The University of Edinburgh
Department of Computer Science
System Level Integration Practical

April 16, 2001

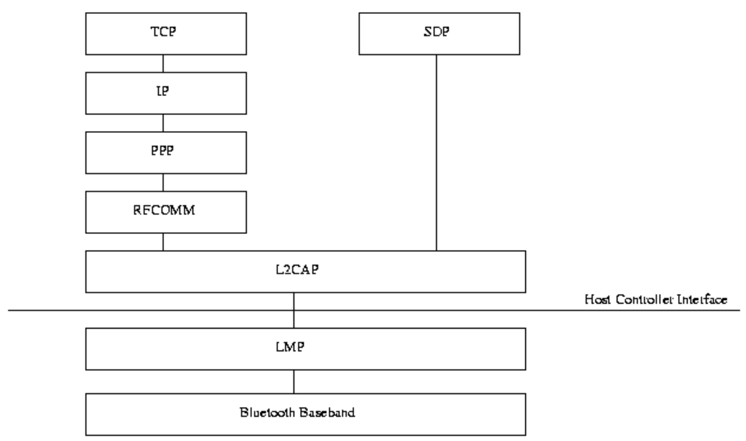
Implementation of Link Manager Protocol and Host Controller Interface

Morten Bek

Contents

1 Introduction

The Link Manager is responsible for controlling the Baseband and maintain physical and logical connections to other devices. In the protocol stack the Link Manager is just above the Baseband and the two are very closely tied. The Host Controller Interface is the interface to control the Link Manager and is also used to send data through once a connection has been set up. The Host Controller Interface defines methods to inquire for other Bluetooth devices, connecting and disconnecting, accept or reject requests for connections, asking for the supported feature of the remote device, etc. It communicates in the other direction with the upper layer through a system of event messages. In our implementation of the protocol stack, the Host Controller Interface is used by the L2CAP layer, which is implemented by Benjamin Hanney, and all decisions for the interface between the two has been discussed with him. Furthermore the last stage of implementation, testing and bug fixing has been done by the two of us in cooperation.



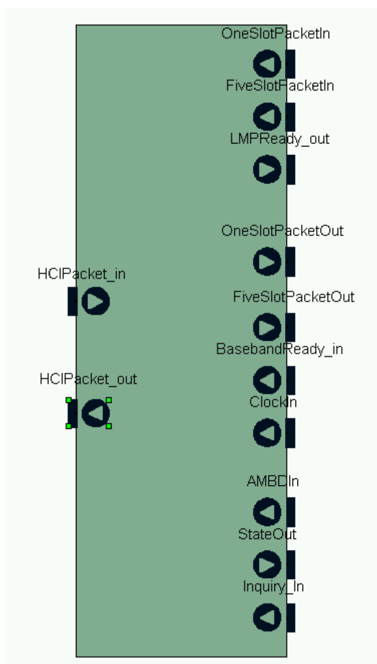
2 Implementation issues

First step to implementing a protocol must of course start with a thorough investigation of the protocol specification. During the first few weeks I tried to figure out exactly what was the purpose of the Link Manager as well as the Host Controller.

As both the Host Controller and Link Manager uses the Baseband directly and the Host Controller furthermore is in charge of initiating Link Manager functionality I have chosen to tie these two together in one symbol in the implementation. As far as I can see that is actually what is intended in the specification as there is no indication of how the Host Controller should communicate with the Link Manager and since the Host Controller Interface in the protocol stack is just shown as a line and not an individual protocol layer.

2.1 The Symbol

The HCI/LMP symbol has changed many times since the initial design – the only thing that has not undergone any changes is the interface to the upper layer which Benjamin Hanney and I decided upon very early in the course of implementation.



In the final version of the symbol I am using all the ports supplied by the original Baseband symbol and have furthermore added three ports AMBDIn, StateOut and Inquiry_In which all were necessary for vital communication between the Baseband and the Link Manager. These three ports are all used for sending data that is private to the Link Manager and the Control part of the Baseband and not something which is specified for either the protocols. This together with the fact that the Link Manager's main task is to control the state of the Baseband leads me to consider that these two might actually be supposed to be tied closer together (IE. let the Link Manager Symbol be part of the Baseband). This makes good sense to me as the Baseband actually does not work without some form of management (in the old implementation of the Baseband some of the Link Manager's tasks were performed to set up the connections when initialising). Either this or an interface between Link Manager and Baseband should be specified (in the Bluetooth Specification the Link Manager is just assumed to have full control of the workings of the Baseband).

2.2 The Interface

2.2.1 Upper

The upper interface for the symbol corresponds to that of the Host Controller Interface and contains the following ports:

HCIPacket_in
Direction: Input
Arguments: HCIPacket
Operation: Used to receive a packet for processing.

HCIPacket_out
Direction: Output
Arguments: HCIPacket
Operation: Used to send events or data from Host Controller to Host.

2.2.2 Lower

The lower interface for the symbol is modelled to match the upper interface of the already existing Baseband symbol. Extra ports have been added to exchange crucial information between the Link Manager and the Baseband.

OneSlotPacketIn
Direction: Input
Arguments: TOneSlotBasebandPacket
Operation: Used to receive a raw single slot packet to be processed.

FiveSlotPacketIn
Direction: Input
Arguments: TFiveSlotBasebandPacket
Operation: As above but receives five slot packets.

BasebandReady_in
Direction: Input
Arguments: Dummy
Operation: Used to signify that the lower layer is ready for operation.

OneSlotPacketOut
Direction: Output
Arguments: TOneSlotBasebandPacket
Operation: Used to send a raw single slot packet to the other side.

FiveSlotPacketOut
Direction: Output
Arguments: TFiveSlotBasebandPacket
Operation: As above but sends five slot packets.

LMPReady_out
Direction: Output
Arguments: Dummy
Operation: Used to tell the Baseband that the Link Manager is ready for operation.

ClockIn
Direction: Input
Arguments: CLKN
Operation: Used to receive the clock signal emitted by the baseband protocol.

AMBDIn
Direction: Input
Arguments: TAMBD_ADDR
Operation: Used by the Baseband to inform the Link Manager of the Active Member address decided upon by the Baseband.

StateOut
Direction: Output
Arguments: TStateIn
Operation: This output port is used to set the state of the Baseband.

Inquiry_In
Direction: Input
Arguments: HCIPacket
Operation: This input port is used to receive results of inquiries done by the Baseband.

3 The code

3.1 Local data types

con: This struct contains all information needed about each connection. This includes the Connection Handle, Active Member Address, BD Address, type of connection (ACL or SCO), the intersection of supported features on the Link Manager in each end of the connection, the currently used packet type, the packet types allowed by the initiator of the connection and the opcode of the last LMP message sent.

pdu: Used to typecast the memory chunk of an LMP command to it's logical contents.

3.2 Helper functions

A whole bunch of helper functions has been added to look up a specific connection based on the different available data (AM address, BD address, connection handle). Functions to add and remove connection structures are also added as well as a function for looking up room for a connection in a bitmap of available connections. Having to add standard functions like `memset` and `strlen` was also necessary (or at least easier than figuring out how to get access to them in VCC). Finally a function for choosing the packet type to be used based on the shared features of the connected devices and the wish of the creator of the connection is also implemented. In this function my preference of packet type is encoded, so that if several packet types are available it will choose the largest packet size.

3.3 Initialisation

Besides from initialising all global state variables one thing is done here which is not specified that I have to do in the protocol specification. This is making the master device inquire for all slaves in the pico net. I do this because this initialises all devices and puts them in PAGESCAN mode and thus ready for connection. The Host Controller actually provides a function for doing this but neither of the above layers seem to be interested in knowing the addresses of the other devices (and as there is no way of converting between BD addresses and IP addresses I can see why they are not interested). It is still completely fine for the above layers to use this function later (if there for example are new devices in the pico net (which won't happen in VCC but might happen in a real network)).

3.4 Runtime function

3.4.1 Basic Concepts

As the Baseband layer is not always ready to receive data the LMP command to be executed next is always kept in a global variable until it is sent. Likewise data packets sent by the above layer is kept in a buffer until sent. Whenever the Baseband is ready any waiting packet will be sent. The LMP command has priority above any waiting data as specified in paragraph 1 of the Link Manager Protocol, p. 191.

Notice that as there is only one variable for storing the LMP command to be executed this will be overwritten if the above layer executes several HCI commands without waiting for a Command Complete or Command Status event. I take this to be the correct behaviour as it on p. 533 is stated that '[on] power-on the Host can send a maximum of one outstanding HCI Command Packet until a Command Complete or Command Status event has been received.' Furthermore there is nowhere any mention of an LMP command queue and there isn't any reason to have one if the above layer behaves nicely. This is one of the (many) places where I find the specification a bit vague, explaining the behaviour superficially in many words with little content.

It is stated in paragraph 2 of the Link Manager Protocol specification that in case an SCO connection exists LMP commands may be sent in DV packets, otherwise they have to be sent in DM1 packets. DV packets are synchronous transfer combined data and voice packets which are not implemented in the Baseband so this we have to ignore. As it is only a 'may' the usage of DM1 packets for all LMP commands is still fine. DM packets were not supported in the Baseband either, but these I have included by adding 2/3 FEC encoding for DH like packets.

3.4.2 Host Controller Interface functions

As there are functions in the specification of the HCI Link Control Commands that does not make sense to us (stuff to do with encryption, power management etc.) it is very lucky that not all commands needs to be supported in order for this to be a valid implementation of the Host Controller Interface. This is due to the command Read_Remote_Supported_Features. With this you can specify a subset of the specified instruction set which is supported and the above layers are thus restricted to only using these commands. Here follows a description of the supported commands.

Inquiry Makes the Baseband enter the INQUIRY mode. This will generate the Command Status event plus an Inquiry Result event for each device that answers the inquiry. At timeout or when Num_Responses devices have answered the Inquiry Complete event will be generated.

Inquiry_Cancel Makes the Baseband enter STANDBY mode. Generates a Command Complete event.

Periodic_Inquiry_Mode Works similarly to calling Inquiry periodically with a random time between Min_Period_Length and Max_Period_Length between each. Generates same events as Inquiry but does so continuously times until the periodic inquiry mode is exited.

Exit_Periodic_Inquiry_Mode Exits the periodic inquiry mode immediately (not waiting to send Inquiry Complete if an inquiry is currently running). Generates a Command Complete event.

Create_Connection Tries to establish an ACL connection to the device with the specified BD address by making the Baseband PAGE the device. A connection structure is initialised. When the Baseband has assigned the device an Active Member address it will let the LM know and thus we both know this address and that the device is initialised and we can continue to set up the connection. A Command Status event will be generated as will a Connection Complete event on both connected devices once the Link Manager has finished to set up the connection.

Disconnect If an existing connection is specified as the Connection_Handle parameter this initiates the termination sequence. The parameter Reason should contain information about why the disconnection is requested.

Add_SCO_Connection Adds a logical connection handle to the ACL connection specified by the Connection_Handle parameter. The connection will use one of the packet types implied by the Packet_Type parameter. At least one packet type which is known to be available should be supplied and then the Link Manager will choose the most appropriate of the listed available types.

Accept_Connection_Request Accepts a connection to the requesting device with the BD address in BD_ADDR. If the connection requested allows for role switching, the accepting device can indicate it's desired role.

Reject_Connection_Request As the above, but the above layer has to supply a reason for the rejection.

Change_Connection_Packet_Type Try to set the packet type for the connection indicated by Connection_Handle to one of the types specified in Packet_Type. This should be used after Read_Remote_Supported_Features in order to be sure that the requested packet type is available.

Remote_Name_Request Will initiate the procedure of acquiring the user friendly name of a remote device.

Read_Remote_Supported_Features Will initiate the procedure of exchanging the list of supported features of each device and report the intersection of these to each device.

3.4.3 Link Manager Commands

The Link Manager commands are only used by the protocols internally and are as such not of much interest to the reader. The curious reader can look at the code in appendix A where these are listed in the code.

3.4.4 Data Packets

When sending data across a connection, the data is first put into an output buffer. In the first packet sent on a connection, the total length of the data to be transferred is written. Depending on the size of the packet type of the current connection the data from the upper layer will be split into as many chunks as required and sent through the baseband one by one. On the other end the incoming data is collected in an input buffer until the last packet have been received. When this happens the collected data will be sent to the higher layer and both ends of the connection are informed that the data was sent successfully.

4 Changes made to the Baseband implementation

4.1 Control

For the Link Manager to work it has to be able to set the state of the Baseband (Inquiry, Standby and Page) which has been implemented by adding a port for this specific purpose. An Inquiry will expire after a timeout value or when the indicated number of slaves has responded. If in the Page state, we will look through the list of slaves that has formerly responded to an inquiry, but even if the slave is not found we allow the address to be paged in order to allow for small networks in which all devices are known and no inquiry is necessary.

The code previously ignored information about the Active Member address, and instead had an extra piece of information (DestinationBDADDR) in each packet header to hack around this problem. I have corrected the use of Active Member addresses and thus added an extra port to let the higher levels know of the AM address to use for a given BD address. There still needs to be cleaned up in one place – namely in the Composer where null packets are sent without information about the AM address of the receiver – to solve this problem the internal workings of the Baseband would have to be changed (since the Composer would need to know of the BD to AM address pairing) which I did not want to start doing as the problem is not that big (only happens during setup and does not increase the amount of traffic considerably).

Five slot packets were incorrectly handled (seems there was a bit of copying and pasting from the one slot packet handling code that went wrong), which meant that the information in them would be cropped. This has been fixed.

Upon initialisation the Baseband informs the Link Manager of it's Bluetooth Device Address.

Disconnection wasn't really possible when there was no use of the Active Member address (as disconnection simply means to give up one's status as an active member). This now makes sense.

4.2 Transmit and Receive

The DM1 and DM5 packets have been added. These contain less data than the corresponding DH1 and DH5 packets as the data is FEC2/3 encoded. As I did not have time to get into the coding and encoding or their implementation in the Baseband I have made support for these by copying bits and pieces from the DH packet code and other packets that uses the encoding. As the data comes through unmodified I assume this to work, but as I have not have a single look at the specification for this I cannot guarantee it (sorry!).

5 Testing

A known issue of conformance is the implementation of SCO links. In a real life implementation of the protocol, SCO links has Quality of Service and the like which only makes sense when one have the concept of bandwidth. As I cannot see a way of representing this concept in our simulation environment I have cheated and let the SCO links behave like ACL links except that they can only be created when the physical channel exists and they have to be disconnected before the ACL link.

Another issue when modelling in VCC is the use of data structures which are kind of an abstraction from the raw bytes that are described in the specification. This of course eases the understanding of symbols and the process of making changes to data types and interfaces, but it is unlike any real life implementation where there is only dealt with raw data and there is therefore easier integration with other devices (in real life any two implementations of different layers should be able to inter operate, whereas in our simulation environment we have logical ports that have to agree on data types etc.)

