

Universität Karlsruhe (TH)

Research University · founded 1825

System Architecture Group
Department of Computer Science

Study Thesis

Implementation of Flow Control in the Linux Bluetooth Stack BlueZ

by

cand. inform.

Martin Röhrich

Supervisor:

Prof. Dr. Frank Bellosa

Dr. Thomas Fuhrmann

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, October 01, 2006

.....
Martin Röhrich

Acknowledgements

I would like to thank my supervisor Dr. Thomas Fuhrmann and Prof. Dr. Frank Bellosa who gave me the opportunity to work on the Linux kernel in a real life university's project. Furthermore I would like to thank the maintainer of the Bluetooth subsystem in the Linux kernel, Marcel Holtmann, as well as my fellow students Johannes Singler and Christoph Mallon for the help they provided me during my first steps and setups.

This work could have never been accomplished without the invaluable support of my parents and my girlfriend Karin. Thank you so much.

Contents

| | |
|---|-----------|
| 1. Introduction | 4 |
| 1.1. The Bluetooth Protocol Stack | 5 |
| 1.2. The Logical Link Control and Adaptation Protocol (L2CAP) | 5 |
| 1.2.1. Terminology | 8 |
| 1.3. Related Work | 8 |
| 2. Linux Kernel Development | 9 |
| 2.1. Socket Buffers | 9 |
| 2.1.1. Lists of Socket Buffers | 11 |
| 2.2. Portability | 12 |
| 2.2.1. Alignment and Endianness | 13 |
| 2.2.2. Concurrency and Race Conditions | 14 |
| 3. Configuration Process | 16 |
| 3.1. Information Retrieval | 16 |
| 3.1.1. Information Request | 16 |
| 3.1.2. Information Response | 16 |
| 3.1.3. Extended Features Mask | 17 |
| 3.2. Configuration Request | 18 |
| 3.3. Configuration Response | 20 |
| 3.4. Retransmission and Flow Control Option | 21 |
| 4. Procedures for Flow Control | 24 |
| 4.1. Variables and Sequence Numbers | 27 |
| 4.1.1. The sending peer | 27 |
| 4.1.2. The receiving peer | 28 |
| 4.2. Retrieving Data | 29 |
| 4.2.1. Invalid Frame Detection | 29 |
| 4.2.2. Process the Send Sequence Number TxSeq | 30 |
| 4.2.3. Process the Receive Sequence Number ReqSeq | 30 |
| 4.2.4. Reassemble the SDU | 31 |
| 4.3. Transmitting Data | 33 |
| 4.3.1. Sending Acknowledgments | 35 |
| 4.3.2. Monitor and Retransmission Timer | 36 |

Contents

| | |
|---|-----------|
| 5. Summary | 38 |
| 5.1. Testing and Analysis | 38 |
| 5.2. Conclusion and Future Work | 40 |
| A. Architectures' Data Types | 41 |
| B. Development Notes | 42 |
| B.1. Bluetooth Adapters Used | 43 |

Executive Summary

Bluetooth is going to be used as one of the major digital data transport protocols mainly for small and embedded devices. The focus was laid on small hardware development costs, low power consumption and a space-saving design. Bluetooth adapters are included nowadays in almost every new laptop and mobile phone. With the rising acceptance of this standard the needs to fulfill different tasks rised, as well. Besides the basic functionality of sending and receiving data from a technical point of view, it became urgent for companies to see some quality of service being specified by the Bluetooth Special Interest Group (SIG). One of these extensions is flow control mode. This mode is meant to prevent devices flooding the remote device with data, that is sent so quickly, that the receiving device is not able to process the incoming data faster than new data is passed into its buffer. Flow control instead provides mechanisms, that enable devices to negotiate appropriate values for their buffers to protect them from buffer overflows.

The Linux kernel, which builds the heart of the GNU/Linux operating system, offers its users an existing implementation of the Bluetooth software stack, called BlueZ. This implementation is the equivalent to so many existing proprietary Bluetooth software stacks, but being open source and freely available by no costs makes Linux an ideal development platform.

This thesis is about an implementation of Flow Control in the L2CAP layer for Linux' Bluetooth subsystem BlueZ.

1. Introduction

The Bluetooth technology was developed by Ericsson, Nokia, Intel, IBM and Toshiba in the mid-1990's. These companies built the core of the Bluetooth Special Interest Group (SIG) which serves as an official consortium and is composed of 6123 members in the meantime.¹

The Bluetooth wireless technology is focused mainly on short-range communication and domain-specific applications. Because of its low energy consumption and low costs it perfectly fits the needs of small embedded devices such as Notebooks, PDAs or mobile phones. The official Bluetooth website *bluetooth.com* describes the technology as follows:

“Bluetooth wireless technology is a short-range communications technology intended to replace the cables connecting portable and/or fixed devices while maintaining high levels of security. The key features of Bluetooth technology are robustness, low power, and low cost.”²

The domain specific applications—also known as *Profiles*—build a characteristic foundation of the technology which distinguishes it from most other wireless technologies. Instead of just specifying the core technical parts, the Bluetooth specification defines some core profiles for which the technology is meant to be used and which may be supported by Bluetooth compliant devices. Currently there are 33 unique profiles defined (e.g. Audio/Video Remote Control Profile (AVRCP), Basic Imaging Profile (BIP), Basic Printing Profile (BPP), Bluetooth Network Encapsulation Protocol (BNEP), Dial-up Networking Profile (DUN), Fax Profile (FAX), File Transfer Profile (FTP), Human Interface Device Profile (HID), Object Exchange (OBEX), Personal Area Networking Profile (PAN), Radio Frequency Communication (RFCOMM) or Service Discovery Protocol (SDP)).³

The use of fixed, defined profiles enables companies to guarantee specific interoperability by fulfilling the corresponding qualification processes. This is an advantage in the low end sector of embedded devices [see also [Wollert, 2002](#), page 27].

¹This statistics arise from September 2006. The membership is separated into *Promoter Members* (8 companies), *Associate Members* (231) and *Adopter Members* (5884)—only the latter one is free of charge. The Promoter Members consist of the founder companies (with Lenovo (Singapore) Pte Ltd. instead of IBM) plus Agere Systems, Microsoft Corporation and Motorola, Inc.

²See <http://www.bluetooth.com/Bluetooth/Learn/Basics/> (last visit 27 Sep 2006)

³For a complete list with short explanations refer to http://bluetooth.com/Bluetooth/Learn/Works/Profiles_Overview.htm (last visit 27 Sep 2006)

Figures within this text are taken from the official specification if not otherwise stated.

1.1. The Bluetooth Protocol Stack

The Bluetooth standard has many protocols which are separated into loosely coupled layers. The Bluetooth stack (see figure 1.1) does neither follow the ISO/OSI model, nor is it applicable to the TCP/IP model [see [Tanenbaum, 2003](#), Chapter 4.6.3]. The IEEE standardization committee mapped the core Bluetooth technology into its own 802 model, so that Bluetooth became an own IEEE standard 802.15.

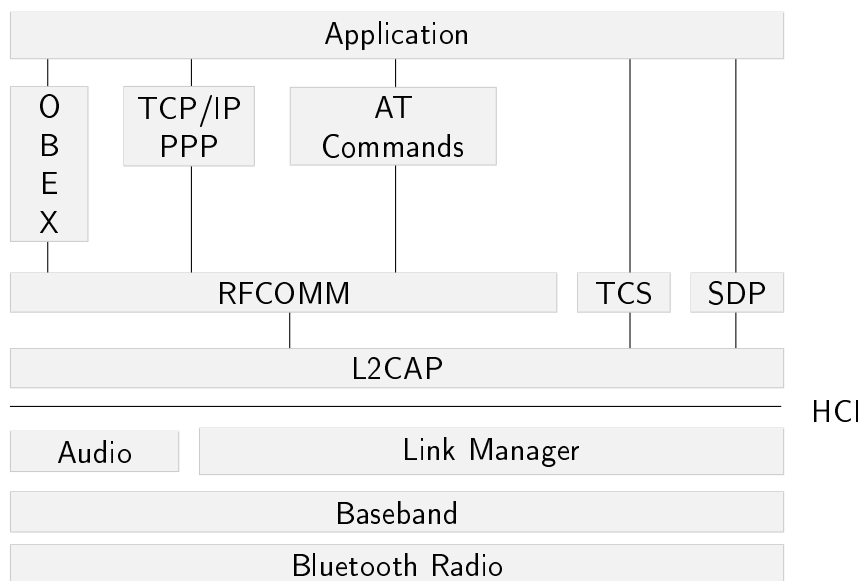


Figure 1.1.: Overview of the Bluetooth Stack

1.2. The Logical Link Control and Adaptation Protocol (L2CAP)

All Bluetooth devices provide only the lower protocol layers within the controller's firmware. The higher level protocols and layers are implemented in a separated software stack. The Host Controller Interface (HCI) builds the bridge between the lower layers and the L2CAP layer. Hence profiles are implemented within the upper layers. [see also [Wollert, 2002](#), Chapter 4].

The L2CAP layer has three main functions:

1. Introduction

1. L2CAP accepts frames with up to 64KB of length. Applications may usually require to send data packets that exceed this limitation, whereas the baseband on the other hand has an even more intense constraint towards smaller frame sizes. This forces the L2CAP layer to segment big data packets into smaller frames on the sending side, and to reassemble frames on the receiving side.
2. Higher level protocol multiplexing. Due to the lack of an appropriate *type* field in the lower layers identifying higher level protocols it is L2CAP's responsibility to distinguish protocols such as RFCOMM, SDP, or Telephony Control on its own.
3. L2CAP is responsible for all parts of Quality of Service. This includes the connection setup as well as necessary negotiations regarding sending and receiving parameters and aspects like flow control and retransmission.

L2CAP supports connection-oriented data services as well as connectionless ones for higher layer protocols. The concept is based on per-connection channels which provides the ability to transfer multiple protocols over one asynchronous link simultaneously.

Figure 1.2 gives an overview of the internal L2CAP architectural block as described by the specification. The Channel Manager is responsible for all kinds of signalling. It provides the control functionality and manages signals to and from the lower layers, the upper layers and the Resource Manager. The Retransmission and Flow Control part is located within the Resource Manager. This block may provide applications per-channel flow control and retransmission. The Resource Manager is responsible for all transmission and reception activities of packets related to L2CAP channels.

Some core parts of the L2CAP layer are to be explained in more detail.

Segmentation and Reassembly This technique takes place between the upper layer protocols that are directly related to applications (like TCP/IP or RFCOMM) and the L2CAP layer. The data packet that is exchanged between those layers is called a Service Data Unit (SDU) with the Maximum Transmission Unit (MTU) as its maximum size. The application is totally decoupled from the required segmentation that needs to be done to map the application packets into the lower layer frames.

Flow Control per channel L2CAP offers the optional feature of per-channel Flow Control which is the base of this work. Providing this technique within a protocol layer for all tasks prevents applications from being forced to implement it on their own. Instead of a simple stop-and-go Flow Control scheme as employed in the baseband, a more efficient window-based Flow Control scheme is provided. This feature is an optional part of the specification.

