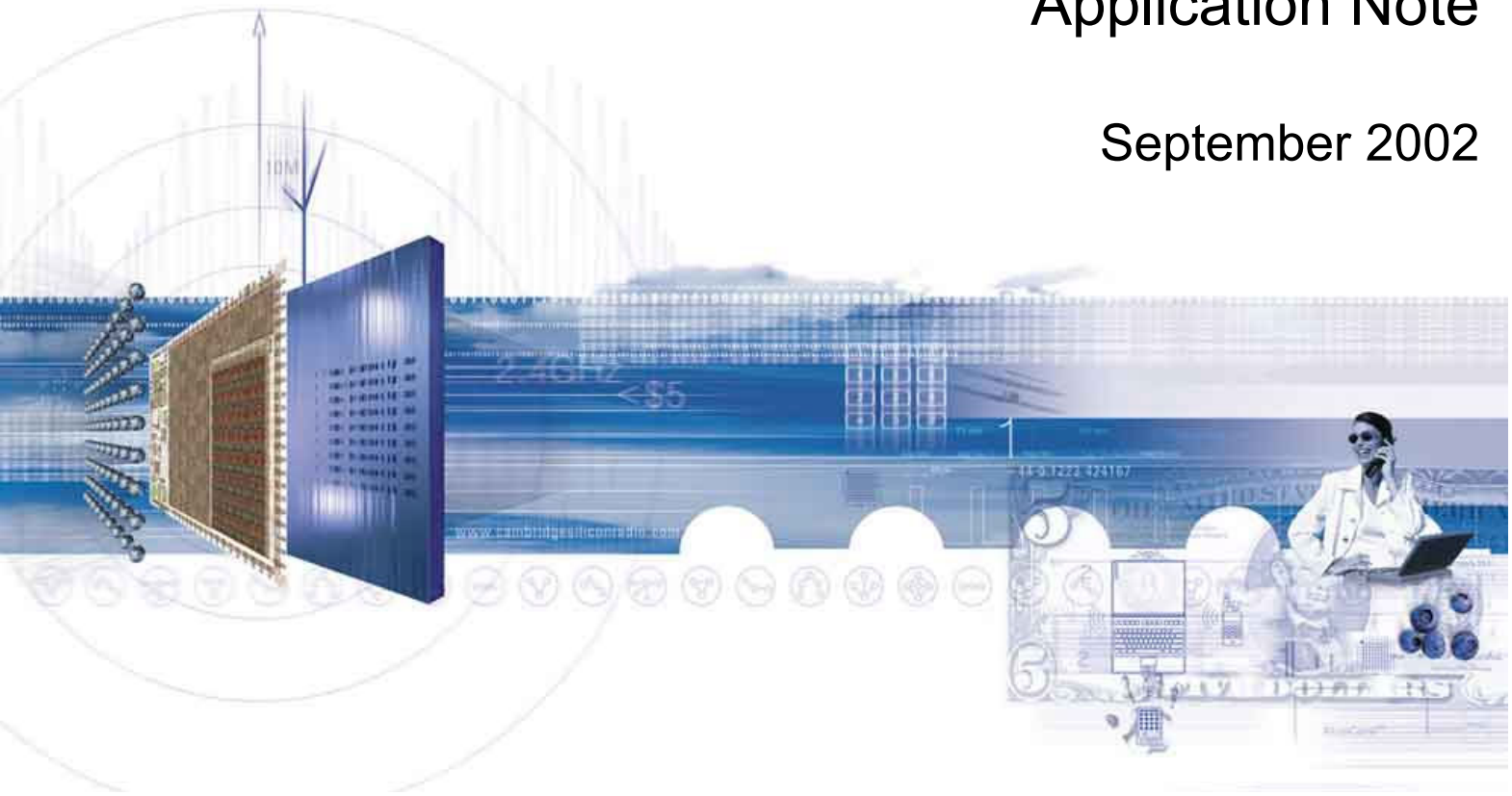
BlueCore™

Accessing RFCOMM Using RFCLI and TCL

Application Note

September 2002



CSR

Unit 400 Cambridge Science Park
Milton Road
Cambridge
CB4 0WH
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

Contents

List of Figures

List of Tables

1 Introduction

This document is an application note specifically written to cover using RFCLI and test tool command language (TCL) to access the RFCOMM layer within the BlueStack® Bluetooth™ protocol stack. The application note examines the important aspects of the RFCOMM layer of the Bluetooth protocol stack outlined in Figure 1.1 and covers the following topics:

- Understanding the RFCOMM protocol layer
- Credit based flow control
- RFCOMM firmware build
- Accessing RFCOMM functionality using RFCLI, TCL and fully functional example scripts

The aim of this application note is to give a reasonable understanding of the RFCOMM protocol layer and the concepts behind it, including credit based flow control. It then documents a fully working example to demonstrate the key concepts required in connecting two Bluetooth devices and passing data between them. The inclusion of the example and its documentation allows users to explore further concepts when prototyping other features or profiles from a known starting point.

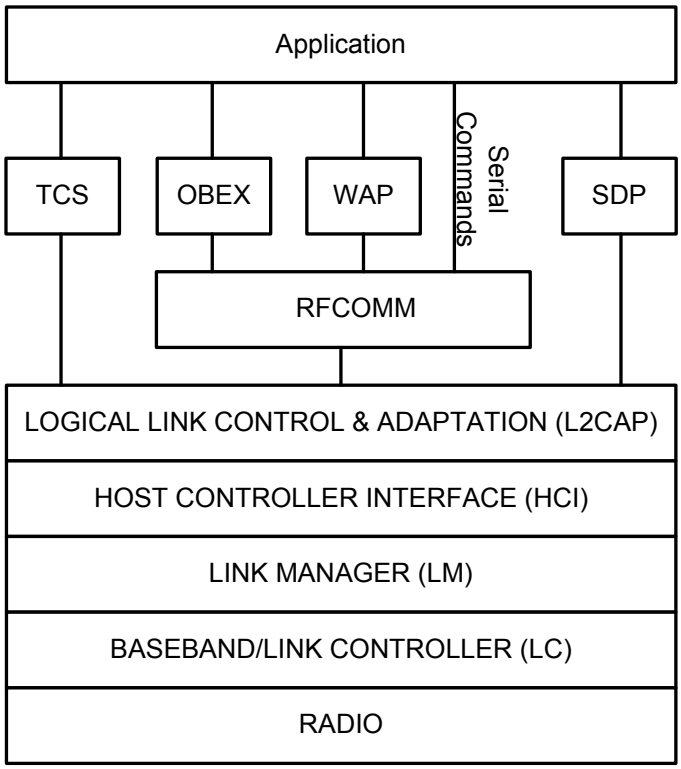


Figure 1.1: Bluetooth Protocol Stack

2 RFCOMM Protocol Layer

The RFCOMM protocol layer is part of the Bluetooth protocol stack outlined in Figure 1.1. It is implemented in BlueStack and for this application note is based on the RFCOMM protocol defined in the Bluetooth specification v1.1.

The RFCOMM protocol is designed to emulate the serial cable lines and the states that are available in an RS-232 serial port over the L2CAP protocol. This means that as well as the data signals being emulated the non-data lines such as clear to send (CTS) are also handled. It is based on the ETSI GSM specification TS 07.10.

The RFCOMM architecture supports the two types of devices that are outlined in Figure 2.1. The type 1 RFCOMM device protocol stack contains the port emulation entity (PEE) that maps the API for the system communications interface to the RFCOMM services. A type 1 interface would normally exist at the end point of a communications path, such as found on a computer or printer. The type 2 interface on the other hand sits in the middle of a communication path and has a port proxy entity (PPE) that is designed to relay data from the RFCOMM to the external RS-232 interface linked to another device. The communications between a serial interface and a piece of data communications equipment (DCE) like a modem is an example of a type 2 device.

Although the RFCOMM does not contain the PEE or the PPE, the RFCOMM API interfaces directly to these two entities and can work with both either singly or running concurrently on a single device.

Another way to view the two types of RFCOMM architecture could be to look at them in terms of whether they have a host or not. The type 1 device would be seen as being hostless, where the system is completely embedded, an example of such a system would be a headset device. Conversely, the type 2 RFCOMM device is a hosted system with an external processor; an example could be the embedded audio gateway part of a mobile phone. Here, the DCE device is represented by the processor inside the mobile phone chipset and is interfaced to the **BlueCore™** device, which is running a RFCOMM firmware build (see Section 1 for further details on RFCOMM firmware).

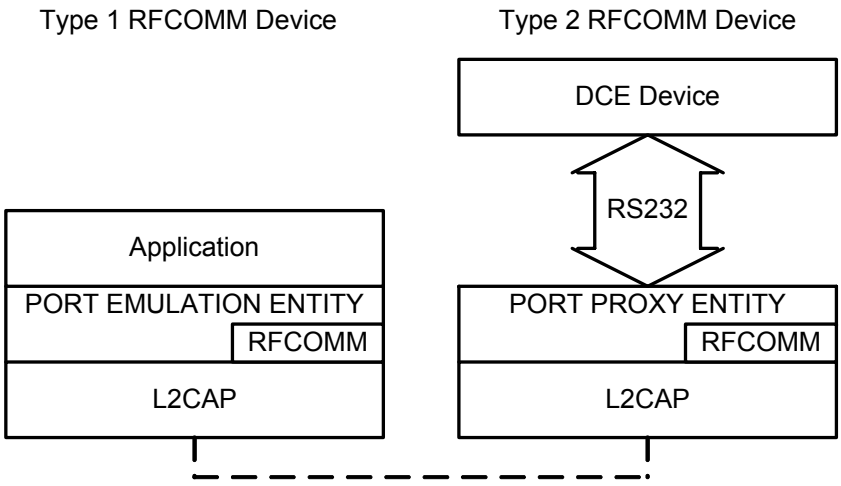


Figure 2.1: Type 1 and Type 2 RFCOMM Devices

As stated earlier in this section the RFCOMM layer has the responsibility of emulating the RS-232 serial ports. The RS-232 contains 9 circuits defined by the pins indicated in Table 2.1, which includes not just the transmit and receive data lines but also the various control signals. The RFCOMM layer is a simple and reliable transport protocol containing provision for:

- Framing
- Multiplexing
- Modem status lines of RTS, CTS, DSR, DTR, DCD and ring
- Remote line status of break, overrun and parity
- Remote port settings such as baud rate, parity, number of data bits
- Parameter negotiation such as frame size

RS232 Pin (25 Way)	RS232 Pin (9 Way)	V.24 Code	Circuit Name
7	5	102	Signal Common
2	3	103	Transmit Data (TD)
3	2	104	Received Data (RD)
4	7	105	Request to Send (RTS)
5	8	106	Clear to Send (CTS)
6	6	107	Data Set Ready (DSR)
20	4	108	Data Terminal Ready (DTR)
8	1	109	Data Carrier Detect (CD)
22	9	125	Ring Indicator (RI)

Table 2.1: RS-232 Circuits Emulated by RFCOMM

The Bluetooth over the air link is seen as the replacement of the wired interface in the ETSI GSM specification, therefore the RFCOMM layer can be seen as providing a serial cable replacement. The ETSI GSM specification allows for the connection of a communications link between a GSM phone and another computing device, such as a laptop computer, via a standard serial COM port interface, this link allows a laptop to use the GSM phone as a radio modem and the RFCOMM layer replaces the standard serial COM port interface.

Although RFCOMM is based on the ETSI GSM specification it is not restricted to just GSM phones. RFCOMM is seen as fundamental to many of the Bluetooth profiles, it forms the basis for the serial port profile (SPP) that is core to a number of the profiles. The position of the SPP with respect to other profiles is shown in Figure 2.2 and from this diagram it can be seen that all the profiles that reside within the SPP box have this profile as a mandatory requirement.

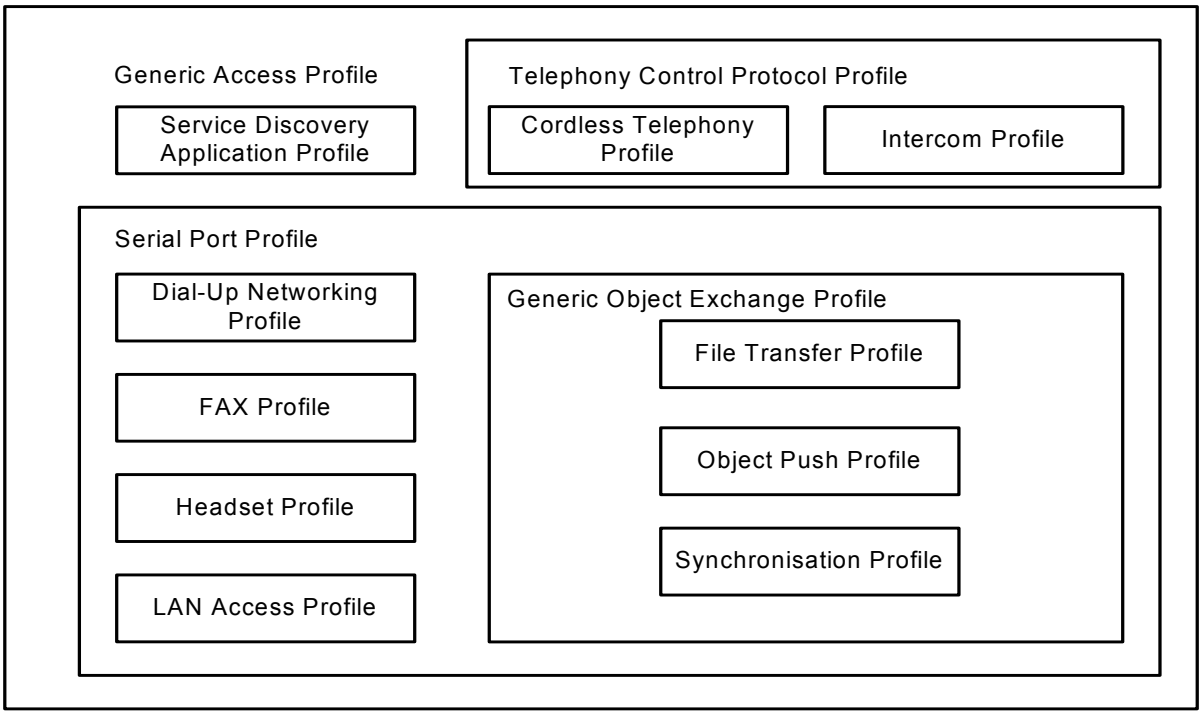


Figure 2.2: Bluetooth Profiles

The RFCOMM implementation, that is based on the ETSI GSM specification, from the perspective of the non data signals of the RS-232 interface does not distinguish between a type 1or a type 2 device i.e. it does not differentiate between a DTE or DCE equipment. The implication this has is that the control signals are sent and received as independent DCE/DTE signals and therefore the RFCOMM layer does not distinguish between DSR or DTR, or RTS and CTS. The way in which the ETSI GSM specification serial port control signals are defined and their corresponding RS-232 control signals are listed in Table 2.2. Transfer of these control signals between devices of the same kind by implication will create a null modem, an example of the wiring between two DTE devices with the null modem emulation is shown in Figure 2.3. It has to be noted that not one individual null modem wiring strategy will fit all schemes, even though it may be applicable in the majority of cases.

