the controller sends a status event back to indicate that the command is still running. After the command completed a command complete event or a special event is

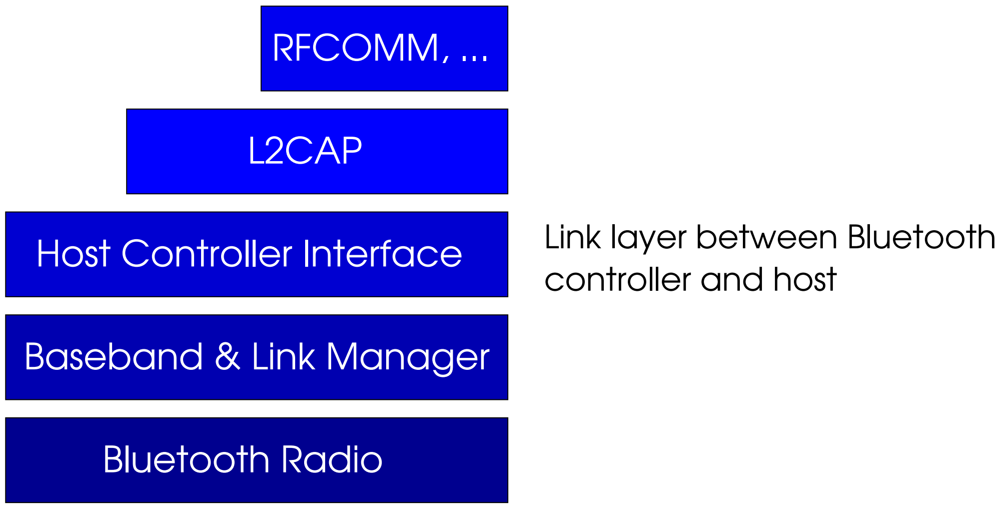
RFCOMM, ...
L2CAP
Host Controller Interface
Baseband & Link Manager
Bluetooth Radio
Link layer between Bluetooth controller and host

Figure 2.2: The different bluetooth layers

generated.
Overall there are three different kinds of packets:

Command packets: This packet can only be sent from host to controller. It is used to read and write general parameters like device name, time-outs and device class and to instantiate inquiries and to initialise data connections.
Commands are logically ordered into different domains. The OGF (Operation code Group Field) specifies the domain of the command and the OCF (Operation code Command Field) points to the actual command in the specified domain. It is possible to distinguish between different commands with the OGF / OCF fields. Connection specific commands also use a distinct connection handle. Some examples of logical domains are host and link control, link policy and informational parameters.

Event packets: This packet is only sent from the controller to the host as a response of a previously sent command or as a request from an other device. Events are used to transport results and information from the controller to the host. The packets consist of a distinct event code and some specific parameters like OGF/OCF, connection handle and return value.

Data packets: There are to different modes of data connections. The first option is a synchronous channel (Synchronous Connection-Oriented - SCO). With this method the upload and download bandwith is the same. This option is mainly used for mobile head sets.

The second possibility is asynchronous (Asynchronous Connection-Less - ACL). ACL connections are used for data transmission where a higher upload than download rate (or vice versa) is needed.

Bluetooth devices can communicate with each other over the baseband connection. This connection is opened when at least two devices are linked as master and slave.
Two ways are possible to create a new connection. The first possibility is to directly open a connection to a specific Bluetooth address (that may be discovered during an inquiry) with the *create connection* command. The callee has then to acknowledge this connection request. The second possibility is to receive a *connection request* event from an other device and then send an *accept connection request* command back.
The disconnection can be either controlled, with a *disconnection request* event and then a *disconnection complete* event. If the disconnection is terminated by an application or by the btstack. The other possibility is that a device runs out of battery or leaves the transmission range. In this case only a *disconnection complete* event will be sent. This indicates an unexpected end of the connection.
The HCI packet size is adjustable between 17 Byte and 339 Byte payload. Two different packet types exist, an error correcting one and an error detecting one.

2.1.2 Logical Link Control and Adaptation Protocol (L2CAP)

The L2CAP layer resides directly above the HCI layer and sends and receives ACL data packets. This layer offers multiple higher layers a connection oriented data stream.
Additionaly this layer offers a Protocol and Service Multiplexing (PSM) that introduces a type field that selects a specific protocol, so L2CAP is capable to differ between different services like SDP or RFCOMM, because these protocols have an already predefined service number.
The L2CAP layer itself uses two different kinds of packets, data packets and signalisation packets. Signalisation packets are used for connection handling and configuration. To the HCI layer both kinds of packets appear as ACL packets.
Out of performance reasons the L2CAP layer is capable to fragment the given packets, so higher protocols do not need to send their (potentially large) header multiple times. The maximum packet size on the L2CAP layer is 64 KB. These packets are then fragmented and passed to the HCI layer.

2.1.3 Higher layers

Higher layers include RFCOMM and SDP.
RFCOMM is used as an emulation layer for applications that need a serial line. RFCOMM emulates up to 60 RS232 devices, multiplexed over the L2CAP link. After setup these RS232 devices can be used like normal serial links. With this layer older applications or closed source applications may still benefit from Bluetooth without explicit knowledge about Bluetooth.
The Service Discovery Protocol (SDP) is used to identify and to keep track of available services. There is no central directory and everything is dynamically discovered. Services can register special attributes like name, service class list (e.g. PrintService, PostscriptPrinter, ...) and provider name.