The validation of correctness of the protocols is not an easy task when dealing with something as big as the LMP and the HCI. Luckily an implementation of LMP can be a subset of all that is described in the specification and thus the implementation has been cut down to what is relevant in a simulation environment. Each command has been tested with a simple test harness so that upon correct use, the correct behaviour can be observed. Also every possible error event that can be generated for each command has been provoked. This of course is in no way an exhaustive test of the protocol but signifies that upon correct use, the behaviour is correct. An exhaustive test would have to include all permutations of the commands, extensive tests of parameter values plus testing if something might possibly go wrong with the buffering of commands and data when many devices speak to each other simultaneously. A test harness that did any of this would most likely be bigger than the implementation of the protocol and as this would be too great a task for this course, my efforts have gone to make sure that the implementation works in our current protocol stack. I will, however, dare to estimate that my implementation follows the specification closely and I cannot think of any problems with integration with other protocols than the L2CAP. As for the downwards interface there really is no specification of an interface, which to me suggests that the Link Manager and Baseband really should be closer linked than they are in our protocol stack. The interface that is defined between Baseband and LM is thus fairly tight, and it would not be an easy task to replace the Baseband. If the Baseband ever needs replacement, I would rather suggest that at least the Link Manager should become part of it.

6 Analysis and conclusion

The memory required for this implementation is very limited. Besides space for input and output buffers (each 12kb) there is only needed memory for the information about seven connections plus a few intermediate variables. Even with the compiled code I would think this less than 30kb. The size of the buffers are greater than what is used in the current implementation of the protocol stack, but I have left them at that size to support future enhancements.

The performance of my implementation seems about as good as can be. There are no unnecessary copying of memory, no more advanced data manipulation than quick table lookups and with the Host Controller Interface and the Link Manager in one symbol we save the extra traffic and the need for memory buffers in both stages.

Altogether the result of the LMP and HCI implementation are very satisfactory. The implementation lies very close to my interpretation of the protocol description and I could not imagine the speed and size to be much better. In our implementation of the protocol stack the Link Manager and Host Controller are fully sufficient and even offer functionality that as of yet is completely unused but might come in handy for future users of the protocol stack.

7 Using VCC

Although being a very powerful tool for simulating execution and communication of behaviours across different platforms, VCC has many issues that makes programming, version control and especially debugging extremely (and seemingly unnecessarily) difficult. In my opinion way too much development time was wasted fighting battles with the program. Most of these problems were due to unspecific error messages (IE. in the worst of all the battles in which I was involved, Benjamin and I spent six hours merging our code which was very hard because of an error message which did not make any sense to us ("Error executing pinpointu.exe") and this even being used for two different problems (one of them simply being that the path name was too long) so that the true problem was extremely hard to identify), but also problems such as VCC being unable to use newly added data types before exiting and restarting the program made the implementation into a very time consuming task. That each and every behaviour in a simulation has to be recompiled before simulating meant that debugging included waiting approximately six minutes every time you change some code (I know that the solution to this is just writing correct code in the first place, but this is not always realistic to do).

One annoying problem is that the whitebox C is not ANSI C compliant. This means that for example the following does not work:

```
(&variable+1)=value;
```

whereas the following is fine:

```
pointer = &variable;
(pointer+1)=value;
```

When you know this you can of course easily avoid any problems, but I cannot see the point in implementing something which is almost but not entirely ANSI C.

Another place where whitebox C differs from other standards is when using `memcpy`. In this implementation it seems that the memory areas may overlap (which they mustn't if conforming to the ISO 9899 standard). Something like this will be likely to cause problems when moving from the simulation environment to a real C language and might even produce incorrect simulation results as the correct function to use in this case is `memmove` which must be about twice as slow as `memcpy` (but which I can't even find in VCC). I am probably being too pedantic about this.

8 Canny Edge Detection

The Canny Edge Detection is a general algorithm for edge detection that consists of several different algorithms and is generally considered to be the best approach for edge detection (although certain steps of the algorithm may vary according to preference). The Department of Artificial intelligence have a great web page (http://www.dai.ed.ac.uk/homes/rbf/HIPR/) explaining the involved steps, and actually a lot of the code for my own version of the edge detection is a direct translation of java code supplied on this site for an interactive demonstration. As my own involvement in the edge detection has only been to make a prototype version in standard C, I will leave the explanation of the different stages to Alan Stanley (http://www.dcs.ed.ac.uk/home/slipc/Application/implementation.html) who has implemented each step in VCC and connected it with the application and a web camera.