ETSI GSM Specification Signals	Corresponding RS-232 Control Signals
RTC	DSR, DTR
RTR	RTS, CTS
IC	RI
DV	DCD

Table 2.2: ETSI GSM Specification Serial Port Control Signals

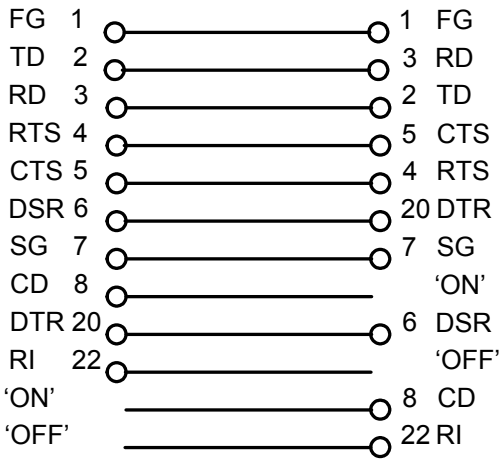


Figure 2.3: Null Modem Pin Out

The Bluetooth specification allows two Bluetooth devices that are communicating using RFCOMM to open multiple emulated serial ports. Within the specification the RFCOMM layer will be permitted to open up to a maximum 60 emulated ports, but the specific numbers of ports used on a device though will be application specific.

Each port opened at the RFCOMM layer is identified by a Data Link Connection Identifier (DLCI) and signifies an ongoing link between a client and a server application. A 6bit number in the range of 2 to 61 represents the value of the DLCI that is used to identify the emulated port. The DLCI values of 1, 62 and 63 are reserved, with the DLCI value of 0 representing the control channel.

The reserved value of DLCI as 1 is unusable in the normal sense due to the concept of server channels being employed within RFCOMM. The server channel concept allows for the situation of both the client and server applications residing on both sides of the RFCOMM session and being able to independently make connections with respect to each other. Figure 2.4 show that the DLCI value space of the ETSI GSM specification is divided between the two linked device using the RFCOMM server channel and the direction bit D. For any RFCOMM session the initiating device is given a direction bit D a value of one and the other device is has its direction bit D set to zero.

Alongside the direction bit, the server channel in RFCOMM is seen as a subset of the DLCI. Server applications registering with an RFCOMM service are assigned a channel number of 1 to 30. The channels 0 and 31 are not permitted in order to remain compatible with the ETSI GSM specification that has these corresponding DLCIs reserved within its specification. The implication of setting the direction bit is to effectively partition the DLCI space such that the server applications on the initiating space can be reached through odd values of DLCIs in the range 3 to 61 and the server applications on the non-initiating device even values in the range 2 to 60. The server channel assignment is carried out by RFCOMM and so the application must register with RFCOMM first before the application can register with the service discovery protocol (SDP).

Bit Number	1	2	3	4	5	6	7	8
TS 07.10	EA	C/R	DLCI					
RFCOMM	EA	C/R	D	Server Channel				

Figure 2.4: Format of the Address Field

2.1 Local Server Channel and Mux ID

Between any two Bluetooth devices there will be a single multiplexor per device. In the case of a master device outlined in Figure 2.5 it depicts the master connected to two slaves. Here the master is running multiple multiplexor sessions but still has only a single multiplexor per slave device, therefore it uses MUX 1 to communicate to SLAVE 1 and MUX 2 to communicate to SLAVE 2.

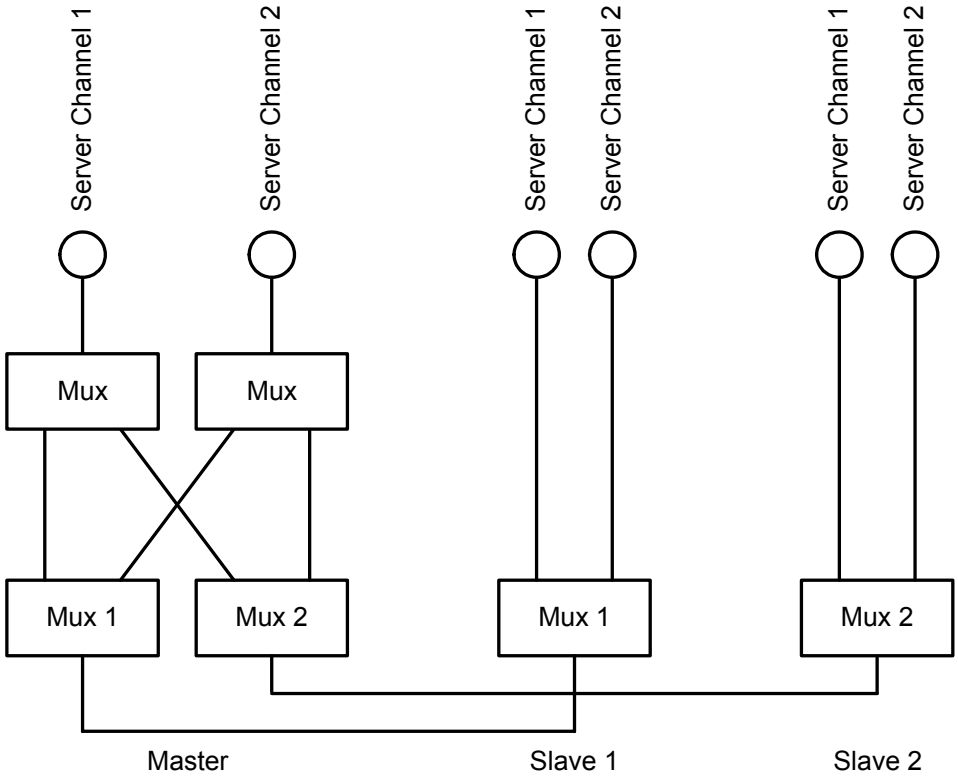


Figure 2.5: Multiplexor and Server Channels

If in Figure 2.5 the system consisted of just the master and Slave 1 then the communications over the RFCOMM layer with multiple emulated serial ports would look like the system depicted in Figure 2.6. The RFCOMM specification, although optional, allows the running of multiple sessions of multiplexors. What this implies is that a Bluetooth device supporting multiple emulated serial ports is permitted to have connection end points in different Bluetooth devices, this is the system set-up already shown in Figure 2.5, but can be redrawn in the style of Figure 2.6 to produce Figure 2.7 showing multiple multiplexor sessions of the emulated port. It has to be noted that each multiplexor session will possess its own L2CAP channel ID (CID)

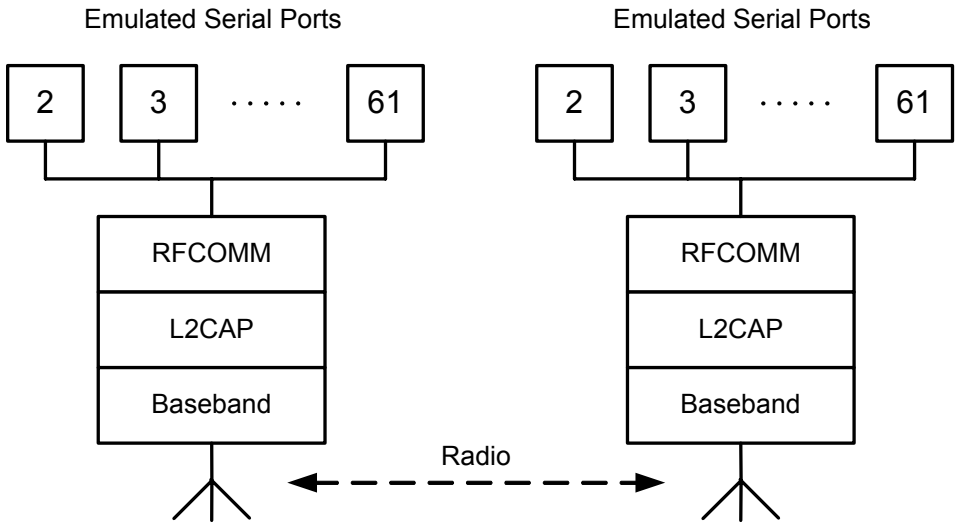


Figure 2.6: Multiple Emulated Serial Ports Between Two Bluetooth Devices

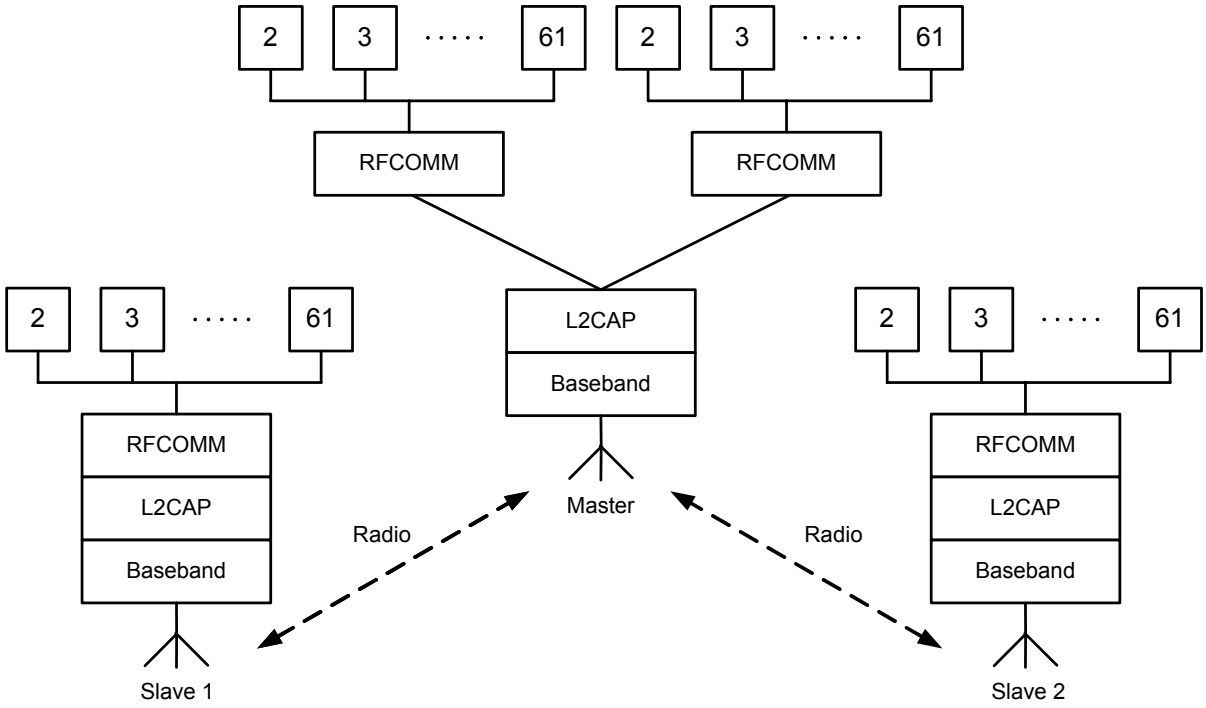


Figure 2.7: Multiple Emulated Serial Ports with Multiple Multiplexor Sessions

In the Bluetooth specification v1.1 the RFCOMM section describes a service definition model the diagram of which is shown in Figure 2.8. From this diagram it can be seen that the application layer is the item that makes use of the serial port communications interface. The PEE layer maps a system specific communication to the RFCOMM services. The RFCOMM layer as described earlier multiplexes multiple emulated serial ports providing a transparent data stream and control channel over an L2CAP channel. Taking the PEE and the RFCOMM layer together this combination is also known as the port driver.

The service registration/discovery layer is where the server applications register on the master Bluetooth device. The server applications register their services here in order to make them discoverable to client applications. The information provided is associated with the SDP and is how to reach the server applications on other devices.

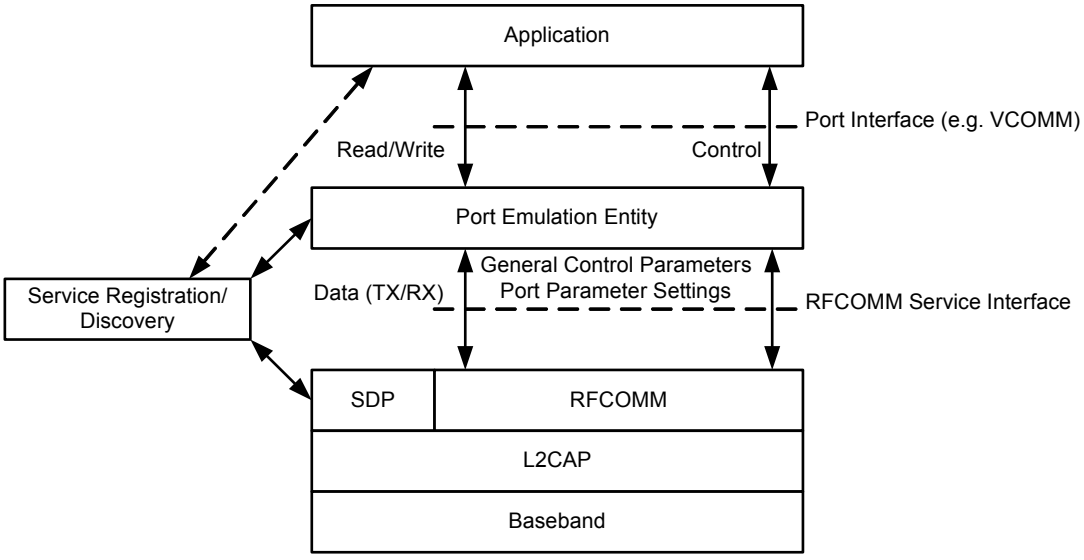


Figure 2.8: RFCOMM Service Definition Model

In order to form association between a service and a logical ID in RFCOMM a local server channel is used, see Figure 2.5. The applications that have been registered within RFCOMM will have provided a server channel

identity to the RFCOMM layer and these server channel identities should be registered via the service registration layer into the Service Discovery Database (SDD). Alongside registration of the server channel identity with RFCOMM the application will also provide a protocol handle (phandle), which is used so that events can be sent to the service. In Figure 2.5 a master device is shown communicating with two slave devices, and on this master it contains two server channels, server channel 1 and server channel 2 that connect to specific services. In this example it is possible for either slave to access the services on the master through the associated server channels. The mux id being the unique identifier to each of the slaves.

3 Flow Control

The RFCOMM layer offers two forms of flow control for data transfer. The initial form of flow control introduced in v1.0 of the Bluetooth specification is based on an aggregate data flow scheme that uses flow control commands to enable data flow to and from the RFCOMM layer. A description of this scheme is available in Part F-1 Section 6.3 of the Bluetooth specification v1.1. BlueStack primitives `RFC_FCON` and `RFC_FCOFF` are used to enable and disable the dataflow respectively.