2.2 BTnode hardware

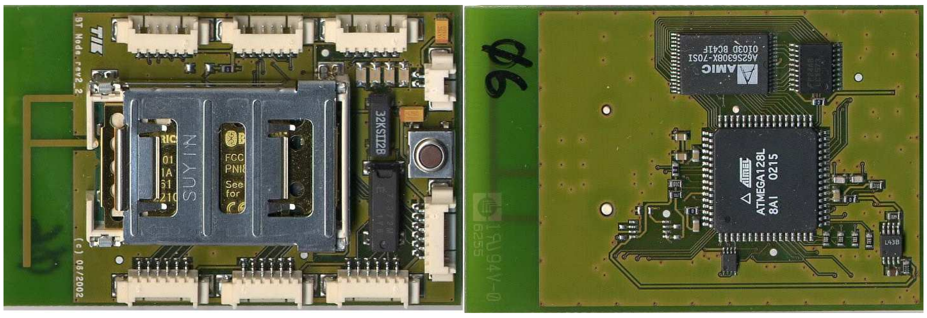

Figure 2.3: Front and back of a BTnode (on the left you see the included antenna)

The used microcontroller for the BTnode hardware is an Atmel ATmega 128L running at 8 MHz, resulting in 8 MIPS. This microcontroller is able to address and use 64 KB of RAM for application data and 128 KB of Flash ROM for program code. For external communication it offers many interfaces like two UART connections, SPI, I2C and 4 LEDs.
For communication an Ericsson ROK 101 007 Bluetooth radio module is attached.
The microcontroller can be programmed in C and standard libraries may be used. The core is a RISC processor, a limitation is that no memory management unit (MMU) is available.
For a picture of the front and back of a btnode see figure 2.3.

2.3 Nut/OS

Nut/OS [2] is an embedded open-source operating-system for microcontrollers, licensed by egnite software [3] under the BSD license. It is allowed to modify the source code and ship binary software, as long as the original copyright notice is included.
Nut/OS offers many features to the programmer like dynamic memory allocation, threading, formatted output and many more. But to ease portability of the stack itself as few Nut/OS specific features as possible were used.
The operating system also offers a **modular design**, so that only needed parts are linked together and no unneeded parts are loaded into the limited program space of the host.
Another advantage is the support for **cooperative multi threading**. The application itself runs in a separate thread and there is no need for the application programmer to cope with events, dispatching and other confusing techniques.
Cooperative multi threading means that the program runs as long as it wants to. If an interrupt occurs, an I/O operation is called or event handling is used, the interrupt gets processed and the control is returned to the running program. If this program wants to pass control, it has to explicitly yield. Then the control is given to the highest waiting thread.
One of the used features includes **thread queues** with asynchronous posting and wake up. Multiple threads can wait on a queue and block until an event is posted to this queue. Then either one or all threads wake up.
Nut/OS also supports **dynamic memory management** to allocate and free memory on the heap, although this is not used to ease portability. **Timer support** and **stream I/O functions** are also supported.
In the Nut/OS part serial device drivers and a PPP stack is included. Additionally Nut/OS features a Nut/Net part, including a fully functional TCP/IP stack providing ARP, IP, UDP, ICMP and TCP protocol, automatic configuration via DHCP, a HTTP server with CGI functions and TCP and UDP socket API for the application programmer.

Chapter 3

Related work

All together many implementations of the Bluetooth specification exist. Below some of these implementations will be discussed and some features will be highlighted.
First the old Bluetooth stack for the BTnodes will be explained. Then two commercial products will be presented. After these a port of TinyOS [9] to the BTnode [13] platform will be shown. Then lwBT [10], a Bluetooth network interface to lwIP [11] is mentioned. At last some larger implementations will be noted.

3.1 Old BTnode Bluetooth Protocol Stack

The first Bluetooth Protocol Stack was explicitly written for devices with very few memory. It featured a monolithic design and included an own operating system. This operating system was built on an event dispatcher. Every time a given event occured a special code fragment was called. This was very troublesome because every program that was to be executed on the BTnodes had to be written with these events in mind. And it was hard to maintain and keep the overview over all the different code fragments.
Another drawback is that the actual packet is copied several times to different locations in memory. On a microcontroller memory accesses take a long time to complete.
The implementation of the protocol stack offered a lot. The HCI and L2CAP layer were nearly complete (all the needed commands were implemented) and RFCOMM was somehow functional.
The stack is licences under the GNU General Public License.

3.2 IAR Embedded Bluetooth Protocol Stack

The Bluetooth Protocol Stack from IAR [6] systems is highly portable to different types of system platforms and offers a complete implementation of

the Bluetooth specifications.
It is possible to port the stack to a personal computer operating system, an operating system for embedded devices and to embedded devices without operating system.
The stack is splitted into different parts that operate independently and communicate with signals and message queues.
The licence is non free and costs for one developer about 10000 €. The generated programs may then only be distributed as binary, no source code can released.

3.3 BeeCon MicroBlue - A Bluetooth protocol stack for embedded systems

The Bluetooth stack from BeeCon [7] is a rather complete implementation of the Bluetooth specification. Additionaly network protocols like IP are implemented.
One of the special features of this stack is that it allows stream processing of the packets. As soon as enough bytes are read from the UART to determine the packet type, the higher layer is called and it can then read the next bytes.
The licence of this stack is non free.

3.4 TinyBT - TinyOS meets BTnodes

This is a port of the TinyOS [9] to the BTnode platform. TinyOS is an event driven operating system for microcontrollers that have to handle multiple inputs. It is designed for wireless embedded sensor networks.
The event driven model makes it hard to program, if one is accustomed to sequential programming.
This implementation provides HCI core functions and keeps the event handling from the controller and passes these to the application.
TinyOS and TinyBT [8] are both open source.

3.5 lwBT - A light weight Bluetooth stack for lwIP

The lwBT [10] Bluetooth stack is mainly an addon to lwIP [11]. The lwIP is a lightweight implementation of the IP protocol and small operating system for embedded systems. The Bluetooth stack is a straight forward implementation of the lower layers up to the IP transport. The Bluetooth part is only intended to be used with lwIP together. The Bluetooth stack only acts as a network interface to lwIP and one of the main goals was to keep the Bluetooth part as small as possible.

This implementation provides HCI, L2CAP (including segmentation), SDP, RFCOMM and PPP. All the functions needed to transport IP packets are implemented. It is possible for the application to register callbacks to keep track of connections.
The lwBT stack is licensed under the BSD license.