A Code for LMP/HCI

```
#include "white_interface.h"
#include <TBasebandPacketTYPE.h>
#include <HCI_Packet.h>
#include <HCI_CommandPacket.h>
#pragma vccType TBasebandPacketTYPE MooToothTypes
#pragma vccType HCI_Packet LMP_Types
#pragma vccType HCI_CommandPacket LMP_Types


#define MAX_CONNECTIONS 20
#define NULL 0

#define debug_memcpy(x, y, z) \
/*vccPrintPdxDebugInfo("%_START_ About to debug_memcpy, length %d, line %d %_END_", z, __LINE__);*/\
memcpy(x, y, z);\
/*vccPrintPdxDebugInfo("%_START_ memcpied. %_END_");*/\

int i;
/*#include <string.h>
 */
#define vcc_union    __union
#define vcc_unionTag __unionTag
#define LM                 3 /*The code of L_CH in LMP packet. 0b11*/
#define L2CAPStart         2
#define L2CAPContinue      1

#define DM1 0x0008
#define DH1 0x0010
#define DM3 0x0400
#define DH3 0x0800
#define DM5 0x4000
#define DH5 0x8000
#define HV1 0x0020
#define HV2 0x0040
#define HV3 0x0080

#define hDM1 0x03
#define hDH1 0x04
#define hDM3 0x0A
#define hDH3 0x0B
#define hDM5 0x0E
#define hDH5 0x0F


/* Supported features bit mask as described on page 234 */
#define feat_3SlotPackets 0x01
#define feat_5SlotPackets 0x02
#define feat_encryption 0x04
#define feat_slotOffset 0x08
#define feat_timingAccuracy 0x10
#define feat_switch 0x20
```

```
#define feat_holdMode 0x40
#define feat_sniffMode 0x80

#define feat_parkMode 0x01
#define feat_RSSI 0x02
#define feat_channelQualityDrivenDataRate 0x04
#define feat_SCOLink 0x08
#define feat_HV2Packets 0x10
#define feat_HV3Packets 0x20
#define feat_ulawLog 0x40
#define feat_AlawLog 0x80

#define feat_CVSD 0x01
#define feat_pagingScheme 0x02
#define feat_powerControl 0x04

int mystrlen(char *mystring){
 int i=0;
 while ((*mystring++)!=0)
        i++;
 return i;
}


#define opLMP_accepted      3
#define opLMP_in_rand       8
#define opLMP_comb_key      9
#define opLMP_au_rand      11
#define opLMP_sres         12
#define opLMP_encryption_mode_req     15
#define opLMP_encryption_key_size_req 16
#define opLMP_start_encryption_req    17

#define opLMP_name_req 1
#define opLMP_name_res 2
#define opLMP_not_accepted 4
#define opLMP_clkoffset_req 5
#define opLMP_clkoffset_res 6
#define opLMP_detach 7
#define opLMP_unit_key 10
#define opLMP_temp_rand 13
#define opLMP_temp_key 14
#define opLMP_stop_encryption_req 18
#define opLMP_switch_req 19
#define opLMP_hold 20
#define opLMP_hold_req 21
#define opLMP_sniff 22
#define opLMP_sniff_req 23
#define opLMP_unsniff_req 24
#define opLMP_park 25
#define opLMP_park_req 26
#define opLMP_set_broadcast_scan_window 27
#define opLMP_modify_beacon 28
```

```
#define opLMP_unpark_BD_ADDR_req 29
#define opLMP_unpark_PM_ADDR_req 30
#define opLMP_incr_power_req 31
#define opLMP_decr_power_req 32
#define opLMP_max_power 33
#define opLMP_min_power 34
#define opLMP_auto_rate 35

#define opLMP_version_req 37
#define opLMP_version_res 38
#define opLMP_features_req 39
#define opLMP_features_res 40
#define opLMP_SCO_link_req 43
#define opLMP_max_slot 45
#define opLMP_max_slot_req 46
#define opLMP_timing_accuracy_req 47
#define opLMP_timing_accuracy_res 48
#define opLMP_setup_complete 49
#define opLMP_use_semi_permanent_key 50
#define opLMP_host_connection_request 51
#define opLMP_slot_offset 52
#define opLMP_page_mode_req 53
#define opLMP_page_scan_mode_req 54
#define opLMP_supervision_timeout 55


int num_connections;

enum conType {
  ACL,
  SCO
};

enum conState {
  NO_CONNECTION,
  CONNECTION_REQUESTED,
  CONNECTED
} State;

struct con {
  int Connection_Handle;
  int AMADDR;
  enum conState state;
  enum conType type;
  unsigned int features;
  int encrypted;
  LMP_Packet_Type_t packetType;
  LMP_Packet_Type_t availablePacketTypes;
  TBluetoothDeviceAddress BDADDR;
  int LastLMPOp;
} Connections[MAX_CONNECTIONS];
```

```
struct con *llist_lookup(int handle){
  int i;
  for (i=0;i<MAX_CONNECTIONS;i++){
    if(handle==Connections[i].Connection_Handle)
      return &Connections[i];
  }
  vccPrintPdxDebugInfo("%_START_ expect crash line %d %_END_",__LINE__);
  return NULL;
}

TBluetoothDeviceAddress *llist_get_bd_by_am(int AM_ADDR){
  int i;
  for (i=0;i<MAX_CONNECTIONS;i++){
    if (AM_ADDR==Connections[i].AMADDR)
      return &Connections[i].BDADDR;
  }
  vccPrintPdxDebugInfo("%_START_ expect crash line %d %_END_",__LINE__);
  return NULL;
}
int llist_get_am_by_bd(TBluetoothDeviceAddress BD_ADDR){
  int i;
  for (i=0;i<MAX_CONNECTIONS;i++){
    if ((BD_ADDR.UAP==Connections[i].BDADDR.UAP) &&
                (BD_ADDR.LAP==Connections[i].BDADDR.LAP) &&
                (BD_ADDR.NAP==Connections[i].BDADDR.NAP))
      return Connections[i].AMADDR;
  }
  return 0;
}

struct con *llist_lookup_amaddr(int AM_ADDR){
  int i;
  for (i=0;i<MAX_CONNECTIONS;i++){
    if (AM_ADDR==Connections[i].AMADDR)
      return &Connections[i];
  }
  vccPrintPdxDebugInfo("%_START_ expect crash line %d %_END_",__LINE__);

  return NULL;
}


struct con *llist_lookup_bdaddr(TBluetoothDeviceAddress BD_ADDR){
  int i;
  for (i=0;i<num_connections;i++){
    if ((BD_ADDR.UAP==Connections[i].BDADDR.UAP) &&
                (BD_ADDR.LAP==Connections[i].BDADDR.LAP) &&
                (BD_ADDR.NAP==Connections[i].BDADDR.NAP))
      return &Connections[i];
  }
  vccPrintPdxDebugInfo("%_START_ expect crash line %d %_END_",__LINE__);
  return NULL;
}
```

```
struct con *llist_add(enum conType type,
                   TBluetoothDeviceAddress bdaddr,
                   int handle){

  if(num_connections<MAX_CONNECTIONS){
          Connections[num_connections].BDADDR.UAP = bdaddr.UAP;
          Connections[num_connections].BDADDR.LAP = bdaddr.LAP;
          Connections[num_connections].BDADDR.NAP = bdaddr.NAP;
          Connections[num_connections].type = type;
          Connections[num_connections].encrypted = 0;
          Connections[num_connections].packetType = DH1;
          Connections[num_connections].state=NO_CONNECTION;
          Connections[num_connections].LastLMPOp = 0;
          Connections[num_connections].Connection_Handle = handle;
          num_connections++;
          return &Connections[num_connections-1];
  } else {
  vccPrintPdxDebugInfo("%_START_ expect crash line %d %_END_",__LINE__);
    return NULL;
  }
}

void llist_remove_connection(int handle){
  int i;
  for (i=handle+1; i<num_connections; i++){
    memcpy(&Connections[i], &Connections[i+1], sizeof(struct con));
  }
}


vcc_bool basebandReady;
vcc_bool basebandInquiryComplete;
int send; /* Use this to describe packet type as well */
vcc_bool receive;

int disconnect = 0;

typedef struct _LMP_PDU_
{
  unsigned char opcode;
  unsigned char body[20];
}TLmpPdu;

vcc_bool initializeBDADDR;

int maxConnectionHandle;
TOneSlotBasebandPacket           oneSlot;
TFiveSlotBasebandPacket          fiveSlot;
HCI_Packet hcipacket;
TBasebandDH1Packet                 *pdh1;
TBasebandDH5Packet                 *pdh5;
TStateIn stateout;
```

```
TBluetoothDeviceAddress myBDADDR;
char nameBuf[249]; /* 248 plus room for terminating 0 */

unsigned char outbuffer[12000];
int OutBufferTotalLength;
int OutBufferCount;
int OutBufferReceiverAMADDR;

unsigned char inbuffer[12000];
int InBufferTotalLength;
int InBufferCount;
int InBufferFromAMADDR;

void memset(char *a, int b, int count){
  int i;
  for (i=0;i<count;i++)
    a[i]=(char)b;
}

void choosePacketType(struct con *c){
  /* I have decided the order of these - if you have another
     preference feel free to change them */

  if ((c->features&feat_5SlotPackets)&&(c->availablePacketTypes&DH5))
    c->packetType = DH5;
  else if ((c->features&feat_5SlotPackets)&&(c->availablePacketTypes&DM5))
    c->packetType = DM5;
  else if ((c->features&feat_3SlotPackets)&&(c->availablePacketTypes&DH3))
    c->packetType = DH3;
  else if ((c->features&feat_3SlotPackets)&&(c->availablePacketTypes&DM3))
    c->packetType = DM3;
  else if (c->availablePacketTypes&DH1)
    c->packetType = DH1;
  else
    c->packetType = DM1;
}


/* Init Function */
void poin_entry_Init() {
  stateout.State = STANDBY;
  num_connections = 0;
  maxConnectionHandle = 0;
  basebandReady = vcc_false;
  basebandInquiryComplete = vcc_false;
  send = 0;
  memset((char*)&oneSlot, 0, sizeof(TOneSlotBasebandPacket));
  memset((char*)&hcipacket, 0, sizeof(HCI_Packet));
  OutBufferTotalLength = 0;
  OutBufferCount = 0;
  InBufferTotalLength = 0;
  InBufferCount = 0;
  initializeBDADDR = vcc_true;
```

```
  if (Master_Value()){
        stateout.State=INQUIRY;
        stateout.max_inquiry_slave=2;
        stateout.inquiry_length=100000000;
        StateOut_Post(&stateout);
  }
}

/* Run Function */

void poin_entry_Run() {
  int receiverAMADDR;
  TLmpPdu pdu;
  struct con *tempCon;
  LMP_Packet_Type_t packetType;
  TBluetoothDeviceAddress otherdeviceBDADDR;
  HCI_CommandPacket *hcicommand;
  HCI_EventPacket *hcievent;
  int length = 0;
  TAMBD_ADDR *tmpAMBD;
  int L_CH = 0;
  int packetSize = 0;
  int TYPE = 0;

  memset((char*)&pdu, 0, sizeof(TLmpPdu));

  /* Since no one in the higher layers wish to inquire for devices I
     decided to do it myself -- in a true baseband implementation
         this event should be passed to the higher layer. */
  if (Inquiry_In_Enabled()){
    switch(Inquiry_In_Value()->vcc_unionTag){
    case HCI_Packet_Event_tag:
      hcievent = &(Inquiry_In_Value()->vcc_union.Event);
      switch(hcievent->vcc_unionTag){
            case HCI_EventPacket_Inquiry_Complete_tag:
              basebandInquiryComplete = vcc_true;
                  break;
          }
        }
  }


  if(AMBDIn_Enabled()){
    if(initializeBDADDR){
          initializeBDADDR = vcc_false;
          myBDADDR = AMBDIn_Value()->BD_ADDR;
        } else {
            if (Master_Value()){
              /* A slave has had assigned an am address */

        /*  vccPrintPdxDebugInfo("%_START_ slave has been assigned amaddr %d %_END_", AMBDIn_Value()->AM_ADDR);
          vccPrintPdxDebugInfo("%_START_ slave bdaddrlap %d %_END_", AMBDIn_Value()->BD_ADDR.LAP);
```

```
  vccPrintPdxDebugInfo("%_START_ slave bdaddruap %d %_END_", AMBDIn_Value()->BD_ADDR.UAP);
  vccPrintPdxDebugInfo("%_START_ slave bdaddrnap %d %_END_", AMBDIn_Value()->BD_ADDR.NAP);
 */

      tempCon = llist_lookup_bdaddr(AMBDIn_Value()->BD_ADDR);
          tempCon->AMADDR = receiverAMADDR = AMBDIn_Value()->AM_ADDR;
          tempCon->LastLMPOp = pdu.opcode = opLMP_host_connection_request;

        length=1;

          send = DM1;
          oneSlot.vcc_unionTag=TOneSlotBasebandPacket_DH1Packet_tag;
          pdh1=&(oneSlot.vcc_union.DH1Packet);
          pdh1->header.AM_ADDR=receiverAMADDR;
          pdh1->header.TYPE=hDM1;
          pdh1->payload.L_CH=LM;
          pdh1->payload.FLOW=0;
          pdh1->payload.LENGTH=length;
          debug_memcpy(pdh1->payload.payloadBody,&pdu,length);

          } else {
 /*
  vccPrintPdxDebugInfo("%_START_ master has assigned amaddr %d %_END_", AMBDIn_Value()->AM_ADDR);
  vccPrintPdxDebugInfo("%_START_ master bdaddrlap %d %_END_", AMBDIn_Value()->BD_ADDR.LAP);
  vccPrintPdxDebugInfo("%_START_ master bdaddruap %d %_END_", AMBDIn_Value()->BD_ADDR.UAP);
  vccPrintPdxDebugInfo("%_START_ master bdaddrnap %d %_END_", AMBDIn_Value()->BD_ADDR.NAP);
 */
          tempCon = llist_add(ACL, AMBDIn_Value()->BD_ADDR, ++maxConnectionHandle);
      tempCon->AMADDR = AMBDIn_Value()->AM_ADDR;
  }
 }
}


if (HCIPacket_in_Enabled()){
 switch(HCIPacket_in_Value()->vcc_unionTag){
 case HCI_Packet_Command_tag:
  hcicommand = &(HCIPacket_in_Value()->vcc_union.Command);
  switch(hcicommand->vcc_unionTag){
  case HCI_CommandPacket_Create_Connection_in_tag:
       otherdeviceBDADDR = hcicommand->vcc_union.Create_Connection_in.BD_ADDR;
       tempCon =  llist_add(ACL,
                                    otherdeviceBDADDR,
                                    ++maxConnectionHandle);
       tempCon->availablePacketTypes = hcicommand->vcc_union.Create_Connection_in.Packet_Type;
       stateout.State=PAGE;
       stateout.BD_ADDR= hcicommand->vcc_union.Create_Connection_in.BD_ADDR;
       break;

  case HCI_CommandPacket_Disconnect_in_tag:
       pdu.body[0]=hcicommand->vcc_union.Disconnect_in.Reason;
       pdu.opcode=opLMP_detach;
       tempCon = llist_lookup(hcicommand->vcc_union.Disconnect_in.Connection_Handle);
```

```
            length=2;
        receiverAMADDR = tempCon->AMADDR;
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Disconnection_Complete_tag;
            hcievent->vcc_union.Disconnection_Complete.Status = 0x00; /* Disconnection has occured */
            hcievent->vcc_union.Disconnection_Complete.Connection_Handle =tempCon->Connection_Handle;
            hcievent->vcc_union.Disconnection_Complete.Reason =
        hcicommand->vcc_union.Disconnect_in.Reason;
            HCIPacket_out_Post(&hcipacket);
            disconnect = tempCon->Connection_Handle;
            break;

case HCI_CommandPacket_Inquiry_in_tag:
            stateout.State=INQUIRY;
            stateout.max_inquiry_slave=hcicommand->vcc_union.Inquiry_in.Num_Responses;
            stateout.inquiry_length=hcicommand->vcc_union.Inquiry_in.Inquiry_Length;
            break;

case HCI_CommandPacket_Inquiry_Cancel_in_tag:
            stateout.State=STANDBY;
            stateout.max_inquiry_slave=0;
            stateout.inquiry_length=0;
            StateOut_Post(&stateout);
            break;
case HCI_CommandPacket_Periodic_Inquiry_Mode_in_tag:
            break;
case HCI_CommandPacket_Exit_Periodic_Inquiry_Mode_in_tag:
            break;
case HCI_CommandPacket_Add_SCO_Connection_in_tag:
            /* SCO connections are simulated on top of ACL
               as we have no true bandwidth etc.. */
            tempCon = llist_lookup(hcicommand->vcc_union.Add_SCO_Connection_in.Connection_Handle);
            if (tempCon){
                    struct con *newCon = llist_add(SCO,
                                                                     otherdeviceBDADDR,
                                                                          ++maxConnectionHandle);
                    newCon->availablePacketTypes = hcicommand->vcc_union.Add_SCO_Connection_in.Packet_Type;
                    newCon->AMADDR = tempCon->AMADDR;
                newCon->LastLMPOp = pdu.opcode = opLMP_SCO_link_req;
            length=1;
                send = DM1;
                    receiverAMADDR = newCon->AMADDR;
            } else {
                hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
                hcievent=&(hcipacket.vcc_union.Event);
                hcievent->vcc_unionTag=HCI_EventPacket_Connection_Complete_tag;
                hcievent->vcc_union.Connection_Complete.Status = 0x01;            /* Connection failed */
                hcievent->vcc_union.Connection_Complete.Encryption_Mode = 0x00;   /* Return invalid  */
                hcievent->vcc_union.Connection_Complete.Connection_Handle =          /* Connection_Handle */
                      hcicommand->vcc_union.Add_SCO_Connection_in.Connection_Handle;
                hcievent->vcc_union.Connection_Complete.Link_Type = 0x00;
                HCIPacket_out_Post(&hcipacket);
```

```
        }
        break;
case HCI_CommandPacket_Accept_Connection_Request_in_tag:
        pdu.body[0]=opLMP_host_connection_request;
        pdu.opcode=opLMP_accepted;
        length=2;
        tempCon = llist_lookup_bdaddr(hcicommand->vcc_union.Reject_Connection_Request_in.BD_ADDR);
        receiverAMADDR = tempCon->AMADDR;
        break;
case HCI_CommandPacket_Reject_Connection_Request_in_tag:
        tempCon = llist_lookup_bdaddr(hcicommand->vcc_union.Reject_Connection_Request_in.BD_ADDR);
        pdu.body[0]=opLMP_host_connection_request;
        pdu.body[1]=hcicommand->vcc_union.Reject_Connection_Request_in.Reason;
        pdu.opcode=opLMP_not_accepted;
        length=3;
      receiverAMADDR = tempCon->AMADDR;
        hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
        hcievent=&(hcipacket.vcc_union.Event);
        hcievent->vcc_unionTag=HCI_EventPacket_Connection_Complete_tag;
        hcievent->vcc_union.Connection_Complete.BD_ADDR = tempCon->BDADDR;
        hcievent->vcc_union.Connection_Complete.Status = 0x01;           /* Connection failed */
        hcievent->vcc_union.Connection_Complete.Encryption_Mode = 0x00;   /* Rest of the parameters */
        hcievent->vcc_union.Connection_Complete.Connection_Handle = 0x00; /* can be ignored */
        hcievent->vcc_union.Connection_Complete.Link_Type = 0x01;
        HCIPacket_out_Post(&hcipacket);
        llist_remove_connection(tempCon->Connection_Handle);

        break;
        /*  This has to do with encryption - ignore */
case HCI_CommandPacket_Link_Key_Request_Reply_in_tag:
case HCI_CommandPacket_Link_Key_Request_Negative_Reply_in_tag:
case HCI_CommandPacket_PIN_Code_Request_Reply_in_tag:
case HCI_CommandPacket_PIN_Code_Request_Negative_Reply_in_tag:
        break;
case HCI_CommandPacket_Change_Connection_Packet_Type_in_tag:
        break;
case HCI_CommandPacket_Set_Connection_Encryption_in_tag:
        /*  This has to do with encryption - ignore */
        break;
case HCI_CommandPacket_Change_Connection_Link_Key_in_tag:
        /*  This has to do with encryption - ignore */
        break;
case HCI_CommandPacket_Master_Link_Key_in_tag:
        /*  This has to do with encryption - ignore */
        break;
case HCI_CommandPacket_Remote_Name_Request_in_tag:
        tempCon = llist_lookup_bdaddr(hcicommand->vcc_union.Reject_Connection_Request_in.BD_ADDR);
        pdu.body[0]=0;
        pdu.opcode=opLMP_name_req;
        length=1;
        receiverAMADDR = tempCon->AMADDR;
        break;
```

```
    case HCI_CommandPacket_Read_Remote_Supported_Features_in_tag:
              tempCon = llist_lookup(hcicommand->vcc_union.Read_Remote_Supported_Features_in);
              pdu.body[0]=feat_5SlotPackets;
              pdu.body[1]=0;
              pdu.body[2]=0;
              pdu.opcode=opLMP_features_req;
              length=3;
              receiverAMADDR = tempCon->AMADDR;
              break;

    case HCI_CommandPacket_Read_Remote_Version_Information_in_tag:
              break;
    case HCI_CommandPacket_Read_Clock_Offset_in_tag:
              break;
    case HCI_CommandPacket_Authentication_Requested_in_tag:
              /*  This has to do with encryption - ignore */
              break;
    }
    if (length>0){
                send = DM1;
                /* Luckily in our implementation of DM1 packets we use
                   DH1 packets, just setting header.TYPE to DM1 and
                   restraining ourselves to using 18 bytes..
                */
                oneSlot.vcc_unionTag=TOneSlotBasebandPacket_DH1Packet_tag;
                pdh1=&(oneSlot.vcc_union.DH1Packet);
                pdh1->header.AM_ADDR=receiverAMADDR;
                pdh1->header.TYPE=hDM1;
                pdh1->payload.L_CH=LM;
                pdh1->payload.FLOW=0;
                pdh1->payload.LENGTH=length;
                debug_memcpy(pdh1->payload.payloadBody,&pdu,length);
