Bluetooth BCSP Implementation in Timber

Håkan Evertsson
Abstract

To meet the need for smaller designs, Bluetooth chips emerge that implement the entire Bluetooth stack. Timber is a program language developed for embedded systems, where the size of the system often is critical. If these two are combined, a small sized embedded system with a possibility to communicate with the surrounding can be created.

This thesis describes the structure of Bluetooth, with a short description of the different layers. The BlueCore Serial Protocol, used to communicate with Bluetooth chips, and BlueStack is described, together with an example of how they can be used to make a Bluetooth implementation in Timber.
Acknowledgements

During the work of this Master thesis at EISLAB, systemteknik, LTU some people meant much for the work. Therefore, I would like to thank my supervisor Per Lindgren for the help and guidance through the work. I also want to thank Jens Eliasson for help with BCSP and communication with the hardware and Johan Nordlander for helping out with Timber problems. Finally, I would like to thank my family for their support during the work.
# Table of Contents

1 Introduction ...................................................................................................... 5  
  1.1 Background .................................................................................................. 5  
  1.2 Goal ............................................................................................................. 5  
  1.3 Demarcations ............................................................................................... 5  
  1.4Disposition .................................................................................................. 5  

2 Bluetooth 6  
  2.1 Bluetooth stack ............................................................................................ 6  
     2.1.1 Radio ....................................................................................................... 6  
     2.1.2 Baseband ................................................................................................ 6  
     2.1.3 Link Controller ...................................................................................... 7  
     2.1.4 Audio ...................................................................................................... 7  
     2.1.5 Link Manager ......................................................................................... 8  
     2.1.6 HCI – Host Controller Interface ............................................................... 8  
     2.1.7 L2CAP – Logical Link Control and Adaptation ..................................... 10  
     2.1.8 RFCOMM - ........................................................................................... 11  

3 BCSP – BlueCore Serial Protocol ................................................................... 13  
  3.1 BCSP packets ............................................................................................ 13  
  3.2 BCSP stack ................................................................................................ 14  
     3.2.1 SLIP – Serial Line Internet Protocol ...................................................... 14  
     3.2.2 Packet Integrity Layer ........................................................................... 14  
     3.2.3 MUX Layer ........................................................................................... 15  
     3.2.4 Sequencing Layer .................................................................................. 15  
     3.2.5 Datagram Queue Layer ........................................................................ 15  

4 BCCMD – BlueCore Command ..................................................................... 16  

5 BlueStack 17  
  5.1 To set up a connection ................................................................................ 17  
  5.2 Connection establishment example............................................................. 17  

6 Timber 18  

7 Implementation ............................................................................................... 19  
  7.1 Implementation work ................................................................................. 19  
  7.2 Implementation .......................................................................................... 19  

8 Conclusions .................................................................................................... 21  
  8.1 Summary ................................................................................................... 21  
  8.2 Problems .................................................................................................... 21  
  8.3 Future work ............................................................................................... 21  

9 References 22  

10 Shortenings ..................................................................................................... 24  

Appendix A – BlueStack instructions ................................................................. 25  
  Packet order for the master ............................................................................. 25  
  Packet order for the slave .............................................................................. 25  
  Device Manager commands ............................................................................ 26  
  RFCOMM commands ..................................................................................... 28  

Appendix B – BCSP in Timber UML ................................................................. 35  

Appendix C – BlueStack in Timber UML .......................................................... 36
Introduction

1 Introduction

1.1 Background

When creating small systems for measurement and controlling, there is a need for the system to be able to communicate with its environment. Bluetooth [1], [3], [4] is a practical technology for this kind of communication since it is wireless and widely used, making it easier to build networks. Timber [10] is a programming language under development that is suitable for Real-Time Embedded Systems. Timber is reactive and has support for timing events. The company CSR has Bluetooth chips that hold the entire Bluetooth stack and it can be accessed through the BlueCore Serial Protocol (BCSP) [6]. By putting Bluetooth, BCSP and Timber together, a possible connection ability for embedded systems is created.

1.2 Goal

The primary goals of the thesis were to learn about Timber and Bluetooth, and to make a simple Bluetooth implementation in Timber.

1.3 Demarcations

A decision was made to focus on the higher layers, since the implementation communicates with the higher layers, hence an overall understanding of the lower layers is enough. The implementation was limited to create a connection and to make a successful transfer of data. Since the Timber compiler was under development during the work, the implementation was made for a Timber interpreter [11].

1.4 Disposition

The remaining thesis starts with some information about Bluetooth and how the Bluetooth stack is designed. This is followed by information about BCSP packets and the structure of a BCSP stack. The BlueCore Command Protocol is described in chapter 7, followed by some information about BlueStack. After a short description of Timber, the implementation is presented. The thesis is rounded up with conclusions, including problems during the work and future work.
2 Bluetooth

2.1 Bluetooth stack

The Bluetooth stack transforms the packets between the antenna, that is below the Radio layer, and the higher layers and applications (Figure 1). Incoming packets are sent from the Radio layer to the Audio, L2CAP, RFCOMM or SDP layer and packets to be transmitted are sent in the other direction.

![Bluetooth Stack Diagram](image)

**Figure 1: Bluetooth stack**

2.1.1 Radio

Bluetooth uses the globally available ISM band. The band is situated between 2.4 and 2.4835 GHz. France, as the last country, have released the full ISM band, but still have power regulations restricting the usage of the band [2]. The ISM band is split up in 79 channels and Bluetooth uses frequency hopping to change between these channels. This means that every new packet is transmitted on a new frequency decided by a scheme.

2.1.2 Baseband

Bluetooth uses basic time units for its operations. A basic slot is 625µs and almost all transmit or receive operations uses one, three or five slots. The exceptions are the communication while setting up new connections. A Bluetooth network has one Master and up to seven Slaves. The Master controls the exchange of data and the Slave responds to the Master. A basic network with one Master and some Slaves is called a piconet. To extend the piconet, which only can hold eight members, the Slaves can be Masters of their own piconet, or they can be Slaves in two piconets.
Bluetooth

(Figure 2). This is called a scatternet. To make time based frequency hopping possible, the Bluetooth devices have clock offsets. The offsets are used to synchronise the device clock to the Master clock. The Master only has to synchronise to the Slave clock during the set up of the connection, before the Slave has synchronised to the Master. Nominally, the devices resynchronise every eight hundred slot, but in reality, it can be more seldom due to different circumstances, for example power saving modes.

![Figure 2: Scatternet with shared slave (left) and a scatternet with a slave that also works as a master (right)](image)

2.1.3 Link Controller

To find other devices, a Bluetooth device can enter an Inquiry state and send out inquiries. Bluetooth also enters an Inquiry Scan state periodically. In the Inquiry Scan state the device is able to respond to inquiries. To avoid collisions with responses from other devices, Bluetooth uses a random delay before a response is sent. When a device knows about other devices it can try to connect to them. The request to connect is addressed to a specific device. The device that takes the initiative to connect will be the Master. When creating the connection, the devices frequency hopping is synchronised. When a device is connected, it can be in one of four modes. In the Active mode the device is fully functional. The Hold mode makes the device inactive in the piconet for a period. During this time, the device can search for other devices or enter a low power sleep mode. The device keeps its piconet address while it is in the hold mode. When the device enters the Sniff mode it only listen to traffic during periods that it defines when it enters the Sniff mode. The last mode is the Park mode. When a device enters the Park mode it disconnects from the Master, but it tries to remain synchronised to make the reconnection faster and more power efficient. The Sniff and Park modes are used when more communication is expected later, but there is not any communication now.

2.1.4 Audio

Audio packets are packets that need to be delivered with as small delay as possible. Since the handling of the information in the layers takes time the Audio layer works as an alternative for the Synchronous Connection Oriented (SCO) links to
lower the delay. The Audio layer supplies a less advanced delivery, but for audio transfers in real time the lower delay is more important than the high quality delivery that is provided otherwise.

### 2.1.5 Link Manager

To manages the setup, configuration, control and transfers of links there is a Link Manager. The Link Manager also handles the power saving modes, piconet managing and Master/Slave switches. The Link Manager relies on the Link Controller for a reliable link. Since the Link Controller does not guarantee the time taken for a message to be delivered, the Link Manager acknowledge its messages to be able to synchronize during Master/slave switches.

### 2.1.6 HCI — Host Controller Interface

A common way of dividing the stack is to put the Link Manager and lower layers in the Bluetooth module and the higher layers in a separate host processor. To make the communication between the host and the Bluetooth module possible, an interface is needed. This interface is the Host Controller Interface. The standard for HCI controls the three packet types and the transport layer, which carries the packets. The three packet types are Command packets, Event packets and Data packets.

The Command packets carry commands from the host to the Bluetooth module (Figure 3). A Command packet consists of a two byte long OpCode, a one byte long Parameter Total and a number of Parameters each an integer of bytes long. The OpCode in turn consists of ten bits with an OpCode Command Field, identifying a command in the group, and six bits with an OpCode Group Field, identifying a group of commands. The Parameter Total field gives the total length of the following Parameters.

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Parameter Total</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 bytes)</td>
<td>(1 byte)</td>
<td>(X bytes)</td>
<td>(Y bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: HCI Command packet
Bluetooth

The Event packets are sent from the Bluetooth module to the host (Figure 4). They consist of a one byte Event Code, a one byte Parameter Total Length and a number of Parameters, each an integer of bytes long. The Event Code identifies an event and the Parameter Total Length gives the length of the following Parameters.

<table>
<thead>
<tr>
<th>Event Code</th>
<th>Parameter Total</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 byte)</td>
<td>(1 byte)</td>
<td>(X bytes)</td>
<td>(Y bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: HCI Event packet

The third packet type is the Data packet (Figure 5). Data packets are sent both from the Bluetooth module to the host and from the host to the Bluetooth module. There are two types of Data packets, Asynchronous Connectionless (ACL) data packets and Synchronous Connection Oriented (SCO) data packets. The ACL data packets start with a twelve bits long Connection Handle followed by a Packet Boundary (PB) flag and a Broadcast (BC) flag, each two bits long. Then there is a two byte long Data Total Length field and last a Data field. The Connection Handle identifies the ACL connection, the PB flag indicates if the packet is the start of a L2CAP packet or if the packet is a fragment belonging to the previous L2CAP packet. The BC flag indicates if the packet is point-to-point data, active broadcast data or piconet broadcast data. The Data Total Length field gives the length of the Data field in bytes. Since the Data Total Length is two bytes long, the Data field has a theoretical upper limit of 65535 bytes, but many Bluetooth buffers are too small to be able to accept that large packets. The minimum size of a Data field that a Bluetooth module can handle is 255 bytes.

<table>
<thead>
<tr>
<th>Connection Handle</th>
<th>Packet Boundary flag</th>
<th>Broadcast flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12 bits)</td>
<td>(2 bits)</td>
<td>(2 bits)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Total Length</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 bytes)</td>
<td>(X bytes)</td>
</tr>
</tbody>
</table>

Figure 5: HCI Data packet

The SCO data packets are similar to the ACL data packets. The difference is that instead of a PB flag and a BC flag there are one Reserved field of four bits and the Data Total Length field is only one byte. The reason for making the Data Total Length smaller is that SCO links are for regular exchange of data, e.g. streaming,
where too large packets would make the transmission time too long.

Bluetooth defines three transport layers, Universal Serial Bus (USB), RS-232 and Universal Asynchronous Receiver Transmitter (UART). The USB transport layer maps different types of HCI packets on to the logical endpoints of the USB standard. RS-232 and UART are serial interfaces. The difference between them is that RS-232 has error correction while UART does not.

Some transport interfaces offer a higher data rate than the Bluetooth radio can handle. By using Flow control, the data transfers are reduced when the Bluetooth module buffer is full. Since the commands are for the Bluetooth module and do not have to use the radio, commands and data are controlled separately. This implies that the commands do not have to wait because the radio is too busy. The data Flow control is dependent on the type of the data packets. The buffer is divided into two parts, one for ACL data packets and one for SCO data packets. The host asks the Bluetooth module for the buffer size and the reply is how many ACL packets and how many SCO packets the buffer can hold. The host then keeps track of how many free places there are in the buffer. Sending one packet makes the buffer counter count down and receiving a Command complete makes the buffer counter count up. The Command complete event indicates that the command has been handled.