3.6 Implementations for PCs and PDAs

Multiple other implementations of the Bluetooth specifications already exist (including several free ones for the linux operating system and also some for Microsoft Windows [12]).
The free ones include bluez [4], the official linux Bluetooth stack that is already integrated into the kernel. As an alternative OpenBT [5] exists. But the main problem of these solutions is, that they were not developed with BTnodes and microcontrollers in mind. They use lots of memory and are feature blown.

Chapter 4

Concept

Below some features are stated that a Bluetooth stack for BTnodes should have.

1. The stack should take care of the **BTnode constraints**. Namely the stack should have a small memory footprint and should not be too complex, because memory and processing power are both very limited on microcontrollers (8 MHz speed and about 64KB of RAM).

2. The stack should be correct regarding **the Bluetooth specification**. The implemented functions and methods should comply with the corresponding functions in the Bluetooth specification.
   To ease programming with the stack some local additions like unique virtual connection-handles were made.

4.1 Memory usage and organisation

There are too possible ways to organise the available memory. Either a **dynamic** or a **static allocation** approach can be taken.
The dynamic method differs between heap and stack. The stack keeps local variables and processor registers, the heap is dynamically used to store application data. The programmer can dynamically get more memory or free memory with functions like *malloc* or *free*.
The second possibility offers no dynamic method to change the layout of the heap. Everything is handled static by the programmer.
Although Nut/OS provides an implementation of malloc and free to use dynamic allocation the decision was to use static allocation. This has the advantage that porting to a different operating system is easier, because there will be no need to implement dynamic allocation. The memory usage of the stack is static, therefore it can easily be defined at compile time.
This method has one disadvantage, namely that the number of buffers the

host can handle is limited at compile time. But this limitation is not that severe because the microcontroller does not have too much RAM to implement many more buffers.

4.2 Calling conventions

For network specific functions there normally exist two different calling methods. Either **blocking** or **non-blocking**.
A blocking call halts the application as long as the call needs. This means that it waits until the data is sent and the response is received. This could result in quite a long waiting time.
Non-blocking calls return immediately and normally return a handle with whom one can keep track of the call. The programmer explicitly needs to check if the call already completed. It is still possible to wait for that specific handle to complete, if there is nothing else to do.
This btstack offers a dual concept to the programmer. All functions, that are transmitted between two (or more) Bluetooth controllers can be called synchronously or asynchronously. The programmer can specify the convention when he calls the function.

4.3 Functional range

On the transport layer the UART transport specification was implemented. This one offers the initialisation of a serial device and send and get packet methods for higher layers. For an USB transport specification some precautions were taken, but it was not implemented.
A wide range of HCI functions were also implemented. Remaining functions can be added in a structured and easy way.
When a HCI command is completed by the controller, it sends back an asynchronous event. These events are fetched in the event handler. The event handler then sets the return value. If a command is called synchronously, the stack sends the command to the controller and then sets the calling thread into a sleep state. The thread sleeps until the return value is set by the returning event and then gets woken up by the btstack.
If a new HCI command is to be implemented one needs to write a calling part that sends an actual HCI command packet to the controller and an event handling part that is inserted into the event handler.
In the current event handler many different return values for HCI commands are already implemented. Code fragments exist for void, unsigned char (8 bit) and unsigned short (16 bit). All these return values are also implemented with a connection handle as additional parameter to match a given connection. To use these general return values only an additional label has to be inserted into the event handler.

4.4 Organisation of different layers

There exist two possible ways of organising the different layers, either a **monolithic attempt**, unioning all the different layers into one or a **layered one**, offering flexibility and interchangeability of the different layers.
In this implementation the layered approach was used.
It would also have been possible to use a monolithic organisation. This would interleave all layers and offer therefore only one layer, with all the functions, to the application programmer. This has the disadvantage that it is not possible to leave out or write own layers. But this approach has the advantage that less memory is used, because there is no need for function pointers and to administrate the different layers and the interaction of these.
As stated above the Bluetooth specification consists of multiple layers. This offers a simple partition of the different parts of the stack.
The layers offer to other layers a clearly specified interface. These interfaces will be used by other layers over header files.
Every layer has a function that will register a higher layer. The higher layer calls the lower layer at initialisation and registers a function pointer that gets called when data for the higher layer is available.
This concept has the advantage that every layer can be exchanged, disabled or replaced by a different implementation. The disadvantage is, that for very high layers the packet may pass through several layers and every layer calls upwards, therefore some additional memory on the stack is needed.
But as there are not too many layers this problem can be disregarded.

4.5 Connection handles

The Bluetooth module itself keeps track of open connections. Every connection has a handle with 12 bit range, as stated in the Bluetooth specification. The problem with the controller is that it may reuse the connection handle. For example if a connection with handle 7 was closed and a new connection (maybe to a different device) is opened, then this connection gets also the handle number 7. This makes it almost impossible to the application to differ between different connections, as new connections may have the same number as old connections.
So a global counter was introduced that sets an unique connection handle to every new connection. As this counter is global, even if multiple stacks are running the connection handles are unique over all stacks on a host.

4.6 Buffer handling

As one of the main constraints was to keep the memory usage as low as possible, the host allocates as few buffers as possible. The HCI layer allocates

one event and one command buffer, as only one thread may write to the UART at a given time.
The HCI layer itself does not keep track of ACL or SCO packets. But when the ACL or SCO layer is initialised it has to pass the HCI layer an ACL or SCO buffer.

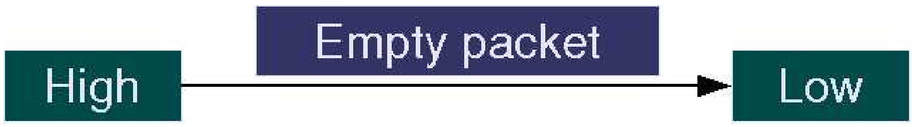

Figure 4.1: When a layer is registered, it has to pass a pointer to a free buffer to the lower layer.