The second form of flow control implementation was introduced in v1.1 of the Bluetooth specification and can be found in Part F-1 Section 6.5 of the Bluetooth Specification v1.1. This form of flow control, known as credit based flow control, its description and implementation is described in Section 3.1. CSR's proprietary extensions to credit based flow control include a flow control layer are described in Section 3.3.

3.1 Credit Based Flow Control

Credit based flow control has been introduced into v1.1 of the Bluetooth specification and is a mandatory requirement for conformity to this version of the specification. This form of flow control is provided on a per RFCOMM channel basis, i.e. each data link connection (DLC) will have flow control associated with it.

The starting point for the use of credit based flow control is at the port entity, which is either the PEE or the PPE. The port entity requests the use of credit based flow control and issues the initial number of credits to the RFCOMM layer. These parameters are passed to the RFCOMM layer through the `RFC_PARNEG` primitive, which can be seen as one of the initial steps in Figure 3.1. The `RFC_PARNEG` primitive is a structure within BlueStack of type `DLC_PAR_T` that contains the DLC parameters. The fields within the structure required for initiating the flow control are:

- `credit_flow_control`: Enables credit based flow control when set to `TRUE`
- `initial_credits`: Determines the initial number of credits for data transfer, 16bit parameter (8bits useable). Initial number of credits issued depends upon the application, in resource limited systems the number of credits issued is minimised (4 to 7 credits), occasionally such systems may wish to issue no credits until the RFCOMM connection has been established.

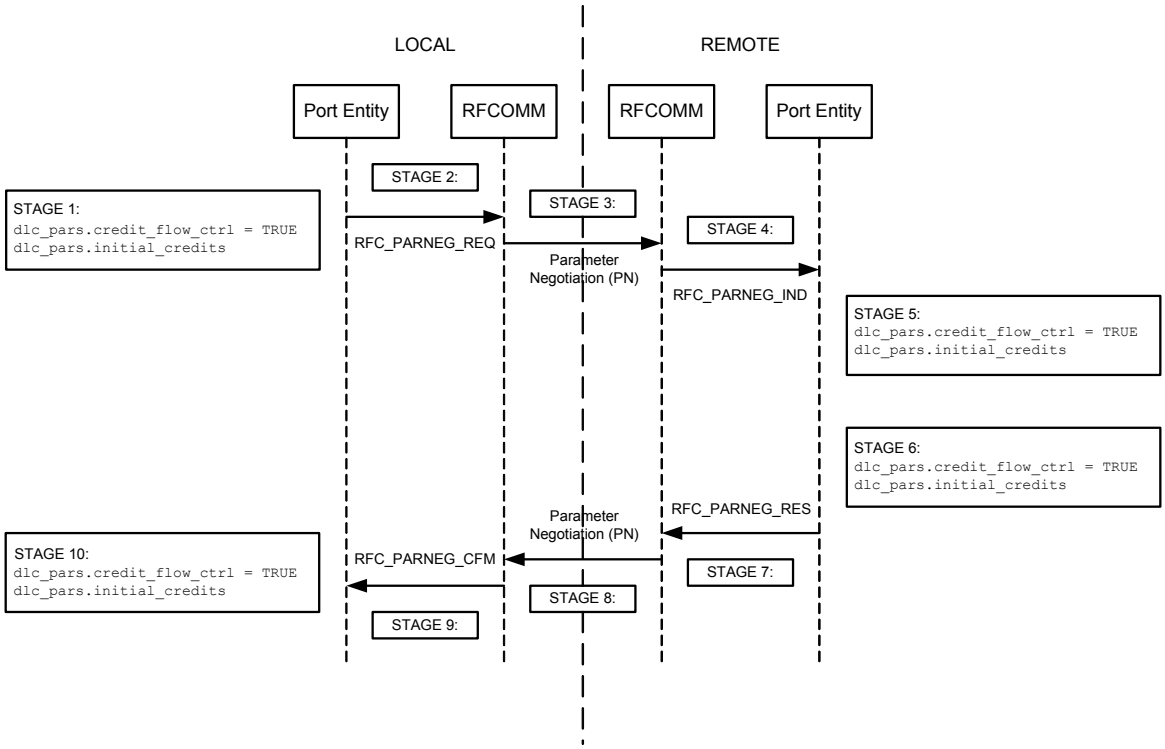


Figure 3.1: Credit Based Flow Control Negotiations

Figure 3.1 shows the initialisation of credit based flow control between a local device and its corresponding remote peer. The steps through which the local and remote port entities go through to establish credit based flow control along a DLC are as follows:

1. Assume that a RFCOMM connection already exists between the local and the remote device. See Section 1 for further details on how to set up a RFCOMM connection.
2. At STAGE 1 in Figure 3.1 the local port entity within the local device requests the use of credit based flow control by setting:
   - `dlc_pars.credit_flow_ctrl` to `TRUE`
   - The initial credits for upstream packets to the RFCOMM in the parameter `dlc_pars.initial_credits`. This initial value represents the number of credits for data receipt by the local port entity.
3. At STAGE 2 the local port entity passes the `DLC_PARS_T` structure containing the credit flow control enable and the initial credits in a `RFC_PARNEG_REQ` primitive to the local RFCOMM layer.
4. At STAGE 3 the local RFCOMM layer communicates with the remote RFCOMM layer using the parameter negotiation (PN) command. The PN command is used to configure the DLC.
5. At STAGE 4 the remote RFCOMM layer passes a `RFC_PARNEG_IND` primitive to the remote port entity containing the requests for credit based flow control from the RFCOMM layer on the local peer device. It receives the parameters `dlc_pars.credit_flow_ctrl` and `dlc_pars.initial_credits` in the `DLC_PARS_T` structure.
6. At STAGE 5 the parameters from the `RFC_PARNEG_IND` primitive are received at the remote port entity. Here the initial credits represent the number of credits for data transmission by the remote port entity.
7. At STAGE 6 the remote port entity responds to the `RFC_PARNEG_IND` primitive by setting:
   - `dlc_pars.credit_flow_ctrl` to `TRUE` if it is willing to accept the request for credit based flow control
   - The initial credits permitted for the number of packets destined for receipt by the remote port entity in the parameter `dlc_pars.initial_credits`. This initial value represents the number of credits for data receipt by the remote port entity.
8. At STAGE 7 the parameters set at STAGE 6 are passed in RFC_PARNEG_RES primitive from the remote port entity to the remote RFCOMM.
9. At STAGE 8 the PN command is used to communicate the parameters between the two peer RFCOMM layers.
10. At STAGE 9 the local RFCOMM layer finally sends a `RFC_PARNEG_CFM` primitive as a response to the `RFC_PARNEG_REQ` primitive issued at STAGE 2.
11. At STAGE 10 with the parameter `dlc_pars.credit_flow_ctrl` set to `TRUE` then the negotiation for credit based flow control has been successful. The value `dlc_pars.initial_credits` is the number of credits that the local RFCOMM layer gives the local port entity for downstream packets, which it may want to transmit. In other words it is the number of credits for data transmission by the local port entity.

The above steps outline a negotiation for the use of credit based flow control and in this scenario the negotiation has been successful. Now what has to be noted at this point is the following:

- The actual number of credits set is not negotiated at any stage. The parameter of `dlc_pars.initial_credits` that used to pass initial credits between the various port entities and the their corresponding RFCOMM layers just reports the initial credits that it has.
- If at STAGE 5 the remote port entity is unable to support credit based flow control then at STAGE 6 the `DLC_PAR_T` parameters will be set as follows and these values will also appear at STAGE 10 set to these values:
  - `dlc_pars.credit_flow_ctrl` is set to `FALSE`
  - `dlc_pars.initial_credits` is set to 0
- Negotiation for credit based flow control is on a per DLC basis referenced to a server channel. RFCOMM does not support multiple flow control methods on a multiplexor, therefore once a flow control method is initially set for a multiplexor then any proceeding DLC channels set up on this multiplexor are forced to use the initial flow control method that has been set.

Once credit based flow control has been set up successfully between two peer RFCOMM layers over a DLC channel there needs a mechanism that exists to update the credits. Figure 3.2 shows how the credits are updated using the credits field within the `RFC_DATA_REQ` and `RFC_DATA_IND` primitives. In the example shown in Figure 3.2 there is no user data exchanged, here the mechanism is being used to transfer the credits to update the credit value. There is no reason why these primitives should not contain data being transferred along side the credit values being update. Therefore it is perfectly acceptable to see along a typical DLC channel, the `RFC_DATA_REQ` and `RFC_DATA_IND` primitives with data being transferred combined with credit values being updated, mixed with primitives that are just updating the credit values.

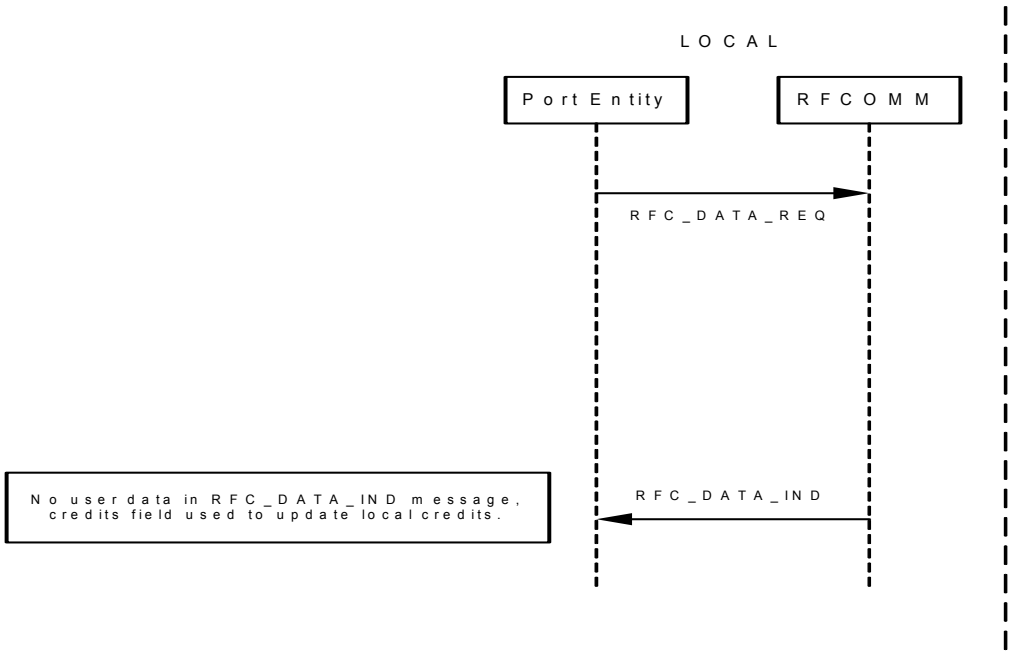


Figure 3.2: Updating Flow Control Credits

Using the example shown in Figure 3.2 the credit update scheme works as follows:

- The port entity will have been given an initial credit value as was shown in STAGE 10 in Figure 3.1. This initial credit value is the amount of packets that the port entity can transmit.
- Assume the initial credit value for the port entity is set to 5
- The port entity transmits 3 packets leaving remaining credit value of 2.
- The port entity receives a `RFC_DATA_IND` primitive with the credits parameter set to 2
- The port entity now increases its remaining credit value by the credit parameter contained in the `RFC_DATA_IND` primitive therefore its new credit value is 4

In the above example it showed how the port entity transmit credit value is updated via the `RFC_DATA_IND` primitive, the transmit credit value for port entity will also be decremented for each packet sent and will be initialised when the RFC_PARNEG_CFM primitive is received. The piece of C type pseudo code below shows these three cases and their interaction on the transmit credit value called `tx_credits`:

```
switch(status)
{
  case RFC_PARNEG_CFM:
  {
    if(dlc_pars.credit_flow_control)
    {
      tx_credits = dlc_pars.initial_credits;
    }
  }
  break;

  case RFC_DATA_IND:
  {
    tx_credits += credits;
  }
  break;

  case Data to Transmit:
  {
    if(tx_credits > 0)
    {
      send data to RFCOMM in a RFC_DATA_REQ;
      tx_credits--;
    }
  }
  break;
}
```

3.2 Maximum Frame Size

During initialisation of credit based flow control the maximum frame size of the RFCOMM packet to be used between the master and slave is negotiated. This negotiation occurs at the PN stages, which are STAGE 3 and STAGE 8 shown in Figure 3.1. During the PN stages details are passed between the master and slave using a PN packet. In general, one device sends a PN message and the other devices replies with another PN message. The device that sends the first PN message proposes a maximum frame size and the replying device may respond with a smaller value for maximum frame size, but it is not allowed to propose a larger value. PN messages may be exchanged until the device that initiated the first message is satisfied with the parameters it receives.

In the Bluetooth Specification v1.1 the default value for the maximum frame size is 127bytes, however this value may be too large for some applications. Some of the profiles built from RFCOMM have smaller maximum frame sizes, for example a headset application might typically have a maximum frame size around 32bytes.The smaller maximum frame size have to be taken into account when attempting an RFCOMM connection between two devices. For the link to function correctly the maximum frame size for the link must be set to the smallest maximum frame size value of the two devices.

3.3 Flow Control Layer

Before data is to be transmitted over the air it is buffered in random access memory (RAM) on BlueCore devices. The RAM available for this buffering is a limited resource and therefore it is important that the data being streamed to the peer does not overflow this memory resource. In earlier releases of the RFCOMM firmware the firmware governed the flow of credits in order to regulate the amount of data on the chip.

The new scheme that is outlined in Figure 3.4 is a modification to the standard credit based flow control scheme described in Section 3.1. This scheme introduces the concept of a Flow Control Layer (FCL) that sits on the host between the application layer and the transport layer. The main responsibility of the FCL is for checking that the BlueCore device is not being overloaded with RFCOMM data requests or other primitives. The FCL monitors the number of primitives that have been issued to BlueCore and the number of primitives that have been consumed. In Section 3.1 it was shown how the number of credits is updated via the `RFC_DATA_IND` primitive, this same strategy is used to signal to the host the number of primitives that have been consumed. This is done through the use of a special case of the `RFC_DATA_IND` primitive called the flow control token (FCT). The FCT is identified by the fact that the parameter `mux_id` has been set to `0xff` and the `payload_length` is set to `0`.