When the host wants a broadcast connection, the host sends the data to a connection handle that is not in use. This connection handle will then be the connection handle for the broadcast transmission. The advantage of this is that no handle numbers have to be reserved in the lower layers. One possible problem with this is that a broadcast data packet from the host and a connection complete event from the Bluetooth module can overlap. The connection complete event informs when a connection is created and what handle number that is used. If this happens, the host stops sending new broadcast packets with the conflicting connection handle, waits for all old broadcast packets with the conflicting handle to be sent and allocates a new broadcast channel handle.[1]

2.1.7 L2CAP – Logical Link Control and Adaptation

L2CAP relies on ACL connections, but L2CAP uses multiplexing over the connections. This means that more than one application can share an ACL connection. The L2CAP packets consists of three fields. One two byte Length field that gives the length of the packet, one two byte Channel Identifier field to identify the channel and a Data field. The Data field is between zero and 65535 bytes long. The L2CAP commands hold OpCode, Identifier, Length and Data fields. The OpCode is one byte long and indicates what operation to perform. The Identifier is also one byte long and unique. The Identifier indicates in what order the commands are sent and cannot be reused for six minutes. Commands that do not come through are retransmitted and therefore can be out of order. The Length field is two bytes long and gives the length of the data field. The Commands are sent together in a signalling packet. The signalling packet looks like the standard L2CAP packets but the Channel Identifier field contains a Connection ID field. The Maximum Transmission Unit (MTU) indicates how long the payload of the packet is allowed to be. The whole packet is
rejected if the length of the packet is longer than the MTU allows. Other reasons for packets to be rejected are that the command is not understood or the connection identifier is invalid.

The L2CAP allows the transmitting applications to send bigger packets than the Bluetooth module can handle. To be able to do this, the L2CAP splits the packets to sizes that fits the Bluetooth module. Packets coming from the lower layers are reassembled to bigger packets. This reduces the overhead added by the L2CAP since more data is using the same header.

The L2CAP can disconnect due to two reasons. The first reason is that it receives a disconnect request. Then the L2CAP answers with a disconnect response. The other reason to disconnect is that a transmission times out. The timeout is normally one minute, but the other side can change the timeout time if more time is needed.

2.1.8 RFCOMM -

RFCOMM emulates a RS232 serial port with cable settings and status. By relying on the L2CAP to handle multiplexing, RFCOMM can provide multiple concurrent connections. For every channel in RFCOMM, RFCOMM provides a separate flow control. No error control is provided, instead RFCOMM relies on L2CAP to provide error free channels. RFCOMM handles two types of devices, Internal emulated serial port or Intermediate device with physical serial port.

RFCOMM uses frames when communicating. There are five different types of frames and they are based on the GSM TS 07.10 protocol [5]. The Start Asynchronous Balanced Mode (SABM) frame is used to start the communication, Unnumbered Acknowledgement (UA) frame is used to respond when connected, Disconnect (DISC) frame is used to disconnect, Disconnect Mode (DM) frame is used as a response to commands while disconnected and Unnumbered Information with Header check (UIH) frames are used to send control messages. Each channel in RFCOMM has a Data Link Connection Identifier (DLCI). The control messages always sets the DLCI to zero.

<table>
<thead>
<tr>
<th>Address (1 byte)</th>
<th>Control (1 byte)</th>
<th>Length (1 byte)</th>
<th>Length or Data (1 byte)</th>
<th>Data (0-32767 bytes)</th>
<th>FCS (1 byte)</th>
</tr>
</thead>
</table>

Figure 6: Basic RFCOMM frame

The Address field consists of an extend address bit, a command/response bit a DLCI bit and five server channel bits (Figure 6). The extended address bit indicates if the address is longer than this byte (zero) or just this byte (one). In Bluetooth, the extended address bit is always set to one since Bluetooth only serves 30 channels and the five channel bits in the RFCOMM frames are enough to cover that. The command response bit indicates if the frame is a command or response. The initiator sets the bit
to one when sending a command and zero when sending a response. The responder device marks the bits in the opposite way with one when responding to the initiators command. The DLCI bit indicates in which direction the frame is sent. The initiator sets the DLCI bit to one and the respondent sets the DLCI bit to zero. One exception from this is the above mentioned control messages. The server channel bits have 30 channels for different services. The reason for only having 30 channels and not 32 is that the channel zero is reserved for control information and channel 31 is reserved by the TS 07.10 standard. The reservation of channel 31 is made to remain compatibility with TS 07.10 applications.

The Control field is used to identify the type of the frame. The different types have different fix values of the control field, but the fifth bit is a poll/final bit. This bit is called poll bit if the frame is a command and final bit if the frame is a response. The poll bit is set to one if a response is wanted and zero if no response is wanted. The final bit is always set to one.

The Length field starts with one bit that indicates how long the length field is. If the first bit is set to one, it is followed by seven bits that indicates the length of the frame in bytes. If the first bit is set to zero instead, the length of the frame length field is 15 bits. This makes the length field either one or two bytes long.

The Credit field is only used when credit based flow control has been negotiated. Each device keeps track of how many credits it has. Each time a RFCOMM frame is sent a credit is used and each time a RFCOMM frame is received, with a credit field that is non-zero, the value in the credit field is added to the device credits.

The Data field is only used in UIH frames. It consists of between zero and 32767 bytes. The Maximum Transmission Unit for the L2CAP limits the size of the frame.

The Frame Check Sequence is calculated from the Address, Control and Length field in SABM, DISC, UA and DM frames. In UIH frames, the Frame Check Sequence is only calculated from the Address and the Control field.
3 BCSP – BlueCore Serial Protocol

BCSP is a protocol from CSR which purpose is to enable communication between a Bluetooth host and a Bluetooth host controller. BCSP carries Bluetooth protocols over the UART to the correct Bluetooth layer. Because CSR have implemented the whole Bluetooth stack on their chips, the software implementation can be reduced. More information about BCSP can be found on the homepage [6].

3.1 BCSP packets

Packets in BCSP consist of three parts, header, payload and an optional CRC field (Figure 7). The header starts with a set of flags. The first bit indicates the Protocol Type, which is if the packet is sent as reliable or unreliable. The second bit indicates if CRC is used. The CRC field in the end of the packet is removed if CRC is not used. The last two flags are the three bit acknowledgement field and the three bit sequence field. The acknowledgement field number is the next wanted sequence number and the sequence field number is used to identify the packet so that the receiver knows that all packets have arrived.

The second byte in the header consists of two parts, each four bits. The first part is the four least significant bits of the payload length and the second part is the Protocol Identifier. The Protocol Identifier indicates what channel is used. The third byte in the packet header is the eight most significant bits of the payload length. Last in the header comes the checksum field. The checksum is a control sum, used to check if the packet header is intact after transmission. The checksum is calculated as the inverted sum of the three other header bytes.

<table>
<thead>
<tr>
<th>Protocol Type (1 bit)</th>
<th>CRC Present (1 bit)</th>
<th>ACK (3 bits)</th>
<th>SEQ (3 bits)</th>
<th>Payload Length (4 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol Identifier (4 bits)</td>
<td>Payload Length (1 byte)</td>
<td>Checksum (1 byte)</td>
<td>Payload (n bytes)</td>
<td>CRC (2 bytes)</td>
</tr>
</tbody>
</table>

Figure 7: BCSP packet
3.2 BCSP stack

The BCSP stack consists of the different steps in BCSP. Each layer in the BCSP stack takes care of some part of the final packet (Figure 8).

![BCSP stack overview](image)

3.2.1 SLIP – Serial Line Internet Protocol

SLIP is a simple protocol that does not address or compress the packets. The packets get an end-byte at the beginning and at the end to indicate where the packet starts and stops. To avoid the problem when the end value appears inside the packet all these bytes are changed to an escape byte and an escape-end byte. Escape bytes are changed in the same way to avoid errors with the escape bytes. SLIP is not accepted as a formal standard, but the guidelines in [7] are a de facto standard.

3.2.2 Packet Integrity Layer

The task for the Packet Integrity Layer is to check that packets are intact. The first test is to control the checksum in the header to see that the header is received correctly. Then the packet length is checked by comparing the length with the payload length field and the CRC present field in the packet. The third and last test when receiving a packet is the CRC check. The CRC field is checked so that it holds the correct value. If any of the tests fail, the packet is silently discarded. When packets are transmitted, the Packet Integrity Layer sets the payload length field, the checksum field, the CRC Present field and the CRC field.
3.2.3 MUX Layer

The MUX Layer sorts the packets that are received. The packets are sorted according to their Protocol Type. Reliable packets are sent to the Sequencing Layer and unreliable packets are sent to the Datagram Queue Layer. The MUX Layer publishes the time for the last received packet. The received acknowledgement value is reported to the Sequencing Layer for every received packet and the acknowledgement value to send is fetched from the Sequencing Layer for every packet that is sent. When sending packets the MUX Layer sets the Protocol Type field to make it possible for the MUX Layer on the other side to make the same sorting of the packets. Packets sent through the Datagram Queue Layer are always given priority over the packets sent through the Sequencing Layer. This is because the Datagram Queue Layer is used for time critical packets. If the Choke signal is set, the MUX Layer only lets packets through that manage the UART link. The MUX Layer also has a command that forces it to send an acknowledgement packet. Acknowledgement packets are packets without content and are only sent to inform the other side of how many packets that have been received. If the MUX Layer are ordered to send an acknowledgement packet and there are other packets to send too, the MUX Layer only sends the other packets since they also carries the acknowledgement.

3.2.4 Sequencing Layer

The Sequencing Layer works as a queue for packets that are going to be sent over a reliable flow. The queue has room for eight packets and each packet has a sequence number that indicates the place in the queue. The MUX Layer fetches packets from the queue and transmits them to the peer. The packets are not removed from the queue when they are transmitted, but remains in the queue until an acknowledgement is received that indicates that the packet with the given sequence number was delivered. When packets are received, they are passed on to the higher layers running BCSP and if they are accepted, the acknowledgement number to be transmitted is updated. The Sequencing Layer notifies the higher code layers if the link to the peer fails.

3.2.5 Datagram Queue Layer

Packets that are sent without flow control are sent through the Datagram Queue Layer. When a packet is going to be sent it is put in a queue. If the queue is full the oldest packet is overwritten. Packets that are received are passed on to the higher code layers.
BCCMD can be used to control and monitor the chips from CSR. The commands are sent over the BlueCore Serial Protocol. BCCMD packets are made to fit in the XAP processor that is based on 16 bit blocks. This implies that the fields in the packets are multiples of 16 bits instead of eight. The packets [9] consist of a header and a payload. The header in turn has five fields, each being two bytes long. The fields hold the packet type, length, sequence number, variable identifier and the status of the transaction. There are three types of messages. The two first types are when the client requests to be given or set variables specified by the variable identifier. The third type is the server response to one of the other messages. The length field indicates the length of the whole packet, including both the header and the payload lengths. The client uses the sequence number to match the responses with the requests. The server only copies the sequence number from the request to the response. The variable identifier indicates what variable the message wants to change. If the value of the variable identifier is zero, it means there is no specific variable that is going to be changed. The status of the transaction must always be zero for the get requests and the set requests. This means that everything is OK. When the server responds, the status indicates if everything is OK or if something went wrong. For example, this could be that the variable identifier had a bad value or that the request tried to write to an unwritable variable. The payload is always at least four times 16 bits long, but it might be longer if needed to fit the variable identifier value. More information about BCCMD can be found in [5].

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Sequence number</th>
<th>Variable identifier</th>
<th>Status</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: BCCMD packet
5 BlueStack

BlueStack is used to control Bluetooth chips. The packets are sent over BCSP and use the different channels in BCSP to control the different parts of the Bluetooth stack. The Device Management channel carries packets for controlling chip settings. To enable Device Management packets, and thereby letting a higher layer control the chip, a registration request for Application Management must be sent to the chip. The control of the RFCOMM is sent over the RFCOMM channel. The RFCOMM control allows to change and read the settings of different parameters and to send packets to and receive packets from a remote device.