Error Control and Retransmission An extra error checking mechanism within the L2CAP layer is provided for applications that require a lower error rate than the baseband can deliver. An error check (based on a CRC checksum) is used in

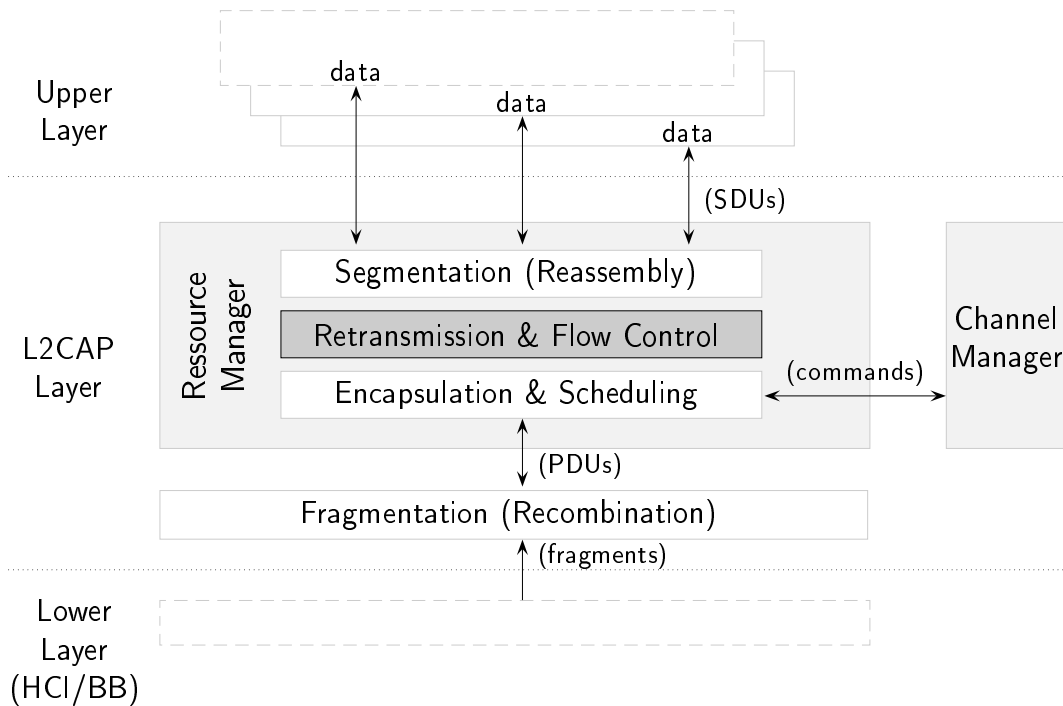


Figure 1.2.: L2CAP architectural block [see [BlueSpec](#), Vol. 4 page 22]

the Retransmission and Flow Control mode. The optional Retransmission mode shall provide a reliable connection by protecting the device from packet loss.

Fragmentation and Recombination Exchanging data between L2CAP and the Host Controller Interface is performed by using Protocol Data Units (PDUs) which contain the entire L2CAP protocol information besides the optional upper layer information data. Splitting PDUs into smaller fragments is called fragmentation; the reverse operation is called recombination. Both techniques ensure proper collaboration with the lower layer. Note that the specification leaves it open to the actual implementation if this procedure is part of the L2CAP layer itself, or if it occurs in the HCI driver, and/or in the Controller. Fragmentation and recombination is not part of the BlueZ L2CAP layer.

L2CAP implements major parts of the data link layer of the ISO/OSI model, even if the ISO/OSI stack cannot be compared strictly to the one of the Bluetooth specification. Those parts include the insurance of a reliable connection by splitting the byte stream into blocks (named frames) and using sequence numbers and frame check sequences. The ability to send acknowledgements and to retransmit frames that are considered lost, as well as a window based flow control mechanism can be accomplished by L2CAP's new retransmission and flow control option on which this thesis is based.

1. Introduction

1.2.1. Terminology

It is indeed important to keep the terminology in mind. The *SDU* (Service Data Unit) is the data packet that is to be exchanged between the L2CAP layer and the upper layers (refer to figure 1.2). The SDU is the plain data unit that is received by L2CAP from the upper layer and does not contain any L2CAP specific header information. Therefore the techniques applied to SDUs are called segmentation and reassembly.

Instead, the *PDU* (Protocol Data Unit) is the data packet containing L2CAP specific protocol information, control information and/or upper layer information data and is always preceded by an L2CAP header. The PDU can be of types *B-frame*, *C-frame*, *G-frame*, *I-frame* and *S-frame*—the latter two are used within Flow Control mode, the first one (B-frame) is used in basic L2CAP mode. Fragments are used in the delivery of data to and from the lower layer. Therefore the corresponding actions applied to fragment PDUs are called fragmentation and recombination.

The *MTU* (Maximum Transmission Unit) describes the maximum size of payload data that can be exchanged between the L2CAP layer and the upper layer entity—this corresponds to the maximum SDU size respectively.

The *MPS* (Maximum PDU Payload Size) on the other hand corresponds to the size that the L2CAP layer entity is capable of accepting from the lower layer entity. Note that either in Basic L2CAP mode, or if no segmentation is used in Flow Control mode, the MTU equals the MPS.

1.3. Related Work

Multiple Bluetooth stacks are widely used. Especially for the Microsoft Windows operating system exist several different proprietary Bluetooth stacks albeit none of them is open source. Besides the other common open source BSD implementations there are two Bluetooth stack implementations for Linux which are both released under the terms of the General Public License (GPL). One of them is called AFFIX⁴ and was primarily developed by Nokia. This stack is totally independent of the currently used official stack BlueZ on which this thesis is based. AFFIX already implements some *parts* of the newly introduced features of the Bluetooth specification 1.2, e.g. Retransmission and Flow Control. Further development of this stack slowed down during the past years—only security patches were introduced since the last major release 3.2.0 in July 2004.

⁴<http://affix.sourceforge.net>

2. Linux Kernel Development

Developing inside the kernel of an operating system like GNU/Linux involves some deeper knowledge and a solid understanding of specific parts of the development process as a whole. We will now cover some of these aspects which are essential for any kernel developer. Helpful in this regard are [Love, 2005] for a good overview of the kernel's implementation and design, [Bovet and Cesati, 2006] for a very detailed explanation of the Linux kernel, [Corbet et al., 2005] and [Quade and Kunst, 2006] for the purpose of Linux device driver's development and finally [Benvenuti, 2006] for the networking internals of the kernel.

2.1. Socket Buffers

The socket buffer structure `sk_buff` is a critical data structure that is used widely within all networking code in the Linux kernel. Its main use lies in an abstract and generic packet storage for all different network layers. Within this data structure all headers and the information payload are stored along with further information for some internal coordination. Some of the fields are not self-explanatory, that's why we will discuss this data structure in detail and go over some of the functions and macros that manipulate those fields. For a more detailed discussion refer to [Benvenuti, 2006, Chapter 2].

The `sk_buff` structure represents headers for data that has been received or is to be transmitted. As it is used by all different layers and by all different protocols makes it a huge structure with a tremendous number of variables. The main diagram of this structure is shown in figure 2.1.

The `head` pointer points to the first byte that is returned by `kmalloc` at buffer allocation time and is never be adjusted afterwards. The `data` pointer returns the beginning of the *data* which consists of headers and the information payload. This pointer can be adjusted to each layer's needs. `tail` points to the end of the data section within the buffer, precisely to the first byte after the data section. This pointer may be adjusted. The `end` pointer points to the beginning of the so-called `skb_shared_info` structure which is used to keep additional information about the data block. This structure is mainly used to transfer paged data but is not involved in the work of this thesis.

Working within the L2CAP layer means we use an already allocated socket buffer in case of frame reception (via `l2cap_data_channel()`) but have to allocate memory if an application needs to send data (via `l2cap_sock_sendmsg()`). Memory allocation for

2. Linux Kernel Development

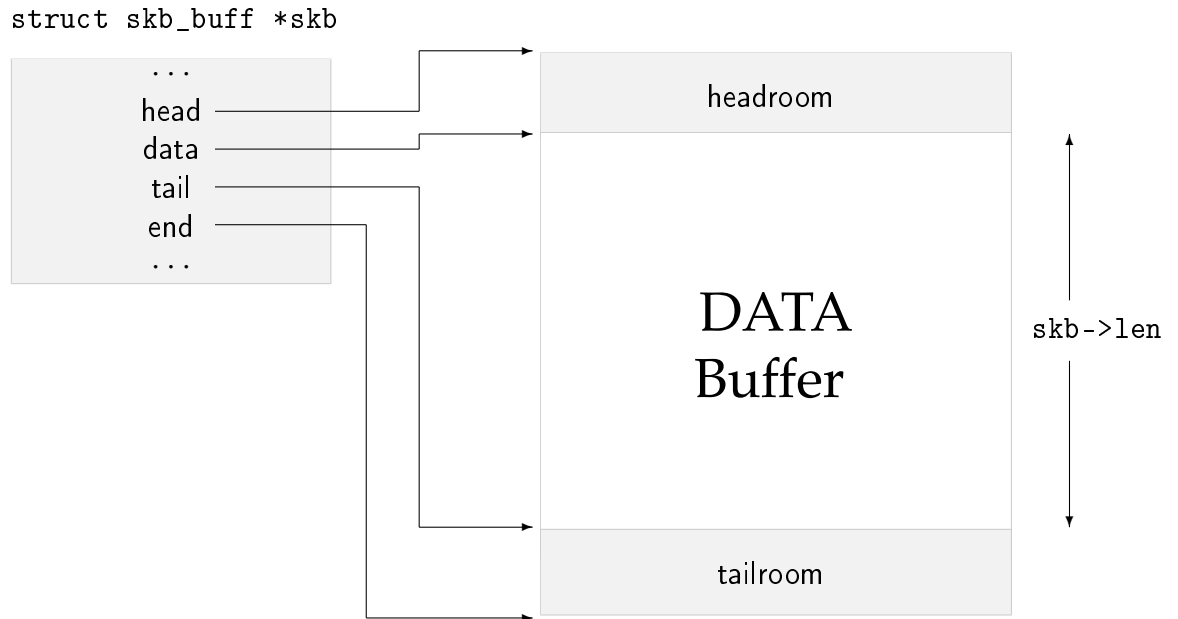


Figure 2.1.: The `sk_buff` struct and their head/end versus data/tail pointers

socket buffers is done in BlueZ by using `bt_skb_send_alloc()`. After that we make use of three different management functions. Note that none of these functions actually copies data from or into the buffer—only pointers are adjusted accordingly.

- `skb_put` extends the tail pointer by a given length, increases the `len` field and returns the old position of the tail pointer. This is a major function used whenever we copy some data into the socket buffer e.g.:

```
memcpy(skb_put(l2cap_pi(sk)->sdu, skb->len), skb->data,
        skb->len);
```

or if we reserve some space for a given struct, e.g.

```
/* Create L2CAP header */
lh = (struct l2cap_hdr *) skb_put(skb, L2CAP_HDR_SIZE);
lh->cid = __cpu_to_le16(l2cap_pi(sk)->dcid);
lh->len = __cpu_to_le16(len + (hlen - L2CAP_HDR_SIZE));
```

- `skb_pull` is used to pull the data pointer towards the tail by a given length and returns the new data pointer; the `len` field is decreased respectively. This can be used to remove data from the front of a buffer and is widely used in each layer to remove the layer's header from the packet.
- `skb_trim` trims the socket buffer to a given length and moves the tail pointer towards the head. This function is useful if we want to cut a trailer off the packet like a given frame check sequence.

The latter two functions are used in combination within `l2cap_reassembly()`:

```
/* remove header, control and fcs from i-frame */
skb_pull(skb, L2CAP_HDR_SIZE + L2CAP_CONTROL_SIZE);
skb_trim(skb, skb->len - L2CAP_FCS_SIZE);
```

Figure 2.2 illustrates how an I-frame is stored within a socket buffer. The headroom and tailroom is not regarded and may or may not be filled with data.