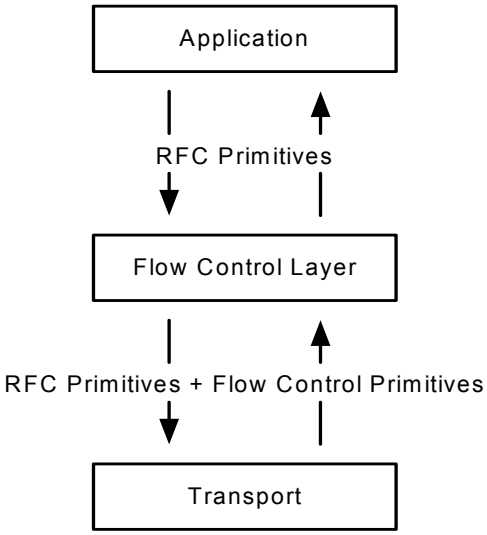


Figure 3.3: Flow Control Layer

It was mentioned earlier in this section that introducing the FCL was a modification to the standard credit based flow control scheme outlined in Section 3.1. The implication this has for the overall system is that FCL layer needs to be implemented as part of the host design. In theory the application layer may not need to be modified much when the FCL layer is introduced. It can still issue and receive RFCOMM data and primitives but instead of sending these directly to the RFCOMM layer these are passed via the FCL. The FCL will then strip out, modify and act on the flow control primitives such as the FCT and therefore ensure that should the application issue more data than the BlueCore device can handle the FCL buffers the data until memory resource becomes available.

The simplest way the FCL controls the data flow and decides whether it needs to buffer the primitives before passing them onto BlueCore due to the RFCOMM buffer being full is through the use of windows. The FCL layer aims to ensure that the number of slots in the windows is less than or equal to the number of primitives issued minus the number of primitives consumed i.e.

window >= (Number of primitives issued)$^{(1)}$ – (Number of primitives consumed)$^{(2)}$

Now the window will be proportional to the number of primitives that can be stored within the RFCOMM buffer, so if the primitive size is based on the largest primitive i.e. RFC_DATA_REQ, then the maximum primitive size will be equal to header size which is 12 bytes plus the frame size n bytes negotiated during the RFCOMM parameter negotiation. Therefore,

window = (RFCOMM buffer size$^{(3)}$) / (Maximum primitive size)

As stated earlier in this section the FCL must count all primitives issued to the chip, this includes the RFCOMM initialisation and channel setup primitives. Until a frame size has been negotiated a "working" window of 1 can be set to allow these primitives to be sent. However, the window should then be recalculated once a frame size has been negotiated.

Note:

(1) Is the count of RFCOMM primitives sent to BlueCore. It is therefore essential as part of the application design or as part of the FCL that a count of the number of primitives issued is kept.

(2) Can be read from the credits field of the FCT

(3) The RFCOMM buffer size is the parameter RFCOMM_BUFFER_SIZE = 1024 - 256 bytes, this is a fixed firmware parameter that is not modifiable by the user.

4 RFCOMM Firmware Build

To be able to use RFCLI or access other layers of the Bluetooth protocol stack above the HCI layer, a RFCOMM firmware build needs to be loaded onto a Bluecore device. In general there are two forms of firmware release available from CSR for the BlueCore devices. These firmware releases are based on splits in the Bluetooth protocol stack shown in Figure 1.1. The usual split is at the HCI or RFCOMM level and therefore the two firmware releases available are known as either the HCI firmware or the RFCOMM firmware builds. This section describes the fundamentals of an RFCOMM firmware build.

The RFCOMM firmware build is based on BlueStack and gives the application programming interface (API) access to the RFCOMM layer, L2CAP layer, SDP and device manager. The primary uses for the RFCOMM firmware is for developing profiles and VM applications. It is not suitable for third party stacks, as it does not give full or direct access to the HCI interface layer.

The RFCOMM firmware is usually supplied as two versions one that supports BCSP and the other supports H4 the Bluetooth UART protocol. There are then two encryption variants of each of the RFCOMM firmware versions, these cover 56bit and 128bit encryption. An example of the RFCOMM firmware is RFCOMM1.1v13.10.5, this is the top level name and its name can be split down to identify it. The ‘RFCOMM’ meaning that it is RFCOMM firmware, ‘1.1’ indicates the version of the Bluetooth specification it supports, ’13.10’ is the level of HCI firmware the RFCOMM firmware is based upon i.e. here it is HCI Stack1.1v13.10 and ‘.5’ means that this is fifth software build. The top level description for the firmware will then have four builds associated with it and for this example RFCOMM1.1v13.10.5 has build identifiers of 339, 340, 341 and 342 that represent whether the build is for BCSP or H4 and whether it is using 56bit or 128bit encryption.

In the RFCOMM firmware there are a few restrictions that are noticeable:

- Full access to the HCI layer is not support, limited interface to HCI commands is supported through the device manager
- The USB transport layer is not supported
- There may be no or limited DFU support
- Bandwidth is limited in comparison to HCI firmware. The new credit based flow control with the FCL achieves around 300kbps. Bandwidths can be varied depending on frame size employed.
- There is Piconet support but generally the RFCOMM build usually defaults to point to point devices such as headset, DUN, cable replacement etc. The settings for PSKEYS can be modified in order to achieve Piconet support, at present this is limited to three slave devices.
- Not all the built in self test functions (BIST) are supported
- Restrictions for a particular RFCOMM firmware build can be found in its software release note

Note:

To find the latest version and documentation on the RFCOMM firmware see the CSR website for further information. This includes the latest qualified version of RFCOMM to the Bluetooth specification and the protocol implementation confirmation statement (PICS) for baseband, LM, L2CAP, RFCOMM, SPP, SDP and GAP.

5 Accessing RFCOMM Functionality Using RFCLI and TCL

The aim of this section is to explore in a step-by-step manner the basic principles required to use RFCLI and TCL to create an RFCOMM connection and pass data between two BlueCore devices. Alongside the use of RFCLI and TCL, the results are examined using the output data captured by an over air sniffer. The output data is included to confirm that the correct RFCOMM packets are sent and received over the air between the two BlueCore devices.

The over air sniffer also has the ability to capture the serial traffic on the BCSP links and therefore this data is included so that the complete passage of data between the two BlueCore devices can be thoroughly understood.

5.1 RFCLI

RFCLI is a MSDOS based software tool that allows access to various layers of the Bluetooth protocol stack through the BlueStack API. To fully understand RFCLI there is a RFCLI User Guide that covers its features with worked examples and executable source scripts, that allow a user to get up and running fully. The RFCLI tool consists of the following elements:

- The TCL interpreter
- The FCL
- A primitive converter, used to convert the primitives entered in RFCLI to the correct structure used by BCSP, BlueCore and BlueStack and select the correct BCSP channel
- A BCSP driver

5.2 TCL

The RFCLI tool contains a TCL Interpreter v8.3, which allows the sending and receiving of primitives to various layers of BlueStack. TCL is a simple scripting language that extends RFCLI by providing programming facilities such as variables, loops, procedures and libraries. TCL structure and syntax is similar to the C language and therefore lends itself to reasonably rapid understanding and prototyping of functions.

For further information on TCL then there is plenty of information available:

- On the world wide web
- In the examples in the RFCLI User Guide
- In the book Tcl and the Tk Toolkit by John K. Ousterhout

The use of TCL to extend RFCLI allows the user to build extra functionality into RFCLI in terms of procedures that can be kept in specific libraries and used as required. Within the libraries supplied with RFCLI there is extra functionality to control specific profiles such as the headset and HID, as well as more basic procedures that group single primitives together into one procedure call e.g. the procedure `rfc_connect_mst` available in the library covers the functionality of three pairs of primitives:

- RFC_START_REQ and RFC_START_CFM
- RFC_PARNEG_REQ and RFC_PARNEG_CFM
- RFC_ESTABLISH_REQ and RFC_ESTABLISH_CFM

5.3 Setting Up RFCOMM Link

This section brings together the use of the RFCLI and TCL to demonstrate how to form an RFCOMM link between two BlueCore devices and then pass data between them. In section 6 the principles developed here are extended further to include items such as the credit based flow control.

RFCOMM as outlined in Section 2 is based on the GSM TS 07.10 specification. This specification is a asymmetrical protocol that allows the multiplexing of streams of data onto one serial cable. RFCOMM differs in the fact that it is designed as a symmetrical specification and sends TS 07.10 frames of data over the L2CAP

layer using a subset of the feature frames and commands of TS 07.10 that have been adapted for Bluetooth usage.

RFCOMM layers communicate with each other through the use of frames and these frames make up the data payload of a L2CAP packet. There are five types of RFCOMM frames and these are:

1. The start asynchronous balanced mode (SABM) frame that is used as the start up the link command
2. The unnumbered acknowledgement (UA) frame that is used as a response when connected
3. The disconnect command (DISC)
4. The disconnected mode (DM) that is the response to a command when disconnected
5. The unnumbered information with header check (UIH) that is used for data

RFCOMM makes use of channels, which all have a unique DLCI. The DCLI of 0 is a special case that is used for RFCOMM control signalling, therefore if data is sent in a UIH frame along the DCLI channel of 0 then this data contains a control message. The SABM, UA, DM and DISC frames are control frames that also utilise DCLI = 0.

RFCOMM frames are carried in L2CAP payloads and to be able to aid this the RFCOMM specification in Bluetooth has reserved a PSM value for L2CAP of 0x0003. This means that if the L2CAP should receive frames with the PSM = 0x0003, the frame will be passed on to the RFCOMM layer for processing.

Figure 5.1 outlines the overall generic message sequence chart that is required between two RFCOMM layers that exist on separate BlueCore devices in order to form a data channel between them. The local RFCOMM device in Figure 5.1 is responsible for initiating the link and the remote RFCOMM device is responsible for responding to the initiating device. With reference to Figure 5.1, in order to set up a RFCOMM link between two Bluetooth devices the following steps are required:

1. Set up an L2CAP connection between the local and the remote device, this requires setting up the L2CAP channel for RFCOMM with PSM = 0x3000
2. The first frame sent by the local RFCOMM is a SABM frame and this is sent on DLCI = 0. When a command is sent from RFCOMM an acknowledgement timer (T1) is started. The timeout value of T1 value is normally set between 10 to 60 seconds and if a command is not acknowledged before T1 reaches the timeout value then the connection is shutdown.
3. The remote RFCOMM when it receives the SABM frame and is willing to connect enters asynchronous balanced mode (ABM) and sends back a UA frame in response. Should the remote device not wish to connect then it will return a DM frame.
4. Once the local RFCOMM receives the UA frame as acknowledgement, the connection has succeeded and the local and remote devices enter the parameter negotiation phase that is outlined in STAGE 3 and STAGE 8 of Figure 3.1 and described in Section 3.1. Although the parameter negotiation stage is optional for all channels that are set up, this first parameter negotiations phase would usually occur here, as the parameters set up here are often left unchanged for the rest of the channels that are set up. The parameter negation command is passed as a message on a UIH frame on DLCI = 0. At the point when the parameter negotiation is complete then DLCI = 0 is completely up and running.
5. Once DLCI is completely up and running a second DLCI channel needs to be started to create a data link on the RFCOMM channel, therefore a second SABM frame is initiated by the local RFCOMM to the remote RFCOMM.
6. The remote RFCOMM when it receives the SABM frame can optionally go into a LMP authentication and encryption phase. If this phase is required then the timeout value on the acknowledgement timer T1 needs to be extended to typically between 60 to 300 seconds to allow for authentication and encryption to complete successfully before timeout occurs. If the remote device is willing to connect it enters ABM and sends back a UA frame in response. Should the remote device not wish to connect then it will return a DM frame.
7. Once the UA frame has been received at the local RFCOMM layer an optional modem status commands (MSC) can be exchanged between the local and the remote RFCOMM via a control message on DLCI = 0.
8. Once the MSC phase is complete the data channel is available to exchange data on the connection or an optional parameter negotiation phase can be entered prior to the data exchange starting.

To summarise the above steps to create a RFCOMM channel so that data can be passed between two devices:

1. The control channel needs to be set up and the parameters for the link negotiated
2. A second channel needs to be set up for data with options available for authentication and encryption, modem status commands and parameter negotiation

In order to shut down an RFCOMM connection a DISC must be sent on the individual channels leaving the control channel on DLCI = 0 until last. The final DISC sent on DCLI = 0 is responsible for shutting down the multiplexor and is also responsible for closing down the L2CAP channel.

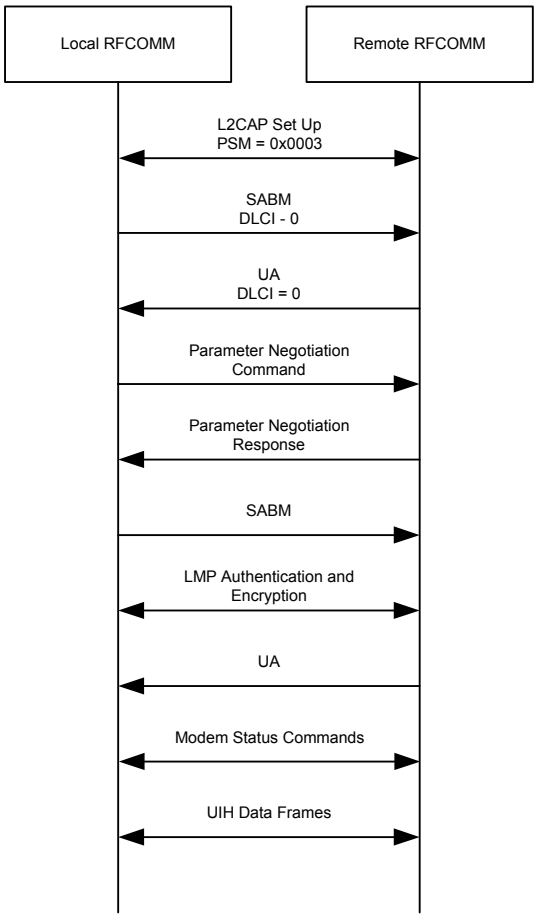


Figure 5.1: RFCOMM Channel Set Up Between Local and Remote Bluetooth Device

In Figure 5.1 this shows a generic set up of an RFCOMM data channel between two Bluetooth devices, this can be minimised by the removing the following optional steps:

- LMP authentication and encryption
- The modem status command exchange between the local and remote RFCOMM