5.1 To set up a connection

Setting up a connection between the Bluetooth chip and the application involves a series of requests. The first thing to do is to register a protocol handle with RFCOMM by sending a RFC_REGISTER_REQ. The request is answered with a confirmation that indicates if the protocol handle was accepted or not and what server channel to use. The next request is to initiate the RFCOMM. If the usage of Device Management is wanted, the Device Management registration request is sent next. At this point, the Bluetooth device is ready to be used by the application. The process differs from here depending on whether you want to connect as a master or a slave. It is also possible to change settings of different variables and states.

5.2 Connection establishment example

The following is an example of how to continue a successful connection establishment as a master after the initiation of RFCOMM is done. A start connection is sent to open a new connection. This request indicates what Bluetooth address to connect to and the start parameters for the parameter negotiation. Some status messages are received during the opening of the channel. An indication that an ACL is successfully established is received and then an indication that the remote features have been read. If the opening has been successful, a confirmation that the connection is opened is received. After this it is time to start the parameter negotiation. A request is sent with the desired maximum frame size, if credit flow control is desired and the initial number of credits. The confirmation received holds the values that are accepted by the slave device. If a value was too big for the slave device, the slave’s maximum value is received. The slave is not allowed to respond with a higher value than supported by the master. When both sides have agreed on the parameters to use, the master sends an establishment request. The request is answered with an establishment confirmation. When these steps are finished, the connection is ready for data packet transmissions. A summary of the connection establishment and the connection establishment of a slave can be viewed in Appendix A. More information about different requests and confirmations can be found in [8] and in [9].
6 Timber

Timber is a programming language based on three different programming paradigms. Timber is a functional, concurrent and imperative object-oriented language. Timber is based on the functional programming language Haskell. This makes it possible to use recursive definitions, higher order functions and pattern-matching. Haskell was developed to O’Haskell that supports reactive object-oriented concurrent functional programming. Timber is based on O’Haskell, but applies strict (non-lazy) evaluation and introduces time based deadlines and baselines to support real-time constraints. The support of concurrency allows synchronous and asynchronous communication and the object-orientation allows for the creation of objects. The combination of functional, concurrent and object-orientation together with the possibility of timing, makes Timber suitable for real-time embedded systems.

Objects in Timber are created by executing a template construct [10]. The record Counter defines an interface to an object of the type Template Counter (Figure 10). An action is an asynchronous method. When an action is called, a new thread is created and the sender can continue running immediately. A request is synchronous and can return a value at the end. The caller waits for the request to be completed before it continues. While “record” defines global declarations, “let” defines the local declarations. The methods in “let” can be called from the methods in the template, or the template can call a method in let when the template is called without any instruction of what method to call. More information about Timber and how to program in Timber can be found in [10].

```
record Counter =
    inc :: Cmd()
    set :: Int -> Cmd()
    read :: Cmd Int

counter :: Template Counter
counter =
    template
    value := 0
    in let
    add a = action value:= value + a
    in record
    inc = action add 1
    set i = action value := i
    read = request return value
```

Figure 10: Timber code example
7 Implementation

7.1 Implementation work

The implementation work started with an UML diagram with a first plan. The plan was then changed during the work and the UML diagram was updated with the changes. The final UML for BCSP in Timber and BlueStack in Timber can be found in Appendix B and Appendix C. Eleven weeks were used for the implementation work.

7.2 Implementation

In the implementation, a BlueCore2 chip from CSR containing the entire Bluetooth1.1 stack was used. This makes the implementation easier since only the BCSP stack needs to be implemented.

The code design of the BCSP in Timber implementation is disposed to be similar to the layout of the BCSP as described in [6]. Each layer in BCSP corresponds to a template in the code. The lowest layer is the “uart” template. The task of “uart” is to handle the UART communication between the host and the Bluetooth chip. It sends the packets to the Bluetooth chip and defines a handler that takes care of received packets. The port specification in the handler is the only part of BCSP in Timber that is hardware dependent. The handler sends the packets to the next layer, which is “slip”. In “slip”, the packets are transformed to fit the SLIP model. “pktIntegrity” checks that the packets have arrived correctly by checking the checksum and the length of the packets. When sending packets without CRC, “pktIntegrity” passes the packets on to the SLIP layer. This is because it was easier to set the length field and the checksum field when creating the packet header. When sending packets with CRC, “pktIntegrity” adds the calculated CRC to the end of the packet. “mux” checks that the packets have the correct sequence number and sorts the received packets to “squeue” and “dqueue”. “squeue” is also informed of the received acknowledgement number. “mux” also has a state machine to decide the order in which to send the packets. The state machine gives priority to datagram packets over sequencing packets. If a sequencing packet is not acknowledged it is resent. The whole header of the packets is set in “mux” to avoid the problem of inserting bits later. Both “squeue” and “dqueue” have two queues each. There is one queue for incoming packets and one queue for outgoing packets. The difference between the queues is that “squeue” holds a circular queue and places the packets in the place given by the sequence number and “dqueue” only has a straight queue. “squeue” also holds the acknowledgement variable. When a sent packet has not been acknowledged and it is time to do a retransmission, the queue in “squeue” is set to that no packets having been sent. When an acknowledgement is received, all packets with a lower sequence number are dropped since they have been received and do not need to be retransmitted. The UML for the BCSP in Timber is in Appendix B and the code is in Appendix D.

The BlueStack in Timber implementation is very basic. Only the commands that are necessary to create a connection are implemented. At the bottom of the implementation is “bt_handler” that is used by BCSP in Timber to deliver messages
Implementation

to indicate that a new packet has been received. “bt_handler” fetches the packet and delivers it to the correct template depending on which channel it was received on. Device Management packets are delivered to the “dm” template and RFCOMM packets are delivered to the “rfpack” template. Packets received in the “dm” template and in the “rfpack” template are sorted by the packet type. The variables are modified according to the packet received and the connection counter is updated. The connection counter counts the steps in the connecting procedure described in Appendix A. The application then reacts depending on the counter. The names of the different methods in the “rfpack” and the “dm” templates follow the description in [8]. The connection variables are stored in the “rfpack” template at this moment. The UML for BlueStack in Timber is in Appendix C and the code is in Appendix E.

The implementation is memory efficient since the packets are built in place. The only place the packets are written to the memory is in “squeue” and in “dqueue”. Only variables that are necessary for holding states are used.

The implementation is based on reactive templates. When a package is received in a template, it is treated and sent to the next template. This means that there is not any execution when there is nothing to do. This makes the implementation efficient in the usage of the cpu.

To ensure robustness of the implementation the connection establishment restarts if the connection with the Bluetooth chip is lost. This makes the recovery of the connection possible.
8 Conclusions

8.1 Summary

The result of this work is a working implementation that supports the most necessary instructions to make a connection and to be able to send and receive data. The implementation can easily be changed to fit other hardware. Support of more instructions is easy to add since the only action required is to add a method in the BCSP protocol handler. The BCSP is hidden behind a user interface to make it easier to use. This implies that only knowledge about BlueStack is needed when writing new applications.

8.2 Problems

The first big problem during the work was to get the port communication to work. After learning how to communicate over the ports in Timber, there was trouble with the port settings. To solve this problem the port settings were changed in the source code for the Timber interpreter.

The second big problem was that the format of the BlueStack packets were hard to find. After finding information about what the different requests and confirmations do [8], it was possible to understand the transmission order of the requests in the BlueStack using rfcli. rfcli is a program that uses BlueStack to communicate over Bluetooth. To learn how the packets are built a port sniffer was used while rfcli connected to another Bluetooth device.

8.3 Future work

The continuation of this work is to test the full functionality of BCSP. In the MUX layer the Choke signal, the Reset signal and the time of the last received packet are not tested. Support for CRC needs to be tested in the Packet Integrity layer. The discovery of link failure also needs to be tested to ensure reliable deliveries.

In BlueStack more control methods needs to be added. Support for the other BCSP channels is also needed. The implementation of BlueStack needs to be more general since the current implementation is made for a specific connection procedure.

In the future, a chip implementation could be made. Then some changes in the communication between BCSP and the Bluetooth unit might have to be changed.
9 References


## 10 Shortenings

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>ACL</td>
<td>Asynchronous Connectionless</td>
</tr>
<tr>
<td>BCCMD</td>
<td>BlueCore Command</td>
</tr>
<tr>
<td>BCSP</td>
<td>BlueCore serial Protocol</td>
</tr>
<tr>
<td>DISC</td>
<td>Disconnect</td>
</tr>
<tr>
<td>DLCI</td>
<td>Data Link Connection Identifier</td>
</tr>
<tr>
<td>DM</td>
<td>Disconnect Mode</td>
</tr>
<tr>
<td>HCI</td>
<td>Host Controller Interface</td>
</tr>
<tr>
<td>L2CAP</td>
<td>Logical Link Control and Adaptation</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>RFCOMM</td>
<td>Radio Frequency Communications</td>
</tr>
<tr>
<td>SABM</td>
<td>Start Asynchronous Balanced Mode</td>
</tr>
<tr>
<td>SCO</td>
<td>Synchronous Connection Oriented</td>
</tr>
<tr>
<td>SDP</td>
<td>Service Discovery Protocol</td>
</tr>
<tr>
<td>SEQ</td>
<td>Sequence number</td>
</tr>
<tr>
<td>SLIP</td>
<td>Serial Line Internet Protocol</td>
</tr>
<tr>
<td>UA</td>
<td>Unnumbered Acknowledgement</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>UIH</td>
<td>Unnumbered Information with Header check</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>
Appendix A – BlueStack instructions

Packet order for the master

1. Send a RFC_REGISTER_REQ to get the server channel to use.
2. Receive a RFC_REGISTER_CFM with the server channel.
3. Send a RFC_INIT_REQ to set up a channel.
4. Receive a RFC_INIT_CFM to confirm the channel creation.
5. Send a DM_AM_REGISTER_REQ to enable Device Manager primitives.
6. Receive a DM_AM_REGISTER_CFM to confirm 5.
7. Send a RFC_START_REQ to open a connection.
8. Receive a RFC_START_CFM to confirm that a connection is opened
9. Receive a RFC_PARNEG_REQ to start the parameter negotiation
10. Receive a RFC_PARNEG_CFM with the parameters to use
11. Send a RFC_ESTABLISH_REQ to establish a server channel
12. Receive a RFC_ESTABLISH_CFM when the server channel is established

Packet order for the slave

1. Send a RFC_REGISTER_REQ to get the server channel to use.
2. Receive a RFC_REGISTER_CFM with the server channel.
3. Send a RFC_INIT_REQ to set up a channel.
4. Receive a RFC_INIT_CFM to confirm the channel creation.
5. Send a DM_AM_REGISTER_REQ to enable Device Manager primitives.
6. Receive a DM_AM_REGISTER_CFM to confirm 5.
7. Send a HCI_WRITE_PAGESCAN_ACTIVITY
8. Receive a DM_HCI_STANDARD_COMMAND_COMPLETE
9. Send a HCI_WRITE_SCAN_ENABLE
10. Receive a DM_HCI_STANDARD_COMMAND_COMPLETE
11. Receive a RFC_START_IND when a remote RFCOMM wishes to set-up a mux
12. Send a RFC_START_RES to accept the connection
13. Receive a RFC_START_CMP_IND multiplexer session configuration completed
14. Receive a RFC_START_REQ when a remote RFCOMM wishes to set-up a mux
15. Send a RFC_START_RES to accept the connection
16. Receive a RFC_START_CMP_IND multiplexer session configuration completed
17. Receive a RFC_PARNEG_REQ to answer the parameter negotiation
18. Receive a RFC_ESTABLISH_IND indicates that a server channel is requested
19. Send a RFC_ESTABLISH_RES
### Device Manager commands

**DM_AM_REGISTER_REQ**
- **type** = 0x00
  - `phandle` first bit is one if the application is off-chip (ex. 0x8000)

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
</tr>
</thead>
</table>

**DM_AM_REGISTER_CFM**
- **type** = 0x01
  - `phandle` first bit is one if the application is off-chip (ex. 0x8000)

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
</tr>
</thead>
</table>

**DM_HCI_READ_REMOTE_FEATURES_COMPLETE**
- **type** = 0x042A
  - `bd_addr.lap` Lower Address Part (24 bits)
  - `bd_addr.uap` Upper Address Part (8 bits)
  - `bd_addr.nap` Non significant Address Part (16 bits)