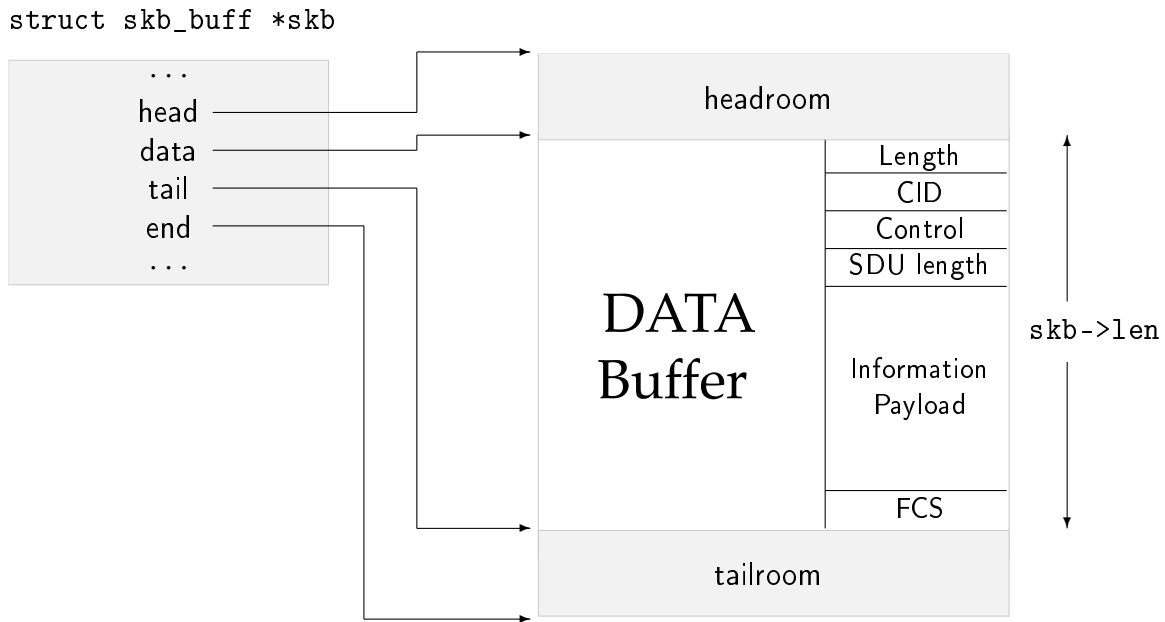


Figure 2.2.: The `sk_buff` struct with the I-frames field in the data section

2.1.1. Lists of Socket Buffers

To store more than one corresponding socket buffer, a so-called socket buffer list (`struct sk_buff_head`) is used. The definition is quite simple:

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff *next;
    struct sk_buff *prev;

    __u32          qlen;
    spinlock_t     lock;
};
```

The implementation follows a doubly linked list where `qlen` represents the number of packets on this list and `lock` can be used for an SMP-aware locking policy. Note that “the first two elements of both `sk_buff` and `sk_buff_head` are the same: the next and

2. Linux Kernel Development

prev pointers. This allows the two structures to coexist in the same list. [...] In addition, the same functions can be used to manipulate both `sk_buff` and `sk_buff_head`.” [Benvenuti, 2006]

Such a list was introduced by this implementation for the transmission *TxQueue* in *l2cap.h*. The functions and macros to manipulate these lists are straight forward. Three of them are used within this implementation:

- `skb_queue_head_init` is an initialization function for a `sk_buff_head` object.
- `skb_queue_tail` puts a socket buffer at the tail of a list.
- `skb_dequeue` removes a socket buffer from the head of the queue and returns it. If the list is empty NULL is returned respectively.

Some more information about how socket buffers work can be found online on Dave M. Miller’s web pages: <http://vger.kernel.org/~davem/skb.html>

2.2. Portability

Linux is an extremely portable operating system which supports all different flavours of hardware. Programming inside the kernel implies programming in a portable manner. One major issue concerning the portability of C Code is the size of specific data types. A *word* is known as the entity which a machine can process in one cycle—this is the size of a general purpose register within a CPU. The size of a word equals the size of a pointer. But the size of a word counted in bits differs from one architecture to another. On widely used Intel IA32 CPUs one word is four bytes long which corresponds to 32 bit. But on a new Intel IA64 CPU a pointer is 64 bits wide. It gets even trickier because if we work on a 64 bit machine we cannot make any assumption about the size of an integer in bits. Appendix A gives an overview of the different data type sizes of different architectures.

For this reason the kernel developers introduced *Opaque Data Types* which do not offer their internal format or their actual size. `pid_t` or `atomic_t` are examples for such data types—the developer should never make any assumptions about the actual size of them. Furthermore there are *Special Types* and *Explicitly Sized Types*. Special Types must be stored in an explicit manner, e.g. the types `jiffy` or `flags` are always stored as `unsigned long`. The Explicitly Sized Types are of much greater interest in this implementation project, especially regarding the networking part of the L2CAP module.

Explicitly Sized Types help the developer to ensure meeting the requirements of the specification. If a networking packet follows a specific composition we want to ensure access of portions of this packet in the right manner. This means we need to know how to access a byte or a 16 bit field. Therefore each architecture within the kernel defines corresponding data types for generic data types with an explicit size. That means that no matter which architecture we are currently running the kernel on,

an unsigned byte is always represented by `u8`, an unsigned 16 bit integer by `u16`; the same applies to `u32` and `u64`, respectively.

2.2.1. Alignment and Endianess

Alignment describes the positioning of a data segment within memory. A variable is naturally aligned if it is stored at a location which is a multiple of its size. For example a 32 bit type is naturally aligned if it is stored at a memory location which is a multiple of 4 (so that the two least significant bits are zero). Some architectures are quite strict concerning these alignments; especially in most RISC architectures a load of mis-aligned data causes an expensive CPU-trap. Modern compilers like GCC avoid those alignment problems automatically.

But this dependancy becomes a problem whenever one uses complex data structures like the commonly used `struct` in C because the alignment of a `struct` is always the alignment of its biggest type. Therefore the compiler uses a technique called padding which enlarges structures with differently sized internal types even though this is not visible by the developer. This is very undesirable if we work with network packets that are usually not naturally aligned—they are packed together by all different fields of different sizes.

But GCC permits us to pack those structures without any padding. Responsible is a compiler specific attribute called `__attribute__((packed))`. This is used within all different kinds of frame and packet structures in the L2CAP layer. For example:

```
struct l2cap_cmd_hdr {
    __u8      code;
    __u8      ident;
    __le16    len;
} __attribute__((packed));
```

Unaligned data within a byte stream should be accessed by using the two macros `get_unaligned()` and `put_unaligned()` defined in `<asm/unaligned.h>`. Those macros work for every data item, no matter if it is one, two, four or eight bytes long (see also [Corbet et al., 2005, page 294]).

Another major issue is the *byte-order* of the underlying architecture. The byte-ordering describes the order in which bytes are stored within one word. The processor can order the bytes within one word either in storing the least significant byte at the first position or at the last position. The two methods are called *Big Endian* and *Little Endian*. Working with networking packets enforces the precise definition in which way data is going to be sent and read over the connection.

The conversion to or from the host to the network is a major concern. Not every architecture is fixed in its endianess—the endianess that should be used on an IA64 machine for example, can be chosen by the user. Therefore the developer has to ensure that data is always processed correctly without assuming anything of the underlying machine. Generic kernel macros help the developer to convert the network data to the host back and forth.

The Bluetooth specification defines its data to be in little endian format. That means we mainly use macros like `__le16_to_cpu()` or `__cpu_to_le16()`. The same can be done with data of size 32 bit or 64 bit.¹ If the underlying architecture uses the same endianness as the network packet, say an Intel IA32 machine with little endian order, the macro is empty and no conversion takes place.

2.2.2. Concurrency and Race Conditions

Concurrency describes the situation in which the system tries to do more than one thing at once. This is not a problem on a simple single processor system, but multiprocessor systems and a preemptible kernel means that we can lose the processor at any given time. Further reasons for concurrency besides multiprocessor systems and kernel preemption are asynchronous Interrupts, Softirqs and Tasklets. [see also [Love, 2005](#), page 176]

“Concurrency-related bugs are some of the easiest to create and some of the hardest to find.” [[Corbet et al., 2005](#), page 106]

Concurrency can cause *Race Conditions*, *Deadlocks* and *Starvation* of different processes.² That means access to shared data has to be controlled by the programmer, otherwise the results are unpredictable. Operations on data structures must be executed atomically. Larger sections of code need to be protected within a critical section; that is code that can be executed by only one thread of execution. The Linux kernel offers a wide variety of different primitives for different needs.

One of the first facilities included was the so-called *big kernel lock*. This turned the entire kernel into one big critical section which allowed the developers to work on SMP systems. But this solution did not scale very well and shortly new techniques were introduced to offer the developer a finer-grained locking scheme.

Today’s Linux kernels come with a variety of mechanisms that allow protection of critical sections and atomic operations. The kernel offers Reader/Writer Semaphores (including Mutexes), atomic variables (type `atomic_t`), atomic bit operations (`set_bit`, `clear_bit`, `change_bit`, ...) Seqlocks and Reader/Writer Spinlocks for code that cannot sleep, like interrupt handlers. Even if all specific needs seemed to be addressed by these techniques,

“fine-grained locking comes at a cost, however. In a kernel with thousands of locks, it can be very hard to know which locks you need—and in which order you should acquire them—to perform a specific operation.” [[Corbet et al., 2005](#), page 123]

Some data structures like sockets and socket buffers provide their own locking scheme. The most critical data structure in our code is the socket specific `struct l2cap_pinfnfo` which maintains the device’s L2CAP channel and socket info.

¹Note that an 8 bit field (one byte) does not need any conversion.

²For an indepth discussion about these topics refer to [[Tanenbaum, 2001](#), chapters 2 and 3]

More about the kernel portability and synchronization issues can be found in [Love, 2005, Chapter 8, 9 and 19], [Corbet et al., 2005, Chapter 5 and 11] and [Quade and Kunst, 2006, Chapter 6.5 and 9.3].

3. Configuration Process

Before we actually use flow control between two entities, the two devices need to negotiate appropriate settings for their channel utilization. This is all done during the configuration process which starts immediately after the connection establishment when we start into a so-called *BT_CONNECT* state. Each device needs to configure specific parameters for its incoming and outgoing logical link before attempting to use the channel for data transmission. Using information requests is an optional way to retrieve specific information about the peer's further abilities albeit it is mandatory to use information requests prior to using extended features such as the retransmission and flow control mechanism in a configuration request. Any incoming information request *shall* be answered by an appropriate information response.

3.1. Information Retrieval

The information retrieval consists of two possible signalling packets, namely information requests and information responses. They are used prior to the actual configuration requests to ask for implementation specific features from the remote entity. If we want to use flow control within the L2CAP layer it is mandatory to use information requests before we put the retransmission and flow control option into a configuration request packet.

3.1.1. Information Request

The information request checks the peer's ability for optional features such as flow control mode. The data packet itself consists of the unique code (0x0A) followed by a unique channel identifier, the fixed length (2 octets) and the information payload in the form of an *InfoType* field which consists of either 0x0001 for the request of a connectionless MTU or 0x0002 for the request of extended features. We are mainly interested in the latter one. All other values are currently reserved for future use.

3.1.2. Information Response

The information response must be sent upon receiving a valid information request. The response data packet consists of the code (0x0B), the unique identifier which

matches the request's one, a variable length field, the InfoType, a result code, and the optional data.

The InfoType field matches the value of the requested information, i.e. 0x0002. The result can basically indicate either success (0x0000) or no available support (0x0001). The contents of the data field are dependant on the InfoType. In case of the retransmission and flow control feature that we are to request it contains 4 octets which consist of the extended features mask.

3.1.3. Extended Features Mask

To make Flow Control usable we have to promote this feature as being accessible in our implementation. This is done by setting the corresponding Bit in the Data field of the Information Response Packet as being described in [Vol. 4 of [BlueSpec](#), section 4.12, page 56]:

“The features are represented as a bit mask in the Information Response data field (see Section 4.11 on page 55). For each feature a single bit is specified which shall be set to 1 if the feature is supported and set to 0 otherwise. All unknown, reserved, or unassigned feature bits shall be set to 0”.

| No. | Supported feature | Octet | Bit |
|-----|-------------------------------------|-------|-----|
| 0 | Flow Control Mode | 0 | 0 |
| 1 | Retransmission Mode | 0 | 1 |
| 2 | Bi-directional QoS | 0 | 2 |
| 31 | Reserved for feature mask extension | 3 | 7 |

Table 3.1.: Extended Features Mask

Incoming information requests are served in our code by `l2cap_information_req()` and incoming information responses are served by `l2cap_information_rsp()`. An information request is issued either after the reception of a connection response to start the configuration process or after the reception of a configuration request if we did not already send an information request to the remote peer.

Once we receive an information request we prepare an appropriate information response immediately. The InfoType field of the corresponding request serves as an indicator for the length and data portion of the response packet. This means either a connectionless MTU size or extended features are requested—everything else will be treated as unsupported in the response's result code.

3.2. Configuration Request

A configuration request is an essential signalling packet for the purpose of negotiation:

“Configuration Request packets are sent to establish an initial logical link transmission contract between two L2CAP entities and also to re-negotiate this contract whenever appropriate”. [BlueSpec, page 47]

Each configuration request is exclusively related to either the incoming or the outgoing traffic. In case of the retransmission and flow control option, such a configuration request determines the incoming values of a sender’s request. For example, consider device *A* sending a request to device *B* wishing to configure retransmission and flow control with a TxWindow size of 32. This value serves as the window size for incoming traffic related from device *B* to device *A* even if the remote peer (here device *B*) certainly has to keep track of negotiated values and is therefore also involved.

Even if all default values are acceptable from the beginning a configuration request needs to be sent in order to proceed with data transmission. In such a case it is perfectly valid to send an empty request and no explicit options.

Device’s configuration requests refer always on their own:

“The configuration request cannot request a change in the parameters the device receiving the request will accept”. [BlueSpec, page 47]

The architecture of a configuration request packet is shown in figure 3.1. It consists of six fields, namely the unique code (0x04), the identifier, the whole packet’s length (the variable size depends on the following options), the destination’s CID, a 16 bit flags field¹, and finally the configuration options.

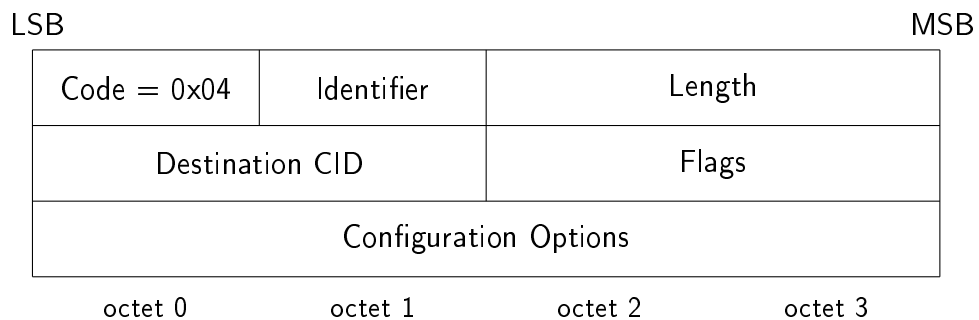


Figure 3.1.: Configuration Request Packet

The specification currently defines four possible configuration options, namely the *Maximum Transmission Unit (MTU)*, the *Flush Timeout*, *Quality of Service (QoS)* and

¹Currently there is only one flag defined which serves as a continuation flag if the receiver’s MTU is not large enough to receive a whole specific configuration request at once and fragmentation of the request’s packet needs to take place.

Retransmission and Flow Control. All options are filed in table 3.2 for reference with their corresponding macroname in this implementation. The RFC option is discussed in depth in section 3.4. Note that the options field may contain an option of a specific type multiple times, for example MTU + RFC + MTU. In such a case we decided to consider only the very first occurrence of each option. To realize that we defined a bitmask (unsigned long options) that is used via the atomic macros `set_bit` and `test_bit`.

| Code | Type | Macroname |
|------|--|---------------------|
| 0x01 | Maximum Transmission Unit (MTU) | L2CAP_CONF_MTU |
| 0x02 | Flush Timeout Option | L2CAP_CONF_FLUSH_TO |
| 0x03 | Quality of Service (QoS) Option | L2CAP_CONF_QOS |
| 0x04 | Retransmission and Flow Control Option | L2CAP_CONF_RFC |

Table 3.2.: Possible type fields for the configuration option format

Once we receive an incoming configuration request (in `l2cap_config_req()`) we set up an array of size 128 bytes to store the individual result codes of all sent configuration options as well as the overall result code. The latter one which is a union of all the separated result codes is stored at `result[0]`. Each possible configuration option's result is stored at its unique place within the array which corresponds to the option's code that is given by the specification. For example the MTU option has code 0x01 and is stored at `result[1]` (`result[L2CAP_CONF_MTU]` in the implementation) whereas the retransmission and flow control option (code 0x04) is stored at `result[4]` respectively.

The helpful side effect is herein that there is no option with code 0x00. We decided to use an array of size 128 because this allows us to handle 127 different options (code 0x00 cannot be used). This size should be sufficient for all future releases as the specification defines the option's code field to be only *one octet* long. Given the octet's most significant bit being used as a *hint*, only seven out of eight bits can be used for the option's code.

We should be prepared for such a big number of options because in case we receive a configuration request wishing to configure parameters for an unknown option, say 0x42, we have to respond with a configuration response informing the remote peer that option 0x42 is unknown. That is why we have to keep track of each possible incoming option. There are currently four possible result codes defined as stated in table 3.3.

All fields within the result array are initialized with 255:

```
memset(result, 255, L2CAP_MAX_OPTS);
```

This value was chosen as it can never be used in a real request and 0 would have stated success for every single option by default which is undesirable.

3. Configuration Process

| Result | Description |
|--------|---|
| 0x0000 | Success |
| 0x0001 | Failure – unacceptable parameters |
| 0x0002 | Failure – rejected (no reason provided) |
| 0x0003 | Failure – unknown option |
| Other | Reserved |

Table 3.3.: Possible configuration response result codes

The result array is passed forward to `l2cap_parse_conf_req()`. This function parses a given configuration request for all different options. It sets the appropriate configuration request parameters from the remote peer if all options are valid and understood. It rejects them if they are malformed and prepares an unknown status for options that we do not support yet. The individual result code for each separate option is set via `l2cap_check_conf_opt()` which checks this specific option for valid parameters.

3.3. Configuration Response

Another part of the configuration process comes into play once we received a valid request and need to build a matching response. This situation occurs when we ran through the aforementioned configuration request parsing process. Within the function `l2cap_config_req()` we call `l2cap_build_conf_rsp()` to build such a response packet. The function takes a pointer to the result array which was just set up. It runs through the following steps:

- (1) Check the value of the overall result code `result[0]` to take appropriate action. Be aware that this is the indicator for the result code of the entire packet:

“The result of the configuration transaction is the *union* of all the result values. All the result values must succeed for the configuration transaction to succeed”. [BlueSpec, page 49]

We distinguish five different cases—the four specified result codes (see table 3.3) plus an empty request that needs to be answered as well.

- (2) Within each case of the overall result code we check every single entry of the array for the same result code. For example consider a configuration request that is treated as successful for all single configuration options and made us set