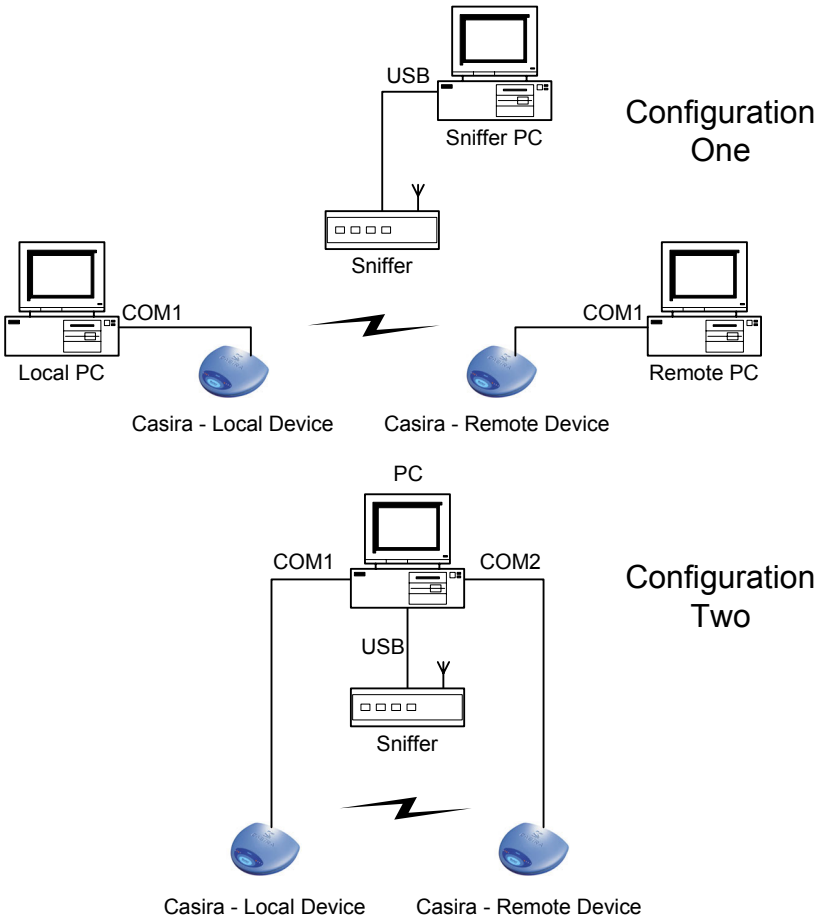


Figure 5.2: System Set Up with Configuration One and Configuration Two

If the above two optional steps are missed out then the messages passed between two RFCOMM layers in order to set up a data link between two BlueCore devices is shown in Figure 5.3. To understand the overall process of setting up an RFCOMM link between two BlueCore devices the message sequence chart in Figure 5.3 can be examined practically with the equipment set-up shown in Figure 5.2. Figure 5.2 has two configuration depending on PC requirements available, if a PC has two standard serial COM ports and a USB port then Configuration Two is the most practical and compact. The system set up requirements to be able to run this example in Configuration Two are as follows:

1. Two Casiras loaded with a BCSP version of RFCOMM firmware are required. The encryption type is not important therefore it may be convenient to use the latest RFCOMM firmware(1) that is available in BlueLab™.
2. A sniffer is connected to the USB port of the PC to as an over air sniffer and record communications traffic on this link. Depending on chosen the sniffer equipment it may be possible to monitor the RFCOMM packets on serial port.
3. The sniffer is set up with the local device set to be the master and the remote device set to be the slave. The sniffer is set recording and left in a state to synchronise to the local and remote device. For further details on setting up the sniffer see the manufacturers user manual.
4. Two RFCLI sessions must be running on a PC where the local device is the Casira attached to COM1 and the remote device is the Casira attached to COM2 of the PC as shown in Figure 5.3.
5. On one RFCLI session run the slave source `exampleslave.tcl` that is shown in Appendix A1 on the remote device by entering the command:

   ```
   source exampleslave.tcl
   ```

6. Wait for the message `Waiting for a connection` to appear on the RFCLI session on the remote device before running a second session on RFCLI
7. On the second RFCLI session run the source script `examplemaster.tcl` that is shown in Appendix A2 on the local device by entering the command:

   ```
   source examplemaster.tcl
   ```

8. The local device script will then attempt to connect to the remote device at the RFCOMM layer
9. Once a successful connection has been made between the local device and remote device, commands and data are passed between the two devices and then the example finishes and the sniffer should have captured either the over air message traffic, the BCSP traffic on COM1 or the BCSP traffic on COM2 depending on the sniffer used and the set up configuration.

Notes:

(1) The RFCOMM firmware used in this example came from BlueLab 2.4. The version used was a BCSP build for Bluecore2. The build ID was 334 and its title was 2xRfc1v1Gccsdk-2-4

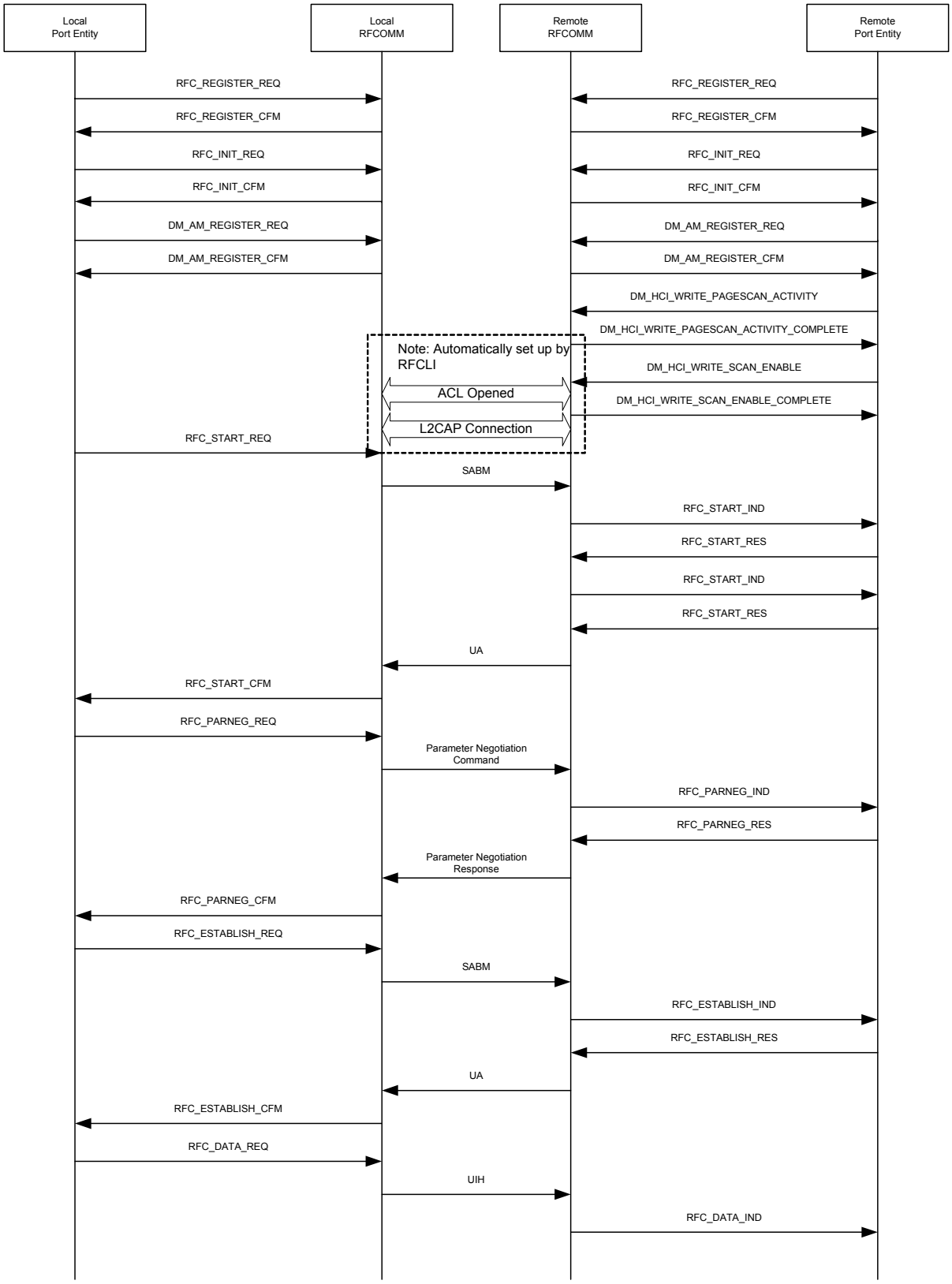


Figure 5.3: Message Sequence Chart For RFCOMM Data Link Set Up Between Two BlueCore Devices

Using Configuration Two in Figure 5.2 and the two TCL source scripts for the local and remote devices listed in Appendix A1 and A2 respectively the interactions of the message sequence chart in Figure 5.3 can be demonstrated and observed. In order to understand the complete example the two scripts can be broken down into key stages as follows:

1. The remote device must be connected to its respective RFCLI session and set up any global variables required, this is not shown in Figure 5.3, the script for the remote device is as follows:

```
#Slave script
#Connect to Casira
puts "Connect to Casira"
BC_connect com2 bcsp 115200
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set max_frame_size 0x7f
set initial_credits 0x07
```

2. The local device must be connected to its respective RFCLI session, this is not shown in Figure 5.3. In the local script the LAP, UAP and NAP must be set up for the remote device, so these three lines will need to change to reflect the address of the device on COM2. The script for local device is as follows:

```
#Master script
#Connect to Casira
puts "Connect to Casira"
BC_connect com1 bcsp 115200
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set bd_addr.lap 0x10e46
set bd_addr.uap 0x5b
set bd_addr.nap 0x02
set max_frame_size 0x7f
set initial_credits 0x07
```

3. The local and the remote device port entities register with RFCOMM. For both port entities this requires sending a RFC_REGISTER_REQ along with a port handle (phandle) to the RFCOMM layer and waiting for a RFC_REGISTER_CFM to be returned. The phandle is usually set to 0x8000 which is the default setting. In the local and remote script the script is the same and the default setting for phandle is used, the script is as follows:

```
#Register with RFCOMM
puts "Register with RFCOMM"
RFC_REGISTER_REQ $phandle
RFC_REGISTER_CFM
```

   The output on RFCLI expected in both the local and remote device RFCLI session is:

```
---- 15:26:56.055 ------------------
RFC_REGISTER_REQ_T
        type =  03
        phandle =  8000

---- 15:26:56.321 ------------------
RFC_REGISTER_CFM_T
        type =  04
        phandle =  8000
        server_chan =  01
        accept =  01
```

4. The local and remote port entities initialise RFCOMM by issuing a RFC_INIT_REQ primitive and wait for a RFC_INIT_CFM primitive to be returned. The RFC_INIT_REQ primitive takes the default parameters for the system except for flow control set up which was set up in step 1. The local and remote scripts are the same and are:

```
#Initialise RFCOMM
puts "Initialise RFCOMM"
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type
$fc_threshold $fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
```

The output on RFCLI expected in both the local and remote device RFCLI session is:

```
---- 15:26:56.337 ------------------
RFC_INIT_REQ_T
        type =  01
        phandle =  8000
        psm_local =  03
        use_flow_control =  01
        fc_type =  01
        fc_threshold =  03
        fc_timer =  01
        rsvd_4 =  00
        rsvd_5 =  00

---- 15:26:56.368 ------------------
RFC_INIT_CFM_T
        type =  02
        phandle =  8000
        psm_local =  03
        fc_type =  8000
        fc_threshold =  03
        fc_timer =  01
        rsvd_4 =  00
        rsvd_5 =  00
```

5. The local and remote port entities initialise the device manager by issuing a DM_AM_REGISTER_REQ primitive and wait for a DM_AM_REGISTER_CFM primitive to be returned. The port entities need to register with the device manager to allow functions such inquiry, inquiry scanning, paging and page scanning to occur. The default phandle value is used and the local and remote scripts are the same:

```
#Register with Device Manager
puts "Register with Device Manager"
DM_AM_REGISTER_REQ $phandle
DM_AM_REGISTER_CFM
```

The output on RFCLI expected in both the local and remote device RFCLI session is:

```
---- 15:26:56.399 ------------------
DM_AM_REGISTER_REQ_T
        type =  00
        phandle =  8000

---- 15:26:56.415 ------------------
DM_AM_REGISTER_CFM_T
        type =  01
        phandle =  8000
```

6. The remote port entity now places the remote device in listening mode by sending primitives to the device manager turning on page scanning and inquiry scanning, this allows the remote device to be found and connected to. At the end of this stage the remote device displays a message `Waiting for a connection` on the RFCLI session and waits for the local device to connect to it. The message sequence chart for this stage can be seen in Figure 5.4 and the script for the slave is as follows:

```
#Enabling page scanning so that the Master can connect to us
DM_HCI_WRITE_PAGESCAN_ACTIVITY 0x800 0x700
DM_HCI_WRITE_PAGESCAN_ACTIVITY_COMPLETE
DM_HCI_WRITE_SCAN_ENABLE 3
DM_HCI_WRITE_SCAN_ENABLE_COMPLETE
#Wait for a connection and respond appropriately
puts "Waiting for a connection"
```

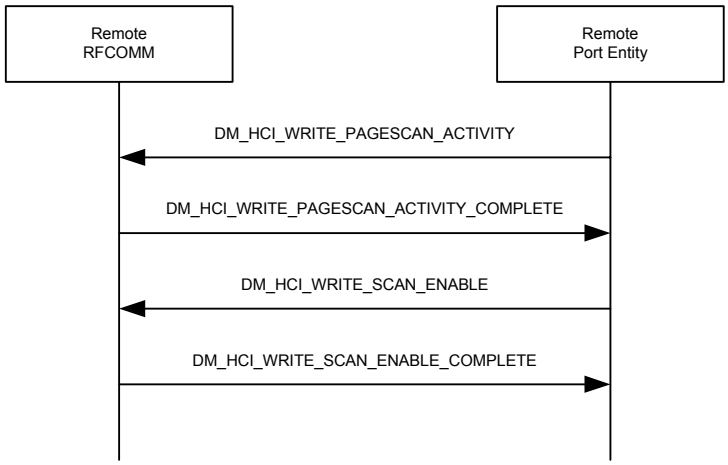


Figure 5.4: Message Sequence Chart to Place Remote Device in Listening Mode

The output on RFCLI expected is:

```
---- 15:26:56.415 ------------------
HCI_WRITE_PAGESCAN_ACTIVITY_T
        HCI_COMMAND_COMMON_T
                op_code =  c1c
                length =  04
        pagescan_interval =  800
        pagescan_window =  700

---- 15:26:56.430 ------------------
DM_HCI_STANDARD_COMMAND_COMPLETE_T
        type =  c53
        phandle =  8000
        status =  00

---- 15:26:56.446 ------------------
HCI_WRITE_SCAN_ENABLE_T
        HCI_COMMAND_COMMON_T
                op_code =  c1a
                length =  02
        scan_enable =  03

---- 15:26:56.462 ------------------
DM_HCI_STANDARD_COMMAND_COMPLETE_T
        type =  c51
        phandle =  8000
        status =  00
```

7. Once the remote device is listening the local device issues a RFCOMM start command by sending a RFC_START_REQ primitive. The local device script has been written so that it sits in a loop waiting for the RFC_START_CFM to be received. Usually the script returns a pending status for the first RFC_START_CFM, so the local device needs to wait for a success status to be returned as a `result_code` before it can continue. At this point the first over air message will be seen when the first SABM frame is sent from the local RFCOMM layer to the remote RFCOMM layer with the DLCI = 0
The script for the local device is:

```
#Request RFCOMM Start
puts "Request RFCOMM Start"
RFC_START_REQ - - - $psm_remote ${sys_pars.port_speed}
${sys_pars.max_frame_size} $respond_phandle
RFC_START_CFM
#If result_code is not success we need to wait
while {$result_code == 1} {
  RFC_START_CFM
}
```