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>status (2 byte)</th>
<th>bd_addr.lap (4 byte)</th>
<th>bd_addr.uap (2 byte)</th>
<th>bd_addr.nap (2 byte)</th>
<th>features (8 byte)</th>
</tr>
</thead>
</table>

**HCI_WRITE_SCAN_ENABLE**
- **op_code** = 0x0C1A
  - `length` = 0x02
  - `scan_enable` off, inquiry, page or inquiry and page

<table>
<thead>
<tr>
<th>op_code (2 byte)</th>
<th>length (2 byte)</th>
<th>scan_enable (2 byte)</th>
</tr>
</thead>
</table>

**HCI_WRITE_PAGESCAN_ACTIVITY**
- **op_code** = 0x0C1C
  - `length` = 0x04
  - `pagescan_interval` (= 800)
  - `pagescan_window` (= 700)

<table>
<thead>
<tr>
<th>op_code (2 byte)</th>
<th>length (2 byte)</th>
<th>pagescan_interval (2 byte)</th>
<th>pagescan_window (2 byte)</th>
</tr>
</thead>
</table>
Appendix A – BlueStack instructions

DM_HCI_STANDARD_COMMAND_COMPLETE
  type = 0x0C51 or 0x0C53
  phandle first bit is one if the application is off-chip (ex. 0x8000)
  status

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>status (2 byte)</th>
</tr>
</thead>
</table>

DM_ACL_OPENED_IND
  type = 0x280D
  phandle first bit is one if the application is off-chip (ex. 0x8000)
  BD_ADDR
    lap Lower Address Part (24 bits)
    uap Upper Address Part (8 bits)
    nap Non significant Address Part (16 bits)
  incoming peer-initiated ACL or locally-initiated
  dev_class class of device for peer

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>bd_addr.lap (4 byte)</th>
<th>bd_addr.uap (2 byte)</th>
<th>bd_addr.nap (2 byte)</th>
<th>incoming (2 byte)</th>
<th>dev_class (4 byte)</th>
</tr>
</thead>
</table>
RFCOMM commands

RFC_INIT_REQ

- **type**: 0x01
- **phandle**: first bit is one if the application is off-chip (ex. 0x8000)
- **psm_local**: 0x03 (identifies RFCOMM over L2CAP)
- **use_flow_control**: 0x01
- **fc_type**: 0x01
- **fc_threshold**: max number of unreported primitives between the FCTs
- **fc_timer**: max time between a primitive and a FCT
- **rsvd_4**: 0x00
- **rsvd_5**: 0x00

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>psm_local (2 byte)</th>
<th>use_flow_control (2 byte)</th>
<th>fc_type (2 byte)</th>
<th>fc_threshold (2 byte)</th>
<th>fc_timer (4 byte)</th>
<th>rsvd_4 (4 byte)</th>
<th>rsvd_5 (4 byte)</th>
</tr>
</thead>
</table>

RFC_INIT_CFM

- **type**: 0x02
- **phandle**: first bit is one if the application is off-chip (ex. 0x8000)
- **psm_local**: default value is 0x03 (identifies RFCOMM over L2CAP)
- **fc_type**: 0x01
- **fc_threshold**: max number of unreported primitives between the FCTs
- **fc_timer**: max time between a primitive and a FCT
- **rsvd_4**: 0x00
- **rsvd_5**: 0x00

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>psm_local (2 byte)</th>
<th>fc_type (2 byte)</th>
<th>fc_threshold (2 byte)</th>
<th>fc_timer (4 byte)</th>
<th>rsvd_4 (4 byte)</th>
<th>rsvd_5 (4 byte)</th>
</tr>
</thead>
</table>

RFC_REGISTER_REQ

- **type**: 0x03
- **phandle**: first bit is one if the application is off-chip (ex. 0x8000)

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
</tr>
</thead>
</table>

RFC_REGISTER_CFM

- **type**: 0x04
- **phandle**: first bit is one if the application is off-chip (ex. 0x8000)
- **server_chan**: holds the server channel to use
- **accept**: True or False

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>server_chan (2 byte)</th>
<th>accept (2 byte)</th>
</tr>
</thead>
</table>
RFC_START_REQ

type = 0x05

BD_ADDR
  lap Lower Address Part (24bits)
  uap Upper Address Part (8 bits)
  nap Non significant Address Part (16 bits)

psm_remote

SYS_PAR start parameters for the negotiation

port_speed

max_frame_size

respond phandle first bit is one if the application is off-chip(ex. 0x8000)

<table>
<thead>
<tr>
<th>type</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>psm_remote</th>
<th>port_speed</th>
<th>max_frame</th>
<th>respond_phandle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(4 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_START_RES

type = 0x06

mux_id given by the RFC_START_IND

accept

SYS_PAR start parameters for the negotiation

port_speed

max_frame_size

respond_phandle first bit is one if the application is off-chip(ex. 0x8000)

<table>
<thead>
<tr>
<th>type</th>
<th>mux_id</th>
<th>accept</th>
<th>port_speed</th>
<th>max_frame_size</th>
<th>respond_phandle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_START_IND

type = 0x07

phandle first bit is one if the application is off-chip(ex. 0x8000)

BD_ADDR
  lap Lower Address Part (24bits)
  uap Upper Address Part (8 bits)
  nap Non significant Address Part (16 bits)

mux_id given here

SYS_PAR start parameters for the negotiation

port_speed

max_frame_size

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>mux_id</th>
<th>port_speed</th>
<th>max_frame_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(4 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>
Appendix A – BlueStack instructions

RFC_START_CFM
- type = 0x08
- phandle first bit is one if the application is off-chip (ex. 0x8000)
- BD_ADDR
  - lap Lower Address Part (24bits)
  - uap Upper Address Part (8 bits)
  - nap Non significant Address Part (16 bits)
- mux_id given here
- result_code 0 = success, 1 = pending, 6 = connection failed
- SYS_PAR start parameters for the negotiation
  - port_speed
  - max_frame_size

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>bd_addr.</th>
<th>mux_id</th>
<th>result_code</th>
<th>port_speed</th>
<th>max_frame_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(4 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_STARTCMP_IND
- type = 0x09
- phandle first bit is one if the application is off-chip (ex. 0x8000)
- mux_id given by the RFC_START_CFM or RFC_START_IND
- result_code 0 = success, 1 = pending, 6 = connection failed
- SYS_PAR start parameters for the negotiation
  - port_speed done
  - max_frame_size

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>mux_id</th>
<th>result_code</th>
<th>port_speed</th>
<th>max_frame_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_CLOSE_REQ
- type = 0x0A
- mux_id given by the RFC_START_CFM

<table>
<thead>
<tr>
<th>type</th>
<th>mux_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_CLOSE_IND
- type = 0x0B
- mux_id given by the RFC_START_CFM
- phandle first bit is one if the application is off-chip (ex. 0x8000)
- reason_code the reason for the close indication

<table>
<thead>
<tr>
<th>type</th>
<th>mux_id</th>
<th>phandle</th>
<th>reason_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>
Appendix A – BlueStack instructions

RFC_ESTABLISH_REQ

- **type** = 0x0C
- **mux_id** given by the RFC_START_CFM or the RFC_START_IND
- **loc_server_chan** local server channel
- **rem_server_chan** remote server channel

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>loc_server_chan (2 byte)</th>
<th>rem_server_chan (2 byte)</th>
</tr>
</thead>
</table>

RFC_ESTABLISH_RES

- **type** = 0x0D
- **mux_id** given by the RFC_START_CFM or the RFC_START_IND
- **server_chan** server channel
- **accept**

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>server_chan (2 byte)</th>
<th>accept (2 byte)</th>
</tr>
</thead>
</table>

RFC_ESTABLISH_IND

- **type** = 0x0E
- **phandle** first bit is one if the application is off-chip (ex. 0x8000)
- **mux_id** given by the RFC_START_CFM or the RFC_START_IND
- **server_chan** server channel

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>server_chan (2 byte)</th>
</tr>
</thead>
</table>

RFC_ESTABLISH_CFM

- **type** = 0x0F
- **phandle** first bit is one if the application is off-chip (ex. 0x8000)
- **mux_id** given by the RFC_START_CFM
- **loc_server_chan** local server channel
- **rem_server_chan** remote server channel
- **result_code** 0 = success

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>loc_server_chan (2 byte)</th>
<th>rem_server_chan (2 byte)</th>
<th>result_code (2 byte)</th>
</tr>
</thead>
</table>
Appendix A – BlueStack instructions

RFC_RELEASE_REQ

- 32 -

<table>
<thead>
<tr>
<th>type</th>
<th>mux_id</th>
<th>server_channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_RELEASE_IND

- 32 -

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>mux_id</th>
<th>server_channel</th>
<th>reason_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_DATA_REQ

- 32 -

<table>
<thead>
<tr>
<th>type</th>
<th>mux_id</th>
<th>server_chan</th>
<th>credits</th>
<th>payload_length</th>
<th>payload_length</th>
<th>payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>

RFC_DATA_IND

- 32 -

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>mux_id</th>
<th>server_chan</th>
<th>credits</th>
<th>payload_length</th>
<th>payload_length</th>
<th>payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>
Appendix A – BlueStack instructions

RFC_PARNEG_REQ

type = 0x20
mux_id given by the RFC_START_CFM
loc_server_chan local server channel
rem_server_chan remote server channel
DLC_PAR parameters to negotiate
  max_frame_size gives the maximum payload size
  credit_flow_ctrl True or False
  initial_credits max primitives at a time without receiving a FCT

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>loc_server_chan (2 byte)</th>
<th>rem_server_chan (2 byte)</th>
<th>max_frame_size (2 byte)</th>
<th>credit_flow_ctrl (2 byte)</th>
<th>Initial_credits (2 byte)</th>
</tr>
</thead>
</table>

RFC_PARNEG_RES

type = 0x21
mux_id given by the RFC_START_IND
server_chan holds the server channel that is used
DLC_PAR parameters to negotiate
  max_frame_size gives the maximum payload size
  credit_flow_ctrl True or False
  initial_credits max primitives at a time without receiving a FCT

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>server_chan (2 byte)</th>
<th>max_frame_size (2 byte)</th>
<th>credit_flow_ctrl (2 byte)</th>
<th>initial_credits (2 byte)</th>
</tr>
</thead>
</table>

RFC_PARNEG_IND

type = 0x22
phandle first bit is one if the application is off-chip (ex. 0x8000)
mux_id given by the RFC_START_IND
server_chan holds the server channel that is used
DLC_PAR parameters to negotiate
  max_frame_size gives the maximum payload size
  credit_flow_ctrl True or False
  initial_credits max primitives at a time without receiving a FCT

<table>
<thead>
<tr>
<th>type (2 byte)</th>
<th>phandle (2 byte)</th>
<th>mux_id (2 byte)</th>
<th>server_chan (2 byte)</th>
<th>max_frame_size (2 byte)</th>
<th>credit_flow_ctrl (2 byte)</th>
<th>initial_credits (2 byte)</th>
</tr>
</thead>
</table>
Appendix A – BlueStack instructions

RFC_PARNEG_CFM
- **type** = 0x23
- **phandle** first bit is one if the application is off-chip (ex. 0X8000)
- **mux_id** given by the RFC_START_CFM
- **server-chan** holds the server channel that is used
- **DLC_PAR** parameters to negotiate
  - **max_frame_size** gives the maximum payload size
  - **credit_flow_ctrl** True or False
  - **initial_credits** max primitives at a time without receiving a FCT