Then every time an ACL or SCO packet arrives, the HCI layer fills the data into the local ACL or SCO buffer. Then it calls the callback to the higher layer with this data buffer. The higher layer has then to return a pointer to an empty ACL or SCO buffer. So the HCI layer always has a pointer to a valid ACL or SCO buffer, but does not need to keep track of the number of buffers.
So the ACL or SCO layer could only use one ACL or SCO buffer. Every time the callback would be called, the data would be processed and the same buffer would be returned to the HCI layer. But to use multiple connections, more buffers are needed.

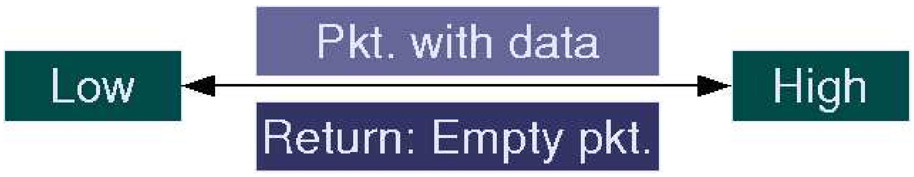

Figure 4.2: If the lower layer receives a packet for a higher layer it saves the data directly in this layer and calls the previously registered callback. This callback has to return a pointer to a free buffer to the lower layer.

Another of the constraints was to keep the copying overhead in memory as low as possible. This means that the data should be written to the memory only once (or as few times as possible). One approach would be, that every layer reads the data from memory and then writes the data into an own local buffer. This would result in a lot of copying and slow the hole stack down. This difficulty is also prevented with the buffer handling mentioned above.
The old btstack was implemented with a lot of copying between different buffers and therefore the resulting speed was not as high as it could have been. This was one of the major drawbacks that was solved with this new stack.

Chapter 5

Implementation

This stack was implemented with the option of handling multiple Bluetooth controllers with one microcontroller in mind. This means that the btstack driver can be used multiple times on one BTnode.
So all functions were implemented in an independent way and the whole stack is kept in a centralised structure. All higher layers register in this structure, so it is possible to differ between different controllers. Every function gets the actual stack context passed as a parameter. This option allows to switch between different instances.
In this chapter some implementation specific details will be presented. First some general information like development platform and how the layers were implemented will be shown. Then some details about the different calling conventions (synchronous / asynchronous) will be stated. The layout and the naming schema of the header files and functions will be highlighted and the memory handling and memory layout presented. At last additional insight into the UART transport layer and the HCI layer will be given.

5.1 Development

The stack was implemented in the language C. To simulate the microcontroller hardware on a personal computer, the Nut/OS emulation for unix operating systems by Matthias Ringwald was used.
But the emulation has some drawbacks. The actual stack size of a thread is ignored. If a thread is specified to have for example 200 Bytes of stack memory, then the emulation ignores this value. This emulation layer was written in parallel with the btstack. This lead to some difficulties and stalls when a part did not work as expected.
A standard AVR-GCC cross compiler chain was set up to transfer the programs onto the BTnode and test them on the real hardware.
After some time of work with archives sent back and forth, a collaborative

[1]CVS [15] system was set up. So multiple people could work together. It was possible to keep track of current versions of different parts and to build applications based on the stack.

5.2 Layer implementation

There is a special intialisation sequence for all the used layers. First the transport layer will be initialised with the given parameters. Then the application programmer can initialise one higher level after another. Every layer is then registered with the lower layer.

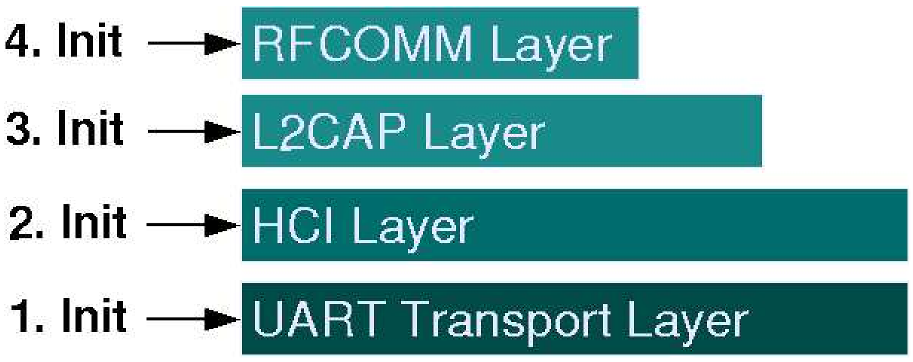

Figure 5.1: Layers are initialised bottom up

The calling of other layers is handled via function pointers in C that are registered during the initialisation of the stack. If no callbacks are registered, then the current layer just drops the received packets. After the registration the lower layer always calls upwards when he receives a packet for the specified layer and passes the data on.
So far only the HCI and the transport layer are implemented, but precautions were taken to handle L2CAP and higher layers.

5.3 Internal naming schema

After checking out the CVS a folder btnut is located in the current directory. The following files are located relatively to this btnut folder.
The header files are in *btnut/btnode/include/bt/* and the c files are in *btnut/btnode/bt/*. Some documented sample applications are located in *btnut/app/*. Please see the README file on how to compile and run the stack and these programs.

[1]Concurrent Versions System: Multiple people can share source files and work together on documents, see bibliography [15] for details.

As an application programmer the files *bt_hci_cmds.h* and *bt_hci_api.h* will be of most interest to you. They include all the needed structures and define all needed HCI commands and other commands. For a description of all implemented HCI functions, please have a look at these files.
The most general file is *bt_defs.h*, it contains all Bluetooth and stack specific options and definitions. Then *bt_hci_defs.h* contains all HCI layer specific definitions that are not yet in *bt_hci_cmds.h* or *bt_hci_api.h.* All event handling function and constant definitions are located in *bt_hci_event.h* and most operating system dependent definitions and some initialisation definitions are located in *bt_dispatch.h.* The file *bt_semaphore.h* gives Bluetooth friendly implementations of semaphores and mutexes. With this implementation it is possible to increase a semaphore by a given value and set an already initialised semaphore to a specific value. Finally, the header file *bt_hci_transport_uart.h* contains the definitions for the UART transport layer.
Generally the naming schema for functions and files is as follows:

1. **bt**, indicating that this belongs to the Bluetooth stack
2. **layer**, indicating which layer the file or function belongs to (e.g. hci or l2cap), multiple layers may be concatenated.
3. **meaning**, the last part contains a descriptive function name or file name, if no meaning is given, then the file or function is general for this layer.