The output on RFCLI expected is shown below, notice that an ACL connection has been opened automatically and the remote features of the remote device read without user intervention:

```
---- 15:28:54.380 ------------------
RFC_START_REQ_T
        type =  05
        BD_ADDR_T
                lap =  10e46
                uap =  5b
                nap =  02
        psm_remote =  03
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  7f
        respond_phandle =  8000

---- 15:28:57.020 ------------------
DM_ACL_OPENED_IND_T
        type =  280d
        phandle =  8000
        BD_ADDR_T
                lap =  10e46
                uap =  5b
                nap =  02
        incoming =  00
        dev_class =  00

---- 15:28:57.036 ------------------
RFC_START_CFM_T
        type =  08
        phandle =  8000
        BD_ADDR_T
                lap =  10e46
                uap =  5b
                nap =  02
        mux_id =  00
        result_code =  01
        SYS_PAR_T
                port_speed =  00
                max_frame_size =  00
```

```
---- 15:28:57.067 ------------------
DM_HCI_READ_REMOTE_FEATURES_COMPLETE_T
        type =  42a
        phandle =  8000
        status =  00
        BD_ADDR_T
                lap =  10e46
                uap =  5b
                nap =  02
        features [c] =  ffff
        features [c] =  0f
        features [c] =  00
        features [c] =  00

---- 15:28:57.223 ------------------
RFC_START_CFM_T
        type =  08
        phandle =  8000
        BD_ADDR_T
                lap =  10e46
                uap =  5b
                nap =  02
        mux_id =  00
        result_code =  00
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  7f
```

8. The remote device in response to the SABM will see a couple of RFC_START_IND primitives at the remote port entity to indicate that RFCOMM has started. To each indication primitive the remote port entity needs to respond with a RFC_START_RES primitive to the remote RFCOMM layer. Having responded correctly the RFCOMM layer will issue an UA frame to the local RFCOMM layer and issue a RFC_STARTCMP_IND to indicate the RFCOMM start has completed. The remote script is:

```
RFC_START_IND
RFC_START_RES
RFC_START_IND
RFC_START_RES
RFC_STARTCMP_IND
```

The output on RFCLI expected is shown below, as was noticed in the local device the remote device also has an ACL connection opened automatically and the remote features of the local device are read without user intervention:

```
---- 15:28:57.036 ------------------
DM_ACL_OPENED_IND_T
        type =  280d
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        incoming =  01
        dev_class =  00
```

```
---- 15:28:57.067 ------------------
RFC_START_IND_T
        type =  07
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        mux_id =  00
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  00

---- 15:28:57.098 ------------------
RFC_START_RES_T
        type =  06
        mux_id =  00
        accept =  01
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  00
        respond_phandle =  8000

---- 15:28:57.098 ------------------
DM_HCI_READ_REMOTE_FEATURES_COMPLETE_T
        type =  42a
        phandle =  8000
        status =  00
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        features [c] =  ffff
        features [c] =  0f
        features [c] =  00
        features [c] =  00

---- 15:28:57.145 ------------------
RFC_START_IND_T
        type =  07
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        mux_id =  00
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  7f

---- 15:28:57.161 ------------------
RFC_START_RES_T
        type =  06
        mux_id =  00
        accept =  01
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  7f
        respond_phandle =  8000
```

```
---- 15:28:57.208 ------------------
RFC_STARTCMP_IND_T
        type =  09
        phandle =  8000
        mux_id =  00
        result_code =  00
        SYS_PAR_T
                port_speed =  ff
                max_frame_size =  7f
```

9. The parameter negotiation stage sets the link to use credit based flow control, with an initial credit size of 7 and maximum frame size for RFCOMM packets of 0x7f hex. These parameters are set by the local port entity by issuing a RFC_PARNEG_REQ primitive and waits for the remote device parameters, which are contained in the RFC_PARNEG_CFM. Once the local RFCOMM layer receives the RFC_PARNEG_REQ primitive from the local port entity it issues an over air parameter negotiation command over the air as a control message on DLCI = 0. The script for the local device is:

```
#RFCOMM Parameter Negotiation
puts "RFCOMM Parameter Negotiation"
RFC_PARNEG_REQ $mux_id - - $max_frame_size $use_flow_control
$initial_credits
RFC_PARNEG_CFM
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.255 ------------------
RFC_PARNEG_REQ_T
        type =  20
        mux_id =  00
        loc_server_chan =  01
        rem_server_chan =  01
        DLC_PAR_T
                max_frame_size =  7f
                credit_flow_ctrl =  01
                initial_credits =  07

---- 15:28:57.333 ------------------
RFC_PARNEG_CFM_T
        type =  23
        phandle =  8000
        mux_id =  00
        server_chan =  01
        DLC_PAR_T
                max_frame_size =  7f
                credit_flow_ctrl =  01
                initial_credits =  07
```

10. The parameter negotiation stage in the remote device takes in the set of parameters it receives from the local device and uses these in its response via the over air parameter negotiation frame. If the remote device requires a smaller frame size or needed to set a different value for initial credits these parameters would need to be set in the RFC_PARNEG_RES primitive. The script for the remote device is:

```
RFC_PARNEG_IND
RFC_PARNEG_RES
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.302 ------------------
RFC_PARNEG_IND_T
        type =  22
        phandle =  8000
        mux_id =  00
        server_chan =  01
        DLC_PAR_T
                max_frame_size =  7f
                credit_flow_ctrl =  01
                initial_credits =  07

---- 15:28:57.302 ------------------
RFC_PARNEG_RES_T
        type =  21
        mux_id =  00
        server_chan =  01
        DLC_PAR_T
                max_frame_size =  7f
                credit_flow_ctrl =  01
                initial_credits =  07
```

11. When the parameter negotiation stage is complete the control channel is completely set up. Once the control channel is fully available a data channel can then be established between the two BlueCore devices attached to COM1 and COM2 of the PC. This is done using the RFC_ESTABLISH_REQ primitive that in this example uses default parameters for items such as the multiplexor indentifier and the local and remote server channels. At this point another SABM frame is issued but this time on a different DLCI channel, here DLCI is equal to 1. The local port entity will wait for the RFCOMM layer to receive a UA frame on DLCI = 1 to confirm the link is established. Once the local RFCOMM layer receives the UA frame it issues the RFC_ESTABLISH_CFM primitive that the local port entity is expecting. The script for the local device is:

```
#RFCOMM Establish
puts "RFCOMM Establish"
RFC_ESTABLISH_REQ $mux_id $loc_server_chan $rem_server_chan
RFC_ESTABLISH_CFM
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.364 ------------------
RFC_ESTABLISH_REQ_T
        type =  0c
        mux_id =  00
        loc_server_chan =  01
        rem_server_chan =  01

---- 15:28:57.427 ------------------
RFC_ESTABLISH_CFM_T
        type =  0f
        phandle =  8000
        mux_id =  00
        server_chan =  01
        result_code =  00
```

12. The remote device in the data channel establishment phase receives the SABM frame on DLCI = 1 and passes the RFC_ESTABLISH_IND primitive to the remote port entity. The remote port entity responds with a RFC_ESTABLISH_CFM using default parameters for the multiplexor identifier and server channel and also accepts the link. An UA frame is then sent over the air to accept the link. The remote script is:

```
RFC_ESTABLISH_IND
RFC_ESTABLISH_RES
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.395 ------------------
RFC_ESTABLISH_IND_T
        type =  0e
        phandle =  8000
        mux_id =  00
        server_chan =  01

---- 15:28:57.395 ------------------
RFC_ESTABLISH_RES_T
        type =  0d
        mux_id =  00
        server_chan =  01
        accept =  01
```

13. Once the data link is set up on DLCI = 1 the local device transmits two message to the remote device using the RFC_DATA_REQ primitive. The first message is the numbers 1 to 9 and the second message is “Hello World”. After these two messages have been sent the example script is complete but the data link is left up. The script is:

```
#Send a data primitive
puts "Connection made, starting transfer"
RFC_DATA_REQ - - 0 ? {1 2 3 4 5 6 7 8 9}
#Send another primitive
RFC_DATA_REQ - - 0 ? {"Hello World"}
puts "All done"
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.442 ------------------
RFC_DATA_REQ_T
        type =  18
        mux_id =  00
        server_chan =  01
        credits =  00
        payload_length =  09
 01 02 03 04 05 06 07 08 09

---- 15:28:57.458 ------------------
RFC_DATA_REQ_T
        type =  18
        mux_id =  00
        server_chan =  01
        credits =  00
        payload_length =  0b
 48 65 6c 6c 6f 20 57 6f 72 6c 64
All done
```

14. The remote device once the data link is set up on DLCI = 1 waits for two messages from the local device. Each message will be carried within a RFC_DATA_IND primitive. When each message has been received it displays the payload of the RFCOMM data packet. the local device transmits two message to the remote device. The first message is the numbers 1 to 9 and the second message is “Hello Word”. After these two messages have been received the example script is complete but the data link is left up. The script is:

```
#Wait for an incoming data primitive
puts "Connection made, starting transfer"
set result [RFC_DATA_IND]
#Wait for another incoming data primitive
puts "Received payload: [lindex $result 5]"
set result [RFC_DATA_IND]
puts "Received payload: [lindex $result 5]"
puts "All done"
```

The output on RFCLI expected is shown below:

```
---- 15:28:57.536 ------------------
RFC_DATA_IND_T
        type =  19
        phandle =  8000
        mux_id =  00
        server_chan =  01
        credits =  00
        payload_length =  09
 01 02 03 04 05 06 07 08 09

Received payload: 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09

---- 15:28:57.552 ------------------
RFC_DATA_IND_T
        type =  19
        phandle =  8000
        mux_id =  00
        server_chan =  01
        credits =  00
        payload_length =  0b
 48 65 6c 6c 6f 20 57 6f 72 6c 64

Received payload: 0x48 0x65 0x6c 0x6c 0x6f 0x20 0x57 0x6f 0x72 0x6c
0x64
All done
```

15. Whilst executing the two TCL scripts there will be an output on the RFCLI screen that is in the same form as the message shown below. This message is the FCT that is explained in Section 3.3:

```
---- 15:28:57.177 ------------------
RFC_DATA_IND_T
        type =  19
        phandle =  8000
        mux_id =  ff
        server_chan =  ff
        credits =  04
        payload_length =  00
```

5.4 Over Air Sniffer

To be able to verify that the correct RFCOMM frames are sent over the air between the local and the remote RFCOMM layers an over air sniffer is used. There are numerous sniffers available from manufacturers such as CATC™, Agilent™, Digianswer™ and Frontline™ to name just a few.

Section 5.3 was a step-by-step guide to setting up an RFCOMM link between two BlueCore devices using two Casiras and passing data from the local device to the remote device. It also shows with examples the outputs expected in RFCLI for each phase as re-assurance. Further assurance can be gained by the use of a sniffer. If the system being used to do the investigations is as per Configuration Two of Figure 5.2 then the sniffer being used can be configured as an over air sniffer or as a serial line sniffer. The over air sniffer has the ability to record the frames transmitted from the RFCOMM layer such as the SABM, UA, PN and UIH frames as well as other layers. Therefore the sniffer is being used here as a tool that can examine what is going on in various parts of the whole link and is especially useful in investigating when the link operates incorrectly. The sniffer used top capture data used in this section and Section 5.5 was the Frontline sniffer but the results could equally have been captured using another manufacturers sniffer. In order to set up a different manufacturers refer to their user guide and compare their set up against the Frontline Sniffer Quick Start User Guide available from CSR.

In this section the example in Section 5.3 is investigated using the sniffer to look at the over air RFCOMM frames If the over air RFCOMM frames are extracted from the captured over air trace, the RFCOMM packets captured can be compared to the message sequence chart in Figure 5.3 verifying correct set up of the RFCOMM link and transmission of expected data. Table 5.1 represents the RFCOMM packet sequence shown in Figure 5.3, it shows in sequential order the frame type and the contents of each RFCOMM frame that have been extracted. Note that the last two frames represent the two data messages - these two messages would be contained within UIH frames.

Frame Type	**RFCOMM Packet Details**
SABM	Role: Master Address: 1 Address: 03 DLCI: 00 Server Channel: 0 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Set Async Balanced Mode Poll/Final Bit: 1 Length Extension: Not Extended Length: 0 FCS: 1c
UA	Role: Slave Address: 1 Address: 03 DLCI: 00 Server Channel: 0 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Acknowledgement Poll/Final Bit: 1 Length Extension: Not Extended Length: 0 FCS: d7

Frame Type (continued)	RFCOMM Packet Details (continued)
UIH	Role: Master Address: 1 Address: 03 DLCI: 00 Server Channel: 0 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Info with Header Check Poll/Final Bit: 0 Length Extension: Not Extended Length: 10 UIH Command/Response: Command Type: Parameter Negotiation Command/Response: Command Type Extension: Not Extended Length Extension: Not Extended Length: 8 Parameter Negotiation: DLCI: 2 Credit Based Flow Control: Sender Supports CFC Type of Frame for Information: UIH Frames Priority: 0 Acknowledgement Timer: 0 Maximum Frame Size: 127 Maximum Number of Retransmission: 0 Initial Number of Credits: 7 FCS: 70

Frame Type (continued)	RFCOMM Packet Details (continued)
UIH	Role: Slave Address: 1 Address: 01 DLCI: 00 Server Channel: 0 Direction: Responder to Initiator Command/Response: Responder Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Info with Header Check Poll/Final Bit: 0 Length Extension: Not Extended Length: 10 UIH Command/Response: Command Type: Parameter Negotiation Command/Response: Response Type Extension: Not Extended Length Extension: Not Extended Length: 8 Parameter Negotiation: DLCI: 2 Credit Based Flow Control: Responder Supports CFC Type of Frame for Information: UIH Frames Priority: 0 Acknowledgement Timer: 0 Maximum Frame Size: 127 Maximum Number of Retransmission: 0 Initial Number of Credits: 7 FCS: aa
SABM	Role: Master Address: 1 Address: 0b DLCI: 01 Server Channel: 1 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Set Async Balanced Mode Poll/Final Bit: 1 Length Extension: Not Extended Length: 0 FCS: 59

Frame Type (continued)	RFCOMM Packet Details (continued)
UA	Role: Slave Address: 1 Address: 0b DLCI: 01 Server Channel: 1 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Acknowledgement Poll/Final Bit: 1 Length Extension: Not Extended Length: 0 FCS: 92
UIH	Role: Master Address: 1 Address: 0b DLCI: 01 Server Channel: 1 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Info with Header Check Poll/Final Bit: 0 Length Extension: Not Extended Length: 9 FCS: 9a
UIH	Role: Master Address: 1 Address: 0b DLCI: 01 Server Channel: 1 Direction: Responder to Initiator Command/Response: Initiator Started C/R Sequence Extension Bit: Not Extended Frame Type: Unnumbered Info with Header Check Poll/Final Bit: 0 Length Extension: Not Extended Length: 11 FCS: 9a

Table 5.1: RFCOMM Packets Extracted from Over Air Sniifer

5.5 Serial Sniffer

To complete the analysis of the set up of the RFCOM link, a serial port sniffer can be used to monitor the serial port traffic and some sniffers have the ability to decode BCSP packets. This means that the traffic on the serial ports for both Casiras on COM1 and COM2 for Configuration Two in Figure 5.2 can be monitored. The serial line sniffer used to capture the data in this section can only record the serial packets on one BCSP serial link at a time, not both.