<table>
<thead>
<tr>
<th>type</th>
<th>phandle</th>
<th>mux_id</th>
<th>server_chan</th>
<th>max_frame_size</th>
<th>credit_flow_ctrl</th>
<th>initial_credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
<td>(2 byte)</td>
</tr>
</tbody>
</table>
Appendix C – BlueStack in Timber UML

```
<StdEnv -> Counter -> DM -> RfPackaging -> Template Actions>
  Application

<Template Counters>
  counter
  *get_step(): Cmd Int
  *set_step(int): Cmd()

<StdEnv -> Counter -> Template DM>
  dm
  *dm_receive(pkt:[Uint8]): Cmd()
  *dm_setup:bt(R)]: Cmd()
  *dm_am_register_reg(int): Cmd()
  *cl_writeالمعادل: Scan_activity(): Cmd()

<StdEnv -> Counter -> Template RfPackaging>
  rfp
  *rf_receive(pkt:[Uint8]): Cmd()
  *rf_setup:bt(R)]: Cmd()
  *rf_close(): Cmd()
  *rf_get:pm(): Cmd Int
  *rf_init_reg(): Cmd()
  *rf_read_reg(): Cmd()
  *rf_start_reg(): Cmd()
  *rf_stop:pm(): Cmd()
  *rfﭖ:pm(): Cmd()

<StdEnv -> Template BTHandlers>
  bt_handler
  *close(): Cmd()
  *call(int): Cmd
  *set:bt(R)]: Cmd()
```
 Appendix D – BCSP in Timber code

import "PacketType.ti"

record BTHandler =
  clos :: Cmd()
  deli :: Int -> Cmd()
  set :: BCSP -> Cmd()

-- UARTDriver
-- Handles the received list of Int
recvHandler :: StdEnv -> SLIP -> Template Connection
recvHandler env sl =
  template
  pkt := [] :: [UInt8]
  old := [] :: [UInt8]
  in record
    close = action
      env.putStrLn("Closed")
    deliver list = action
      pkt := pkt ++ (map ord list)
      if (length pkt>1) then
        -- remove unwanted bytes (noise)
        if head pkt == end then
          old := [head pkt] ++ check (tail pkt)
        else
          while not(pkt == [] || head pkt == end) do
            pkt := tail pkt
            if not(last old == end) then
              old := []
            pkt := drop (length old) pkt
            if (length old>1) then
              sl.recv old
              old := []
          done
    where
      check [] = []
      check (x::xs)
        | x == end = [x]
        | otherwise = x : check xs

-- UART

record UART =
  uart_send :: [UInt8] -> Cmd()
  uart_setup :: SLIP -> Cmd()
  uart_close :: Cmd()
  uart :: StdEnv -> Template UART
  uart env =
    template
      outfile := record
        deliver d = action
env.putStrLn("FEL!!!")
done
  close = action done
infile := record
  close = action done

in record
  uart_send pkt = action
    outfile.deliver (map chr pkt)

uart_setup sl = action
  o <- do env.devicesoutfile "/dev/ttyS0"
  handle e -> error ("Exception: " ++ show e)
  outfile := o

hand <- recvHandler env sl
  i <- do env.devicesinfile "/dev/ttyS0" hand
  handle e -> error ("Exception: " ++ show e)
  infile := i
done

uart_close = action
  outfile.close
  infile.close
done

******************************************************************************
* SLIP                                                       *
******************************************************************************
record SLIP =
  send :: [UInt8] -> Cmd()
  recv :: [UInt8] -> Cmd()
  slpi_close :: Cmd()

------------------------------------------------------------------
--slip sends and receives packets.
------------------------------------------------------------------
slip :: StdEnv -> UART -> PI -> Template SLIP
slip env u pi =
  template
    pk := [] :: [UInt8]
in record
  send pkt = do
    u.uart_send ([end]++(concat [check x |x<-pkt])++[end])
    where
      check x
      | x == end = [esc, esc_end]
      | x == esc = [esc, esc_end]
      | otherwise = [x]

recv pkt = action
  if ((head pkt)== end && (last pkt)==end) then
    pk := pkt
    pi.recv (check (tail pkt))
    where
      check (x:xs)
      | x == end = []
| x == esc = escfound xs |
| otherwise = x:(check xs) |

escfound (x:xs) |
| x == esc_end = end : (check xs) |
| x == esc_esc = esc : (check xs) |
| otherwise = [] -- makes length wrong => dropped later |

slpi_close = action
u.uart_close

--**************************************************************--
--* Packet_Integrity                                       *--
--**************************************************************--
-- The send part has the header functions moved to the MUX layer
-- No support for CRC yet
--**************************************************************--

record PI < SLIP =
  pi_setup :: SLIP -> MUX -> Cmd()

pktIntegrity :: StdEnv -> Template PI
pktIntegrity env =
  template
  sl := record
    send a = action done
    recv a = action done
    slpi_close = action done

  mx := record
    mux_send_link a = action done
    mux_send = action done
    mux_recv a = action done
    mux_close = action done

  in record
    send (h:pkt) = request
    if not((primAndInt h 0x40)== 0) then
      -- add crc to the end and send
      sl.send ((h:pkt) ++ (crc_ccitt 0xFF (h:pkt)))
    else
      sl.send (h:pkt)

    recv pkt = action
    if (checksum pkt == (head (drop 3 pkt))) then
      -- is CRC present
      if ((primAndInt (primShrInt (head pkt) 6) 1)==1) then
        -- then control that the CRC is correct
        if ((crc_ccitt 0xFF (take ((length pkt)-2) pkt)) == (drop ((length pkt)-2) pkt))
          -- is the payload length correct?
          if (payloadlengthcheck (take ((length pkt)-2) pkt)) then
            mx.mux_recv (take ((length pkt)-2) pkt)
          else
            mx.mux_recv pkt
        else
          if (payloadlengthcheck pkt) then
            mx.mux_recv pkt
      else
        if (payloadlengthcheck pkt) then
          mx.mux_recv pkt

    pi_setup l m = action
    sl := l
Appendix D – BCSP in Timber code

```haskell
mx := m
done

slpi_close = action
sl.slpi_close

--**************************************************************--
--* MUX.ti                                                     *
--**************************************************************--
-- Records in the MUX
------------------------------------------------------------------
record MUX =
  mux_send_link :: [UInt8] -> Cmd()
  mux_send :: Cmd()
  mux_recv :: [UInt8] -> Cmd()
  mux_reset :: Cmd()
  mux_close :: Cmd()
  mux_getLastRxTime :: Cmd Time
------------------------------------------------------------------
-- mux.recv sorts the received packets
------------------------------------------------------------------
mux :: StdEnv -> PI -> SQueue -> DQueue -> BCSPlink -> Template MUX
mux env pi sq dq bl =
template
  mut := True
  rxTime := 0 :: Time
  crc := 0 :: UInt8
in let
  idle = action
  d <- bl.getChoke
  if d then
    after 4000000 idle
  else
    i <- dq.len
    if i > 0 then send_datagram
    else
      j <- sq.len
      if j > 0 then send_seq
      else
        t <- sq.getsendack
        if t then send_ack
        else mut := True
  done

send_datagram = action
  msg <- dq.getpkt --(channel number, payload)
  a <- sq.txack
  sendPkt 0 0 a 0 (fst(snd msg)) (snd(snd msg))
  idle

send_seq = action
  msg <- sq.getpkt --(sequence number,(channel number, payload))
  a <- sq.txack
  sendPkt 1 crc a (fst msg) (fst(snd msg)) (snd(snd msg))
  b <- sq.totlen
  if b > 0 then
```