        }
    break;

  case HCI_Packet_Data_tag:
    tempCon = llist_lookup(HCIPacket_in_Value()->vcc_union.Data.ConnectionHandle);
        vccPrintPdxDebugInfo("%_START_ data packet type %d , conhandle %d%_END_", tempCon->packetType, HCIPacket_in_Value()->vcc_union.Data.ConnectionHandle);
    if (OutBufferCount==0){
              OutBufferTotalLength = HCIPacket_in_Value()->vcc_union.Data.DataTotalLength+4;
              OutBufferReceiverAMADDR = tempCon->AMADDR;
              debug_memcpy(outbuffer, &(OutBufferTotalLength), 4);
              debug_memcpy(outbuffer+4, &(HCIPacket_in_Value()->vcc_union.Data.DataBytes), OutBufferTotalLength/*-4*/);
    }
    break;

  default:
    break;
  }
}

if (OneSlotPacketIn_Enabled()){
      switch (OneSlotPacketIn_Value()->vcc_unionTag){
```

```
        case TOneSlotBasebandPacket_DH1Packet_tag:
              pdh1=&(OneSlotPacketIn_Value()->vcc_union.DH1Packet);
      L_CH = pdh1->payload.L_CH;
              TYPE = pdh1->header.TYPE;
              tempCon = llist_lookup_amaddr(pdh1->header.AM_ADDR);
              switch (TYPE){
                case hDM1:
                  packetSize = 18;
                  break;
                case hDH1:
                  packetSize = 27;
                      break;
              }
              break;

      }
}
if (FiveSlotPacketIn_Enabled()){
      switch (FiveSlotPacketIn_Value()->vcc_unionTag){
        case TFiveSlotBasebandPacket_DH5Packet_tag:
              pdh5=&(FiveSlotPacketIn_Value()->vcc_union.DH5Packet);
      L_CH = pdh5->payload.L_CH;
              TYPE = pdh5->header.TYPE;
          tempCon = llist_lookup_amaddr(pdh5->header.AM_ADDR);
              switch (TYPE){
                case hDM5:
                  packetSize = 226;
                  break;
                case hDH5:
                  packetSize = 341;
                      break;
              }
              break;
      }
}

if (L_CH==LM){
    debug_memcpy(&pdu,pdh1->payload.payloadBody,pdh1->payload.LENGTH);
    switch (pdu.opcode){
          case opLMP_accepted:
                if (pdu.body[0]==tempCon->LastLMPOp){
                  switch(pdu.body[0]){

                    case opLMP_host_connection_request:
                            pdu.opcode=tempCon->LastLMPOp=opLMP_setup_complete;
                                length=1;
                            receiverAMADDR = tempCon->AMADDR;
                                break;
                    case opLMP_SCO_link_req:
                          hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
                          hcievent=&(hcipacket.vcc_union.Event);
                          hcievent->vcc_unionTag=HCI_EventPacket_Connection_Complete_tag;
                          hcievent->vcc_union.Connection_Complete.Status = 0x00;
```

```
                        hcievent->vcc_union.Connection_Complete.Encryption_Mode = 0x00;
                        hcievent->vcc_union.Connection_Complete.Connection_Handle = tempCon->Connection_Handle;
                        hcievent->vcc_union.Connection_Complete.Link_Type = 0x00;
                        HCIPacket_out_Post(&hcipacket);
                            break;
                       }
          }
            break;
case opLMP_auto_rate:
            break;
case opLMP_clkoffset_req:
            break;
case opLMP_clkoffset_res:
            break;
case opLMP_detach:
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Disconnection_Complete_tag;
            hcievent->vcc_union.Disconnection_Complete.Status = 0x00; /* Disconnection has occured */
            hcievent->vcc_union.Disconnection_Complete.Connection_Handle = tempCon->Connection_Handle;
            hcievent->vcc_union.Disconnection_Complete.Reason = pdu.body[0];
            HCIPacket_out_Post(&hcipacket);
            disconnect = tempCon->Connection_Handle;
            break;
case opLMP_features_req:
            pdu.body[0]&=feat_5SlotPackets;
            pdu.body[1]&=0;
            pdu.body[2]&=0;
            pdu.opcode=opLMP_features_res;
            length=3;
            tempCon->features = pdu.body[0]+pdu.body[1]^8+pdu.body[2]^16;
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Read_Remote_Supported_Features_Complete_tag;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.Connection_Handle = tempCon->Connection_Handle;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.Status = 0x00;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.LMP_Features = tempCon->features;
            HCIPacket_out_Post(&hcipacket);
            receiverAMADDR = tempCon->AMADDR;
            choosePacketType(tempCon);
            break;
case opLMP_features_res:
            tempCon->features = pdu.body[0]+pdu.body[1]^8+pdu.body[2]^16;
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Read_Remote_Supported_Features_Complete_tag;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.Connection_Handle = tempCon->Connection_Handle;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.Status = 0x00;
            hcievent->vcc_union.Read_Remote_Supported_Features_Complete.LMP_Features = tempCon->features;
            HCIPacket_out_Post(&hcipacket);
            choosePacketType(tempCon);
            break;
case opLMP_hold:
```

```
        break;
case opLMP_hold_req:
        break;
case opLMP_host_connection_request:
        hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
        hcievent=&(hcipacket.vcc_union.Event);
        hcievent->vcc_unionTag=HCI_EventPacket_Connection_Request_tag;
        hcievent->vcc_union.Connection_Request.BD_ADDR = tempCon->BDADDR;
        hcievent->vcc_union.Connection_Request.Class_of_Device = 0x00;
        hcievent->vcc_union.Connection_Request.Link_Type = 0x01;
        HCIPacket_out_Post(&hcipacket);

        break;
case opLMP_max_slot:
        break;
case opLMP_max_slot_req:
        break;
case opLMP_name_req:
        {
                char *cStr = Name_Value();
                int cLen = 16>mystrlen(cStr)-pdu.body[0]?mystrlen(cStr)-pdu.body[0]:16;
                pdu.opcode = opLMP_name_res;
                pdu.body[1] = mystrlen(cStr);
                debug_memcpy(pdu.body+2, cStr+pdu.body[0], cLen);
                length=cLen+2;
                receiverAMADDR=tempCon->AMADDR;
        }
        break;
case opLMP_name_res:
        debug_memcpy(nameBuf+pdu.body[0], pdu.body+2, pdu.body[1]);
        if (pdu.body[1]==16){
                pdu.opcode = opLMP_name_req;
                pdu.body[0]+=16;
                length=1;
        } else {
                /* 0-terminate string */
                nameBuf[pdu.body[0]+pdu.body[1]]=0;
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Remote_Name_Request_Complete_tag;
            hcievent->vcc_union.Remote_Name_Request_Complete.BD_ADDR = myBDADDR;
            hcievent->vcc_union.Remote_Name_Request_Complete.Status = 0x00;
            hcievent->vcc_union.Remote_Name_Request_Complete.Remote_Name = nameBuf;
            HCIPacket_out_Post(&hcipacket);
        }
        break;
case opLMP_not_accepted:
        if (pdu.body[0]==tempCon->LastLMPOp){
          switch(tempCon->LastLMPOp){
          case opLMP_host_connection_request: /* Connection refused  */
            hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
            hcievent=&(hcipacket.vcc_union.Event);
            hcievent->vcc_unionTag=HCI_EventPacket_Connection_Complete_tag;
```

```
            hcievent->vcc_union.Connection_Complete.BD_ADDR = tempCon->BDADDR;
            hcievent->vcc_union.Connection_Complete.Status = 0x01;            /* Connection failed */
            hcievent->vcc_union.Connection_Complete.Encryption_Mode = 0x00;   /* Rest of the parameters */
            hcievent->vcc_union.Connection_Complete.Connection_Handle = 0x00; /* can be ignored */
            hcievent->vcc_union.Connection_Complete.Link_Type = 0x01;
            HCIPacket_out_Post(&hcipacket);
            llist_remove_connection(tempCon->Connection_Handle);
            break;
          }
        }
        break;
case opLMP_page_mode_req:
        break;
case opLMP_page_scan_mode_req:
        break;
case opLMP_SCO_link_req:
        /* Always accept SCO connections */
          pdu.opcode=opLMP_accepted;
          length=1;
          receiverAMADDR=tempCon->AMADDR;
        break;
case opLMP_setup_complete:
        hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
        hcievent=&(hcipacket.vcc_union.Event);
        hcievent->vcc_unionTag=HCI_EventPacket_Connection_Complete_tag;
        hcievent->vcc_union.Connection_Complete.BD_ADDR = tempCon->BDADDR;
        hcievent->vcc_union.Connection_Complete.Link_Type = (tempCon->type==ACL)?0x01:0x00;
        hcievent->vcc_union.Connection_Complete.Status = 0x00; /* Success */
        hcievent->vcc_union.Connection_Complete.Encryption_Mode = tempCon->encrypted;
        hcievent->vcc_union.Connection_Complete.Connection_Handle = tempCon->Connection_Handle;
        HCIPacket_out_Post(&hcipacket);

        if (tempCon->LastLMPOp!=opLMP_setup_complete){
          pdu.opcode=opLMP_setup_complete;
          length=1;
          receiverAMADDR=tempCon->AMADDR;
        }
        break;
case opLMP_slot_offset:
        break;
case opLMP_supervision_timeout:
        break;
case opLMP_switch_req:
        break;
case opLMP_timing_accuracy_req:
        break;
case opLMP_timing_accuracy_res:
        break;

    case opLMP_version_req:
        break;
case opLMP_version_res:
        break;
```

```
/*  The following has to do with encryption - ignore */
      case opLMP_temp_key:
      case opLMP_temp_rand:
      case opLMP_unit_key:
      case opLMP_use_semi_permanent_key:
      case opLMP_sres:
      case opLMP_start_encryption_req:
      case opLMP_stop_encryption_req:
      case opLMP_in_rand:
      case opLMP_encryption_key_size_req:
      case opLMP_encryption_mode_req:
      case opLMP_au_rand:
      case opLMP_comb_key:
                break;

/* Parking is not supported by baseband - ignore */
          case opLMP_unpark_PM_ADDR_req:
          case opLMP_unpark_BD_ADDR_req:
      case opLMP_modify_beacon:
      case opLMP_set_broadcast_scan_window:
          case opLMP_park:
          case opLMP_park_req:
                break;

/* Sniffing is not supported by baseband - ignore */
          case opLMP_unsniff_req:
      case opLMP_sniff:
      case opLMP_sniff_req:
                break;

/*  This has to do with power management - ignore */
      case opLMP_incr_power_req:
      case opLMP_max_power:
      case opLMP_min_power:
      case opLMP_decr_power_req:
                break;

      }

    if (length>0)
      {
                /* As above - in our implementation of DM1 packets we
                   use the DH1 structure and set header.TYPE to DM1
                 */
                send = DM1;
        oneSlot.vcc_unionTag=TOneSlotBasebandPacket_DH1Packet_tag;
                pdh1=&(oneSlot.vcc_union.DH1Packet);
                pdh1->header.AM_ADDR=receiverAMADDR;
                pdh1->header.TYPE=hDM1;
                pdh1->payload.L_CH=LM;
```

```
          pdh1->payload.FLOW=0;
          pdh1->payload.LENGTH=length;
          debug_memcpy(pdh1->payload.payloadBody,&pdu,length);
          length=0;

      }
}


if (L_CH==L2CAPStart) {
  if (InBufferTotalLength==0){
        int templength;
          InBufferFromAMADDR = tempCon->AMADDR;

          debug_memcpy(&(InBufferTotalLength), &(pdh1->payload.payloadBody), 4);
          templength  = (InBufferTotalLength)>packetSize?packetSize:(InBufferTotalLength);
          debug_memcpy(inbuffer, &(pdh1->payload.payloadBody), templength);

          InBufferCount+=templength;
          if (InBufferCount>=InBufferTotalLength){
            hcipacket.vcc_unionTag=HCI_Packet_Data_tag;
            hcipacket.vcc_union.Data.ConnectionHandle=tempCon->Connection_Handle;
            hcipacket.vcc_union.Data.BCFlag=0x00; /* Only used between host and host-controller */
            hcipacket.vcc_union.Data.PBFlag=0x10; /* Always first packet as we can fit all data at once. */
            hcipacket.vcc_union.Data.DataTotalLength=InBufferTotalLength;
            debug_memcpy(&(hcipacket.vcc_union.Data.DataBytes), inbuffer+4, InBufferTotalLength);
            InBufferTotalLength=0;
            InBufferCount=0;
            HCIPacket_out_Post(&hcipacket);
            memset(inbuffer, 0, sizeof(inbuffer));
            memset((char *)&(hcipacket.vcc_union.Data.DataBytes), 0, sizeof(L2CAP_Packet));

          }
  } else {
          /* We are interrupted in a transmission -
             current strategy is to ignore the
             intruder..
          */
  }
}
if (L_CH==L2CAPContinue) {
  int templength = (InBufferTotalLength-InBufferCount)>packetSize?packetSize:(InBufferTotalLength-InBufferCount);
  if (InBufferFromAMADDR==tempCon->AMADDR){
          debug_memcpy(inbuffer+InBufferCount, &(pdh1->payload.payloadBody), templength);
          InBufferCount+=templength;
          if (InBufferTotalLength && InBufferCount>=InBufferTotalLength){
            hcipacket.vcc_unionTag=HCI_Packet_Data_tag;
            hcipacket.vcc_union.Data.ConnectionHandle=tempCon->Connection_Handle;
            hcipacket.vcc_union.Data.BCFlag=0x00; /* Only used between host and host-controller */
            hcipacket.vcc_union.Data.PBFlag=0x10; /* Always first packet as we can fit all data
                                                     at once. */
            hcipacket.vcc_union.Data.DataTotalLength=InBufferTotalLength;
            debug_memcpy(&(hcipacket.vcc_union.Data.DataBytes), inbuffer+4, InBufferTotalLength);
```

```
                InBufferTotalLength=0;
                InBufferCount=0;
                HCIPacket_out_Post(&hcipacket);
                memset(inbuffer, 0, sizeof(inbuffer));
                memset((char *)&(hcipacket.vcc_union.Data.DataBytes), 0, sizeof(L2CAP_Packet));
              } else {
                /* We are interrupted in a transmission -
                   current strategy is to ignore the
                   intruder..
                */
              }
    }
  }


if (BasebandReady_in_Enabled ()){
  basebandReady = vcc_true;
}


if (basebandInquiryComplete&&stateout.State==PAGE){
      StateOut_Post(&stateout);
  basebandReady = vcc_false;
      stateout.State=STANDBY;
}
if (basebandReady&&send){
  basebandReady=vcc_false;
      switch (send){
    case DM1:
        case DH1:
          OneSlotPacketOut_Post(&oneSlot);
      memset((char*)&oneSlot, 0, sizeof(TOneSlotBasebandPacket));
              break;
    case DM5:
        case DH5:
          FiveSlotPacketOut_Post(&fiveSlot);
      memset((char*)&fiveSlot, 0, sizeof(TFiveSlotBasebandPacket));
              break;
  }
  send = 0;
}
if ((basebandReady) &&
    (OutBufferTotalLength>OutBufferCount) &&
    (tempCon = llist_lookup_amaddr(OutBufferReceiverAMADDR))){

  basebandReady=vcc_false;

        vccPrintPdxDebugInfo("%_START_ sending with packettype %d %_END_", tempCon->packetType);

      switch (tempCon->packetType){
        case DM1:
          packetSize = 18;
          length=(OutBufferTotalLength-OutBufferCount)>packetSize?packetSize:(OutBufferTotalLength-OutBufferCount);
```

```
  oneSlot.vcc_unionTag=TOneSlotBasebandPacket_DH1Packet_tag;
  pdh1=&(oneSlot.vcc_union.DH1Packet);
  pdh1->header.AM_ADDR=OutBufferReceiverAMADDR;
  pdh1->header.TYPE=hDM1;
  if (OutBufferCount==0)
    pdh1->payload.L_CH=L2CAPStart;
  else
    pdh1->payload.L_CH=L2CAPContinue;

  pdh1->payload.FLOW=0;
  pdh1->payload.LENGTH=length;

  debug_memcpy(pdh1->payload.payloadBody,outbuffer+OutBufferCount,length);

  OneSlotPacketOut_Post(&oneSlot);
  memset((char*)&oneSlot, 0, sizeof(TOneSlotBasebandPacket));

  break;
case DH1:
  packetSize = 27;
  length=(OutBufferTotalLength-OutBufferCount)>packetSize?packetSize:(OutBufferTotalLength-OutBufferCount);
  oneSlot.vcc_unionTag=TOneSlotBasebandPacket_DH1Packet_tag;
  pdh1=&(oneSlot.vcc_union.DH1Packet);
  pdh1->header.AM_ADDR=OutBufferReceiverAMADDR;
  pdh1->header.TYPE=hDH1;
  if (OutBufferCount==0)
    pdh1->payload.L_CH=L2CAPStart;
  else
    pdh1->payload.L_CH=L2CAPContinue;

  pdh1->payload.FLOW=0;
  pdh1->payload.LENGTH=length;

  debug_memcpy(pdh1->payload.payloadBody,outbuffer+OutBufferCount,length);

  OneSlotPacketOut_Post(&oneSlot);
  memset((char*)&oneSlot, 0, sizeof(TOneSlotBasebandPacket));
      break;
case DM5:
  packetSize = 226;
  length=(OutBufferTotalLength-OutBufferCount)>packetSize?packetSize:(OutBufferTotalLength-OutBufferCount);
  fiveSlot.vcc_unionTag=TFiveSlotBasebandPacket_DH5Packet_tag;
  pdh5=&(fiveSlot.vcc_union.DH5Packet);
  pdh5->header.AM_ADDR=OutBufferReceiverAMADDR;
  pdh5->header.TYPE=hDM5;
  if (OutBufferCount==0)
    pdh5->payload.L_CH=L2CAPStart;
  else
    pdh5->payload.L_CH=L2CAPContinue;

  pdh5->payload.FLOW=0;
  pdh5->payload.LENGTH=length;
```

```
          debug_memcpy(pdh5->payload.payloadBody,outbuffer+OutBufferCount,length);

          FiveSlotPacketOut_Post(&fiveSlot);
          memset((char*)&fiveSlot, 0, sizeof(TFiveSlotBasebandPacket));
          break;
        case DH5:
          packetSize = 341;
          length=(OutBufferTotalLength-OutBufferCount)>packetSize?packetSize:(OutBufferTotalLength-OutBufferCount);
          fiveSlot.vcc_unionTag=TFiveSlotBasebandPacket_DH5Packet_tag;
          pdh5=&(fiveSlot.vcc_union.DH5Packet);
          pdh5->header.AM_ADDR=OutBufferReceiverAMADDR;
          pdh5->header.TYPE=hDH5;
          if (OutBufferCount==0)
            pdh5->payload.L_CH=L2CAPStart;
          else
            pdh5->payload.L_CH=L2CAPContinue;

          pdh5->payload.FLOW=0;
          pdh5->payload.LENGTH=length;

          debug_memcpy(pdh5->payload.payloadBody,outbuffer+OutBufferCount,length);

          FiveSlotPacketOut_Post(&fiveSlot);
          memset((char*)&fiveSlot, 0, sizeof(TFiveSlotBasebandPacket));
          break;
      }
  OutBufferCount += length;
  if (OutBufferTotalLength<=OutBufferCount){
    OutBufferTotalLength=0;
    OutBufferCount=0;
    hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
    hcievent=&(hcipacket.vcc_union.Event);
    hcievent->vcc_unionTag=HCI_EventPacket_Number_of_Completed_Packets_tag;
    hcievent->vcc_union.Number_of_Completed_Packets.Number_of_Handles = 1;
    hcievent->vcc_union.Number_of_Completed_Packets.Connection_Handle = tempCon->Connection_Handle;
    hcievent->vcc_union.Number_of_Completed_Packets.HC_Num_of_Completed_Packets = 1;
    HCIPacket_out_Post(&hcipacket);
        memset(outbuffer, 0, sizeof(outbuffer));
  }
}

if (disconnect&&basebandReady){
  /* get ready for new connections if we are a slave
     (disconnection is not really supported in
     our baseband implementation, so we can't really
     disconnect a master..) */
  if (!Master_Value()){
    stateout.State=PAGESCAN;
    StateOut_Post(&stateout);
  }

  llist_remove_connection(disconnect);
  disconnect = 0;
```

```
  }

  LMPReady_out_Post (0);
}
```