The RFCOMM packets that are carried along as BCSP packets on the serial ports of the local and remote devices represent the RFCOMM primitives sent from each port entity to their corresponding RFCOMM layer. In order to set up the serial port sniffer refer to manufacturer's user manual and compare against Frontline Sniffer Quick Start User Guide for reference.

Once data has been captured for the serial port for the connection between the port entity and the RFCOMM layer on both the local and the remote devices. The captured data can be analysed to check that the packets captured correspond to the primitives expected in Figure 5.3.

For example the frame shown below was taken from the file that captured the serial traffic between the local port entity and the RFCOMM layer on the local BlueCore device:

```
--- Frame 23 (DTE)--- Length: 11 --- Errors: 0 --- Time: 24/09/2002
17:16:27.471  ---

Summary Header Labels
---------------------
BlueCore Serial Protocol,Ack,Seq,Length,Channel

Layer Summaries
---------------
BlueCore Serial Protocol,0,0,4,RFCOMM

Protocol Decodes
----------------
Physical Frame:
      db dc 49 00 f6 03 00 00 80 da 8a

BlueCore Serial Protocol (logical frame):
      c0 49 00 f6 03 00 00 80 da 8a

BlueCore Serial Protocol:
      Flags:
            Protocol Type: Reliable Datagram Stream
            CRC Present: Yes
            Ack: 0
            Sequence: 0
      Payload Length: 4
      Channel: RFCOMM
      Checksum: f6
      CRC: da8a

Data:
      Hex: 03 00 00 80
      ASCII: .  .
```

This frame represents the RFC_REGISTER_REQ primitive shown in Figure 5.3 that is sent from the local port entity and the local RFCOMM layer. This can be decoded from the data section of the frame, which has the four hex values of 03, 00, 00 and 80. As the data is 16bit and in Little Endian format the two data values then become the hex values of 0x0003 and 0x8000. The first 16bit value represents the RFCOMM primitive, where 0x0003 is the value for RFC_REGISTER_REQ. This value can be checked by looking in the header file **rfcomm_prim.h** included as part of BlueLab. As the RFC_REGISTER_REQ requires a 16bit parameter `phandle` then the hex value of 0x8000 represents this parameter.

6 Example of Accessing RFCOMM Using RFCLI and TCL

As stated previously one of the layers accessible using RFCLI is the RFCOMM layer. This section and the application note Accessing RFCOMM Using μBCSP cover examples of how to access the RFCOMM layer. The same examples used in this section and the application note Accessing RFCOMM Using μBCSP cover both access via RFCLI on a PC and via an external host using BCSP/μBCSP. These examples are based on a type 2 RFCOMM device depicted in Figure 2.1.

The example developed in this section is written in TCL and is designed to transfer a series of commands and data from a local device to a remote device over the RFCOMM layer in each peer device. The local device sends out a series of requests that may include data to the remote device, which will respond to the various commands with an answer to the local device. In this example the aspects of credit based flow control and the interaction with the FCL are highlighted if the options `-s` and `-e` are used when RFCLI is invoked. In order to use the FCL in a correct manner the use of a buffer is also included in this example.

To be able to run this example two Casiras loaded with BCSP version of RFCOMM firmware are required. The encryption type is not important therefore it may be convenient to use the latest RFCOMM firmware that is available in BlueLab.(1) Two RFCLI sessions must be running on a PC and the local device is the Casira attached to COM1 and the remote device is the Casira attached to COM2 of the PC as shown in Configuration Two in Figure 6.1. In order to run this example the following steps are required:

1. On one RFCLI session run the source script `rfcslv.tcl` on the remote device by entering the command

   ```
   source rfcslv.tcl
   ```

2. Wait for the message ‘Waiting for a connection’ to appear on the RFCLI session on the remote device before running a second session on RFCLI

3. On the second RFCLI session run the source script `rfcmst.tcl` on the local device by entering the command

   ```
   source rfcmst.tcl
   ```

4. The local device script will then attempt to connect to the remote device at RFCOMM layer the sequence of which is explained in Section 6.1.

5. Once a successful connection has been made between the local device and remote device commands, data is passed between the two devices and the example finishes.

Note:

(1) The RFCOMM firmware used in this example came from BlueLab 2.4. The version used was a BCSP build for BlueCore2. The build ID was 334 and its title was 2xRfc1v1Gccsdk-2-4

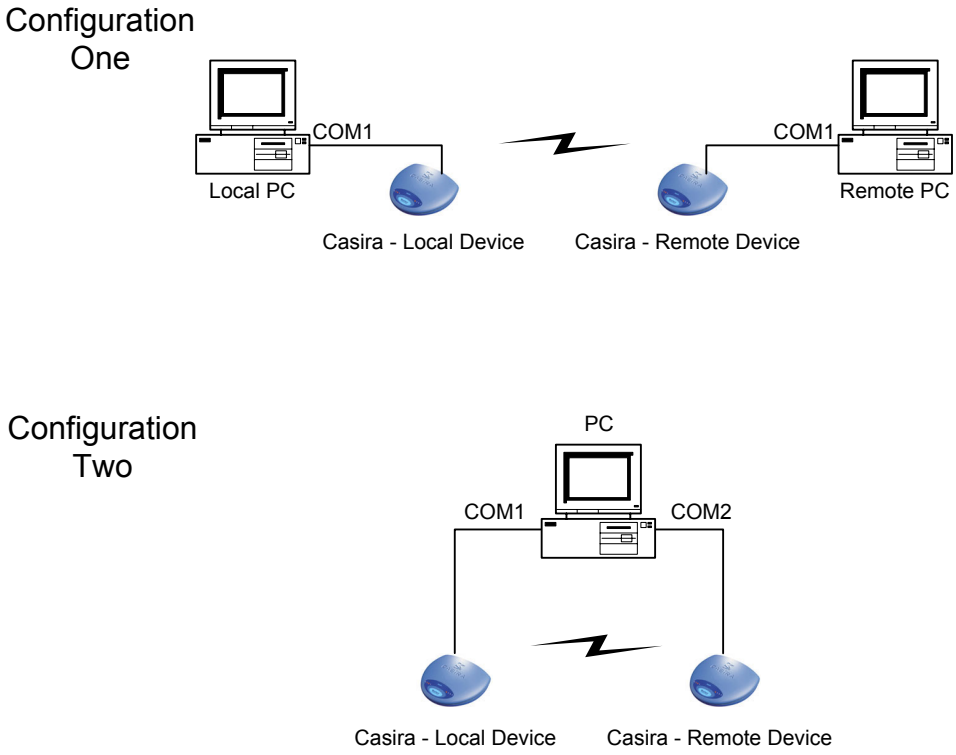


Figure 6.1: Set Up for Example

6.1 Details of the RFCLI Source Script Example

The following sections examine in detail the output from the RFCLI scripts `rfcmst.tcl` and `rfcslv.tcl`. The overall functionality of the system is described in terms of how the component parts interact. The main functionality of the example is contained within the local device and the majority of these are replicated and re-used inside the remote device script.

6.2 Local and Remote Device Source Script

The flow of functionality of the local device script breaks down into the two state transition diagrams Figure 6.2 and Figure 6.4. Figure 6.2 contains the various states that make up the initialisation of the local device. All these initialisation states shown in Figure 6.2 are contained within the state called 'Initialisation of RFCOMM Layer' in Figure 6.4. The script for the local device script `rfcmst.tcl` is contained in Appendix B1 and the comments clearly indicate where each state is within the script.

The remote device source script is very similar in structure to that of the local device source script and the subsections of this section will highlight the difference from the local device script file on initialisation of the RFCOMM layer and the main loop and state machine. The script for the remote device script `rfcslv.tcl` is contained in Appendix B2.

6.3 Initialisation of RFCOMM Layer in the Local Device

Figure 6.2 depicts the states that need to be transitioned in order to initialise the RFCOMM layer as well as the system variables and buffers.

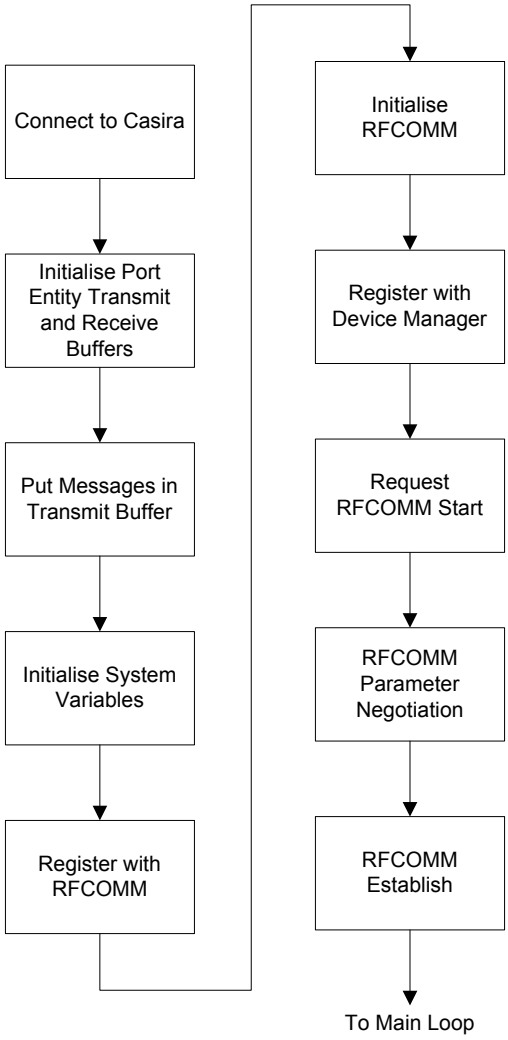


Figure 6.2: Initialisation of RFCOMM Layer

In the example script the initialisation of the local device is sequenced and each state completes before going onto the next state, at the end of the initialisation phase the local device will be connected to the remote device. If any state is unsuccessful either RFCLI causes the script to stop, or it may pause indefinitely waiting for a confirmation message. A description of each state of Figure 6.2 is described in Sections 6.3.1 to Section 6.3.10.

6.3.1 Connect to Casira

This state is very important, as it is responsible for connecting the RFCLI session running on the PC to the BlueCore device running on the Casira on serial port COM1. This BlueCore device represents the local device. This state issues a `BC_connect` command to connect to the Casira on serial COM1 at 115Kbaud and the script is shown below:

```
#Connect to Casira
puts "Connect to Casira"
BC_connect com1 bcsp 115200
```

This state completes when the `BC_connect` completes successfully, otherwise the script will end and no further states are executed.

6.3.2 Initialise Port Entity Transmit and Receive Buffers

This state initialises the transmit and receive buffers and fills them with the value `0xc0ffee`, sets up the pointers for the top and bottom of each buffer, the number of packets and buffer size. The script for both local and the remote device is:

```
#Initialise Port Entity Transmit Buffer
puts "Initialise Port Entity Transmit Buffer"
set tx_buffer(noofpackets) 0
set tx_buffer(txcredits) 0
set tx_buffer(pointer) 0
set tx_buffer(buffersize) 10
set tx_buffer(topptr) 0
set tx_buffer(bottomptr) 0
set tx_buffer(transmitted) 0
for {set cnt 0} {$cnt < $tx_buffer(buffersize)} {incr cnt} {
  set tx_buffer($cnt) 0xc0ffee
}
#Initialise Port Entity Receive Buffer
puts "Initialise Port Entity Receive Buffer"
set rx_buffer(noofpackets) 0
set rx_buffer(txcredits) 0
set rx_buffer(pointer) 0
set rx_buffer(topptr) 0
set rx_buffer(bottomptr) 0
set rx_buffer(buffersize) 10
set rx_buffer(received) 0
for {set cnt 0} {$cnt < $rx_buffer(buffersize)} {incr cnt} {
  set rx_buffer($cnt) 0xc0ffee
}
```

The state will transition in to the next state when all the space inside each buffer inside transmit and receive buffers are initialised. The size of each buffer is set by the `buffersize` for each buffer.

6.3.3 Put Messages in Transmit Buffer

This script forms part of the initialisation of the local device script only. It demonstrates how to load the complete transmit buffer. It makes use of the Load Packet in Transmit Buffer in Section 1.1.1 and loads all slots in the buffer dictated by the buffer size value stored in `tx_buffer(buffersize)`. In this script the `tx_buffer(buffersize)` is set ten so ten messages are loaded into the transmit buffer to be sent to the remote device. The data to be loaded in the buffer is first loaded into the variable `txmsg` and this is then placed in the transit buffer by calling the Load Packet in Transmit Buffer state:

```
#Put Messages in Transmit Buffer
puts "Put Messages in Transmit Buffer"
for {set cnt 0} {$cnt < $tx_buffer(buffersize)} {incr cnt} {
   switch $cnt {
      0 {set txmsg {1}}
      1 {set txmsg {2}}
      2 {set txmsg {3 "Name changed by RFCLI"}}
      3 {set txmsg {1}}
      4 {set txmsg {2}}
      5 {set txmsg {3 "Name changed by RFCLI"}}
      6 {set txmsg {1}}
      7 {set txmsg {2}}
      8 {set txmsg {3 "Name changed by RFCLI"}}
      9 {set txmsg {0}}
      default {set txmsg {0}}
   }
   if {[putpktpetxbuffer $txmsg]} {
      puts "Message failed to load in PE Tx Buffer"
   }
}
```

This state will transition on to the next state when the transmit buffer is completely loaded with all messages.

6.3.4 Initialise System Variables

This state initialises variables such as the Bluetooth address of the remote device that is being connected to and the maximum frame size and initial credits. The script is below and the variables `bd_addr.lap`, `bd_addr.uap` and `bd_addr.nap` represent the lap, uap and nap respectively of the Bluetooth address of the remote device, these will need to be modified depending on the address of the user's remote device:

```
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set bd_addr.lap 0x10e46
set bd_addr.uap 0x5b
set bd_addr.nap 0x02
set max_frame_size 0x7f
set initial_credits 0x07
set state 1
puts "Connection Attempts: $connectionAttempts"
```

The state transitions when all variables are set.