---

- 40 -
Appendix D – BCSP in Timber code

``` timber
  after 4000000 resend
  idle

  send_ack = action
  i <- sq.txack
  sendAck i
  idle

  sendAck ack = action
  pi.send (ackpkt ack)

  sendPkt ptype crcpr acknr seqnr pid payload = action
  pi.send (bcsppkt ptype crcpr acknr seqnr (length payload) pid payload)

  resend = action
  d <- sq.sqresend
  case d of
    0 -> if mut then
      mut := False
      idle
    1 -> done          -- nothing to resend
    otherwise -> bl.linkFailed
    done              -- link lost

  lastRxPktTime = action
  t <- primTimeOfDay
  rxTime := primTime2Time t
  done

  in record
    mux_send_link msg = action
    sendPkt 0 0 0 0 1 msg

    mux_send = action
    if mut then
      mut := False
    idle

    mux_recv pkt = action
    lastRxPktTime
    -- 0 Datagram, 1 Sequencing
    case (primShrInt (head pkt) 7) of
      0 ->
        -- If Link Establishment
        if ((primAndInt 0x0F (head(drop 1 pkt))) == 1) then
          bl.receiveBL (drop 4 pkt)
        done
        else
          sq.rxack (primShrInt (primAndInt (head pkt) 0x38) 3)
          if (length pkt) > 4 then
            -- sequence number, channel and payload
            dq.insrec 0 (primAndInt 0x0F (head(drop 1 pkt))) (drop 4 pkt)
            done
        end
      1 ->
        i <- (sq.rxseq (primAndInt (head pkt) 0x07))
        if i then
```
Appendix D – BCSP in Timber code

```
sq.rxack (primShrInt (primAndInt (head pkt) 0x38) 3)
  -- sequence number, channel and payload
sq.insrec (primAndInt (head pkt) 0x07)
  (primAndInt 0x0F (head(drop 1 pkt)))
  (drop 4 pkt)
  done

mux_reset = action
rxTime:=0
done

setcrcprecent i = action
crc:=i

mux_close = action
  pi.sipi_close

mux_getLastRxTime = request
  return rxTime

--**************************************************************--
--* Datagram_Queue.ti                                          *
--**************************************************************--
------------------------------------------------------------------
-- Records in Datagram_Queue.ti
------------------------------------------------------------------
record DQueue =
  getpkt :: Cmd (Int,(UInt8,\[UInt8\]))
  inspkt :: UInt8 -> \[UInt8\] -> Cmd Int
  len    :: Cmd Int
    -- for received packets
  getrec :: Cmd (UInt8,\[UInt8\])
  insrec :: Int -> UInt8 -> \[UInt8\] -> Cmd Int
  lenrec :: Cmd Int
  reset  :: Cmd()

------------------------------------------------------------------
-- Datagram Queue
------------------------------------------------------------------
datagram_queue :: StdEnv -> BTHandler -> Template DQueue
datagram_queue env bt =
  template
    queue := []
    -- queue for received packets
    recqueue := []
    -- helping variable
    ret := (0,[])
  in record
    getpkt = request
      ret := (0,[]) 
      if (length queue > 0) then
        ret := (head queue)
        queue := drop 1 queue 
      return (0,ret)
    inspkt ch i = request

- 42 -
queue := queue ++ [(ch,i)]
if (length queue) > 8 then queue := drop 1 queue
return 1

len = request return (length queue)

-- for received packets
getrec = request
ret := (0,[])
if (length recqueue > 0) then
ret := (head recqueue)
recqueue := drop 1 recqueue
return ret

insrec s ch i = request
recqueue := recqueue ++ [(ch,i)]
if (length recqueue) > 8 then recqueue := drop 1 recqueue
bt.deli 0
return 1

lenrec = request return (length recqueue)

reset = action
queue := []
recqueue := []
ret := (0,[])

--**************************************************************--
--* Sequencing_layer.ti                                       *--
--**************************************************************--
------------------------------------------------------------------
-- Records in Sequencing_layer.ti
------------------------------------------------------------------
record SQueue < DQueue =
  txack :: Cmd Int
  setsendack :: Cmd()
  getsendack :: Cmd Bool
  -- for received packets
  rxack :: Int -> Cmd()
  rxseq :: Int -> Cmd Bool
  totlen :: Cmd Int
  sqresend :: Cmd Int
------------------------------------------------------------------
-- Sequencing Layer
------------------------------------------------------------------
squeue :: StdEnv -> BTHandler -> Template SQueue
squeue env bt =
  template
queue := [(0,[]),(0,[]),(0,[]),(0,[]),
  (0,[]),(0,[]),(0,[]),(0,[])] :: [(UInt8,[UInt8])]}
h := 0 :: Int -- head of queue
t := 0 :: Int -- tail of queue
full := False :: Bool-- tells if the queue is full
allsent := False :: Bool -- tells if all pkts in the queue has been sent
seq := 0 :: Int-- sequence number
ack := 0 :: Int-- ack number (se recqueue comment)
sendack := False :: Bool

tup := (0,(0,[])) -- queue element deliverer
-- queue for received packets. tail of receivequeue == ack
recqueue := [(0,[]),(0,[]),(0,[]),(0,[]),
(0,[]),(0,[]),(0,[]),(0,[])] :: [(UInt8,[UInt8])]
rech := 0 :: Int -- receive queue head
recfull := False :: Bool
-- resend variables
tries := 0 :: Int -- resend counter

in let
setSendAck = action
  sendack := True
in record
-- packet queue
getpkt = request
tup := (0,(0,[]))
if (not(allsent)) then
  tup := (seq,element seq queue)
  seq := mod (seq + 1) 8
  allsent := (seq == t)
return tup
where
  element 0 q = head (take 1 q)
  element i q = head (drop i (take (i+1) q))

inspkt ch pkt = request
if (not full) then
  queue := (take t queue) ++ [(ch,pkt)] ++ (drop (t+1) queue)
t := mod (t+1) 8
full := (t==h)
allsent := False
return 1
else return 0
where
  element 0 q = head (take 1 q)
  element i q = head (drop i (take (i+1) q))

len = request
if (full && (t-seq == 0)) then return 8
else return (mod (t-seq) 8)
totlen = request
if full then return 8
else return (mod (t-h) 8)

-- ack number
txack = request
  return ack
-- send_ack_command
setSendack = action
  setSendAck
getsendack = request
if sendack then
sendack := False
return True
else return False

-- for received packets

-- ignores false acks
rxack i = action
if (i>=0 && i<8 && (mod (i-h) 8) <= (mod (seq-h) 8) && (full || not(h==i))) then
full := False
h := i
tries := 0

rxseq i = request
if (i>=0 && i<8 && ack == i) then
ack := i
return True
else
return False

getrec = request
if (recfull || (mod (ack-rech) 8)>0) then
recfull := False
tup := (0,element rech recqueue)
rech := mod (rech + 1) 8
return (snd tup)
else return (0,[])

where
element 0 q = head (take 1 q)
element i q = head (drop i (take (i+1) q))

insrec s ch pkt = request
if (s==ack && not(recfull)) then
recqueue := (take ack recqueue) ++ [(ch,pkt)] ++ (drop (ack+1) recqueue)
setSendAck
ack := primAndInt 0x7 (s+1)
recfull := (ack==rech)
b.t.deli 1
return 1
else return 0

lenrec = request
if recfull then return 8
else return (mod (ack-rech) 8)

sqresend = request
if (tries < retryLimit) then
if (allsent && (seq == h))
return 1
else
seq := h
allsent:= (seq==t)&&(not full)
tries := tries + 1
return 0
else return 2

-- reset
reset = action
queue := [(0,[]),(0,[]),(0,[]),(0,[]),
         (0,[]),(0,[]),(0,[]),(0,[])]
h := 0
t := 0
full := False
allsent := False
seq := 0
ack := 0
sendack := False
tup := (0,(0,[]))
-- queue for received packets
recqueue := [(0,[]),(0,[]),(0,[]),(0,[]),
             (0,[]),(0,[]),(0,[]),(0,[])]
rech := 0
recfull := False

--**************************************************************--
--* BCSP_Link_Establishment                                     *--
--**************************************************************--

data State = Shy | Curious | Garrulous
          deriving (Eq)

record BCSPlink =
establish :: MUX -> Action -> Cmd() -- Action is the program to send the messages to
receiveBL :: [UInt8] -> Cmd()
getChoke   :: Cmd Bool
linkFailed :: Cmd()

bcsp_link_establishment :: StdEnv -> Template BCSPlink
bcsp_link_establishment env =
tshy := 500000
tconf := 500000
choke := True
state := Shy
mx := record
    mux_send_link a = action done
    mux_send = action done
    mux_recv a = action done
    mux_close = action done
pr := action done

in let
    shy = action
    choke := True
    mx.mux_send_link sync_pkt
    after tshy send_state
done

    curious = action
    mx.mux_send_link conf_pkt
    after tconf send_state
done

    garrulous = action
    choke := False


send_state = action
if (state == Shy) then
    shy
else
    if (state == Curious) then
        curious
    else
        garrulous

in record
establish m p = action
    mx := m
    pr := p
    state := Shy
    shy
    done

receiveBL pkt = action
if (comp pkt sync_pkt) then
    if (state == Shy) then
        mx.mux_send_link sync_resp_pkt
done
    else
        if (state == Curious) then
            mx.mux_send_link sync_resp_pkt
done
        else
            state := Shy
            shy
            done
    else
        if (comp pkt sync_resp_pkt) then
            if (state == Shy) then
                done
            else
                if (state == Curious) then
                    done
                else
                    done
        else
            if (comp pkt conf_pkt) then
                if (state == Shy) then
                    done
                else
                    if (state == Curious) then
                        mx.mux_send_link conf_resp_pkt
done
                    else
                        mx.mux_send_link conf_resp_pkt
done
                    else
                        if (comp pkt conf_resp_pkt) then
                            done

else
    if (comp pkt conf_resp_pkt) then
if (state == Shy) then
done
else
if (state == Curious) then
state := Garrulous
done
else
done

getChoke = request return choke

linkFailed = action
state := Shy
shy
done

------------------------------------------------------------------
-- To compare packets
------------------------------------------------------------------
comp :: [UInt8] -> [UInt8] -> Bool
comp [] [] = True
comp (x:xs) (y:ys) = (x==y && comp xs ys)

--**************************************************************--
--* template for external use                                    *--
--**************************************************************--
record BCSP =
setup :: Action -> Cmd Int
send_datagram :: UInt8 -> [UInt8] -> Cmd Int
send_sequencing :: UInt8 -> [UInt8] -> Cmd Int
get_dq :: Cmd (UInt8,[UInt8])
get_sq :: Cmd (UInt8,[UInt8])
sendack :: Cmd()
activate_crc :: Cmd()
inactivate_crc :: Cmd()
close_down :: Cmd()

bcsp :: StdEnv -> BTHandler -> Template BCSP
bcsp env bt =
  template
  sq <- squeue env bt
dq <- datagram_queue env bt
u <- uart env
pi <- pktIntegrity env
sl <- slip env u pi
bl <- bcsp_link_establishment env
mx <- mux env pi sq dq bl

in let
  send_ack = action
  mx.mux_send

in record
setup pr = request
  pi.pi_setup sl mx
u.uart_setup sl
bl.establish mx pr
return 1

send_datagram ch pkt = request
choke <- bl.getChoke
if not(choke) then
dq.inspkt ch pkt
mx.mux_send
return 1
else return 0

send_sequencing ch pkt = request
choke <- bl.getChoke
if not(choke) then
sq.inspkt ch pkt
mx.mux_send
return 1
else return 0

get_dq = request
d <- dq.getrec
return d

get_sq = request
s <- sq.getrec
return s

sendack = action
sq.setsendack
after 8000000 send_ack

activate_crc = action
mx.setcrcpercent 1

inactivate_crc = action
mx.setcrcpercent 0

close_down = action
mx.mux_close
import "BCSP_com1.ti"

-- Handles incoming packets
bt_handler :: StdEnv -> DM -> RFpacking -> HCI -> Template BTHandler
bt_handler env d rf hc =
  template
    bc := record
      setup a = request return 1
      send_datagram a b = request return 1
      send_sequencing a b = request return 1
      get_dq = request return (0,[1])
      get_sq = request return (0,[1])
      sendack = action done
      activate_crc = action done
      inactivate_crc = action done
      close_down = action done
    in record
      clos = action
      env.putStrLn("Closed")
    -- tells what to do when a packet is received
    deli i = action
    -- if the packet is a datagram packet i==0 else it's a sequencing packet
    if i==0 then
      pkt <- bc.get_dq
      -- checks which BCSP channel the packet belongs too
      case (fst pkt) of
        HCI_SCO -> hc.hci_receive_sco (snd pkt)
        _       -> env.putStrLn("Packet received dq: ("++show (fst pkt))
                  ++show (map trans2 (snd pkt)))
                  env.putStrLn("")
    done
    else
      pkt <- bc.get_sq
    bc.sendack
    -- checks which BCSP channel the packet belongs too
    case (fst pkt) of
      HCI_CMD   -> hc.hci_receive_cmd (snd pkt)
      HCI_ACL   -> hc.hci_receive_acl (snd pkt)
      HCI_SCO   -> hc.hci_receive_sco (snd pkt)
      RFCOMM    -> rf.rf_receive (snd pkt)
      _         -> env.putStrLn("Unknown packet received: ("++show (fst pkt))
                  ++show (map trans2 (snd pkt)))
                  env.putStrLn("")
    done

-- defines the bcsp template to use
set b = action
bc := b

--**************************************************************--
--* DM                                                        *
--**************************************************************--
record DM =
dm_receive_cmd :: [UInt8] -> Cmd()
dm_setup :: BCSP -> Cmd()
dm_am_register_req :: Cmd()
hci_write_pagescan_activity :: Cmd()
hci_write_scan_enable :: Cmd()

dm :: StdEnv -> RFpacking -> Template DM
dm env rf=
  template
  bc := record
    setup a = request return 1
    send datagram a b = request return 1
    send sequencing a b = request return 1
    get dq = request return (0, [1])
    get sq = request return (0, [1])
    sendack = action done
    close_down = action done
  pagescan_interval := 0x800
  pagescan_window := 0x700

in let
  -- TODO: add more device management functionality
  dm_am_register_cfm pkt = action
    env.putStrLn("DM_AM_REGISTER_CFM")
    step <- rf.get_step
    if step < 7 then
      rf.set_step 7
    done

  dm_hci_read_remote_features_complete pkt = action
    env.putStrLn("DM_HCI_READ_REMOTE_FEATURES_COMPLETE")
    -- no implementation yet
    done

  dm_hci_qos_setup_cfm pkt = action
    env.putStrLn("DM_HCI_QOS_SETUP_CFM")
    -- no implementation yet
    done

  dm_hci_switch_role_complete pkt = action
    env.putStrLn("DM_HCI_SWITCH_ROLE_COMPLETE")
    -- no implementation yet
    done

  dm_hci_standard_command_complete pkt = action
    env.putStrLn("DM_HCI_STANDARD_COMMAND_COMPLETE")
    step <- rf.get_step
    if step < 9 then rf.set_step 9
    else
      if step < 11 then rf.set_step 11

- 51 -
Appendix E – BlueStack in Timber code

done

\texttt{dm\_acl\_opened\_ind\ pkt = action}
env.putStrLn("DM\_ACL\_OPENED\_IND")
-- no implementation yet
done

\texttt{dm\_acl\_closed\_ind\(pkt = action)
env.putStrLn("DM\_ACL\_CLOSED\_IND")
-- no implementation yet
done

in record
\texttt{dm\_receive\_cmd\ pkt = action}
if (comp (take 2 pkt) \[0x01, 0x00\]) then dm\_am\_register\_cfm\ pkt
if (comp (take 2 pkt) \[0x2A, 0x04\]) then dm\_hci\_read\_remote\_features\_complete\ pkt
if (comp (take 2 pkt) \[0x10, 0x08\]) then dm\_hci\_qos\_setup\_cfm\ pkt
if (comp (take 2 pkt) \[0x13, 0x08\]) then dm\_hci\_switch\_role\_complete\ pkt
if (comp (take 2 pkt) \[0x51, 0x0C\]) then dm\_hci\_standard\_command\_complete\ pkt
if (comp (take 2 pkt) \[0x53, 0x0C\]) then dm\_hci\_standard\_command\_complete\ pkt
if (comp (take 2 pkt) \[0x0D, 0x28\]) then dm\_acl\_opened\_ind\ pkt
if (comp (take 2 pkt) \[0x0E, 0x28\]) then dm\_acl\_closed\_ind\ pkt
done

\texttt{dm\_setup\ b = action}
\texttt{bc := b}

\texttt{dm\_am\_register\_req = action}
env.putStrLn("DM\_AM\_REGISTER\_REQ")
\texttt{ph <- rf.rf\_get\_phandle}
\texttt{bc.send\_sequencing\ device\_mgt\ ([0x00, 0x00]++
\texttt{turn ph)}
done

where
\texttt{turn ph = [(primAndInt\ ph\ 0xFF),
\texttt{(primAndInt\ (primShrInt\ ph\ 8)\ 0xFF)]}

\texttt{hci\_write\_pagescan\_activity = action}
env.putStrLn("HCI\_WRITE\_PAGESCAN\_ACTIVITY")
\texttt{bc.send\_sequencing\ device\_mgt\ ([0x1C, 0x0C]++
\texttt{[0x04, 0x00]++
\texttt{[primAndInt\ 0xFF\ pagescan\_interval]++
\texttt{[primShrInt\ pagescann\_interval\ 8]++
\texttt{[primAndInt\ 0xFF\ pagescan\_window]++
\texttt{[primShrInt\ pagescan\_window\ 8])

done

\texttt{hci\_write\_scan\_enable = action}
env.putStrLn("HCI\_WRITE\_SCAN\_ENABLE")
\texttt{bc.send\_sequencing\ device\_mgt\ ([0x1A, 0x0C]++
\texttt{[0x02, 0x00]++
\texttt{[0x03, 0x00])

done

---************************************************************---

-- ^ HCI
---
Appendix E – BlueStack in Timber code

record HCI =
    hci_send_reset :: Cmd()
    hci_send_read_buffer_size :: Cmd()
    hci_send_cmd :: [UInt8] -> Cmd()
    hci_send_acl :: [UInt8] -> Cmd()
    hci_send_sco :: [UInt8] -> Cmd()
    hci_receive_cmd :: [UInt8] -> Cmd()
    hci_receive_acl :: [UInt8] -> Cmd()
    hci_receive_sco :: [UInt8] -> Cmd()
    hci_setup :: BCSP -> Cmd()

-- TODO: implement HCI commands

hci :: StdEnv -> Template HCI

hci env =
    template
        bc := record
            setup a = request return 1
            send_datagram a b = request return 1
            send_sequencing a b = request return 1
            get_dq = request return (0,[1])
            get_sq = request return (0,[1])
            sendack = action done
            close_down = action done
        in let
            send_cmd pkt = action
                bc.send_sequencing hci_cmd pkt
                done
            send_acl pkt = action
                bc.send_sequencing hci_acl pkt
                done
            send_sco pkt = action
                bc.send_sequencing hci_sco pkt
                done
        in record
            hci_send_reset = action
                send_cmd [0x03, 0x0C, 0x00, 0x00]
            hci_send_read_buffer_size = action
                send_cmd [0x05, 0x10, 0x00, 0x00]
            hci_send_cmd pkt = action
                send_cmd pkt
            hci_send_acl pkt = action
                send_acl pkt
            hci_send_sco pkt = action
                send_sco pkt
            hci_receive_cmd pkt = action
                env.putStrLn("HCI CMD/ENV: "+show (map trans2 pkt))
                done
Appendix E – BlueStack in Timber code