B New Control code

```
#include "white_interface.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define GIAC 0X9E8B33  /*Reserve LAP value for General Inquiry Access Code*/
#define PN1           0xBBCC54FC     /*Because the Pseudo-random noise sequence is too long*/
#define PN2           0x83848D96     /*so I cut it to two part PN1,first 32 bit, PN2 second 32 bit*/
#define G                          "260534236651" /*Generator polynomial*/
#define MAX_LEN       100
#define MAX_SIZE      20


/*The two definition are used for the union type in VCC.*/
#define vcc_union    __union
#define vcc_unionTag __unionTag
#define MAX_SLAVE    7
#define MAX_PICO     10    /*Though there is not restriction in BB, such definition can make it easier to manage.*/
int     maxInquirySlave;



/*Here W_PAGE_SCAN SHOULD BE 8000,be it same to be to slow in this simulation environment, so I use 200.*/
int W_PAGE_SCAN;                        /*From page 100, we know Tw_page_scan is 1.28s around 8000 clk cycle.(each is 3.25us)*/
/*The same situation as before.*/
int W_INQUIRY_SCAN;             /*From page 109, we know Tw_inquiry_scan is 2.56s around 16000 clk cycle.(each is 3.25us)*/


/* Init Function */
TBluetoothDeviceAddress slaveBD_ADDR[MAX_SLAVE];                            /*These two varible are used for master units.*/
int                                             master_estime_clockoffset[MAX_SLAVE];

TBluetoothDeviceAddress masterBD_ADDR;                      /*This varible is used for slave units.*/
TBluetoothDeviceAddress connectSlaveBDADDR;

int                                             slave_estime_clockoffset[MAX_PICO];

/*This variable is used for slave, which is used to record the am_addr assigned by the master.*/
int                                             slave_am_addr[MAX_PICO];

int                                     current_pico;               /*Mark down which piconet were participating.*/
int                     pico_no;


int                     slave_counter;           /*The number of slave units.*/
int                     current_slave;           /*The number of current connecting Slave.*/
TBasebandState                  State;
int                                             N_Inquiry=0;
int                                             N_Page=0;
```

```
int                                    Slave_No;
int                            clkn27;                  /*The variable is used to record the native clock of the bluetooth unit.*/
int                            CLKInquiryTimeout;
TBasebandFHSPacketPayload fhs;
TOneSlotBasebandPacket    oneSlot;
int                       RAND;                  /*The random number is used for inquiry reponse, avoid collision.*/
int                                             inquiry_counter;
int                       clock_offset;
int                                     random_delay;         /* The boolean is used to whether the slave random has been delayed a random time slot.*/

/*This variable is used for master.*/
int                                             am_addr=1;            /* The active member address,0 is reserved for broadcast messages.*/


TOneSlotBasebandPacket    OneSlotBuffer[MAX_SIZE];                                             /*This buffer is used to store the data from Upper layer One Slot packet*/
TFiveSlotBasebandPacket   FiveSlotBuffer[MAX_SIZE];                                            /*This buffer is used to store the data from Upper layer Five slot packet*/
int                                             numInOneSlotBuffer=0;                              /*Counter of the buffer.*/
int                                             numInFiveSlotBuffer=0;
TBasebandInternalState    inStates;


int                       Tsco;
int                       scoCounter=0;
int                                             EncryptSetupCompleted;

/*This operation is use to cut some bits of a binary digit*/
int bit_opp(unsigned long src,int start,int end,int step)
{
  int i,j;
  unsigned long temp,result;
  if (end<start) return -1;
  temp=src>>start;
  result=0;
  for (i=0,j=0;i<end-start+1;i=i+step,j++)
    result=(1<<j)*(((1<<i)&temp)!=0)+result;
  return result;

}

/* The function is used for generating the ACCESS code of the
   bluetooth unit from the BD_ADDR.
*/
void AnyToBinary(char *from, char from_type, char *to)
{
  char *hexstr[16]={"0000","0001","0010","0011","0100","0101","0110","0111",
                    "1000","1001","1010","1011","1100","1101","1110","1111"};
  char *octstr[8]={"000","001","010","011","100","101","110","111"};
  unsigned int i,len=0;
  switch (from_type)
    { case 'h':
    case 'H':
      for (i=0;i<strlen(from);i++)
        {
```

```
        if (from[i]>='0'&&from[i]<='9')
          memcpy(to+len,&hexstr[from[i]-'0'][0],4);
        else if (from[i]>='a'&&from[i]<='f')
          memcpy(to+len,&hexstr[from[i]-'a'+10][0],4);
        else if (from[i]>='A'&&from[i]<='F')
          memcpy(to+len,&hexstr[from[i]-'A'+10][0],4);
        len=len+4;
      }
      break;
  case 'o':
  case 'O':
    for (i=0;i<strlen(from);i++)
      {
        if (from[i]>='0'&&from[i]<='8')
          memcpy(to+len,&octstr[from[i]-'0'][0],3);
        len=len+3;
      }
    break;
  }
  to[len]=0;
}

/*The function is used to compare the to binary numbers*/
int lt(char *from, char *to)
{
  int len,i,borrow=0;
  char src[200],sub[200];
  len=strlen(to)>strlen(from)?strlen(to):strlen(from);
  memset(src,0,200);
  memset(sub,0,200);
  memset(sub,'0',len);
  memset(src,'0',len);
  memcpy(src+len-strlen(from),from,strlen(from));
  memcpy(sub+len-strlen(to),to,strlen(to));
  for (i=len-1;i>=0;i--)
    if (src[i]<sub[i]+borrow) borrow=1;
    else  borrow=0;
  return borrow;
}

/*It's a Modulo 2 division, used in page 146,step3 */
void unsigned_divide(char *dividend, char *divisor, char *quotient, char *remainder)
{
  unsigned int  num_bits,index=0,i,len;
  sprintf(remainder,dividend);
  memset(quotient,0,100);
  num_bits = strlen(dividend);
  len=strlen(divisor);
  while (num_bits>=len) {
    quotient[index]=remainder[0];
    index++;
    if (remainder[0]=='1')
      for (i=0;i<len;i++)
```

```
        remainder[i]=((remainder[i]-'0')^(divisor[i]-'0'))+'0';
    memcpy(remainder,remainder+1,num_bits-1);
    num_bits--;
    remainder[num_bits]=0;
  }
}

/*The function is used to convert string binary to unsigned long hex*/
int BinToHex(char *from, unsigned int *result)
{
  int i,j,counter;
  for (i=0;i<strlen(from);i++)
    if (from[i]!='0')
      break;
  sprintf(from,from+i);
  for (counter=0,i=strlen(from)-1;i>=0;i=i-32,counter++)
    {
      result[counter]=0;
      for (j=0;j<32&&(i-j)>=0;j++)
        result[counter]=(from[i-j]-'0')*(1<<j)+result[counter];
    }
  return counter;
}

/*Convert the common hex to the HEX whose LSB is in the rightmost.*/
void HexToLSBRight(unsigned int *result)
{
  unsigned int temp1,temp2=0;
  int  i;
  temp1=*result;
  for (i=0;i<32;i++)
    {
      if ((temp1>>31)&0x1)
        temp2=temp2+(1<<i);
      temp1=temp1<<1;
    }
  *result=temp2;
}



void Gen_Syncword(unsigned int lap, unsigned int *syncword)
{
  unsigned int sync;
  char     hex[100],dividend[100],c[100],divisor[100],quotient[100];
  int      i;

  if (bit_opp(lap,23,23,1)==0)    /*if BIT LAP23==0,TAIL=001101,a0a1...a23,001101*/
    sync=lap+(0x2c<<24);
  else                            /*if BIT LAP23==1,TAIL=110010*/
    sync=lap+(0x13<<24);
  sync=sync^bit_opp(PN2,34-32,63-32,1); /*Here I minus 32, which is used match the 34-63 bit of PN,page 145,figure13.2*/
  sprintf(hex,"%x",sync);                             /*Convert the code word in string Hex*/
```

```
  AnyToBinary(hex,'h',dividend);        /*Convert the code word from Hex to Binary(in string)*/
  dividend[strlen(dividend)+34]=0;        /*Shift 34 bits to left*/
  memset(dividend+strlen(dividend),'0',34);
  AnyToBinary(G,'o',divisor);                 /*Convert the divisor from Octal to binary mode */

  for (i=0;i<strlen(divisor);i++)
    if (divisor[i]=='1') break;
  sprintf(divisor,divisor+i);           /*Remove the zero in the head*/
  unsigned_divide(dividend,divisor ,quotient,c);
  BinToHex(c,syncword);
  syncword[0]=syncword[0]^PN1;
  syncword[1]=(bit_opp(syncword[1],0,1,1)+(sync<<2))^PN2;

  /*Convert the Sync word from common mode to LSB as rightmost bit*/
  HexToLSBRight(&(syncword[0]));
  HexToLSBRight(&(syncword[1]));
  syncword[1]=syncword[1]&0x3;

}



void poin_entry_Init()
{
  TAMBD_ADDR AMBD;
  /* At first Master do not any information  about other
     bluetooth device,so a Inquiry operation is necessory*/

/*      Outcommented by Morten Bek - trying to do this through LMP..
    if (Master_Value())
            State=INQUIRY;
        else
                State=INQUIRYSCAN;
*/
  if (Master_Value())
    State=STANDBY;
  else
    State=INQUIRYSCAN;

  Slave_No=0;
  clkn27=0;
  slave_counter=0;
  random_delay=0;  /*This is a boolean.*/

  memset(OneSlotBuffer,0,sizeof(TOneSlotBasebandPacket)*MAX_SIZE);
  memset(FiveSlotBuffer,0,sizeof(TFiveSlotBasebandPacket)*MAX_SIZE);

  memset(&inStates,0,sizeof(inStates));
  inStates.state=State;
  inStates.clk27=clkn27;
  inStates.AM_ADDR=0;
  if (!EncryptionEnabled_Value())
    EncryptSetupCompleted=1;
```

```
  else
    EncryptSetupCompleted=0;

  srand( (unsigned)time( NULL )%BD_ADDR_Value()->LAP );
  current_slave=0;
  maxInquirySlave=max_inquiry_slave_Value();
  W_PAGE_SCAN=PageTO_Value();
  W_INQUIRY_SCAN=InquiryTO_Value();
  current_pico=0;
  memset(slave_estime_clockoffset,0,sizeof(slave_estime_clockoffset));

  Tsco=T_sco_Value();
  pico_no=1;
  if  (InquiryPeriod_Value()>0)
    pico_no=InquiryPeriod_Value();

        /* initialize lmps , added by Morten Bek*/
    AMBD.BD_ADDR.LAP = BD_ADDR_Value()->LAP;
    AMBD.BD_ADDR.UAP = BD_ADDR_Value()->UAP;
    AMBD.BD_ADDR.NAP = BD_ADDR_Value()->NAP;

    AMBDOut_Post(&AMBD);
}

void Master_Inquiry()
{

}
/* Run Function */
void poin_entry_Run()
{
  HCI_Packet hcipacket;
  HCI_EventPacket *hcievent;

  TAMBD_ADDR AMBD;
  THopParams hParams;
  int        i;
  char       mytest[200];

  /* Enable state changing from higher
     layer.
         Added by Morten Bek
  */

  if (StateIn_Enabled()){
    State = StateIn_Value()->State;
    if (State==INQUIRY){
      maxInquirySlave=StateIn_Value()->max_inquiry_slave;
      CLKInquiryTimeout = (clkn27 + StateIn_Value()->inquiry_length<<12)%134217728;
    }
    if (State==PAGE){
      int found =0;
      connectSlaveBDADDR = StateIn_Value()->BD_ADDR;
```

```
    for (i=0; i<slave_counter;i++){
      if(connectSlaveBDADDR.LAP==slaveBD_ADDR[i].LAP &&
         connectSlaveBDADDR.UAP==slaveBD_ADDR[i].UAP &&
         connectSlaveBDADDR.NAP==slaveBD_ADDR[i].NAP){
        current_slave = i;
        found = 1;
        break;
      }
    }
    if (!found){
      slaveBD_ADDR[slave_counter]=connectSlaveBDADDR;
      current_slave=slave_counter;
      slave_counter++;
    }
  }

}


if (EncryptReadyIn_Enabled()&&EncryptionEnabled_Value())
  {
    EncryptSetupCompleted=1;
    ReadMessageOut_Post(0);  /*We can continue to receive the packets.*/
  }

if (OneSlotPacketIn_Enabled()&&State==CONNECTION&&EncryptSetupCompleted)
  /*
    OneSlotPacketIn_Value()->vcc_unionTag!=TOneSlotBasebandPacket_FHSPacket_tag&&
    OneSlotPacketIn_Value()->vcc_unionTag!=TOneSlotBasebandPacket_NULLPacket_tag)
  */
  {

    /*be careful, such routing functions are not
      supported by Bluetooth Baseband protocol.*/
    /* Added by Morten Bek: This is a _very_ bad
       hack! Why not use the AM_ADDR?? I thought
       that was what they were for!? */

   /* test fix by Morten Bek - seems to do the job
          (except that everyone gets random NULL-
               packets from somewhere.. Dunno what they
               are for.. They are created in the
               composer if you want to fix it.. At least
               they should contain an AM address so that
               not everybody receives them..)

       */

   if (OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.AM_ADDR==0||Master_Value()){
         ULOneSlotPacketOut_Post(OneSlotPacketIn_Value());
       } else {
         if (OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.AM_ADDR==slave_am_addr[current_pico]){
           ULOneSlotPacketOut_Post(OneSlotPacketIn_Value());
```

```
          } else
            ReadMessageOut_Post(0);
      }
    }


                                                           /*
      if (OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.UAP!=0||
          OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.LAP!=0||
          OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.NAP!=0)
        {
          if (OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.UAP==BD_ADDR_Value()->UAP&&
              OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.LAP==BD_ADDR_Value()->LAP&&
              OneSlotPacketIn_Value()->vcc_union.NULLPacket.header.DestBD_ADDR.NAP==BD_ADDR_Value()->NAP)
            {
              / This packet is what I want. /
              ULOneSlotPacketOut_Post(OneSlotPacketIn_Value());
            }
          else if (Master_Value())
            {
              / OK, A master receive such packet, it will check whether the
                packet is for one of its slave. If so, retranster it . /
              if (numInOneSlotBuffer<MAX_SIZE)
                {
                  ReadMessageOut_Post(0);
                  memcpy(&OneSlotBuffer[numInOneSlotBuffer],OneSlotPacketIn_Value(),sizeof(TOneSlotBasebandPacket));
                  numInOneSlotBuffer++;
                }
            }
          else
            / Not packet I wanted. /
            ReadMessageOut_Post(0);

        }
      else
        ULOneSlotPacketOut_Post(OneSlotPacketIn_Value());
    }
*/
  if (FiveSlotPacketIn_Enabled()&&State==CONNECTION&&EncryptSetupCompleted)
    ULFiveSlotPacketOut_Post(FiveSlotPacketIn_Value());

  if (ComposerReadyIn_Enabled()&&State==CONNECTION)
    ULReadyOut_Post(0);

  if (ULReadMessageIn_Enabled()&&State==CONNECTION)
    ReadMessageOut_Post(0);

  if (ULOneSlotPacketIn_Enabled()&&State==CONNECTION)
    {

      if (numInOneSlotBuffer<MAX_SIZE)
        {
          memcpy(&OneSlotBuffer[numInOneSlotBuffer],ULOneSlotPacketIn_Value(),sizeof(TOneSlotBasebandPacket));
```

```
        numInOneSlotBuffer++;
      }
  }

if (ULFiveSlotPacketIn_Enabled()&&State==CONNECTION)
  {
    if (numInOneSlotBuffer<MAX_SIZE)
      {
        memcpy(&FiveSlotBuffer[numInOneSlotBuffer],ULFiveSlotPacketIn_Value(),sizeof(TFiveSlotBasebandPacket));
        numInFiveSlotBuffer++;
      }
  }


if (Master_Value ())
  {
    /* Master Operation begins here*/

    /* These operation is activated by the clock*/
    if (CLKNIn_Enabled())
      {
        clkn27=CLKNIn_Value();
        inStates.state=State;
        inStates.clk27=clkn27;
        inStates.BD_ADDR.LAP=slaveBD_ADDR[current_slave].LAP;
        inStates.BD_ADDR.UAP=slaveBD_ADDR[current_slave].UAP;
        inStates.BD_ADDR.NAP=slaveBD_ADDR[current_slave].NAP;

        StateOut_Post(&inStates);

        switch (State)
          {
          case INQUIRY:
            if (clkn27>CLKInquiryTimeout){
              /* generate inquiry complete event */
              hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
              hcievent=&(hcipacket.vcc_union.Event);
              hcievent->vcc_unionTag=HCI_EventPacket_Inquiry_Complete_tag;
              hcievent->vcc_union.Inquiry_Complete.Num_Responses = 1;
              hcievent->vcc_union.Inquiry_Complete.Status = 0x00;
              HCIPacketOut_Post(&hcipacket);
              State = STANDBY;
              break;
            }