All parts are delimited by an underscore (_).

5.4 Memory layout

The chosen system allows the programmer to use as many stacks as he wants to. For every single stack the application programmer has to declare a btstack variable in his program. After the initialisation (e.g. *bt_hci_init(&stack, &devUsart0)*) the programmer just has to pass the right btstack structure to every call.
Below is a copy of the btstack structure, including documentation.

```
/**
 * struct btstack
 * brief Represents one entity of a running stack (more possible)
 * Keeps all data for one stack, like devices, buffers, states, ...
 */
struct btstack {
#if HCI_TRANSPORT == UART
        /** UART-Transport definitions */
        struct bt_hci_uart transport;
#else
        /** USB-transport definitions (intentionally void ->
          * error, because not jet implemented) */
        void transport;
#endif
        /** Number of hci cmds we can send to the controller
          * until he runs out of buffer */
        struct bt_semaphore nr_hci_cmds;
        /** Semaphore for the waiting commands. After one
          * command is sent, the next thread is woken up.*/
        struct bt_semaphore hci_cmd_queue;
        /** Only one command at a time may be sent
          * by the uart */
        struct bt_hci_pkt_cmd cmd;
        /** Only one event at a time may be received by
          * the uart*/
        struct bt_hci_pkt_evt evt;

        /** acl data (state of the acl-layer aka l2cap) */
        void *acl_data;
        /** Pointer to the acl-buffer (local to the hci-layer) */
        struct bt_hci_pkt_acl *acl_pkt;
        /** Initial acl pkt. from this time on, the stack always
          * gets one from the higher level */
        struct bt_hci_pkt_acl acl_init;

        /** Semaphore for the max. number of acl pkts,
          * the bt-controller can handle */
        struct bt_semaphore nr_acl_pkts;
        /** Max. length of a acl-data-pkt the bt-controller will
          * ever send */
        u_short acl_max_len;
        /** Callback for acl-data pkts */
        HCI_ACL_CB;

        /** sco data (state of the sco-layer) */
        void *sco_data;
        /** Pointer to the sco-buffer (local to the hci-layer) */
        struct bt_hci_pkt_sco *sco_pkt;
        /** Initial sco pkt. from this time on, the stack always
          * gets one from the higher level */
        struct bt_hci_pkt_sco sco_init;
        /** Semaphore for the max. number of sco pkts, the
          * bt-controller can handle */
        struct bt_semaphore nr_sco_pkts;
        /** Max. length of a sco-data-pkt the bt-controller will
          * ever send */
        u_char sco_max_len;
        /** Callback for sco-data pkts */
        HCI_SCO_CB;

        /** Reset flag */
        u_char reset;
        struct bt_semaphore inquiry;

        /** Keeps an eye on running pkts */
        struct bt_hci_cmd_response
                *waitqueue[BT_HCI_NR_WAIT_QUEUES];

        /** Flag to indicate if a conn. request is pending
          * (handled by the main thread) */
        u_char conn_request;
        /** Callback for changes in the connection-table */
        HCI_CON_TABLE_CB;
        /** Keeps track on open connections */
        struct bt_hci_connection
                connection[BT_HCI_MAX_NUM_CONN];
};
```

5.5 Calling conventions

To have as much flexibility as possible regarding the administration of multiple stacks every function has an additional parameter with the stack structure. This way the functions are completely independent. Every function can then get their values out of this structure.
All packets that go over the wire (that are processed and sent by the transport layer) have an additional parameter. Every call can be either synchronous or asynchronous. The easier way is to handle everything synchronous, just pass *BT_HCI_SYNC* as parameter. If you pass *BT_HCI_SYNC*, the stack will initiate a synchronous call and the callee will be woken up, when the HCI command returns and will get a return value passed, indicating either an error or the result.
If asynchronous functions are needed, a *bt_hci_cmd_response* structure can be passed. This structure represents a command that is waiting to be completed. Per asynchronous command one such structure is needed. Then it

is possible to pass a pointer to such a structure and the btstack will call this HCI function asynchronous.
As soon as the field *handle* in the structure represents *SIGNALED* the *response* value in the structure is set and holds either the return value or the result. The constant SIGNALED is defined by the Nut/OS event handling routines in the file *./sys/event.h* which resides in the Nut/OS include directory. There are some commands where a special data structure is filled. This structure is also valid as soon as the *handle* is *SIGNALED*. More information on these special commands is available in the documentation of the functions that use these parameters.
After the HCI command has started, it is possible to check, if the handle is already signalled or wait a specific amount of time for the completion. With *NutEventWait(&response−>event, time_ms)* you can wait for a specific amount of time. Time_ms is a value between 60 and NUT_WAIT_INFINITE. The value is in milliseconds and 60 ms is the smallest time resolution possible in Nut/OS. If you pass NUT_WAIT_INFINITE the program will wait until the command has completed.
If the response value is ≥ 0, then it represents a successful execution of the command and contains the return value. If it is < 0 and $\geq -0xFF$ it indicates an error from the controller. If the value is $< -0xFF$ it is an error from the stack. Please have a look at *bt_hci_api.h* for the actual error messages.
Below is the current layout of the structure. Keep in mind, that some fields are for internal use only and should not be changed!

```
/** Struct for async calls */
struct bt_hci_cmd_response {
        /** The ogfocf pair (is set by internal functions,
          * for internal use only!) */
        u_short ogfocf;
        /** The internal handle (is set by internal functions,
          * for internal use only!) */
        u_short handle;
        /** The response value of the function if max. short,
          * values <0 represent an error! */
        long response;
        /** This is the pointer to the result, if it is bigger
          * than short (is set by internal functions,
          * for internal use only!) */
        void *ptr;
        /** This is the wait-queue (is set by internal
          * functions, for internal use only!) */
        HANDLE block;
};
```

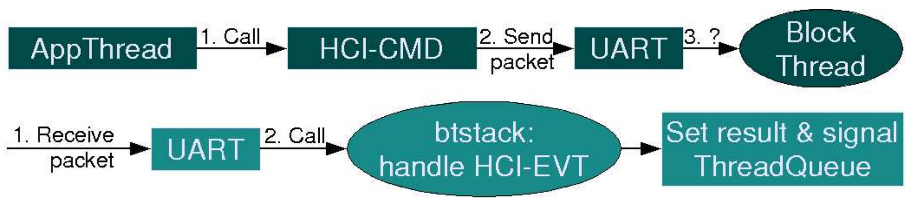

Figure 5.2: HCI Call schema. First the application thread emits an hci command and then may block (until a signal is sent to the thread queue). Then the btstack thread receives the corresponding event and sets the return parameter and signales the thread queue.