```haskell
hci_receive_acl pkt = action
  env.putStrLn("HCI ACL: "++show (map trans2 pkt))
done

hci_receive_sco pkt = action
  env.putStrLn("HCI SCO: "++show (map trans2 pkt))
done

hci_setup b = action
  bc := b
done

--**************************************************************--
--* RFCOMM                                                     *--
--**************************************************************--
record RFpacking =
  rf_send     :: UInt8 -> UInt8 -> [UInt8] -> Cmd()
  rf_receive     :: [UInt8] -> Cmd()
  rf_setup     :: BCSP -> DM -> Cmd()
  rf_get_phandle :: Cmd Int
  rf_init_req     :: Cmd()
  rf_register_req :: Cmd()
  rf_start_req    :: Cmd()
  rf_start_res    :: Cmd()
  rf_close_req    :: Cmd()
  rf_establish_req:: Cmd()
  rf_establish_res:: Cmd()
  rf_release_req  :: Cmd()
  rf_data_req     :: [UInt8] -> Cmd()
  rf_parneg_req   :: Cmd()
  rf_parneg_res   :: Cmd()
get_step          :: Cmd Int
set_step          :: Int -> Cmd()

-- TODO: add more rfcomm functionalities
rfpack :: StdEnv -> Template RFpacking
rfpack env =
  template
    addr := 0 :: UInt8
    ctrl := 0 :: UInt8
    len  := 0 :: UInt8
    fcs  := 0 :: UInt8
    crc_table := crc_table_8
    bc := record
      setup a       = request return 1
      send_datagram a b = request return 1
      send_sequencing a b = request return 1
      get_dq       = request return (0,[1])
      get_sq       = request return (0,[1])
      sendack      = action done
      close_down   = action done
    d_mg := record
      dm_receive_cmd pkt = action done
      dm_setup b       = action done
      dm_am_register_req = action done
```

- 54 -
hci_write_pagescan_activity = action done
hci_write_scan_enable = action done

phandle := 0x8000
bd_addr := [-0x00, 0x0B, 0xAC, 0x08, 0x27, 0x7B]
[-0x00, 0x07, 0x80, 0x80, 0x32, 0x27]
[-0x00, 0x0B, 0xAC, 0x20, 0x3E, 0x5E]
[0x00, 0xA0, 0x96, 0x0A, 0xBC, 0xF4]

psm_remote := 0x03
port_speed := 0xFF
max_frame_size := 0x7f
credit_flow_ctrl := 0x01
initial_credits := 0x04
mux_id := 0x00
loc_server_chan := 0x01
rem_server_chan := 0x01
credits := 0
fc_threshold := 0
fc_timer := 0
payload_length := 0
step := 1
pong_counter := 0

in let
rf_address ea cr d chnl =
    primOrInt (primOrInt (primShlInt ea 7) (primAndInt cr 1) 6))
    (primOrInt (primShlInt (primAndInt d 1) 5) (primAndInt chnl 0x1F))

send cr d pkt = action
addr := (rf_address 1 cr d 1)
ctrl := 0xF7
len := primAndInt 0xff (length pkt)

fcs := primAndInt 0xff (0xff - head (drop (primAndInt 0xff
    ((head (drop (primAndInt 0xff (0xff^addr)) crc_table))^ctrl)) crc_table))

-if (len<=127) then
--bc.send_sequencing rfcomm ([addr] ++ [ctrl] ++ [len] ++ pkt ++ [fcs])
b.c.send_sequencing rfcomm
    ([0x18,0x00,0x00,0x00,0x01,0x00,0x00,0x00]++[len]++[00,00,00]++pkt)
done

rfc_init_cfm pkt = action
env.putStrLn("RFC_INIT_CFM")
if (length pkt == 22) then
    fc_threshold := primOrInt
        (primAndInt (primShlInt (head (drop 11 pkt)) 8) 0xFFF0)
        (primAndInt (head (drop 10 pkt)) 0x00FF)
    fc_timer := primOrInt
        (primAndInt (primShlInt (head (drop 13 pkt)) 24) 0xFFF00000)
        (primAndInt (primShlInt (head (drop 12 pkt)) 16) 0xFFF00000)
        (primOrInt
            (primAndInt (primShlInt (head (drop 15 pkt)) 8) 0xFFF0)
            (primAndInt (head (drop 14 pkt)) 0x00FF))
if step < 5 then step := 5
Appendix E – BlueStack in Timber code

done

rfc_register_cfm pkt = action
  env.putStrLn("RFC_REGISTER_CFM")
  if (length pkt == 8) then
    loc_server_chan := primOrInt
      (primAndInt (primShlInt (head (drop 5 pkt)) 8) 0xFF00)
      (primAndInt (head (drop 4 pkt)) 0x00FF)
    if step < 3 then step := 3
done

rfc_start_ind pkt = action
  env.putStrLn("RFC_START_IND")
  mux_id := primOrInt
    (primAndInt (primShlInt (head (drop 13 pkt)) 8) 0xFF00)
    (primAndInt (head (drop 12 pkt)) 0x00FF)
  port_speed := primOrInt
    (primAndInt (primShlInt (head (drop 15 pkt)) 8) 0xFF00)
    (primAndInt (head (drop 14 pkt)) 0x00FF)
  if step < 12 then
    step := 12
done
  else
    if step==13 then
      step := 14
done

rfc_start_cfm pkt = action
  env.putStrLn("RFC_START_CFM")
  if (length pkt == 20) then
    case (head (drop 14 pkt)) of
      0x00 -> env.putStrLn("Connection established.")
        mux_id := primOrInt
          (primAndInt (primShlInt (head (drop 13 pkt)) 8) 0xFF00)
          (primAndInt (head (drop 12 pkt)) 0x00FF)
        port_speed := primOrInt
          (primAndInt (primShlInt (head (drop 17 pkt)) 8) 0xFF00)
          (primAndInt (head (drop 16 pkt)) 0x00FF)
      0x01 -> env.putStrLn("Connection established. Pending.")
        mux_id := primOrInt
          (primAndInt (primShlInt (head (drop 13 pkt)) 8) 0xFF00)
          (primAndInt (head (drop 12 pkt)) 0x00FF)
        port_speed := primOrInt
          (primAndInt (primShlInt (head (drop 17 pkt)) 8) 0xFF00)
          (primAndInt (head (drop 16 pkt)) 0x00FF)
    if step < 9 then step := 9
done

0x01 -> env.putStrLn("Connection established. Pending.")
  mux_id := primOrInt
    (primAndInt (primShlInt (head (drop 13 pkt)) 8) 0xFF00)
    (primAndInt (head (drop 12 pkt)) 0x00FF)
  port_speed := primOrInt
    (primAndInt (primShlInt (head (drop 17 pkt)) 8) 0xFF00)
    (primAndInt (head (drop 16 pkt)) 0x00FF)
  if step < 9 then step := 9
done
Appendix E – BlueStack in Timber code

0x06 -> env.putStrLn("Connection failed! Retrying.")
    step := 7
    done

_ -> env.putStrLn("Connection failed! "++show (trans2 (head (drop 14 pkt)))))
    step := 0
    done
    done

rfc_startcmp_ind pkt = action
    env.putStrLn("RFC_STARTCMP_IND")
    case (head (drop 6 pkt)) of
        0x00 -> env.putStrLn("Connection established.")
            mux_id := primOrInt
                (primAndInt (primShlInt (head (drop 5 pkt)) 8) 0xFF00)
                (primAndInt (head (drop 4 pkt)) 0x00FF)

        port_speed := primOrInt
            (primAndInt (primShlInt (head (drop 9 pkt)) 8) 0xFF00)
            (primAndInt (head (drop 8 pkt)) 0x00FF)

        if step < 16 then step := 16
        done

    0x01 -> env.putStrLn("Connection established. Pending.")
        mux_id := primOrInt
            (primAndInt (primShlInt (head (drop 5 pkt)) 8) 0xFF00)
            (primAndInt (head (drop 4 pkt)) 0x00FF)

        port_speed := primOrInt
            (primAndInt (primShlInt (head (drop 9 pkt)) 8) 0xFF00)
            (primAndInt (head (drop 8 pkt)) 0x00FF)

        if step < 16 then step := 16
        done

    0x06 -> env.putStrLn("Connection failed!")
        step := 15
        done

_ -> env.putStrLn("Connection failed! "++show (trans2 (head (drop 14 pkt)))))
    step := 0
    done
    done

rfc_close_ind pkt = action
    env.putStrLn("RFC_CLOSE_IND")
    done

rfc_establish_ind pkt = action
    env.putStrLn("RFC_ESTABLISH_IND")
    if step < 19 then step := 19
    done

rfc_establish_cfm pkt = action
    env.putStrLn("RFC_ESTABLISH_CFM")
case head (drop 8 pkt) of
    0x00 -> env.putsLn("rfc_establish_cfm = success")
    _ -> done
  done

done

rfc_release_ind pkt = action
env.putsLn("RFC_RELEASE_IND")
if step < 21 then step := 21
done

rfc_data_ind pkt = action
  env.putsLn("RFC_DATA_IND")
  if not((head (drop 4 pkt)==0xFF) &&
    ((head (drop 10 pkt)==0) ||
     (head(drop 11 pkt)==0))) then
    payload_length := primOrInt
    (primAndInt (primShlInt (head (drop 11 pkt)) 8) 0xFF00)
    (primAndInt (head (drop 10 pkt)) 0x00FF)
  if (length pkt) == (payload_length + 16) then
    env.putsLn("Receiving: "++map chr (drop 16 pkt))
    if (comp (drop 16 pkt) (map ord "close")) then
      step := 20
    if (comp (drop 16 pkt) (map ord "ping")) then
      --env.putsLn("RFC_DATA_REQ")
      env.putsLn("Sending: pong")
      bc.send_sequencing rfcomm ([0x18, 0x00]++
        [mux_id] ++ [0x00] ++
        [loc_server_chan] ++ [0x00] ++
        [credits] ++ [0x00] ++
        [primAndInt 0xFF (length "pong")] ++
        [primShrInt (length "pong") 8] ++
        [0x00, 0x00] ++
        (map ord "pong")
      )
      done
      if (comp (drop 16 pkt) (map ord "pong")) then
        pong_counter := pong_counter + 1
        if pong_counter < 10 then
          --env.putsLn("RFC_DATA_REQ")
          env.putsLn("Sending: ping")
          bc.send_sequencing rfcomm ([0x18, 0x00]++
            [mux_id] ++ [0x00] ++
            [loc_server_chan] ++ [0x00] ++
            [credits] ++ [0x00] ++
            [primAndInt 0xFF (length "ping")]) ++
            [primShrInt (length "ping") 8] ++
            [0x00, 0x00] ++
            (map ord "ping")
          )
          done
        else
          --env.putsLn("RFC_DATA_REQ")
          env.putsLn("Sending: close")
          --bc.send_sequencing rfcomm ([0x18, 0x00]++
            [mux_id] ++ [0x00] ++
            [loc_server_chan] ++ [0x00] ++
Appendix E – BlueStack in Timber code