            /* At first Master do not any information  about other
               bluetooth device,so a Inquiry operation is necessory*/

            hParams.clk=CLKNIn_Value();
            hParams.ulap=GIAC;
            hParams.state=INQUIRY;
            HopParamsOut_Post(&hParams);
            ID_PacketOut_Post(GIAC);
```

```
  ReadMessageOut_Post(0);                                      /* Here I want to wait a FHS back.*/
  break;
case PAGE:
  /*Try connect the first dectected slave, and the one after.*/
  /*First the slave's device addrs is used to determine the page hopping seuqence,page 101*/
  hParams.ulap=slaveBD_ADDR[current_slave].LAP+(slaveBD_ADDR[current_slave].UAP<<24);
  /*For the phoase in the sequence, the master uses an estimate of the slave's clock.
    This estimate can for example be derived from timming information that was exchanged during
    the last encounter with this particular device or from an inquiry procedure.page 101*/
  clkn27=CLKNIn_Value();
  hParams.clk=CLKNIn_Value()+master_estime_clockoffset[current_slave];
  hParams.state=PAGE;
  HopParamsOut_Post(&hParams);
  ID_PacketOut_Post(slaveBD_ADDR[current_slave].LAP);
  break;
case MASTERRESPONSE:
  if (CLKNIn_Value()%2==1) return ;
  hParams.ulap=slaveBD_ADDR[current_slave].LAP+(slaveBD_ADDR[current_slave].UAP<<24);
  clkn27=CLKNIn_Value();
  hParams.clk=CLKNIn_Value()+master_estime_clockoffset[current_slave];
  hParams.state=MASTERRESPONSE;
  HopParamsOut_Post(&hParams);
  break;
case CONNECTION:
  if (scoCounter==0)
    {
      scoCounter=Tsco;
      TscoOut_Post(clkn27);
    }
  scoCounter--;
  hParams.clk=CLKNIn_Value();
  hParams.ulap=BD_ADDR_Value()->LAP+(BD_ADDR_Value()->UAP<<24);
  hParams.state=CONNECTION;
  HopParamsOut_Post(&hParams);
  if (clkn27%2==0)
    {
      if (numInOneSlotBuffer!=0)
        {
          OneSlotPacketOut_Post(&OneSlotBuffer[0]);
          numInOneSlotBuffer--;
          memcpy(&OneSlotBuffer[0],&OneSlotBuffer[1],numInOneSlotBuffer*sizeof(TOneSlotBasebandPacket));

        }
      if (numInFiveSlotBuffer!=0)
        {
          FiveSlotPacketOut_Post(&FiveSlotBuffer[0]);
          numInFiveSlotBuffer--;
          memcpy(&FiveSlotBuffer[0],&FiveSlotBuffer[1],numInFiveSlotBuffer*sizeof(TFiveSlotBasebandPacket));
        }
    }
case STANDBY:
  break;
```

```
        }
      }



    if (ID_PacketIn_Enabled()) {
      switch (ID_PacketIn_Value()) {
      case SLAVERESPONSE:
        if (State==PAGE) {
          /*The master waits for a reply from the slave and when this arrives in step
            2, it will enter the master response in step 3. Note that during the initial message
            exchange, all parameters are derived from the slaves BD_ADDR, and
            that only the page hopping and page_response hopping sequences are used.
          */
          State=MASTERRESPONSE;

/* Inform higher layer of assigned
   am<->bd address mapping.
       Added by Morten Bek
*/
          AMBD.AM_ADDR = am_addr;
          AMBD.BD_ADDR = slaveBD_ADDR[current_slave];
          AMBDOut_Post(&AMBD);


          Gen_Syncword(BD_ADDR_Value()->LAP+(BD_ADDR_Value()->UAP<<24),fhs.parity);
          fhs.LAP=BD_ADDR_Value()->LAP;
          fhs.UNDEF0=0;
          fhs.SR=0;
          fhs.SP=0;
          fhs.UAP=BD_ADDR_Value()->UAP;
          fhs.NAP=BD_ADDR_Value()->NAP;
          fhs.classOfDevice=0;
          fhs.AM_ADDR=am_addr;        /*The usage of AM_ADDR has been show on page.147*/
          am_addr++;
          fhs.CLK27=clkn27;      /*Assign the native clock to the fhs packet.*/
          fhs.pageScanMode=0;   /*From P.58, I will use Mandatory scan mode.*/
          memcpy(&(oneSlot.vcc_union.FHSPacket.payload) , &fhs,sizeof(fhs));
          oneSlot.vcc_unionTag = TOneSlotBasebandPacket_FHSPacket_tag;
          inStates.state=State;
          StateOut_Post(&inStates);
          OneSlotPacketOut_Post(&oneSlot);

        }
            else if (State==MASTERRESPONSE)
              {
                /*
                  if (current_slave>=slave_counter-1)
                  {
                */
                State=CONNECTION;
                inStates.state=State;
                StateOut_Post(&inStates);
```

```
                  ULReadyOut_Post((vcc_trigger)0);
/* Following removed by Morten Bek.
   Should not be done. We require
   explicit connection requests from
   higher layer..
*/
                  /*                                            }
                                                                else
                                                                {
                  */
                  /*OK, continue to page next Bluetooth slave.*/
                  /*current_slave++;
                    State=PAGE;
                    inStates.state=State;
                    StateOut_Post(&inStates);
                    }
                  */
                }

              break;
            default:
              break;
            }
        }

      /* These operation is activated by the OneSlotPacketIn.*/
      if (OneSlotPacketIn_Enabled())
        {
          switch (State)
            {
            case INQUIRY:
              /*The master is inquirying other devices,Now it gets a FHS packet back.*/
              /*OK!Found another bluetooth slave, store the Accesscode.*/
              if (slave_counter<maxInquirySlave){
                slaveBD_ADDR[slave_counter].LAP=OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.LAP;
                slaveBD_ADDR[slave_counter].UAP=OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.UAP;
                slaveBD_ADDR[slave_counter].NAP=OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.NAP;
                master_estime_clockoffset[slave_counter]=clkn27-OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.CLK27;
                slave_counter++;

                /* generate inquiry result event */
                hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
                hcievent=&(hcipacket.vcc_union.Event);
                hcievent->vcc_unionTag=HCI_EventPacket_Inquiry_Result_tag;
                hcievent->vcc_union.Inquiry_Result.Num_Responses = 1;
                hcievent->vcc_union.Inquiry_Result.BD_ADDR.LAP = slaveBD_ADDR[slave_counter-1].LAP;
                hcievent->vcc_union.Inquiry_Result.BD_ADDR.UAP = slaveBD_ADDR[slave_counter-1].UAP;
                hcievent->vcc_union.Inquiry_Result.BD_ADDR.NAP = slaveBD_ADDR[slave_counter-1].NAP;
                hcievent->vcc_union.Inquiry_Result.Clock_Offset = master_estime_clockoffset[slave_counter-1];
                HCIPacketOut_Post(&hcipacket);
              }
              if (slave_counter==maxInquirySlave){
                /* generate inquiry complete event */
```

```
          hcipacket.vcc_unionTag=HCI_Packet_Event_tag;
          hcievent=&(hcipacket.vcc_union.Event);
          hcievent->vcc_unionTag=HCI_EventPacket_Inquiry_Complete_tag;
          hcievent->vcc_union.Inquiry_Complete.Num_Responses = 1;
          hcievent->vcc_union.Inquiry_Complete.Status = 0x00;
          HCIPacketOut_Post(&hcipacket);
          State = STANDBY;
        }
        break;
      }
    }
  }
else
  {
    /*Slave operations begin here.*/

    if (CLKNIn_Enabled())
      {
        if (State!=CONNECTION)
          {
            if (State!=PAGESCAN)
              N_Page++;
            else if (State!=INQUIRYSCAN)
              N_Inquiry++;
            else
              {
                N_Page++;
                N_Inquiry++;
              }
          }

        if (State!=INQUIRYRESPONSE&&State!=SLAVERESPONSE)
          {
            if (N_Page>=W_PAGE_SCAN)
              {
                State=PAGESCAN;
                N_Page=0;

                /* current_pico=0;*/
              }
            else if (N_Inquiry==W_INQUIRY_SCAN)
              {
                State=INQUIRYSCAN;
                N_Inquiry=0;
              }
          }


        clkn27=CLKNIn_Value();
        inStates.state=State;
        if (State==CONNECTION)
          {
            /*
```

```
        if (Sniff_Max_Interval_Value()!=0)
        {
        for (i=0;i<pico_no;i++)
        inStates.clk27=clkn27+slave_estime_clockoffset[current_pico];

        }
        }
        else

        {
      */
      if ((clkn27%pico_no)==current_pico)
        inStates.clk27=clkn27+slave_estime_clockoffset[current_pico];

      /*
        }
      */
    }
  else
    inStates.clk27=clkn27;

  StateOut_Post(&inStates);

  switch (State)
    {
    case INQUIRYSCAN:
      /* Slave Change the frequency in INQUIRYSCAN state*/
      hParams.clk=CLKNIn_Value();
      hParams.ulap=GIAC;
      hParams.state=State;
      HopParamsOut_Post(&hParams);
      break;
    case STANDBY:
      if (inquiry_counter<RAND)
        {
          inquiry_counter++;
        }
      else
        State=INQUIRYRESPONSE;
                       /*In order to avoid collisions between units that do wake up
                         in the same inquiry hop channel simultaneously, the following
                         protocol in the slaves inquiry response is used. If the slave
                         receives an inquiry message, it generates a random number RAND
                         between 0 and 1023. In addition, it freezes the current input
                         value (phase) to the hop selection scheme,*/
      break;
    case PAGESCAN:
/* Slave Change the frequency in INQUIRYSCAN state*/
      hParams.clk=CLKNIn_Value();
      hParams.ulap=BD_ADDR_Value()->LAP+(BD_ADDR_Value()->UAP<<24);
      hParams.state=State;
      HopParamsOut_Post(&hParams);
      break;
```

```
      case SLAVERESPONSE:
        if (CLKNIn_Value()%2==1) return ;
        hParams.clk=CLKNIn_Value();
        hParams.ulap=BD_ADDR_Value()->LAP+(BD_ADDR_Value()->UAP<<24);
        hParams.state=SLAVERESPONSE;
        HopParamsOut_Post(&hParams);
        break;
      case CONNECTION:
        if (scoCounter==0)
          {
            scoCounter=Tsco;
            TscoOut_Post(inStates.clk27);
          }
        scoCounter--;

        hParams.clk=inStates.clk27;
        hParams.ulap=masterBD_ADDR.LAP+(masterBD_ADDR.UAP<<24);
        hParams.state=CONNECTION;
        HopParamsOut_Post(&hParams);

        if (inStates.clk27%2==1&&((((inStates.clk27-1)/2)%(Tsco/2))==(slave_am_addr[current_pico])-1))
          {
            if (numInOneSlotBuffer!=0)
              {
                OneSlotPacketOut_Post(&OneSlotBuffer[0]);
                numInOneSlotBuffer--;
                memcpy(&OneSlotBuffer[0],&OneSlotBuffer[1],numInOneSlotBuffer*sizeof(TOneSlotBasebandPacket));
              }
            if (numInFiveSlotBuffer!=0)
              {
                FiveSlotPacketOut_Post(&FiveSlotBuffer[0]);
                numInFiveSlotBuffer--;
                /* memmove should be used if areas overlap!!
                   memcpy on overlapping areas has undefined behavior */
                memcpy(&FiveSlotBuffer[0],&FiveSlotBuffer[1],numInFiveSlotBuffer*sizeof(TFiveSlotBasebandPacket));
              }
          }
        break;
      }
  }

if (OneSlotPacketIn_Enabled())
  {
    if (State==SLAVERESPONSE)
      {
        ID_PacketOut_Post(BD_ADDR_Value()->LAP);
        /*Response to the FHS packet from Master first.*/
        masterBD_ADDR.LAP=
          OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.LAP;
        masterBD_ADDR.UAP=
          OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.UAP;
        masterBD_ADDR.NAP=
          OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.NAP;
```

```
            slave_estime_clockoffset[current_pico]=
              clkn27-OneSlotPacketIn_Value()->
              vcc_union.FHSPacket.payload.CLK27;
            slave_am_addr[current_pico]=
              OneSlotPacketIn_Value()->vcc_union.FHSPacket.payload.AM_ADDR;


            if (current_pico>=pico_no-1)
              {

                State=CONNECTION;
                inStates.state=State;
                inStates.AM_ADDR=slave_am_addr[current_pico];
                inStates.BD_ADDR.LAP= masterBD_ADDR.LAP;
                inStates.BD_ADDR.UAP= masterBD_ADDR.UAP;
                inStates.BD_ADDR.NAP= masterBD_ADDR.NAP;
                StateOut_Post(&inStates);

/* Inform higher layer of assigned
   am<->bd address mapping.
       Added by Morten Bek
*/
                AMBD.AM_ADDR = slave_am_addr[current_pico];
                AMBD.BD_ADDR = masterBD_ADDR;
                AMBDOut_Post(&AMBD);
                ULReadyOut_Post((vcc_trigger)0);

              }
            else
              {
                State=PAGESCAN;
                inStates.state=State;
                StateOut_Post(&inStates);
                current_pico++;
              }

            /*
              OneSlotPacketOut_Post(&oneSlot);
            */


          }
      }


    if (ID_PacketIn_Enabled())
      {
        switch (ID_PacketIn_Value())
          {
          case INQUIRY:
            /*Receive the Inquiry request from master, The slave will sent a FHS packet to response*/
            /*The procedure of Inquriy Response is described in page.111*/
            if (!random_delay)
```

```
  {
    /*In order to avoid collisions between units that do wake up
      in the same inquiry hop channel simultaneously, the following
      protocol in the slaves inquiry response is used. If the slave
      receives an inquiry message, it generates a random number RAND
      between 0 and 1023. In addition, it freezes the current input
      value (phase) to the hop selection scheme,*/

    RAND=rand()%128;        /* The correct value should be 1024,but it sames to slow in the demo.
                               So I change it to 128.*/
    inquiry_counter=0;
    random_delay=1;
    State=STANDBY;
    return;
  }

if (State==INQUIRYRESPONSE)
  {
    Gen_Syncword(BD_ADDR_Value()->LAP+BD_ADDR_Value()->UAP,fhs.parity);
    fhs.LAP=BD_ADDR_Value()->LAP;
    fhs.UNDEF0=0;
    fhs.SR=0;
    fhs.SP=0;
    fhs.UAP=BD_ADDR_Value()->UAP;
    fhs.NAP=BD_ADDR_Value()->NAP;
    fhs.classOfDevice=0;
    fhs.AM_ADDR=0;        /*The usage of AM_ADDR has been show on page.147*/
    fhs.CLK27=clkn27;        /*Assign the native clock to the fhs packet.*/
    fhs.pageScanMode=0;   /*From P.58, I will use Mandatory scan mode.*/
    memcpy(&(oneSlot.vcc_union.FHSPacket.payload) , &fhs,sizeof(fhs));
    oneSlot.vcc_unionTag = TOneSlotBasebandPacket_FHSPacket_tag;


                  /*
                    current_pico++;

                    if (current_pico>=pico_no)
                    {
                  */


    N_Page= W_PAGE_SCAN-1;  /*Get a Inquiry, we can now change to page.*/
    current_pico=0;

                  /*
                    }
                  */


                  /*
                    random_delay=0;

                    StateOut_Post(&inStates);
```

```
                                OneSlotPacketOut_Post(&oneSlot);
                                State=INQUIRYSCAN;
                              */
                  inStates.state=State;
                  random_delay=0;
                  StateOut_Post(&inStates);
                  OneSlotPacketOut_Post(&oneSlot);
                  State=PAGESCAN;


                }
              break;
            case PAGE:

              /*On recognizing its device access code, the slave enters the slave response in step 2. page 104*/
              State=SLAVERESPONSE;
              inStates.state=State;
              StateOut_Post(&inStates);
              ID_PacketOut_Post(BD_ADDR_Value()->LAP);
              ReadMessageOut_Post(0);
              /*Here I want to wait a FHS back.*/
              break;

            }   /*switch*/
        }    /* if (ID_PacketIn_Enabled()) */
    }

}
```