To hide all event specific details from the application programmer a HCI call can be divided into two different steps. The first step involves the application thread that calls the function, sets the parameters and sends the data to the controller. Then this part is finished and the application thread may do something else (wait for the command to finish or emit other commands). The second part involves the btstack thread. As soon as the result comes back (in form of an event) the btstack thread handles this event, sets the return parameters in the specified response structure and signals the queue in the structure. After the completion of this event the btstack thread waits until new data from the transport layer arrives. See Figure 5.2 for a graphical schema.

5.6 Semaphores

Another feature that is heavily used in this Bluetooth implementation is semaphores (and mutexes, but these are handled as unary semaphores). This stack uses some special add-ons to the standard semaphore implementation. Additionally you can set the semaphore to a specific value, no matter what value it had before. This is needed when the controller tells the stack an explicit value, for example the number of free buffers, and the stack has to set the semaphore accordingly. Another option is to increase the semaphore not only by one, but by a specified value. This feature is needed when the controller tells for example the number of completed packets.

5.7 Transport layer

The transport layer is actually quite simple and easy exchangeable. There are only a few functions one has to implement.
This layer needs to offer a *bt_hci_init_dev(struct btstack ∗stack)* function to initialise the hardware. For the UART layer this function sets flow control, speed, may initialise power and so on.

The function *bt_hci_send_pkt(struct btstack ∗stack, u_char ∗pktptr)* needs the stack and an arbitrary HCI packet as parameter and sends the packet via the transport that is specified in the stack.
At last the function *u_char ∗bt_hci_get_pkt(struct btstack ∗stack)* receives a packet from the transport. This function is called from the main btstack control thread and has to block until a packet is completely received.
There are actually four different kinds of packets that need to be sent to the controller and be received. In the UART transport layer this is handled with an additional byte for each packet that indicates the type of the following packet.
So the send and get packet functions need to do this conversion from an arbitrary HCI packet (event, command, ACL data or SCO data) to the transport format and send it to the controller.

5.8 HCI layer

The HCI layer defines a protocol how the host can access the functions of the controller. With these functions the host can set and read options like local and remote name, timeouts and buffers. It is also suitable to open baseband connections and search for nearby devices (inquiry).
The HCI layer also keeps track of all open connections in a connection table. A special feature of this table is that it offers a unique virtual connection handle in spite of the connection handle presented by the controller. The problem is that the controller reuses the connection handle as soon as the connection is closed. So the application may not be sure if the connection was closed and is now talking to a new device or if it is the same connection. So the stack's connection table was extended to translate between controller handles and host handles.
The host handles are unique for all stacks and every connection running on a host. Currently this number is an u_short and this means that at most 65534 unique connections are possible. After this number the handles will be repeated. But this is enough for practical use.
To keep track of all connection changes (opening, closing and master/slave role changes) a special callback can be registered that gets called every time such an event occours.
Please refer to *bt_hci_defs.h* or the appendix for a list of implemented HCI functions.

5.9 Packet handling

To keep the number of allocated buffers as low as possible per layer only the really needed buffers are allocated. This means that the HCI layer only allocates one event and one command packet that are placed directly in the

btstack structure. Additionly the HCI layer has a pointer to a ACL packet and a pointer to a SCO packet.
When a transport layer is registered, it passes valid buffer addresses to the HCI layer. Every time a packet for the ACL or SCO layer arrives, the HCI layer reads the data directly into the buffer specified by the ACL or SCO layer.
This means that no extra copying of the data is needed, as the data is already in the buffer of the layer where it is acutally needed.
The callback of the ACL or SCO layer has to complete as fast as possible and return a new pointer to an ACL or SCO packet that is stored in the HCI layer.

5.10 BTstack threads

In this section the layout of the application code and the btstack code will be presented. This code is overlayed with the threads and on what aspects of the code these threads are running.
See Figure 5.3 for details. On the left hand side we see that the application threads run most of the time on their own code. But when a HCI command is issued some functions of the btstack code are used. The btstack on the right hand side uses only its own code. But control gets passed to the UART functions and to the Bluetooth controller. The btstack thread also handles the wait queues (for HCI commands) for the application theads such keeping control of the callback and event handling.
The connection between application thread and btstack thread is defined in the waitqeue array inside of the btstack structure. The application thread calls an HCI function inside of the btstack code. This code registers an internal response structure inside of this waitqeue array.
In this response structure the application thread saves the OGC, OCF and if needed the connection handle. This information makes it possible for the stack to assign the correct response structure to the correct incoming event.
After the response comes back from the Bluetooth controller the btstack thread can look into the waitqueue array and is able to track the given response structure down and fill in the return values. With this response structure it is also possible for the btstack thread to awake the sleeping application thread.
This implementation shows that it is possible to masquerade the asynchronic calling conventions of the Bluetooth definition and offer two possibilities to the programmer. With this implementation the programmer can chose every time a Bluetooth function is called if a asynchronous or a synchronous calling method is used.