```
--[credits] ++ [0x00] ++
--[primAndInt 0xFF (length "close")]] ++
--[primShrlnt (length "close") 8] ++
--[0x00, 0x00] ++
--(map ord "close")
step := 20

done

done

rfc_control_ind pkt = action
env putStrLn("RFC_CONTROL_IND")
done

rfc_parneg_ind pkt = action
env putStrLn("RFC_PARNEG_IND")
max_frame_size := min max_frame_size (primOrInt
  (primAndInt (primShrlnt (head (drop 9 pkt)) 8) 0xFF00)
  (primAndInt (head (drop 8 pkt)) 0x00FF))
if step < 17 then step := 17
done

rfc_parneg_cfm pkt = action
env putStrLn("RFC_PARNEG_CFM")
max_frame_size := primOrInt
  (primAndInt (primShrlnt (head (drop 9 pkt)) 8) 0xFF00)
  (primAndInt (head (drop 8 pkt)) 0x00FF)
if step < 11 then step := 11
done

in record
  --rf_send (c/r) (direction bit) (packet)
rf_send cr d pkt = action
send cr d pkt
done

rf_receive pkt = action
case (head pkt) of
  0x02 -> rfc_init_cfm pkt
done

  0x04 -> rfc_register_cfm pkt
done

  0x07 -> rfc_start_ind pkt
done

  0x08 -> rfc_start_cfm pkt
done

  0x09 -> rfc_startcmp_ind pkt
done

  0x0B -> rfc_close_ind pkt
done

  0x0E -> rfc_establish_ind pkt
done
```

- 59 -
Appendix E – BlueStack in Timber code

0x0F -> rfc_establish_cfm pkt  
done

0x11 -> rfc_release_ind pkt  
done

0x19 -> rfc_data_ind pkt  
done

0x1D -> rfc_control_ind pkt  
done

0x22 -> rfc_parneg_ind pkt  
done

0x23 -> rfc_parneg_cfm pkt  
done

_    -> env.putStrLn("Unknown packet:
++show (map trans2 pkt))

done

rf_setup b c = action
  d_mg := c
  bc := b
  done

rf_get_phandle = request
  return phandle

rfc_init_req = action
  env.putStrLn("RFC_INIT_REQ")
  bc.send_sequencing rfcomm ([0x01, 0x00] ++
    [primAndInt 0xFF phandle] ++
    [primShrInt phandle 8] ++
    [3,0,1,0,3,0,0,0,0,1,0,0,0,0,0,0,0,0])
  done

rfc_register_req = action
  env.putStrLn("RFC_REGISTER_REQ")
  bc.send_sequencing rfcomm ([0x03, 0x00] ++
    [primAndInt 0xFF phandle] ++
    [primShrInt phandle 8])
  done

rfc_start_req = action
  env.putStrLn("RFC_START_REQ")
  bc.send_sequencing rfcomm ([0x05, 0x00] ++
    (take 1 (drop 3 bd_addr)) ++ [0x00] ++
    (take 1 (drop 5 bd_addr)) ++
    (take 1 (drop 4 bd_addr)) ++
    (take 1 (drop 2 bd_addr)) ++ [0x00] ++
    (take 1 (drop 1 bd_addr)) ++
    (take 1 bd_addr) ++
    [psm_remote] ++ [0x00] ++
    [port_speed] ++ [0x00] ++
    [max_frame_size] ++ [0x00] ++
    [primAndInt 0xFF phandle] ++

  done
Appendix E – BlueStack in Timber code

done

rfc_start_res = action
env.putStrLn("RFC_START_RES")
bc.send_sequencing rfcomm ([0x06, 0x00] ++
 [mux_id] ++ [0x00] ++
 [0x01, 0x00] ++
 [port_speed] ++ [0x00] ++
 [max_frame_size] ++ [0x00] ++
 [primAndInt 0xFF phandle] ++
 [primShrInt phandle 8])
done

rfc_close_req = action
env.putStrLn("Closing connection.")
bc.send_sequencing rfcomm ([0x0A, 0x00] ++
 [mux_id] ++ [0x00])
done

rfc_establish_req = action
env.putStrLn("RFC_ESTABLISH_REQ")
bc.send_sequencing rfcomm ([0x0C, 0x00] ++
 [mux_id] ++ [0x00] ++
 [loc_server_chan] ++ [0x00] ++
 [rem_server_chan] ++ [0x00])
done

rfc_establish_res = action
env.putStrLn("RFC_ESTABLISH_RES")
bc.send_sequencing rfcomm ([0x0D, 0x00] ++
 [mux_id] ++ [0x00] ++
 [loc_server_chan] ++ [0x00] ++
 [0x01, 0x00])
done

rfc_release_req = action
env.putStrLn("RFC_RELEASE_REQ")
bc.send_sequencing rfcomm ([0x10, 0x00] ++
 [mux_id] ++ [0x00] ++
 [loc_server_chan] ++ [0x00])
done

rfc_data_req pkt = action
if (length pkt <= max_frame_size) then
  env.putStrLn("RFC_DATA_REQ")
  bc.send_sequencing rfcomm ([0x18, 0x00] ++
  [mux_id] ++ [0x00] ++
  [loc_server_chan] ++ [0x00] ++
  [credits] ++ [0x00] ++
  [primAndInt 0xFF (length pkt)] ++
  [primShrInt (length pkt) 8] ++
  [0x00, 0x00] ++
  pkt)
done
else
  env.putStrLn("The payload is too big!\nThe maximum length is " ++ show max_frame_size)
Appendix E – BlueStack in Timber code

env.putStrLn("bytes.")
done

rfc_parneg_req = action
bc.send_sequencing rfcomm ([0x20, 0x00] ++
  [mux_id] ++ [0x00] ++
  [loc_server_chan] ++ [0x00] ++
  [rem_server_chan] ++ [0x00] ++
  [max_frame_size] ++ [0x00] ++
  [credit_flow_ctrl] ++ [0x00] ++
  [initial_credits] ++ [0x00])
done

rfc_parneg_res = action
env.putStrLn("RFC_PARNEG_RES")
bc.send_sequencing rfcomm ([0x21, 0x00] ++
  [mux_id] ++ [0x00] ++
  [loc_server_chan] ++ [0x00] ++
  [max_frame_size] ++ [0x00] ++
  [credit_flow_ctrl] ++ [0x00] ++
  [initial_credits] ++ [0x00])
done

get_step = request return step

set_step i = action
  step := i
done

-- To translate packets to hex
--
trans2 :: Int -> String
trans2 b = trans (primShrInt b 4) ++ trans (primAndInt 0xF b)
where
  trans a =
    case a of
      0 -> "0"
      1 -> "1"
      2 -> "2"
      3 -> "3"
      4 -> "4"
      5 -> "5"
      6 -> "6"
      7 -> "7"
      8 -> "8"
      9 -> "9"
      10 -> "A"
      11 -> "B"
      12 -> "C"
      13 -> "D"
      14 -> "E"
      15 -> "F"
      _ -> "#"

-- To test as a master
test_mst :: StdEnv -> DM -> RFpacking -> HCI -> BCSP -> BTHandler -> Template Action

In let
tes = action

step <- rf.get_step
--env.putStr("step: " ++ show step)
case step of
  1 -> rf.rfc_register_req
       after 4000000 tes
       rf.set_step 2
       done

  3 -> rf.rfc_init_req
       after 4000000 tes
       rf.set_step 4
       done

  5 -> d.dm_am_register_req
       after 4000000 tes
       done

  7 -> rf.rfc_start_req
       after 8000000 tes
       rf.set_step 8
       done

  9 -> rf.rfc_parneg_req
       after 4000000 tes
       rf.set_step 10
       done

 11 -> rf.rfc_establish_req
       after 4000000 tes
       rf.set_step 12
       done

 13 -> rf.rfc_send 1 1 (map ord "ping")
       after 1000000 tes
       env.putStrLn("sending: \"ping\"")
       rf.set_step 14
       done

 20 -> rf.rfc_release_req
       after 1000000 tes
       rf.set_step 21
       done

 21 -> rf.rfc_close_req
       after 1000000 tes
       rf.set_step 22
       done

 22 -> bc.close_down
       after 1000000 tes
       bt.clos
rf.set_step 23
done

23 -> env.quit
done

_ -> if not(step == 24) then after 4000000 tes
done

in tes

------------------------------------------------------------------
-- To test as a slave
------------------------------------------------------------------

test_slv :: StdEnv -> DM -> RFpacking -> HCI -> BCSP -> BTHandler -> Template Action
test_slv env d rf hc bc bt=
template
in let
tes = action
    step <- rf.get_step
    --env.putStr("step: "++show step)
case step of
    1 -> rf.rfc_register_req
        after 1000000 tes
        rf.set_step 2
        done

    3 -> rf.rfc_init_req
        after 1000000 tes
        rf.set_step 4
        done

    5 -> d.dm_am_register_req
        after 1000000 tes
        rf.set_step 6
        done

    7 -> d.hci_write_pagescan_activity
        after 1000000 tes
        rf.set_step 8
        done

    9 -> d.hci_write_scan_enable
        after 1000000 tes
        rf.set_step 10
        done

    12 -> rf.rfc_start_res
        after 1000000 tes
        rf.set_step 13
        done

    14 -> rf.rfc_start_res
        after 1000000 tes
        rf.set_step 15
        done
Appendix E – BlueStack in Timber code

17 -> rf.rfc_parneg_res
    after 1000000 tes
    rf.set_step 18
    done

19 -> rf.rfc_establish_res
    after 1000000 tes
    rf.set_step 0
    after 500000 (env.putsStrLn(
        "012A received \"ping\" is responded with a \"pong\"
      +
        \nA received \"pong\" is responded with a \"ping\"
      +
        \nAfter 10 received \"pong\" the connection is closed
      +
        \nA received \"close\" closes the connection))
    done

20 -> rf.rfc_release_req
    after 1000000 tes
    rf.set_step 21
    done

21 -> rf.rfc_close_req
    after 1000000 tes
    rf.set_step 22
    done

22 -> bc.close_down
    after 1000000 tes
    bt.close
    rf.set_step 23
    done

23 -> env.quit
    done

_ -> if not(step == 23) then after 1000000 tes
    done

in tes

prog :: StdEnv -> RFpacking -> HCI -> Action -> Template Action
prog env rf hc t=
    template
    in let
        pr = action
            after 500000 t
    in pr

main :: StdEnv -> Template Program
main env =
    template
    in record
        start = action
        hc <- hci env
        rf <- rffield env
        d <- dm env rf
        bt <- bt_handler env d rf hc

- 65 -
Appendix E – BlueStack in Timber code

bc <- bcsp env bt
t <- test_mst env d rf hc bc bt
pr <- prog env rf hc t
bt.set bc
bc.setup pr

d.dm_setup bc
hc.hci_setup bc
rf.rf_setup bc d