```
result[0] = L2CAP_CONF_SUCCESS;
```

This makes us check all 127 remaining entries within the array for the code `L2CAP_CONF_SUCCESS`.

- (3) If both cases match check the entry (e.g. `0x01` for `L2CAP_CONF_MTU`) and add this option to the configuration response that we build.

Adding such an option is done via `l2cap_add_conf_opt()`. This function receives a pointer to the location where all options are finally stored. At the end of the function this pointer is increased in length of the pushed option to point to the next available option. The function's signature demands for the option's type and length so that the function itself can be used generically for all different options and to build a proper response packet as well as a valid request.

An incoming configuration response is handled by `l2cap_config_rsp()`. Configuration response packets consist of basically the same fields as a configuration request—instead of a destination CID it uses a source CID and further more there is a one octet field *result* which takes one of the result codes of table 3.3 on page 20. The architecture of a configuration response packet is shown in figure 3.2.

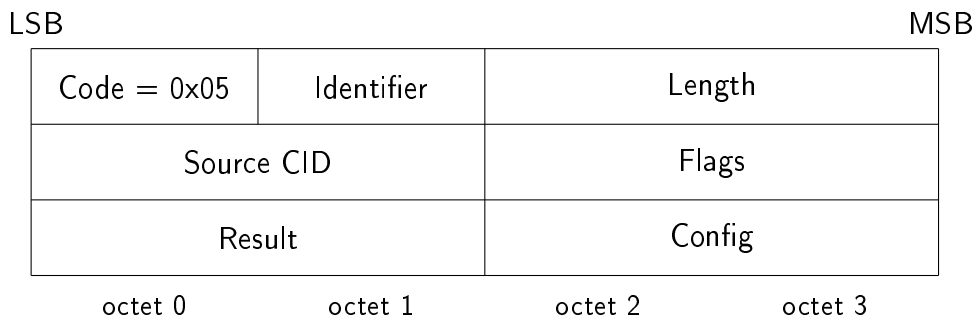


Figure 3.2.: Configuration Response Packet

Depending on the given result code appropriate action is taken. This implies a thorough understanding of the different result codes. In case of *Success* the configuration request is fully accepted by the remote peer which lets us finally set the different parameters to match the requested values. In case of an *Unacceptable Parameters* failure the response contains all “the values that would have been accepted if sent in the original request.” [BlueSpec, page 51] An *Unknown Option* failure indicates missing support for a requested option. It is important to note that this option is to be included within the response to let the requesting side know which of the sent options cannot be understood. The last possible option is a *Reject* failure which does not need any reasons to be provided.

3.4. Retransmission and Flow Control Option

For the purpose of extending configuration parameters L2CAP offers configuration parameter options. The basic format is illustrated in figure 3.3. The *Type* specifies

3. Configuration Process

the option type field of the parameters that are to be configured. The four currently specified options are shown in table 3.2.

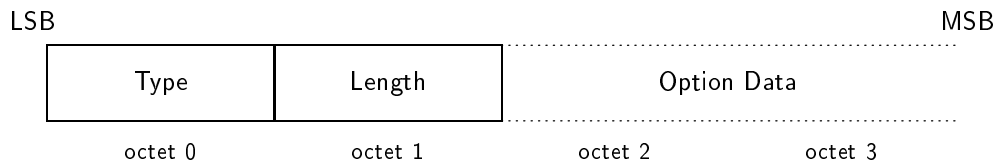


Figure 3.3.: Configuration option format

Note that the *option data* field can consist of multiple options within one single request. The Retransmission and Flow Control option format itself is specified as drawn in figure 3.4. The whole option is 11 bytes long and the different fields are discussed in this section.

| | | | |
|--|--------------------------------|-------------|---|
| 0 | 31 | | |
| 0x04 | Length = 9 | <i>Mode</i> | <i>TxWindow size</i> |
| <i>Max Transmit</i> | <i>Retransmission time-out</i> | | <i>Monitor time-out</i> (least significant byte) |
| <i>Monitor time-out</i> (most significant byte) | <i>Maximum PDU size (MPS)</i> | | |

Figure 3.4.: Retransmission and Flow Control option format

Mode (1 octet) This field is the final instance about the link mode that is to be used for this connection. The different values for this field are specified in table 3.4. The default is the Basic L2CAP Mode and using this mode invalidates all other fields within this option.²

TxWindow (1 octet) The TxWindow size field has to be set accordingly to the available buffer space for flow control. The size reflects the available buffer space counted in frames and may vary between one and 32. For a better channel utilization a larger value is recommended. The sending delay is influenced by this value, too. The transmitter can send as many PDUs as fit within the receiver's TxWindow.

Max Transmit (1 octet) This field's value specifies the number of possible retransmissions of a single I-frame in *Retransmission Mode*—therefore it is of no interest for Flow Control Mode.

²It is not entirely clear why the Extended Features are propagated when it basically comes down to the use of the Basic L2CAP Mode for each connection as seen on the Siemens S75 mobile phone which claims to support all extended features but never allows us to make use of them by signalling a value of 0x00 for the Mode field in the Retransmission and Flow Control Option.

| Value | Description |
|--------------|-------------------------|
| 0x00 | Basic L2CAP Mode |
| 0x01 | Retransmission Mode |
| 0x02 | Flow Control Mode |
| Other values | Reserved for future use |

Table 3.4.: Possible mode definitions within the RFC option

Retransmission Time-Out (2 octets) This value represents the time-out given in milliseconds and is primarily used for the initialization of the Retransmission Timer. In Flow Control mode the purpose of this timer is the supervision of I-frame transmissions. This leads the transmitting side to keep on transmitting I-frames after the time-out value if acknowledgements or I-frames got lost.³

Monitor Time-Out (2 octets) This is the value in milliseconds that is used for the initialization of the monitor timer which ensures that lost acknowledgements (S-frames) are retransmitted. Note that only one of the two timers shall be active at any given time. Upon expiration of the Monitor-Timer the last acknowledgement frame is to be sent again and the timer shall be restarted. The timer shall remain active in the open state and enforces an idle connection to have periodic monitor traffic sent in both directions.

Maximum PDU Size (1 octet) This is the size of the maximum PDU payload size that the device is capable of accepting.

³Note that the purpose of the retransmission timer in Flow Control Mode is actually not for the *retransmission* of lost I-frames but more to prevent the transmission process to stock due to a lost acknowledgements. It enables the sending peer to continue transmission.

an acknowledgement. Within the Retransmission Mode S-frames may request the retransmission of missing I-frames and allow the use of so-called Reject-frames. But both techniques do not apply to Flow Control Mode and are beyond the scope of this work.

| Frame Type | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|------------|-----|----|--------|----|----|----|----|---|---|-------|---|---|---|---|---|---|
| I | SAR | | ReqSeq | | | | | | R | TxSeq | | | | | | 0 |
| S | × | × | ReqSeq | | | | | | R | × | × | × | S | 0 | 1 | |

× denotes reserved bits. Shall be coded 0.

Figure 4.2.: Control Field Formats

The *Segmentation and Reassembly* bits (SAR) are used to distinguish different frames if an L2CAP SDU is segmented. These bits do only occur within the control field of I-frames and represent one of four possible states:

- 00 specifies an unsegmented L2CAP SDU.
- 01 represents the start frame of a segmented L2CAP SDU. When this condition applies the L2CAP SDU length field is present and needs to be set accordingly by the transmitter and recognized by the receiver respectively.
- 10 signals the end frame of a segmented L2CAP SDU.
- 11 is present in all continuation frames of an L2CAP SDU.

The *Retransmission Disable Bit* (R) is part of any Flow Control implementation. When the sender receives a frame with this bit set to one, the receiver indicates that his internal receive buffer is full and that the sender shall disable the retransmission timer and stop transmitting I-frames. The monitor timer is started on the transmitter side to monitor signalling.

The two sequence numbers ReqSeq and TxSeq are used to number frames and acknowledge previously received frames. In Retransmission Mode the receive sequence number would allow the receiver to request retransmission of a specific frame that got lost. But retransmission does not take place in Flow Control Mode.

Finally there is the two bits long *Supervisory function* (S) which marks the type of the S-frame. Currently there are only two of four possible types defined (The encoding 10 and 01 are reserved for future use). One is the RR frame (Receiver Ready, code 00) and the other one is the REJ frame (Reject, code 01). In Flow Control Mode REJ frames shall not be used.

The Frame Check Sequence

The *Frame Check Sequence* (FCS) at the end of each S- and I-frame forms a cyclic redundancy check over the whole frame, including the L2CAP header. The generator polynomial is $g(D) = D^{16} + D^{15} + D^2 + 1$, [see BlueSpec, page 39]. The Linux kernel

4. Procedures for Flow Control

offers a `crc16()` function within `<linux/crc16.h>` which covers exactly the aforementioned generator polynomial. It can be used to create a frame check sequence, for example:

```
/* calculate fcs over information payload plus
 * len, cid and control fields */
fcs = crc16(0, skb->data, len + 6);
```

or to detect invalid frames (in `l2cap_invalid_frame_detection()`):

```
/* contains an fcs error */
fcs = get_unaligned((u16 *) (skb->tail - 2));
if (unlikely(fcs != crc16(0, skb->data, skb->len - 2)))
    err = -3;
```

Figure 4.3 illustrates the segmentation process between different layers. The SDU is composed of multiple information payload fields of different I-frames. Note that the entire protocol overhead, like the L2CAP header, the control field, or the frame check sequence are stripped for the SDU composition. The SDU length field is only present in the first I-frame of a series of I-frames that belong to a segmented SDU and is marked in bright gray within this figure.

For illustrating purpose, parts of the HCI layer and its interaction with the L2CAP layer are shown as well. We see how multiple HCI data payloads compose an I-frame. This is done transparently to the L2CAP header on which this work is based.

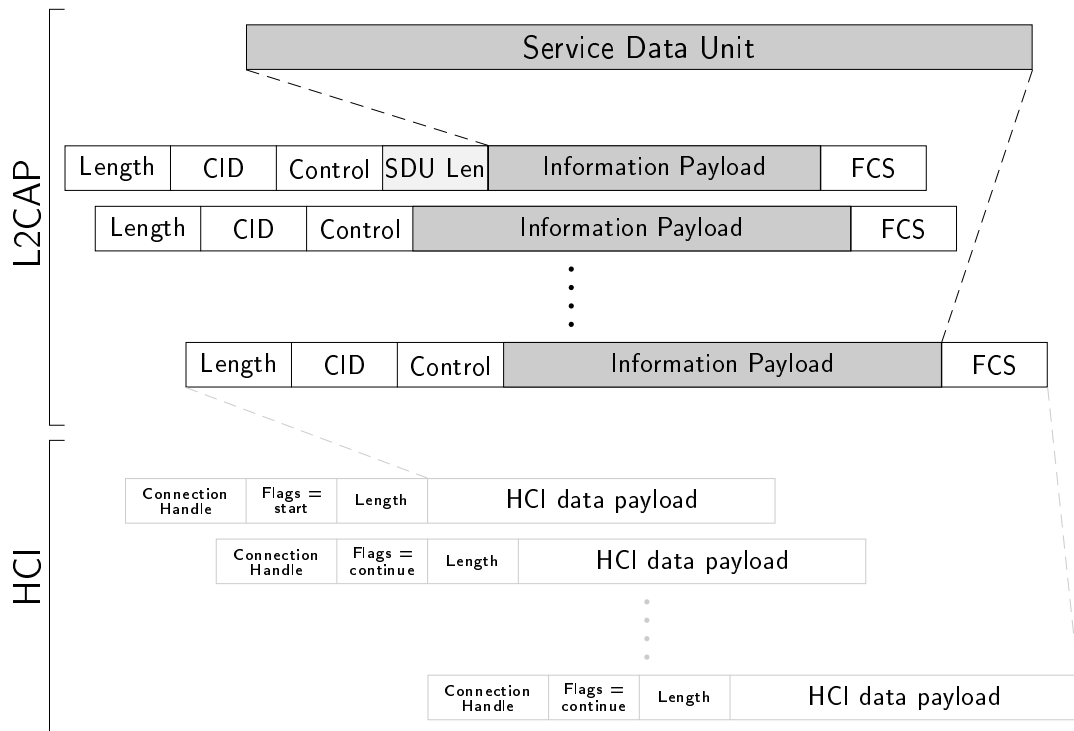


Figure 4.3.: Example of the segmentation process between the L2CAP and HCI layers

4.1. Variables and Sequence Numbers

Implementing Flow Control requires the use of some internal variables and sequence numbers to keep track of the current status of its own buffer and the remote peer's buffer respectively. We distinguish between the sending peer and the receiving peer. The techniques used with specific variables are illustrated by figures 4.4 and 4.5. All variables are in the range of 0 to 63.

4.1.1. The sending peer

The sending peer uses three different internal variables, namely TxSeq, NextTxSeq and ExpectedTxSeq. The send sequence number TxSeq is used to number each transmitted I-frame sequentially and will be provided within the control field of the transmitted I-frame. NextTxSeq states the number of the next new I-frame to be transmitted and shall be maintained for each remote endpoint. Once the sender decides to transmit a new in-sequence I-frame¹ the value of TxSeq is set to the one of NextTxSeq and NextTxSeq is incremented by one. Be aware that “the value of NextTxSeq [...] shall not exceed ExpectedAckSeq by more than the maximum number of outstanding I-frames (TxWindow)” [BlueSpec, page 89].

Finally ExpectedAckSeq reflects the current value of the next I-frame expected to be acknowledged by the receiving peer. This acknowledge state variable shall be maintained for each remote endpoint, too. An acknowledgement of a single I-frame from the remote side yields the value of ExpectedAckSeq + 1 as its ReqSeq value.

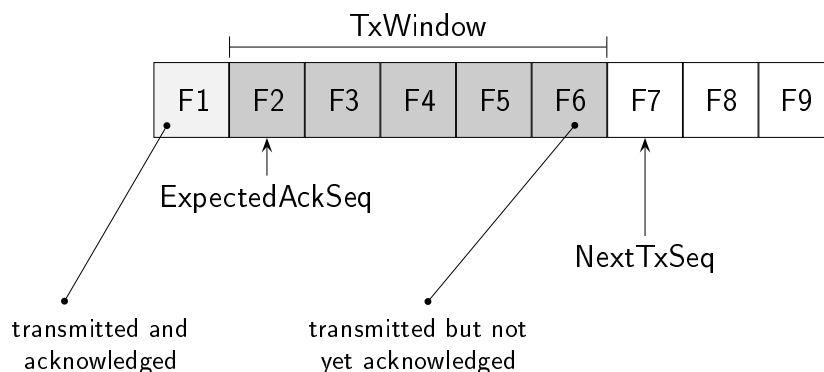


Figure 4.4.: Example of the Transmitter's side

Note the following requirement given by [BlueSpec, page 90]:

“If a valid ReqSeq is received from the peer then ExpectedAckSeq is set to ReqSeq. A valid ReqSeq value is one that is in the range $\text{ExpectedAckSeq} \leq \text{ReqSeq} \leq \text{NextTxSeq}$ ”.

¹Retransmission Mode allows retransmission of already sent I-frames for which the value of TxSeq must not be changed.

4. Procedures for Flow Control

Figure 4.4 illustrates the use of all variables for the transmitter. The negotiated TxWindow size is five and all the frames F2 to F6 have already been transmitted but not yet acknowledged. The first acknowledgement is expected to match frame F2. But note that once we receive an acknowledgement for any frame that still needs to be acknowledged (here frames F2 to F6) all frames *up to (but not including) this frame* are acknowledged even if either one of the frames in front of it or one of the acknowledgements got lost.

For example suppose the value of ExpectedAckSeq to be 2 as seen in the figure mentioned above. If the sending peer receives a valid frame from the receiving peer (of which the ReqSeq value is implicitly counted as an acknowledgement number) which sends an acknowledgement with ReqSeq = 5, we assume the frames 2, 3 and 4 to be implicitly acknowledged. Only within Retransmission Mode action can be taken on the frames 2, 3 and 4, which is beyond the scope of this work.

4.1.2. The receiving peer

Three internal variables are defined for the receiving peer, namely the receive sequence number ReqSeq, the receive state variable ExpectedTxSeq and the buffer state variable BufferSeq. Figure 4.5 shows an example of how all of these variables come into play on the receiver's side.

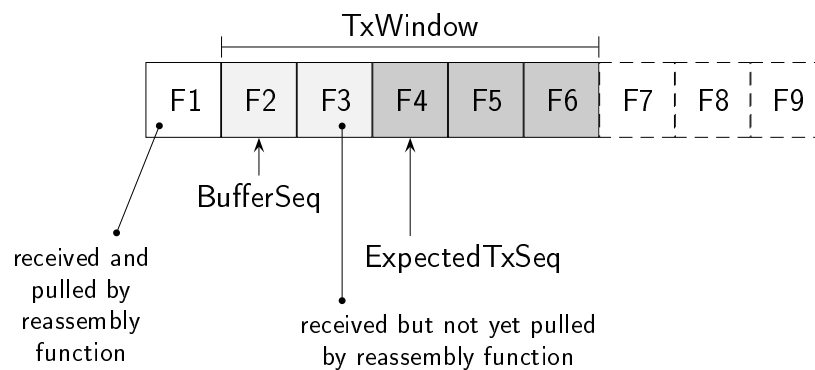


Figure 4.5.: Example of the Receiver's side

ReqSeq is used in I-frames as well as in S-frames to acknowledge correctly received frames with corresponding TxSeq values up to and including ReqSeq - 1. Therefore the value of ReqSeq shall be set to the current value of ExpectedAckSeq or BufferSeq (which of these values are to be used is implementation specific and discussed in the next paragraphs). ExpectedTxSeq reflects the expected TxSeq value of the next in-sequence I-frame.

The buffer state variable BufferSeq provides the delay of acknowledging frames until they have been pulled by the upper layers. This mechanism prevents the receiver of buffer overflow. The values of ExpectedTxSeq and BufferSeq are equal when no segmentation takes place and frames are pushed to the upper layer immediately on

reception. Sequence numbers of I-frames shall be in the range of $\text{ExpectedTxSeq} \leq \text{TxSeq} < (\text{BufferSeq} + \text{TxWindow})$.

The specification states [BlueSpec, page 92]:

“On receipt of an I-frame with TxSeq equal to ExpectedTxSeq, ExpectedTxSeq shall be incremented by 1 regardless of how many I-frames with TxSeq greater than ExpectedTxSeq were previously received”.

4.2. Retrieving Data

Retrieving data in a socket based matter is done in UNIX network programming by a `recvmsg` call. Such a call is matched in the L2CAP layer by a signal handler function which is called `bt_sock_recvmsg` wherein the necessary socket buffer is filled and various checks are fulfilled. Note that this function itself is not part of the L2CAP layer implementation. The L2CAP layer comes into play when the HCI layer beneath invokes the signal handler for `l2cap_recv_acldata()`, namely `recv_acldata`. From this point on the responsibility lies within the L2CAP layer.

The control flow moves on to `l2cap_recv_frame()` where a strict differentiation takes place between three kinds of frames that can be received, based on the given frame header's CID. This means we either step into a signalling channel if the header's CID is `0x0001`, which covers all sorts of signalling packets that we saw during the configuration process. Elsewise we can move into a connectionless channel if the CID is `0x0002`. In all other cases we use a data channel and `l2cap_data_channel()` is called. Herein we check the stage in which we operate and decide upon the result whether we are about to handle an I- or S-frame in Flow Control Mode or if we received an ordinary B-frame if we are in Basic L2CAP Mode. In the latter case—which was already implemented—no more action is to be taken except checking the size of the information payload for validity and passing the socket buffer to the upper layer via `sock_queue_rcv_skb()`.

4.2.1. Invalid Frame Detection

Quite more steps than those in Basic L2CAP Mode are necessary to handle frame reception in Flow Control Mode. The first of a variety of steps consists of a validity check in `l2cap_invalid_frame_detection()`. This function corresponds to [BlueSpec, section 3.3.6] where seven possible reasons are mentioned in which case the PDU shall be regarded as invalid. Those conditions cover errors like a wrong frame check sequence or wrong fields in the frame like an unknown CID or a length that exceeds the maximum PDU payload size (MPS). Whenever one of these error conditions applies the whole PDU is dropped silently. Error reporting is currently used only within the optional debug output but may be implemented later to interact with the user interface for convenience.

4. Procedures for Flow Control

One extra check was implemented within this function. It is the very first check that is performed and covers the assumption that each single socket buffer matches exactly one frame. This condition is validated by the following code snippet:

```
len = get_unaligned((u16 *) skb->data);

if (len != (skb->len - L2CAP_HDR_SIZE))
    err = -1;
```

This assumption is correct because a complete frame is copied into a newly created socket buffer (`conn->rx_skb`) in `l2cap_recv_acldata` and the size of the newly allocated memory area of `conn->rx_skb` is set to the same size that is specified in the frame's header.

4.2.2. Process the Send Sequence Number TxSeq

The send sequence number TxSeq, that can only be present in I-frames, is set by the transmitter to its internal sequence number NextTxSeq. The receiver yields an internal variable called ExpectedTxSeq that reflects the value of TxSeq expected in the next I-frame. Another internal variable on the receiver's side, BufferSeq, is used to delay acknowledgement of received I-frames when segmented I-frames are buffered.

Basically four error checks are done within `l2cap_process_reqseq()`. First of all the sent TxSeq value is compared to the expected one. In case of success the ExpectedTxSeq is incremented by one and everything is fine:

```
if (txseq == pi->exp_txseq) {
    pi->exp_txseq = (pi->exp_txseq + 1) % 64;
}
```

If this condition does not apply we check for an out-of-sequence I-frame and re-adjust the ExpectedTxSeq value:

```
else if (((txseq - pi->buffer_seq + 64) % 64) < pi->itxw) {
    /* out-of-sequence i-frame */
    pi->exp_txseq = (txseq + 1) % 64;
}
```

The last two checks cover the case of a duplicated I-frame or an invalid TxSeq sequence number in which an error is returned and the whole frame is dropped.

4.2.3. Process the Receive Sequence Number ReqSeq

The ReqSeq value within the control field must be processed in I-frames as well as in S-frames. A basic error check is fulfilled at the beginning of `l2cap_process_reqseq()` by checking the status of the right-most bits. In case of an I-frame the very last bit shall be coded 0, and in case of an S-frame the four right-most bits shall be coded 0001 because the two S bits of the supervisory function are always encoded 00 (marking the

S-frame to be of type Receiver Ready) and this is the only valid S-frame type within Flow Control Mode.²

The next error checking mechanism implemented takes [BlueSpec, section 8.5.6.2] into account:

“An ReqSeq sequence error exception condition occurs in the transmitter when a valid S-frame or I-frame is received which contains an invalid ReqSeq value. An invalid ReqSeq is one that is not in the range $\text{ExpectedAckSeq} \leq \text{ReqSeq} \leq \text{NextTxSeq}$.”

The L2CAP entity shall close the channel as a consequence of an ReqSeq Sequence error”.

This task is accomplished by the following code:

```
/* reqseq sequence error */
if (!(reqseq - pi->exp_ackseq + 64) % 64 <=
    (pi->next_txseq - pi->exp_ackseq + 64) % 64)) {
    /* close channel -> send disconnect request */
    struct l2cap_disconn_req req;
    struct l2cap_conn *conn = l2cap_pi(sk)->conn;
    sk->sk_state = BT_DISCONN;
    sk->sk_err = ECONNRESET;
    l2cap_sock_set_timer(sk, HZ * 5);
    req.dcid = __cpu_to_le16(l2cap_pi(sk)->dcid);
    req.scid = __cpu_to_le16(l2cap_pi(sk)->scid);
    l2cap_send_cmd(conn, l2cap_get_ident(conn),
        L2CAP_DISCONN_REQ, sizeof(req), &req);
    return -2;
}
```

If the ReqSeq value is valid and in-sequence, the internal ExpectedAckSeq value is set to the current ReqSeq value.

4.2.4. Reassemble the SDU

Reassembling the SDU of multiple PDUs is actually done in `l2cap_reassembly()`. This function receives an I-frame in form of a socket buffer and the extracted value of the corresponding control field. The first action to be taken on the socket buffer is to remove the unnecessary fields from it, namely the L2CAP header, the control field and the frame check sequence:

```
/* remove header, control and fcs from i-frame */
skb_pull(skb, L2CAP_HDR_SIZE + L2CAP_CONTROL_SIZE);
skb_trim(skb, skb->len - L2CAP_FCS_SIZE);
```

²This error condition would have to be changed slightly whenever Retransmission Mode is going to be implemented and REJ-frames may occur.

4. Procedures for Flow Control

What remains is the information payload and—just for the first frame of a segmented SDU—the 16 bit SDU length field that precedes the information payload.

Depending on the type of the frame, regarding its position within the SDU, we have to distinguish basically four different cases:

- (1) The frame is the first in a sequence of frames that build a complete SDU. This frame is marked with 01 or constant-wise `L2CAP_SAR_START` in the SAR bits of the control field. When this condition applies the SDU length field is present and the information payload field is reduced by two octets in size.
- (2) The frame belongs to a segmented SDU but is neither the first nor the last bit of it. This frame has the bits 11 or constant-wise `L2CAP_SAR_CONTINUE` set in its SAR bits.
- (3) The frame belongs to a segmented SDU and builds the end of it. The SAR bits of its control field are set to 10 or `L2CAP_SAR_END`.
- (4) If the frame is not segmented and builds a SDU by its own, the bits 00 are set in the SAR portion of the control field.

We will now see how these four cases can be worked out in a generic way and how we handle error conditions and exceptions.

The first design decision affects handling these different states. Depending on the SAR bits of the control field we differentiate between two explicitly stated cases, namely `L2CAP_SAR_UNSEGMENTED` and `L2CAP_SAR_START` and put the other two (the last frame and all continuation frames) into a default case.

Receiving an unsegmented frame can be processed straight forward by adjusting the `BufferSeq` value to be the `ExpectedTxSeq` one, passing the entire information payload to the upper layer and transmitting an S-frame as an acknowledgement to the remote peer.

The rest of the function covers the reception of segmented SDUs only. Receiving a start frame requires the SDU length field to be considered and extracted. Building a complete SDU from ground up requires us to have an associated socket buffer available. We declared such a socket buffer in our socket specific structure. The first action to be taken consists of an appropriate error check, because in case we are demanded to build a new SDU from multiple segments but the last reassembling process was not finished correctly, we have to make sure to reset the socket specific SDU buffer.

Under normal circumstances we will now allocate enough buffer space for the entire SDU and copy the first part of the segmented SDU into the newly allocated buffer. Furthermore the socket bound variable `sdu_len` is set to the value of the given SDU length minus the length of the first part of the SDU that was just copied into the buffer. This will help us to ensure that we will never accept segments of length bigger than the size of the allocated buffer. Finally the value of `BufferSeq` is adjusted to the one of `ExpectedTxSeq` and an S-frame is sent as an acknowledgement.

The default case handles both the reception of continuation frames and end frames. The first part consists of two error checks. If there is no remaining space in our allocated SDU buffer (`sdu_len` is 0) we break off the function and return an error. The second condition applies if the frame's length (`skb->len`) is claimed to be larger than the remaining length of the allocated buffer. This causes our implementation to cancel the SDU reassembling process, free the corresponding socket buffer, reset its values, and to break off the function immediately.

If none of the above stated error conditions apply we copy the information payload into the SDU socket buffer, adjust all values accordingly, and transmit an S-frame as an acknowledgement to the remote peer. In case this is an end frame and the SDU is now complete in terms of a completely filled buffer, we pass the entire SDU to the upper layer, free the corresponding buffer afterwards, and reset the values:

```
memcpy(skb_put(l2cap_pi(sk)->sdu, skb->len), skb->data, skb->
    len);
l2cap_pi(sk)->sdu_len -= skb->len;
l2cap_pi(sk)->buffer_seq = l2cap_pi(sk)->exp_txseq;
if (l2cap_send_sframe(sk, l2cap_pi(sk)->buffer_seq))
    return -6;

/* last frame for sdu */
if ((L2CAP_GET_SAR(control) == L2CAP_SAR_END) &&
    (l2cap_pi(sk)->sdu_len == 0)) {
    if (sock_queue_rcv_skb(sk, l2cap_pi(sk)->sdu))
        return -1;
    kfree_skb(l2cap_pi(sk)->sdu);
    l2cap_pi(sk)->sdu = NULL;
    l2cap_pi(sk)->sdu_len = 0;
}
```

4.3. Transmitting Data

Sending data is initially invoked by calling the socket specific `sendmsg()` function from userspace. This call matches a signal handler function in the L2CAP layer called `l2cap_sock_sendmsg()`. Besides some typical basic error checking a lock is acquired for the corresponding socket before the actual implementation of frame transmission takes place. The existing implementation already covered basic L2CAP mode by invoking `l2cap_do_send()` with the message `msg` as the information payload.

Building an equivalent functionality for flow control mode required a complete different approach to fulfill the needs. Fundamental differences covered the way in which frames are created and passed to the lower layer. Basic L2CAP mode uses B-frames which are basically identical—no control field or frame check sequence at the end of a frame is in use, and no SDU length field is to be used for one specific frame. The technique of the existing implementation makes use of a list of fragments in the

4. Procedures for Flow Control

socket buffer shared info structure where the entire message is separated into multiple fragments of information payload fields.

Instead of rewriting the existing function we decided to split the functionality of segmenting the SDU and building the actual I-frames into two separate and new functions, called `l2cap_segment_sdu()` and `l2cap_do_send_rfc()`. Depending on the connection's mode, the corresponding control flow takes place:

```
if (l2cap_pi(sk)->mode == L2CAP_MODE_FLOW) {
    err = l2cap_segment_sdu(sk, msg, len);
    if (err < 0)
        goto out;
    if (l2cap_do_send_rfc(sk) < 0)
        err = -1;
} else {
    err = l2cap_do_send(sk, msg, len);
}
```

The function `l2cap_segment_sdu()` is the counter-part to the function that handles the reassembling process, `l2cap_reassemble_sdu()`. At first we store the value of the minimum between the length of the message and the size of the MTU minus the L2CAP header into a variable called `count`. This variable is needed to keep track of the remaining length of the message that has not yet been put into information payload fields of I-frames.

The first part of the function covers the case in which one single I-frame is enough to hold the entire message. This applies whenever the length of the message equals the size of `count`. In such a case the way to process the I-frame is pretty much straight forward. We allocate enough space for a socket buffer, create the L2CAP header and the individual control field, copy the message from userspace into the buffer via `memcpy_fromiovec()` and calculate the frame check sequence over the header, the control field and the information payload field. Finally we increase the value of `Next-TxSeq` by one and store the created socket buffer at the end of a transmit queue. Once we did all these steps we break out of this function immediately.

It is a whole different story once we need to send a message that exceeds the size of `count`. In such a case we need to segment the SDU and create a bunch of individual I-frames. This enforces us to make use of the SAR bits in the control field and use the SDU length field in the very first I-frame. Before we start actually building all the I-frames we adjust the value of `count` by decreasing it by two bytes and increasing the size of the virtual header's length by two bytes—this is due to the fact that we need two more bytes for the SDU length field and have therefore two bytes less for the information payload field. Further more the SAR bits are initialized to `L2CAP_SAR_START`.

The basic idea on how the segmentation is handled is based on a while loop over the remaining length of the message. The while loop starts with a socket buffer allocation of size `count` plus the header's length. After that the header is created and put into the first four bytes of the newly allocated socket buffer. Afterwards the individual control field is created and put at the following two bytes. If this is the first frame in the series of segments (the SAR bits are set to `L2CAP_SAR_START`), then the next

two bytes are filled with the actual length of the SDU. Now `count` bytes can be copied from userspace into the socket buffer's information payload field:

```
/* Information payload */
if (memcpy_fromiovec(skb_put(skb, count), msg->msg_iov, count
)) {
    err = -EFAULT;
    goto fail;
}
```

After that the frame check sequence is calculated and put at the last two bytes of the socket buffer. The value of `NextTxSeq` is increased by one and the socket buffer is put at the end of the `TxQueue`. We have to re-adjust some variables to ensure the loop is entered with valid values.

After the SDU has been segmented into multiple I-frames which are stored in the `TxQueue`, `l2cap_do_send_rfc()` is invoked to actually send the I-frames awaiting transmission. This function was written so that it can easily be invoked not only by `l2cap_sock_sendmsg` as part of a normal transmission process, but also by the expiration of a retransmission timer.

The function has a local variable `occupied` that computes the value of frames that have been sent but not yet been acknowledged by subtracting the `ExpectedAckSeq` from `NextTxSeq`. If the receiver's `TxWindow` is full (`occupied` equals the value of `otxw`) or if there are no frames in our `TxQueue` that need to be sent, we exit this function.

But whenever we are allowed to send frames and we currently have one or more I-frames in our queue, we dequeue those ones and pass them to the lower layer by invoking `hci_send_acl()`:

```
while ((pi->tx_queue_frames > 0) && (pi->otxw > occupied)) {
    skb = skb_dequeue(&pi->tx_queue);
    if (skb == NULL)
        goto fail;

    if ((err = hci_send_acl(conn->hcon, skb, 0)) < 0)
        goto fail;

    pi->tx_queue_frames--;
    occupied++;
}
```

4.3.1. Sending Acknowledgments

Acknowledgements can either be sent implicitly by using I-frames and their `ReqSeq` values in the control field, or by using S-frames explicitly for the purpose of pure acknowledgements. The latter functionality was implemented by this work. The function that is invoked on the expiration of the monitor timer or within the reassembly function is called `l2cap_send_sframe()`.

4. Procedures for Flow Control

S-frames are simple structures. Figure 4.1 illustrates the four different fields, each 16 bits wide, that compose one S-frame. Note that no information payload field is present and we therefore have a fixed length for the entire frame. Building such a frame consists of allocating the necessary socket buffer, creating the L2CAP header and the individual control field. Figure 4.2 illustrates, that the control field for an S-frame in flow control mode consists only of the ReqSeq bits and the last bit which is set to one.³ Finally the frame check sequence is computed over all preceding bytes and attached to the socket buffer. Then the buffer can be passed to the lower layer by calling `hci_send_acl()`, and the monitor timer is restarted.

4.3.2. Monitor and Retransmission Timer

The monitor and retransmission timers prevent us of from getting stuck in a deadlock by resending necessary acknowledgement frames to the remote peer, and by transmitting I-frames continuously in case sent acknowledgements from the remote peer got lost.

The monitor timer will ensure periodic traffic between both peers as long as the connection is held in an open state which keeps sequence numbers synchronized. The monitor timer is of even more interest in retransmission mode where a specific bit can be used for signalling purposes.

The retransmission timer on the other hand is used to continue transmission of I-frames by the sender even if the current monitored TxWindow of the remote's receiving peer is full. This situation shall prevent us from a deadlock in case either an acknowledgement frame got lost on its way or one of the already sent I-frames was corrupted and could not be acknowledged but does neither occupy a space in the receiver's TxWindow.

Both timers are defined in `l2cap_sock_init`:

```
init_timer(&l2cap_pi(sk)->ret_timer);
l2cap_pi(sk)->ret_timer.function =
    l2cap_retransmission_timer;
l2cap_pi(sk)->ret_timer.data = (unsigned long)l2cap_pi(sk);

init_timer(&l2cap_pi(sk)->mon_timer);
l2cap_pi(sk)->mon_timer.function = l2cap_monitor_timer;
l2cap_pi(sk)->mon_timer.data = (unsigned long)l2cap_pi(sk);
```

The signal handler `l2cap_retransmission_timer` increases the value of `ExpectedAckSeq` by one and starts the monitor timer in case no more I-frames await transmission. This condition applies whenever `ExpectedAckSeq` equals `NextTxSeq`. If there are still I-frames that need to be sent the retransmission timer is started a socket lock is acquired and `l2cap_do_send_rfc()` is called to transmit the next I-frame (this can be done because `ExpectedAckSeq` is now increased).

³The value of the S bits would have to be implemented in retransmission mode, but this is outside the scope of this work.

The monitor timer's signal handler is called `l2cap_monitor_timer`. All its minimal functionality is based on inquiring transmission of an S-frame to re-acknowledge the last valid and in-sequence I-frame received by the remote peer with `ExpectedTxSeq` as its `ReqSeq` value, and finally restart the monitor timer.

Starting both timers is achieved by `l2cap_start_ret_timer` for the retransmission timer, and `l2cap_start_mon_timer` for the monitor timer respectively. The timeout with which they are called is already converted so that it can be compared to the current kernel's *jiffies* value, e.g.:

```
l2cap_start_mon_timer(sk, pi->imon_to * HZ / 1000);
```

5. Summary

5.1. Testing and Analysis

The GNU/Linux system is based on a monolithic kernel, and to work inside kernelspace is always critical and must be done cautiously. The L2CAP layer of Linux' Bluetooth subsystem is a separate module that allows it to be dynamically linked to or removed from the running kernel. But due to the monolithic structure of the kernel itself, a failure in kernelspace, like a NULL pointer dereference, a deadlock or a race condition, can cause fatal exceptions that usually stall the entire system.

Debugging in kernel mode can be done in various ways, although none of them is very comfortable and all debugging facilities have their own drawbacks. We didn't make any use of one of the unofficial kernel debuggers whenever we encountered problems. Instead we used the module's built-in debugging output with extended `printk()` statements.

This was sufficient as long as we didn't make any serious faults. And in fact we didn't encounter any of the more annoying faults like the aforementioned NULL pointer dereferences or equivalent. But one troublemaker stalled the whole development process for a big amount of time. By changing from one kernel version to another that used the PREEMPT option of the kernel's configuration to make the kernel preemptible, the entire system crashed reproducibly. Difficult enough to locate the error because the system just hangs and the X terminals were not fast enough to catch and print the debugging output.

Fortunately the Linux kernel provides its developers the facility to work on the framebuffer console where output is immediately dumped if the console log level is set to an appropriate value.¹ To work only with the console can be tricky when the monitor and/or the console's screen resolution is not big enough to show all necessary information on one page because scrolling is not possible and there is no way to copy the messages into any file or onto another computer. This task may be achieved when working with a serial connection but there was no serial connector available on the main development computer.

Intensive testing was accomplished in each phase of the implementation. The testing environment consisted of two Acer notebooks (Acer Travelmate 803LMiB and

¹The Linux kernel offers the so-called SysRq keys, that can be usually accessed by pressing Alt + Print + *Function* simultaneously, e.g. setting the loglevel to 8 to print all warnings and other non-critical messages.

Acer Travelmate 3002WTMi) with integrated Bluetooth adapters, two mobile phones (Siemens S55 and S75), and an external USB Bluetooth adapter (Cellink BTA-6030). There exist multiple different Bluetooth software stacks in many different versions. We couldn't take all in considerations, but we were able to test our implementation against the two Siemens ones, the Toshiba stack for Windows in versions 3.03.13 and 4.20.01, the original Windows XP stack and finally the stack that comes shipped with Apple's operating system Mac OS X 10.4 (Tiger) on PowerPC.

Especially the Apple iMac G4 that we could use as a test machine enabled us to ensure proper and endianness-aware data transmission, because this iMac still uses a big endian PowerPC microarchitecture whereas the laptops are Intel x86 little endian machines.

Interesting, but even frustrating enough, was the determination that none of all the available test machines from different venturers was able to use flow control mode. The Siemens S55—probably the device with the oldest software stack on it—returned on an information request about support of extended features (like flow control mode) a corresponding information response that stated no success as a result. Stacks like the ones from Toshiba and Apple claimed to use extended features but did not offer any bits inside the data field of the extended features bitmask:

```
l2cap_information_rsp: type 0x0002 result 0x00 ident 2
Extended Features Mask: 00 00 00 00
```

The closest candidate of all these devices was definitely the Siemens S75 mobile phone which not only claimed to support extended features, but also returned a bitmask, indicating support for *every* currently available extended feature (0x0007). Sadly though, it stubbornly refused to configure any flow control feature, by responding to an appropriate configuration request with a failure for unacceptable parameters, indicating only concrete support for Basic L2CAP Mode in the response's retransmission and flow control option.

The test cases consisted of sending and retrieving files between a Linux computer with a patched kernel and the aforementioned devices. The kernel we used to accomplish this task changed over time as new kernel versions were released. The plain *vanilla* kernel was patched with the official patchset of the BlueZ project, which integrates code that is considered to be experimental but is supposed to be integrated into the main kernel tree once it is stable. Such a patched kernel was compiled and started before we built our own L2CAP module against the respective kernel headers and linked the binary module dynamically to the kernel.² Sending and retrieving files was used as an easy method that makes intensive use of the L2CAP layer.

The test results that we achieved with other devices were more or less useful. They helped definitely to encounter device specific behaviour during the configuration process. It was amazing to realize in how many ways devices from different manufactures can behave, although all are based on the same specification. As a consequence of the absence of flow control capability in all our devices, intensive test cases

²See Appendix B for a detailed description of this procedure.

5. Summary

for this last part could not be created. Finally the only tests that worked as expected could be achieved with two patched Linux PCs.

L2CAP flow control mode is definitely a nice feature and may be of even more interest in industrial environments which are limited by resource (e.g. only slow embedded processors) or if the device has to handle multiple high bandwidth connections under tough conditions like large distances with walls between the devices or similar scenarios.

It can be proven that flow control is in action if both the information response and the configuration response are returned with appropriate values.

5.2. Conclusion and Future Work

The implementation of Flow Control Mode for Linux' Bluetooth L2CAP layer was a major task that significantly enlarged the existing functionality. It was an advantage to work on an existing piece of software that was well written and structured. This prevented the undertaking of writing the code for the entire layer from ground up.

Future work may be based at first on finding bugs and improving the code before this part finds its way into the main kernel tree. Besides that, Flow Control Mode for Linux' L2CAP layer can be considered as being complete. However there is some more work that can be done, that is closely related to this implementation. One part covers the Retransmission Mode that makes use of multiple functions and paradigms that were now already—at least partly—implemented. Retransmission Mode is a very interesting feature to provide users reliable channels. Linux' L2CAP layer lacks only the lately introduced functionality like Retransmission Mode, Quality of Service Mode, or the Flush Timeout option.

A. Architectures' Data Types

The following table¹ covers architectural design differences and their implementations within the Linux kernel.

| Architecture | Description | Word | sizeof(int) | sizeof(long) |
|--------------|-------------------|--------------|-------------|--------------|
| alpha | Alpha Processor | 64 Bit | 4 | 8 |
| arm | Arm (old version) | 32 Bit | 4 | 4 |
| arm26 | Arm, StrongARM | 32 Bit | 4 | 4 |
| i386 | Intel <i>x86</i> | 32 Bit | 4 | 4 |
| ia64 | IA-64 | 64 Bit | 4 | 8 |
| m68k | Motorola 68xxx | 32 Bit | 4 | 4 |
| mips | MIPS | 32 or 64 Bit | 4 | 4/8 |
| ppc | PowerPC | 32 Bit | 4 | 4 |
| ppc64 | PPC64 | 64 Bit | 4 | 8 |
| sparc | Sun Sparc | 32 Bit | 4 | 4 |
| sparc64 | Ultra-Sparc | 64 Bit | 4 | 8 |
| x86_64 | AMD64/EMT64 | 64 Bit | 4 | 8 |

Table A.1.: Only some of the more popular architectures and their design differences

¹The complete set for all architectures can be found in [Quade and Kunst, 2006, Chapter 9.3]

B. Development Notes

During the time of this development the patches created for testing purposes had to be applied to a recent kernel version with included latest patchsets of Linux' Bluetooth maintainer Marcel Holtmann. This means a vanilla kernel like 2.6.17.x had to be patched first of all with the most recent patch of the bluez.org website (e.g. patch-2.6.17-mh5), before we could create our own patch covering the two files *l2cap.c* and *l2cap.h*.

The patched vanilla kernel (e.g. linux-2.6.17mh5) had to be compiled. This was done using Ubuntu Dapper Drake 6.06 via the following commands:

```
$ tar xjf linux-2.6.17.tar.bz2
$ cd linux-2.6.17
linux-2.6.17$ patch -p1 < ../patch-2.6.17-mh5
linux-2.6.17$ cp /boot/config-2.6.15-26-686 .config
linux-2.6.17$ make oldconfig
[...]
[edit some config options manually]
linux-2.6.17$ fakeroot make-kpkg kernel_image modules_image
    --initrd --revision mh5 binary
[...]
linux-2.6.17$ cd ..
$ sudo dpkg -i kernel-image-2.6.17mh5_i386.deb
```

After completing this step the new kernel has to be booted. Within this image we can compile and load our L2CAP module dynamically via insmod. The development took place within the directory `/home/martin/kernel`. To compile the necessary kernel module we used the following Makefile:

```
ifneq ($(KERNELRELEASE),)
    obj-m := l2cap.o
else

modules: l2cap.c
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(shell
        pwd) modules

install: l2cap.ko
    install -d $(DESTDIR)/lib/modules/$(shell uname -r)/kernel/
        net/bluetooth
```

```

install -m 644 -c nozomi.ko $(DESTDIR)/lib/modules/$(shell
    uname -r)/kernel/net/bluetooth
/sbin/depmod -e

uninstall:
    rm -rf $(DESTDIR)/lib/modules/$(shell uname -r)/kernel/net/
        bluetooth/l2cap.ko
/sbin/depmod -e

clean:
    rm -rf *.ko *.o *.mod.c *.d *.cmd .tmp_versions Modules.
        symvers

endif

```

With the help of this Makefile we compiled the module by issuing a `make modules` command.

If no warnings or errors appeared we were able to load and test the module. But to do so, we had to make sure no L2CAP module was already running with respect to dependencies. Therefore we created a small shell script that helped unloading and reloading the necessary modules as well as setting up the Bluetooth adapter:

```

#!/bin/bash
/etc/init.d/bluez-utils stop
modprobe crc16
rmmod rfcomm
rmmod l2cap
insmod /home/martin/kernel/l2cap.ko
/etc/init.d/bluez-utils start
hciconfig hci0 up

```

B.1. Bluetooth Adapters Used

The three adapters we used had different hardware capabilities. The output of the command `hciconfig -a` is shown for all three adapters. The first one is about the integrated Bluetooth adapter of the Acer Travelmate 3002WTMi, the second one covers the integrated Bluetooth adapter of the Acer Travelmate 803LMiB and finally the third one shows the output for the external Cellink BTA-6030 USB adapter:

```

hci0: Type: USB
    BD Address: 00:0B:6B:5F:8E:27 ACL MTU: 377:10 SCO MTU: 64:8
    UP RUNNING PSCAN ISCAN
    RX bytes:968409 acl:65326 sco:0 events:18614 errors:0
    TX bytes:8249626 acl:120935 sco:0 commands:145 errors:0
    Features: 0xff 0xff 0x0d 0x38 0x08 0x08 0x00 0x00
    Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3

```

B. Development Notes

```
Link policy: RSWITCH HOLD SNIFF PARK
Link mode: SLAVE ACCEPT
Name: 'karin-laptop-0'
Class: 0x3e0100
Service Classes: Networking, Rendering, Capturing
Device Class: Computer, Uncategorized
HCI Ver: 1.2 (0x2) HCI Rev: 0x69 LMP Ver: 1.2 (0x2) LMP
  Subver: 0x694a
Manufacturer: Broadcom Corporation (15)
```

```
hci0:  Type: USB
      BD Address: 00:02:8A:9E:93:1D ACL MTU: 192:8 SCO MTU:
        64:8
      UP RUNNING PSCAN ISCAN
      RX bytes:385 acl:0 sco:0 events:18 errors:0
      TX bytes:319 acl:0 sco:0 commands:17 errors:0
      Features: 0xff 0xff 0x0f 0x00 0x00 0x00 0x00 0x00
      Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
      Link policy: RSWITCH HOLD SNIFF PARK
      Link mode: SLAVE ACCEPT
      Name: 'bart-0'
      Class: 0x3e0100
      Service Classes: Networking, Rendering, Capturing
      Device Class: Computer, Uncategorized
      HCI Ver: 1.1 (0x1) HCI Rev: 0x20d LMP Ver: 1.1 (0x1)
        LMP Subver: 0x20d
      Manufacturer: Cambridge Silicon Radio (10)
```

```
hci1:  Type: USB
      BD Address: 00:0A:94:02:F2:1F ACL MTU: 384:8 SCO MTU:
        64:8
      UP RUNNING PSCAN ISCAN
      RX bytes:385 acl:0 sco:0 events:18 errors:0
      TX bytes:319 acl:0 sco:0 commands:17 errors:0
      Features: 0xff 0xff 0x8f 0xfe 0x9b 0xf9 0x00 0x80
      Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
      Link policy: RSWITCH HOLD SNIFF PARK
      Link mode: SLAVE ACCEPT
      Name: 'bart-1'
      Class: 0x3e0100
      Service Classes: Networking, Rendering, Capturing
      Device Class: Computer, Uncategorized
      HCI Ver: 1.1 (0x1) HCI Rev: 0x7a6 LMP Ver: 1.1 (0x1)
        LMP Subver: 0x7a6
      Manufacturer: Cambridge Silicon Radio (10)
```

Bibliography

- Ch. Benvenuti. *Understanding Linux Network Internals*. O'Reilly, first edition, 2006. ISBN 0-596-00255-6.
- BlueSpec. *Specification of the Bluetooth System Ver. 2.0+EDR*. Bluetooth SIG, November 2004.
- D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, 2006. ISBN 0-596-00565-2.
- J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition, 2005. ISBN 0-596-00590-3.
- R. D. Love. *Linux-Kernel-Handbuch*. Addison-Wesley, 2005. ISBN 3-8273-2204-9.
- J. Quade and E.-K. Kunst. *Linux-Treiber entwickeln*. dpunkt-Verlag, second edition, 2006. ISBN 3-89864-392-1.
- A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001. ISBN 0-13-031358-0.
- A. S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, 2003. ISBN 0-13-066102-3.
- J. Wollert. *Das Bluetooth Handbuch*. Franzis Verlag, 2002. ISBN 3-7723-5323-1.