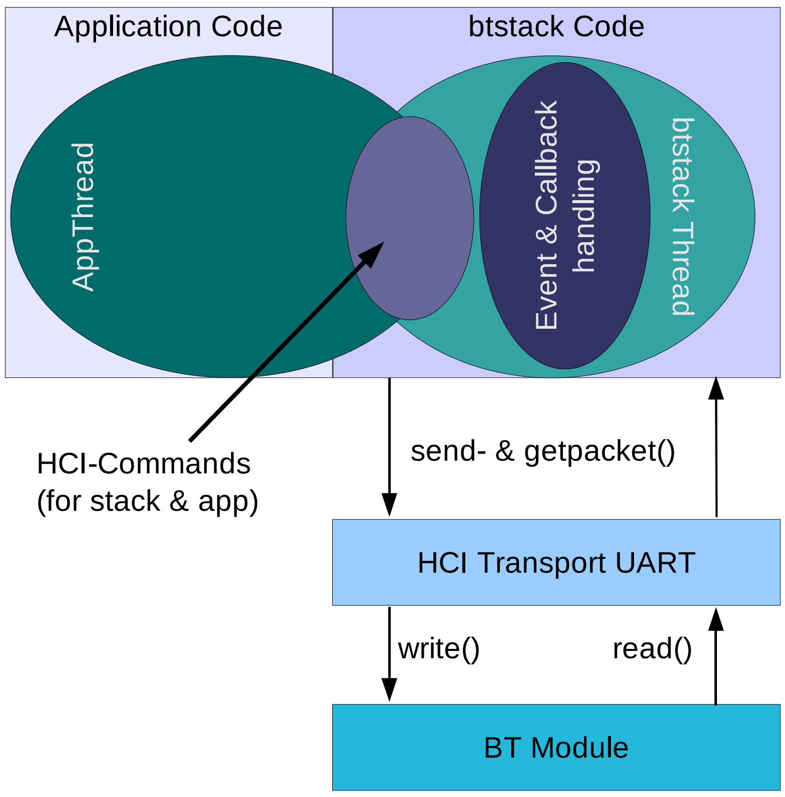

Figure 5.3: Layout of the program code overlayed with the running threads.

Chapter 6

Discussion

The HCI layer reached a stable state and is used in applications. Unfortunately no L2CAP layer is yet available. One of the outstanding features is the highly modular concept, that makes it possible to exchange nearly every part of the stack and plug the layers together as the programmer likes to.
Also the interfaces were designed as clearly as possible, so it is easy to expand the stack and implement further layers. Nearly all the basic stuff is done and the software is ready to be used and extended.
Further work is still required. Namely the L2CAP layer and then the RFCOMM layer should be implemented. But work in this direction is already in progress.
If Nut/OS is still used at this time then it should be quite easy to tunnel TCP/IP over RFCOMM. The implementation of all the network protocols was already done by the Nut/OS people. Except the integration of the Bluetooth stack as a network transport layer.
Finally the conclusion is positive. A portable, highly modular Bluetooth stack was developed in about three months. There is still more work to do, but the skeletal structure is running smoothly. Hopefully the stack will be widely used and enhanced.

The current btstack release and additional information about the BTnodes can be downloaded from *http://www.btnode.ethz.ch/*

Chapter 7

Appendix

7.1 Implemented HCI commands

Following is a list of all implemented HCI commands.
If you are interested in the actual names and parameters of the functions, please have a look at *bt_hci_cmds.h* for all HCI functions and *bt_hci_api.h* for all other functions that are handled by the stack itself and that are not sent to the controller.
Additional and more explaining information is also provided in these header files.

7.1.1 Link Control

All commands with OGF LinkControl keep track of the open connections, how to look for devices and the remote names.

1. **Inquiry**
   Looks for other devices nearby.
2. **Create connection**
   Creates a baseband connection to a specified device.
3. **Disconnect**
   Disconnects an open connection.
4. **Accept connection request**
   Accepts a connection request from a remote device.
5. **Reject connection request**
   Rejects a connection request from a remote device.
6. **Remote name request**
   Asks a remote device for its human readable name.

7.1.2 Link Policy

This OGF section handles link policy settings, like master-slave role changes.

1. **Role discovery**
   Asks the controller if this device is currently master or slave of a specific connection.
2. **Role change**
   Tries to perform a role change. (Master $\rightarrow$ Slave, or Slave $\rightarrow$ Master).
3. **Write link policy settings**
   Sets the link policy settings for a connection. This determines if a role change and hold and sniff mode and park state are possible
4. **Write default link policy settings**
   Sets the link policy settings for all new connections. This determines if a role change and hold and sniff mode and park state are possible

7.1.3 Host Control

This OGF handles some controller parameters.

1. **Reset**
   This command performs a reset of the controller.
2. **Set event filter**
   Sets an event filter, to suppress specific events.
3. **Read PIN type**
   Reads the mode of the pin (variable or fixed).
4. **Write PIN type**
   Sets the mode of the pin (variable or fixed).
5. **Write local name**
   Sets the local human readable name of this device.
6. **Read local name**
   Gets the local human readable name of this device.
7. **Read connection accept timeout**
   Reads the timeout until a connection is accepted.
8. **Write connection accept timeout**
   Writes the timeout until a connection is accepted.

9. **Read page timeout**
   Reads the timeout how long the local link manager will wait for a baseband page response from the remote device.

10. **Write page timeout**
    Writes the timeout how long the local link manager will wait for a baseband page response from the remote device.

11. **Read scan enable**
    Checks if this device can be found by an inquiry or page scans.

12. **Write scan enable**
    Sets if this device can be found by an inquiry or page scans.

13. **Read COD**
    Reads the local class of device.

14. **Write COD**
    Sets the local class of device.

15. **Set host controller to host flow control**
    Enabled or disabled the flow control for ACL and/or SCO connections

16. **Host buffer size**
    Sets the number of packets the host can handle until the buffers are filled.

17. **Host number of completed packets**
    Tells the controller how many packets have been completed on the host side.

18. **Read inquiry mode**
    Checks if the inquiry mode includes the rssi (signal strength) value or not.

19. **Write inquiry mode**
    Sets if the inquiry mode includes the rssi (signal strength) value or not.

20. **Write link supervision timeout** Sets the time until a no longer responding device is noted.