C New Transmit code

D New Receive code

E My own (SDL) version of Canny

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <SDL.h>
#include <math.h>
//#include <SDL_thread.h>
//gcc canny.c -I/usr/local/include/SDL -lpthread -lSDL -lm

// Good values:
//   low threshold:  0.1 (or 0.0)
//   high threshold 0.95
//   Gaussian filter size: <1.4 (1.0 seems fine..)

#define WIDTH 320
#define HEIGHT 200

#define EDGE 255
#define POSSIBLE_EDGE 100
#define NOEDGE 0

struct vector{
  int up, down, left, right;
} vectImg[WIDTH][HEIGHT];


SDL_Color mycolors[256];
Uint8 pixbuf[WIDTH][HEIGHT];
Uint8 smoothedImg[WIDTH][HEIGHT];
int deltaX[WIDTH][HEIGHT], deltaY[WIDTH][HEIGHT];
Uint8 magImg[WIDTH][HEIGHT];
Uint8 result[WIDTH][HEIGHT];


void SuppressNonMax() {
  int r, c, i;         /* for indexing */
  short z1,z2;
  short gx,gy, m00;
  float mag1, mag2, xperp,yperp;

  /* Zero the edges of the result image. */
  for(i=0; i<WIDTH; i++){
    result[i][0] = result[i][HEIGHT-1] = 0;
  }

  for(i=0; i<HEIGHT; i++){
    result[0][i] = result[WIDTH-1][i] = 0;
  }

  /* Suppress non-maximum points. */
  for(r=1; r<HEIGHT-2; r++){
    for(c=1; c<WIDTH-2; c++){
```

```
m00 = magImg[c][r];

if(m00 == 0){
  result[c][r] = NOEDGE;
  gx = gy = 0;
}
else {
  xperp = -(gx = deltaX[c][r])/((float)m00);
  yperp = (gy = deltaY[c][r])/((float)m00);
}

if(gx >= 0){
  if(gy >= 0){
    if (gx >= gy){
      /* 111 */
      /* Left point */

      z1 = magImg[c-1][r];
      z2 = magImg[c-1][r-1];
      mag1 = (m00 - z1)*xperp + (z2 - z1)*yperp;
      /* Right point */

      z1 = magImg[c+1][r];
      z2 = magImg[c+1][r+1];

      mag2 = (m00 - z1)*xperp + (z2 - z1)*yperp;
    } else {
      /* 110 */
      /* Left point */

      z1 = magImg[c][r-1];
      z2 = magImg[c-1][r-1];

      mag1 = (z1 - z2)*xperp + (z1 - m00)*yperp;

      /* Right point */

      z1 = magImg[c][r+1];
      z2 = magImg[c+1][r+1];

      mag2 = (z1 - z2)*xperp + (z1 - m00)*yperp;
    }
  } else {
    if (gx >= -gy) {
      /* 101 */
      /* Left point */

      z1 = magImg[c-1][r];
      z2 = magImg[c-1][r+1];

      mag1 = (m00 - z1)*xperp + (z1 - z2)*yperp;

      /* Right point */
```

```
      z1 = magImg[c+1][r];
      z2 = magImg[c+1][r-1];

      mag2 = (m00 - z1)*xperp + (z1 - z2)*yperp;
    } else {
      /* 100 */
      /* Left point */

      z1 = magImg[c][r+1];
      z2 = magImg[c-1][r+1];

      mag1 = (z1 - z2)*xperp + (m00 - z1)*yperp;

      /* Right point */

      z1 = magImg[c][r-1];
      z2 = magImg[c+1][r-1];

      mag2 = (z1 - z2)*xperp  + (m00 - z1)*yperp;
    }
  }
} else {
  if ((gy = deltaY[c][r]) >= 0) {
    if (-gx >= gy) {
      /* 011 */
      /* Left point */

      z1 = magImg[c+1][r];
      z2 = magImg[c+1][r-1];

      mag1 = (z1 - m00)*xperp + (z2 - z1)*yperp;

      /* Right point */

      z1 = magImg[c-1][r];
      z2 = magImg[c-1][r+1];

      mag2 = (z1 - m00)*xperp + (z2 - z1)*yperp;
    } else {
      /* 010 */
      /* Left point */

      z1 = magImg[c][r-1];
      z2 = magImg[c+1][r-1];

      mag1 = (z2 - z1)*xperp + (z1 - m00)*yperp;

      /* Right point */
      z1 = magImg[c][r+1];
      z2 = magImg[c-1][r+1];

      mag2 = (z2 - z1)*xperp + (z1 - m00)*yperp;
    }
```

```
        } else {
          if (-gx > -gy) {
            /* 001 */
            /* Left point */
            z1 = magImg[c+1][r];
            z2 = magImg[c+1][r+1];

            mag1 = (z1 - m00)*xperp + (z1 - z2)*yperp;

            /* Right point */
            z1 = magImg[c-1][r];
            z2 = magImg[c-1][r-1];

            mag2 = (z1 - m00)*xperp + (z1 - z2)*yperp;
          } else {
            /* 000 */
            /* Left point */

            z1 = magImg[c][r+1];
            z2 = magImg[c+1][r+1];

            mag1 = (z2 - z1)*xperp + (m00 - z1)*yperp;

                /* Right point */

            z1 = magImg[c][r-1];
            z2 = magImg[c-1][r-1];
            mag2 = (z2 - z1)*xperp + (m00 - z1)*yperp;
          }
        }
      }



      /* Now determine if the current point is a maximum point */

      if ((mag1 > 0.0) || (mag2 > 0.0)){
        result[c][r] = (Uint8) NOEDGE;
        //          *resultptr = (unsigned char) NOEDGE;
      } else {
        if (mag2 == 0.0)
          result[c][r] = (Uint8) NOEDGE;
        //*resultptr = (unsigned char) NOEDGE;
        else
          result[c][r] = (Uint8) POSSIBLE_EDGE;
        //            *resultptr = (unsigned char) POSSIBLE_EDGE;
      }
    }
  }
}
```

```
void follow_edges(int c, int r, int lowval)
{
  //   short *tempmag;
  //   unsigned char *tempresult;
   int i, tempc, tempr;
   int x[8] = {1,1,0,-1,-1,-1,0,1},
       y[8] = {0,1,1,1,0,-1,-1,-1};

   for(i=0;i<8;i++){
     tempr = r+y[i];
     tempc = c+x[i];
     //      tempresult = result - y[i]*cols + x[i];
     //      tempmag = mag - y[i]*cols + x[i];
     if ((result[tempc][tempr] == POSSIBLE_EDGE) && (magImg[tempc][tempr] > lowval)){
       result[tempc][tempr] = EDGE;
       follow_edges(tempc, tempr, lowval);
     }
     //      if((*tempresult == POSSIBLE_EDGE) && (*tempmag > lowval)){
     //         *tempresult = (unsigned char) EDGE;
     //         follow_edges(tempresult,tempmag, lowval, cols);
     //      }
   }
}



void Hysteresis(float tlow, float thi)
{
  int r, c, pos, numedges;
  int highcount, lowthreshold, highthreshold, hist[256];
  short maxMag;

/* Initialize the edge map to possible edges everywhere the non-maximal
   suppression suggested there could be an edge except for the border. */

  //   for(r=0,pos=0;r<HEIGHT;r++)
  //     for(c=0;c<WIDTH;c++)
  //       result[c][r] = (edgeImg[c][r] == POSSIBLE_EDGE)? POSSIBLE_EDGE : NOEDGE;
   /* initialize border */
   for(r=0;r<HEIGHT;r++)
     result[0][r] = result[WIDTH-1][r] = NOEDGE;
   for(c=0;c<WIDTH;c++)
     result[c][0] = result[c][HEIGHT-1] = NOEDGE;

   /* Compute the histogram of the magnitude image. Then use the histogram to
      compute hysteresis thresholds. */
   for(r=0;r<256;r++) hist[r] = 0;
   for(r=0;r<HEIGHT;r++)
     for(c=0;c<WIDTH;c++)
       if(result[c][r] == POSSIBLE_EDGE) hist[magImg[c][r]]++;

   /* Compute the number of pixels that passed the nonmaximal suppression. */
   for(r=1,numedges=0;r<256;r++){
```

```
    if(hist[r] != 0) maxMag = r;
    numedges += hist[r];
  }

  highcount = (int)(numedges * thi + 0.5);