21. **Read link supervision timeout** Checks the time until a no longer responding device is noted.

7.1.4 Informational Parameters

This OGF keeps track of informational parameters, offered by the controller.

1. **Read buffe size**
   Reads how many buffers and the maximum packet size the controller can handle for ACL and SCO packets.

2. **Read BD Address**
   Reads the local Bluetooth address.

7.1.5 Status Parameters

This OGF offers some status information.

1. **Read RSSI**
   Returns the received signal strength for a specified connection.

7.1.6 Vendor Specific

This OGF offers some status information.

1. **Set baudrate**
   Changes the baudrate of the controller and the host. Keep in mind, that this command is Ericsson specific.

7.2 Dependencies of the OS

The dependencies on the underlying were kept as small as possible. Only some functions are needed to port this stack to a different operating system. These functions were used in the following files: *bt_hci.c*, *bt_hci_dispatch.c* and *bt_semaphore.c*. In the UART transport some operating system dependent functions are used to set flow control, speed and to do data input and output. These functions are located in *bt_hci_transport_uart.c*.

7.2.1 Threading

The current implementation uses a specific btstack thread to handle incoming data and events. To declare these threads on the Nut/OS platform the reserved keyword *THREAD* was used to specify the main thread of the stack.
When the stack is initialised the function *NutThreadCreate()* is used to create a new btstack thread. The btstack thread passes the control flow to the next waiting thread after the handling of a packet. This is done with *NutThreadYield()*.

7.2.2 Thread queues and Nut/OS events

Nut/OS offers thread queues as a special feature. A running thread can wait onto a queue with *NutEventWait()*. A special parameter is the maximum time the thread will stay in this queue and wait for a signal. This time interval is between 60 ms and infinite. Two possible signals can be sent to the queue, one that wakes the longest waiting thread (*NutEventPost()* or a signal that wakes all threads (*NutEventBroadcast()*).
The sending of these signals can happen in two ways. Either the control gets passed to the thread with the highest priority (if the woken thread has higher priority, it will run). This is done with *NutEventPost()* or *NutEvent-Broadcast()*. Or the thread will only be woken and inserted into the Nut/OS thread queue and wait for the sender to pass control. This queue keeps the runnable threads that are waiting for the CPU. This is done with *NutEvent-PostAsync()* or *NutEventBroadcastAsync()*.

7.2.3 Low level functions

To ensure that no interrupt occurs between critical sections of the stack the feature to disable and reenable interrupts was used. This is done with the functions *NutEnterCritical()* and *NutExitCritical()*.
NutRegisterDevice()) was used to register the UART in Nut/OS, then the function *fopen()* opens the UART and *_fileno()* returns the file number of the UART. With this file handle low level read (*_read()*) and write (*_write()*) functions are then used.
To set speed, parity and other UART specific parameters the function *_ioctl()* was utilised.

7.2.4 Association from functions to files

bt_hci.c: Contains the main thread of the btstack

1. **THREAD** Keyword to define a new thread.
2. **NutThreadYield()** Passes control to the next runnable thread.
3. **NutThreadCreate()** Creates and starts a thread.

bt_hci_dispatch.c: Contains dispatching functions to simplify HCI commands.

1. **NutEventWait()** Waits for a signal.
2. **NutEventBroadcastAsync()** Broadcasts a signal to all waiting threads in the queue and returns control to the callee.
3. **NutEnterCritical()** Disables interrupts.

4. **NutExitCritical()** Reenables interrupts.

bt_semaphore.c: Implements semaphores for the bt stack.

1. **NutEventWait()** Waits for a signal.
2. **NutEventPostAsync()** Posts a signal to the longest waiting thread in the queue and returns the control to the callee.
3. **NutEventBroadcastAsync()** Broadcasts a signal to all waiting threads in the queue and returns control to the callee.
4. **NutEnterCritical()** Disables interrupts.
5. **NutExitCritical()** Reenables interrupts.

bt_hci_transport_uart.c: Implements the UART transport layer and offers standardised functions to the HCI layer.

1. **NutRegisterDevice()** Registers an UART deivce in Nut/OS.
2. **_ioctl()** Changes some device specific values.
3. **fopen()** Opens a specified device for reading and / or writing.
4. **_fileno()** Returns the file number of an opened device.
5. **_read()** Reads a given amount of bytes from the UART. This function blocks the calling thread until at least one byte is read.
6. **_write()** Writes a given amount of bytes to the UART. This function blocks the calling thread, until all bytes are in the write ringbuffer of the operating system.

Bibliography

[1] Bluetooth specification and additional information:
http://www.Bluetooth.org

[2] Nut/OS main page:
http://www.ethernut.de/en/software.html

[3] Egnite software - holder of Nut/OS sources:
http://www.egnite.de

[4] Bluez - Official linux bt stack:
http://bluez.sourceforge.net

[5] OpenBT - Alternative bt stack:
http://developer.axis.com

[6] IAR Bluetooth Stack for embedded systems:
http://www.iar.com/Products/?name=MPBT

[7] MicroBlue - Bluetooth stack by BeeCon:
http://www.beecon.de/produkte/MicroBlue/index.html

[8] TinyBT - TinyOS for BTnodes:
http://www.diku.dk/ leopold/work/tinybt.pdf

[9] TinyOS - Event driven OS for microcontrollers:
http://webs.cs.berkeley.edu/tos/

[10] lwBT - Bluetooth implementation for lwIP:
http://www.sm.luth.se/ conny/lwbt/

[11] lwIP - A Lightweight TCP/IP stack for embedded systems:
http://savannah.nongnu.org/projects/lwip/

[12] Microsoft Bluetooth Implementation and Wireless Information:
http://www.microsoft.com/whdc/device/network/wireless/

[13] BTnodes - A distributed environment for prototyping ad hoc networks:
http://www.btnode.ethz.ch

[14] BTnode Bluetooth stack - Software repository:
http://sourceforge.net/projects/btnode

[15] CVS - A collaborative source administration software:
http://www.gnu.org/software/cvs/