  /* Compute the high threshold value as the (100 * thi) percentage point
     in the magnitude of the gradient histogram of all the pixels that passes
     non-maximal suppression. Then calculate the low threshold as a fraction
     of the computed high threshold value. */

  r = 1;
  numedges = hist[1];
  while((r<(maxMag-1)) && (numedges < highcount)){
    r++;
    numedges += hist[r];
  }
  highthreshold = r;
  lowthreshold = (int)(highthreshold * tlow + 0.5);

  /* looking for pixels above the highthreshold to locate edges and
     then calls follow_edges to continue the edge. */

  for(r=0;r<HEIGHT;r++){
    for(c=0;c<WIDTH;c++){
      if((result[c][r] == POSSIBLE_EDGE) &&
         (magImg[c][r] >= highthreshold)){
        result[c][r] = EDGE;
        follow_edges(c,r,lowthreshold);
      }
    }
  }

/* Set all the remaining possible edges to non-edges.*/
  for(r=0;r<HEIGHT;r++)
    for(c=0;c<WIDTH;c++)
      if(result[c][r] != EDGE)
        result[c][r] = NOEDGE;
}



void ComputeGradient() {
  int r, s, sq1, sq2;
  /* Compute the x-derivative. */

  for(r=0;r<HEIGHT;r++){
     for(s=0;s<(WIDTH-1);s++){
       deltaX[s][r] =
         //      (smoothedImg[s+1][r] >= smoothedImg[s][r]) ?
         smoothedImg[s+1][r] - smoothedImg[s][r];
         //      smoothedImg[s][r] - smoothedImg[s+1][r];
```

```
      }
   }

   /* Compute the y-derivative. */
   for(r=0;r<(HEIGHT-1);r++){
      for(s=0;s<WIDTH;s++){
        deltaY[s][r] =
          //      (smoothedImg[s][r+1] >= smoothedImg[s][r]) ?
          smoothedImg[s][r+1] - smoothedImg[s][r];
          //smoothedImg[s][r] - smoothedImg[s][r+1];

      }
   }

   /* Compute the magnitude */

   for(r=0;r<HEIGHT;r++){
     for(s=0;s<WIDTH;s++){
       sq1 = deltaX[s][r] * deltaX[s][r];
       sq2 = deltaY[s][r] * deltaY[s][r];

       magImg[s][r] = (Uint8)(0.5 + sqrt((float)sq1 + (float)sq2));
     }
   }
}




void MakeKernel(double *kernel, double sigma, int windowSize)
{
  int i, center = windowSize/2;
  double normfactor = 0.0;

  for (i = 0; i < windowSize; i++) {
    kernel[i] = exp(-0.5*(i-center)*(i-center)/(sigma*sigma));
    normfactor += kernel[i];
  }
  /* normalize */
  for (i = 0; i < windowSize; i++)
    kernel[i] /= normfactor;
}


void
GaussSmooth(double sigma){
  int row, col, center, windex;
  int numRows=HEIGHT;
  int numCols=WIDTH;
  int windowSize;
  double convol, normfactor;
  double kernel[20];
  double *imagetemp;
```

```
  windowSize = 1 + 2 * ceil(2.5 * sigma);  /* odd number */
  windowSize = 5;
  MakeKernel(kernel, sigma, windowSize);
  center = windowSize/2;

  imagetemp = (double *) malloc(numRows * numCols * sizeof(double));

  /* smoothing in x-direction */
  for (row = 0; row < numRows; row++) {
    for (col = 0; col < numCols; col++) {
      convol = 0.0;
      normfactor = 0.0;
      for (windex = -center; windex <= center; windex++) {
        if ((col+windex >= 0) && (col+windex < numCols)) {
          convol += pixbuf[col+windex][row]
                    * kernel[center+windex];
          normfactor += kernel[center+windex];
        }
      }
      imagetemp[row * numCols + col] = convol/normfactor;
    }
  }
  /* smoothing in y-direction */
  for (col = 0; col < numCols; col++) {
    for (row = 0; row < numRows; row++) {
      convol = 0.0;
      normfactor = 0.0;
      for (windex = -center; windex <= center; windex++) {
        if ((row+windex >= 0) && (row+windex < numRows)) {
          convol += imagetemp[(row+windex) * numCols + col]
                    * kernel[center+windex];
          normfactor += kernel[center+windex];
        }
      }
      smoothedImg[col][row]=(Uint8) (convol/normfactor);
    }
  }

  free(imagetemp);

}



void follow_edges2(int c, int r, int count, int *minx, int *maxx, int *miny, int *maxy, int *maxcount, int *maxcx, int *maxcy)
{
  int i, tempc, tempr; //, numfound=0;
/*    int x[8] = {1,1,0,-1,-1,-1,0,1}; */
/*    int y[8] = {0,1,1,1,0,-1,-1,-1}; */
  int x[8] = {1,0,-1,0,-1,-1, 1, 1};
  int y[8] = {0,1,0,-1,-1, 1, 1,-1};
  //int x[4] = {1,0,-1,0};
```

```
  //int y[4] = {0,1,0,-1};

  result[c][r] = POSSIBLE_EDGE;
  if (count>16000){
    printf("Stack about to overflow!!\n");
    return;
  }
  if (c>*maxx)
    *maxx=c;
  if (c<*minx)
    *minx=c;
  if (r>*maxy)
    *maxy=r;
  if (r<*miny)
    *miny=r;
  if (count>*maxcount){
    *maxcount = count;
    *maxcx = c;
    *maxcy = r;
  }

  //  printf("Count: %i, row: %i, col: %i\n", count, r, c);
  for(i=0;i<8;i++){
    tempr = r+y[i];
    tempc = c+x[i];
    if (tempr>=0 && tempr<HEIGHT && tempc>=0 && tempc<WIDTH){
      if (result[tempc][tempr] == EDGE){
        //      numfound++;
        follow_edges2(tempc, tempr, count+1, minx, maxx, miny, maxy, maxcount, maxcx, maxcy);
      }
      /*        if (result[tempc][tempr] == POSSIBLE_EDGE){
                numfound++;
        }
      */
    }
   }
  //  if(numfound<1){
  //   printf("Fandt hoejst een paa: %i, %i\n", c,r);
    //    result[c][r]=254;
  //  }
}

void count_edges(int x, int y, int count, int *maxcount, int *maxcx, int *maxcy)
{
  int i, tempx, tempy;
  int dx[8] = {1,0,-1,0,-1,-1, 1, 1};
  int dy[8] = {0,1,0,-1,-1, 1, 1,-1};

  result[x][y] = 254;
  if (count>16000){
    printf("Stack about to overflow!!\n");
    return;
  }
```

```
  if (count>*maxcount){
    *maxcount = count;
    *maxcx = x;
    *maxcy = y;
  }

  //  printf("Count: %i, row: %i, col: %i\n", count, r, c);
  for(i=0;i<8;i++){
    tempy = y+dy[i];
    tempx = x+dx[i];
    if (tempy>=0 && tempy<HEIGHT && tempx>=0 && tempx<WIDTH){
      if (result[tempx][tempy] == POSSIBLE_EDGE){
        count_edges(tempx, tempy, count+1, maxcount, maxcx, maxcy);
      }
    }
   }
}

void vectorize(int x, int y, int maxx, int maxy, int *gotit, int count, int direction, int prevx, int prevy)
{
  int i, tempx, tempy;
  int dx[8] = {1,0,-1,0,-1,-1, 1, 1};
  int dy[8] = {0,1,0,-1,-1, 1, 1,-1};

  result[x][y]=253;

  if (count>16000){
    printf("Stack about to overflow!!\n");
    return;
  }

  if(x==maxx && y==maxy)
    *gotit=1;


  for(i=0;i<8;i++){
    tempy = y+dy[i];
    tempx = x+dx[i];
    if ((tempy>=0 && tempy<HEIGHT && tempx>=0 && tempx<WIDTH) &&
        (result[tempx][tempy] == 254) &&
        (!*gotit)){
      vectorize(tempx, tempy, maxx, maxy, gotit, count+1, (dy[i]==0)?dx[i]*2:dy[i], x, y);
    }
  }

  if(*gotit){
    result[x][y]=90;
    switch(direction){
    case(-2):
      vectImg[x][y].left++;
      vectImg[prevx][prevy].right++;
      break;
```

```
    case(2):
      vectImg[x][y].right++;
      vectImg[prevx][prevy].left++;
      break;
    case(-1):
      vectImg[x][y].down++;
      vectImg[prevx][prevy].up++;
      break;
    case(1):
      vectImg[x][y].up++;
      vectImg[prevx][prevy].down++;
      break;

    }
  }

  //  *prevx=x;
  //  *prevy=y;
}


void floodfill(){
  int i, j, count = 0;
  int gotit = 0;
  int minx, maxx, miny, maxy, maxcx, maxcy, maxcount = 0, mincx, mincy;
  int prevx, prevy;
  int numpixls, up, down,k,l;
  for (i=0; i<WIDTH; i++){
    for (j=0; j<HEIGHT; j++){
      vectImg[i][j].up =  vectImg[i][j].down =  vectImg[i][j].left =  vectImg[i][j].right = 0;
    }
  }
  for (i=0; i<WIDTH; i++){
    for (j=0; j<HEIGHT; j++){
      if (result[i][j]==EDGE){
        count++;
        printf("Object no. %i:\n", count);
        minx = maxx = maxcx = i;
        miny = maxy = maxcy = j;
        follow_edges2(i,j,0, &minx, &maxx, &miny, &maxy, &maxcount, &maxcx, &maxcy);
        printf("MinX=%i, MaxX=%i, MinY=%i, MaxY=%i\n",  minx, maxx, miny, maxy);
        printf("Mincount=%i on %i, %i\n", maxcount, maxcx, maxcy);
        maxcount = 0;
        mincx = maxcx;
        mincy = maxcy;
        count_edges(maxcx, maxcy, 0, &maxcount, &maxcx, &maxcy);
        printf("Maxcount=%i on %i, %i\n\n", maxcount, maxcx, maxcy);
        //      if (abs(mincx-maxcx)>1 || abs(mincy-maxcy)>1){
          //      result[mincx][mincy]=154;
          //      result[maxcx][maxcy]=154;
        //  printf("Der er et hul!\n");
        //}
        gotit=0;
```

```
        vectorize(mincx, mincy, maxcx, maxcy, &gotit, 0, (maxcy==mincy)?((maxcx>mincx)?-2:2):(maxcy>mincy)?-1:1, maxcx, maxcy);
        numpixls=0;
        for (k=minx;k<=maxx;k++){
          up = down = 0;
          for (l=miny;l<=maxy;l++){
            if (vectImg[k][l].up%2==1) up = 1-up;
            if (vectImg[k][l].down%2==1) down = 1-down;
            if (up||down) numpixls++;
          }
        }
        printf("numpixls: %i, magicnumber: %i\n", numpixls, maxcount*maxcount/numpixls);
      }
    }
  }
}

int
main(int argc, char *argv[])
{
  SDL_Surface *srfImage, *srfImage2, *srfPrimary;
  SDL_Color greys[256];

  Uint8 *thescreen, *theimage;
  int up, down;
  int i,j;
  int min, max;
  int color, high;
  float tlo, thi, smooth;
  SDL_Event       event;


  for (i = 0; i<256; i++)
    greys[i].r = greys[i].g = greys[i].b = i;


  /* Initialize SDL */
  if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) {
    fprintf(stderr, "Couldn't initialize SDL: %s\n",SDL_GetError());                exit(1);
  }
  atexit(SDL_Quit);

  tlo = atof(argv[2]);
  thi = atof(argv[3]);
  smooth = atof(argv[4]);

  /*  tlo = 0.1;
      thi = 0.95;
      smooth = 1.2;
  */
  for (i=0;i<256;i++){
    mycolors[i].r = mycolors[i].g = mycolors[i].b = i;
  }
```

```
srfImage = SDL_LoadBMP(argv[1]);
srfPrimary = SDL_SetVideoMode(WIDTH*2, HEIGHT*2, 8, SDL_SWSURFACE|SDL_HWPALETTE);



if ( srfPrimary == NULL ) {
  fprintf(stderr, "Couldn't set 320x200x8 video mode: %s\n",
          SDL_GetError());
  exit(1);
}


SDL_SetColors(srfPrimary, mycolors, 0, 256);

SDL_SetColors(srfPrimary, greys, 0,256);
//  SDL_ShowCursor(0);
SDL_WM_SetCaption("Canny Edge Detection", NULL);


theimage = (Uint8 *)srfImage->pixels;
thescreen = (Uint8 *)srfPrimary->pixels;

min = 256;
max = 0;
for(i=0;i<WIDTH;i++)
  for(j=0;j<HEIGHT;j++){
    pixbuf[i][j]=srfImage->format->palette->colors[theimage[j*srfImage->pitch+i]].r;
    if (pixbuf[i][j]>max)
      max = pixbuf[i][j];
    if (pixbuf[i][j]<min)
      min = pixbuf[i][j];
  }
//  printf("Min: %i, max: %i\n", min, max);

for(i=0;i<WIDTH;i++)
  for(j=0;j<HEIGHT;j++){
    pixbuf[i][j]=((pixbuf[i][j]-min)*255)/(max-min);
  }

GaussSmooth(smooth);
ComputeGradient();
SuppressNonMax();
Hysteresis(tlo, thi);

floodfill();



for(j=0;j<HEIGHT;j++){
  up = down = 0;
  for(i=0;i<WIDTH;i++){
    if (vectImg[i][j].up%2==1) up = 1-up;
```

```
      if (vectImg[i][j].down%2==1) down = 1-down;

/*        thescreen[2*j*srfPrimary->pitch+2*i]= result[i][j];  */
/*        thescreen[(2*j+1)*srfPrimary->pitch+2*i+1]= result[i][j];  */
/*        thescreen[(2*j+1)*srfPrimary->pitch+2*i]= result[i][j];  */
/*        thescreen[2*j*srfPrimary->pitch+2*i+1]= result[i][j]; */


      thescreen[2*j*srfPrimary->pitch+2*i]=
        thescreen[(2*j+1)*srfPrimary->pitch+2*i+1]=
        thescreen[(2*j+1)*srfPrimary->pitch+2*i]=
        thescreen[2*j*srfPrimary->pitch+2*i+1]= 255-(up||down)*255;

      //      thescreen[(2*j+1)*srfPrimary->pitch+2*i+1]|= vectImg[i][j].up*255;
      //      thescreen[2*j*srfPrimary->pitch+2*i+1]|= vectImg[i][j].down*255;

    }
  }
  SDL_SaveBMP(srfPrimary, "grab.bmp");
  SDL_UpdateRect(srfPrimary,0,0,0,0);
  while(1)
    while (SDL_PollEvent(&event)) {
      switch (event.type) {
      case SDL_MOUSEBUTTONDOWN:
        SDL_Quit();
        exit(0);
        break;
      default:
        break;
      }
    }
}
```