6.3.5 Register with RFCOMM

This state registers the local device with RFCOMM layer and waits for confirmation. The registration is done using the routine `register_rfcomm` that is documented in Section 6.7.2 to carry out this function complete:

```
#Register with RFCOMM
puts "Register with RFCOMM"
register_rfcomm
```

Receipt of RFC_REGISTER_CFM within the `register_rfcomm` confirms registration of RFCOMM has completed.

6.3.6 Initialise RFCOMM

This state initialises the RFCOMM connection by setting the flow control and type to credit based flow control in the RFC_INIT_REQ primitive and issues the primitive. The script is

```
#Initialise RFCOMM
puts "Initialise RFCOMM"
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type $fc_threshold
$fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
```

With the introduction of the FCL as documented in Section 3.3 most of the fields in the RFC_INIT_REQ are now redundant. However, two still have meaning:

1. `tx_credits_issue_threshold`, this controls how often the chip will issue an FCT i.e., when the number of unconsumed primitives on-chip passes this threshold, and FCT will be emitted.
2. `tx_credits_issue_timer`, sets the number of microseconds between checks that the number of unconsumed primitives on-chip has changed. If this number has changed, an FCT will be emitted regardless of the setting of tx_credits_issue_threshold.

The state transitions when the receipt of RFC_INIT_CFM to confirm initialisation of RFCOMM has completed.

6.3.7 Register with Device Manager

This state registers with device manager allowing for options such as inquiry, security manager primitives and specific device manager primitives to be available. The script is:

```
#Register with Device Manager
puts "Register with Device Manager"
puts "DM phandle: $phandle"
DM_AM_REGISTER_REQ $phandle
DM_AM_REGISTER_CFM
```

The state transitions when the receipt of DM_AM_REGISTER_CFM to confirm registration of device manager has completed.

6.3.8 Request RFCOMM Start

This state requests to start RFCOMM with port speed and maximum frame size, it calls the `start_request` function that is documented in Section 6.7.1. The script is:

```
#Request RFCOMM Start
puts "Request RFCOMM Start"
start_request ${sys_pars.port_speed} ${sys_pars.max_frame_size}
```

The state transitions when the receipt of RFC_START_CFM inside the `start_request` function confirms that RFCOMM has started.

6.3.9 RFCOMM Parameter Negotiation

This state is the parameter negotiation state for RFCOMM, here is where the use flow control and the initial credits are set for the RFCOMM link and the maximum frame size is negotiated for the link. The script is:

```
#RFCOMM Parameter Negotiation
puts "RFCOMM Parameter Negotiation"
RFC_PARNEG_REQ $mux_id - - $max_frame_size $use_flow_control
$initial_credits
RFC_PARNEG_CFM
puts "Credits: ${dlc_pars.initial_credits}"
```

The state transitions when the receipt of RFC_PARNEG_CFM occurs to confirm the RFCOMM parameter negotiation has completed.

6.3.10 RFCOMM Establishment

This state establishes a RFCOMM connection with the remote device identified with a specific mux id, server channel and remote server channel. The script is:

```
#RFCOMM Establish
puts "RFCOMM Establish"
RFC_ESTABLISH_REQ $mux_id $loc_server_chan $rem_server_chan
RFC_ESTABLISH_CFM
```

The state transitions when the receipt of RFC_ESTABLISH_CFM to confirm RFCOMM has occurred. In this overall system example the RFC_ESTABLISH_CFM is issued once the local device is connected to the remote device that is in the Connect as Slave outlined in Section 6.4.9.

6.4 Initialisation of the RFCOMM Layer of the Remote Device

The initialisation of the RFCOMM layer for the remote device script is shown in Figure 6.3, the differences between the initialisation of the local device shown in Figure 6.2 are:

- The transmit buffer is not loaded with messages during the initialisation phase, the buffer is loaded during the main loop
- The Listen state, enables page scanning so the local device can connect to it
- Output the “Waiting For Connection Message” to indicate that the local device script can be run
- Connect as Slave, sits here until the local device has established a connection with the remote device

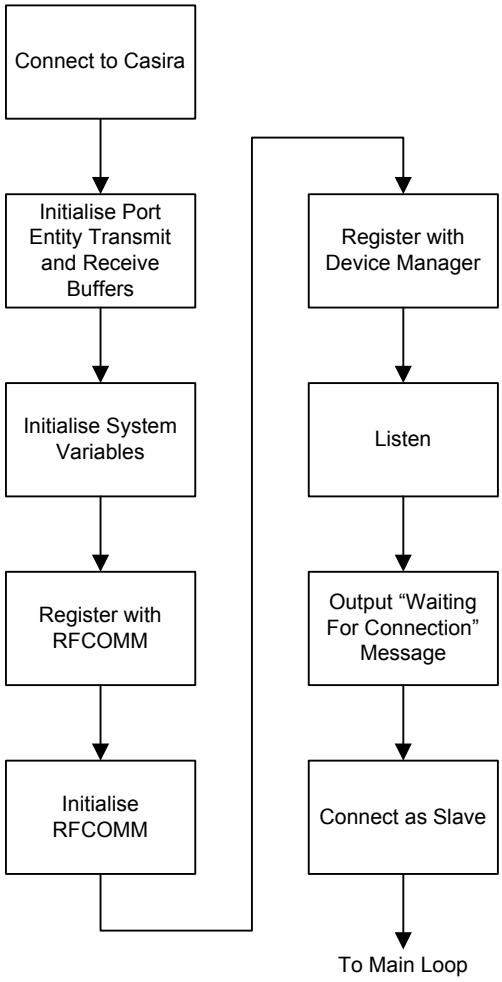


Figure 6.3: Initialisation of the RFCOMM Layer of The Remote Device

In the example script the initialisation of the remote device is in sequence and each state completes before going onto the next state, at the Connect as Slave state of the initialisation phase the remote device will be waiting to be connected to the local device. Once the Connect as Slave is exited the remote and local device will have a RFCOMM connection. If any state is unsuccessful either RFCLI causes the script to stop, or it may pause indefinitely waiting for a confirmation message. A description of each state of Figure 6.3 is documented in the Section 6.4.1 to Section 6.4.9.

6.4.1 Connect to Casira

This state is very similar to the local device script for this state as shown in Section 6.3.1. It is responsible for connecting the RFCLI session running on the PC to the BlueCore device running on the Casira on serial port COM2. This BlueCore device represents the remote device. This state issues a `BC_connect` command to connect to the Casira on serial COM2 at 115Kbaud and the script is shown below:

```
#Connect to Casira
puts "Connect to Casira"
BC_connect com1 bcsp 115200
```

This state completes when the `BC_connect` completes successfully, otherwise the script will end and no further states are executed.

6.4.2 Initialise Port Entity Transmit and Receive Buffers

For the remote device this state is exactly the same as the local device outlined in Section 6.3.2.

6.4.3 Initialise System Variables

This state Initialises the system variables such as the maximum frame size and initial credits for the remote device. The script is:

```
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set max_frame_size 0x7f
set initial_credits 0x07
set state 1
set taskcompleteflag 0
```

The state transitions when all variables are set.

6.4.4 Register with RFCOMM

For the remote device this state is exactly the same as the local device outlined in Section 6.3.5.

6.4.5 Initialise RFCOMM

For the remote device this state is exactly the same as the local device outlined in Section 6.3.6.

6.4.6 Register with Device Manager

For the remote device this state is exactly the same as the local device outlined in Section 6.3.7.

6.4.7 Listen

This state makes a call to the library command `listen` that enables the inquiry scan and page scan so that remote device is discoverable and connectable. The script is:

```
#Listen
listen
```

The state transitions once all scanning is turned on.

6.4.8 Output “Waiting For Connection” Message

This state displays a “Waiting For Connection” Message output to indicate that remote device is waiting for local device to establish a connection. The script is;

```
#Output Waiting For Connection Message
puts "Waiting For Connection Message"
```

The state transitions once the message is displayed.

6.4.9 Connect as Slave

This state calls the library command `rfc_connect_slv`, which waits for a RFCOMM connection to be established from the local device. The script is:

```
#Connect as Remote
rfc_connect_slv
#Wait for an incoming data primitive
puts "Connection made, starting transfer"
```

Whilst the remote device is in the Connect as Slave state it will not transition out of this state until the receipt of the RFC_STARTCMP_IND within the `rfc_connect_slv` library function. The RFC_STARTCMP_IND will be issued once the local device has completed the RFCOMM Establish state described in Section 6.3.10. At the end of the Connect as Slave state the RFCOMM link is complete.

6.5 Main Loop and State Machine

Once the local device has completed its initialisation phase and is connected to the remote device it then enters the main loop and transitions through the states outlined in Figure 6.4. At the heart of the script is essentially a small state machine that has five main states, which are:

1. State 0, all tasks are complete and the script is complete
2. State 1, steady state
3. State 2, the transmit buffer is checked and packets are sent
4. State 3, packets are received
5. State 4, received packets are processed

As shown in Figure 6.4 states 2 and 3 both transition through the Flow Control Layer, in RFCLI the Flow Control Layer is just a simple switch to pass control either to the transmitter or receiver functions. This is due to the fact that RFCLI handles the Flow Control Layer of the credit based flow control function. In application note Accessing RFCOMM using BCSP/μBCSP the Flow Control Layer is more complex when implemented with BCSP/μBCSP, and this needs to be explicitly written. The Flow Control Layer block is drawn Figure 6.4 so that this state transition diagram applies to both this example that uses RFCLI and the one that uses BCSP/μBCSP.

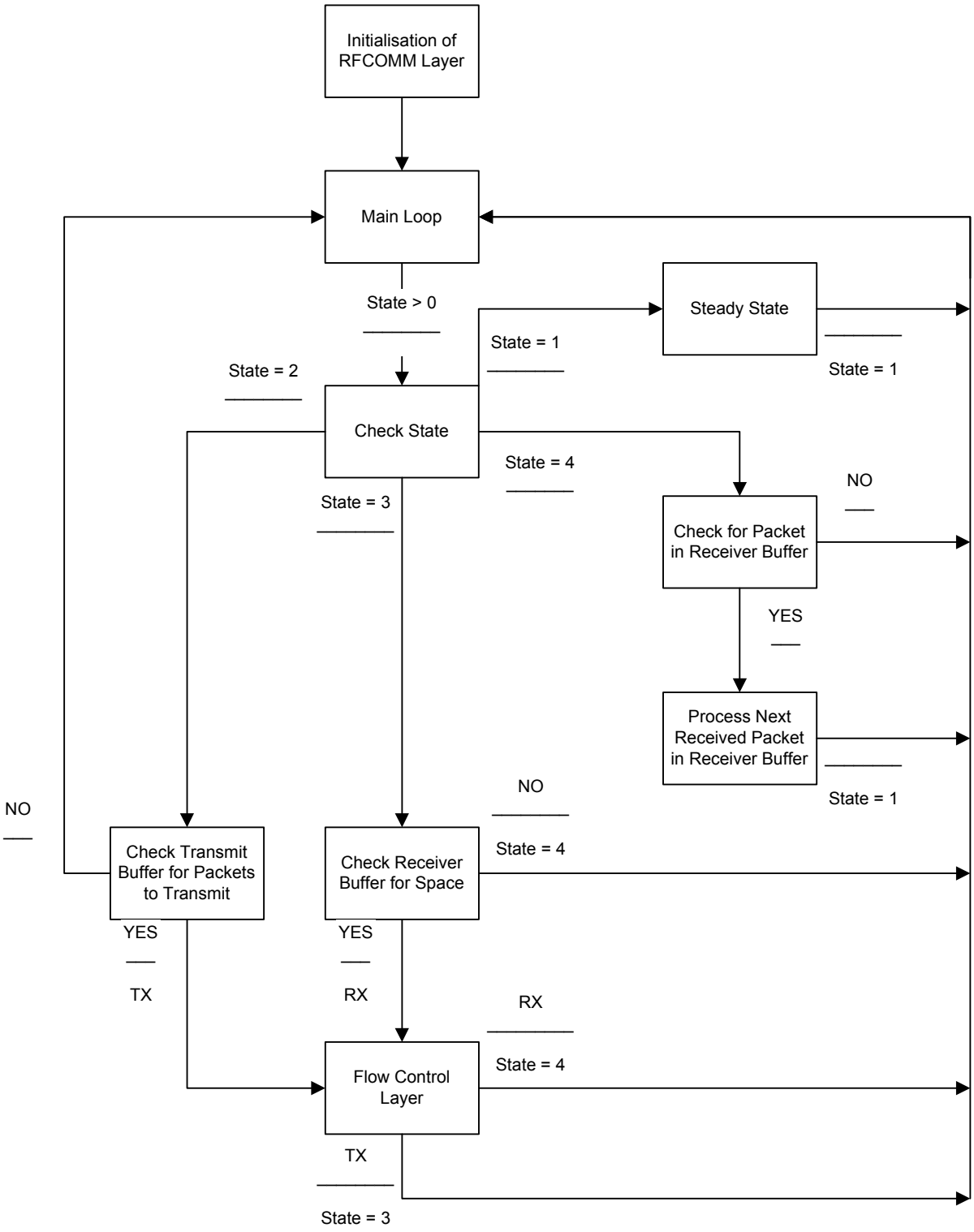


Figure 6.4: Main Control Loop

Table 6.1 describes the various states within the main control loop outlined by Figure 6.4 and highlights what causes the transition between the states and which state it goes onto. The ideal behind the main control loop is to:

1. Transition between state 1, 2, 3 and 4 in sequence whilst there is packets to transmit, packets to receive or received packets to process.
2. The packets stored within the transmit buffer have simple commands that the remote device must respond to, these commands are:
    - 1, this causes the remote device to carry out a read of its own Bluetooth address and respond with 1 and the Bluetooth address to the local device
    - 2, this causes the remote device to carry out a read of its own local name and respond with a 2 plus the local name to the local device
    - 3, this causes the remote device to change its own local name to the name supplied to the local device and will then reply with 3 and the new name
    - 0, this causes the remote device to stop responding to messages as the task is complete, the remote device will then respond with 0 to signify completion
3. Once all packets are transmitted, received and processed the state is set to 0 and the main control is exited and the example is complete.

State	Description	Cause of State Transition	Next State
Main Loop	The start of the main control loop	State >0	Check State
Check State	Checks the state value to see whether needs to transmit, receive or process a packet	State > 0	If state = 2 then Check Transmit Buffer for Packets to Transmit If state = 3 then Check Receiver Buffer for Space If state = 4 then Check for Packet in Receiver Buffer
Check Transmit Buffer for Packets to Transmit	See if there are any packets to transmit to the remote device	No packets or packets to transmit	No packets, return to Main Loop Packets, transition to Flow Control Layer
Flow Control Layer	Decides to call the actual RFCOMM data transmission or reception procedures depending on whether it receives a TX for transmit or RX for receive	Data transmitted or received	Return to Main Loop
Check Receiver Buffer for Space	Does not allow data to be received if space is not available in the receive buffer.	Either space available in receive buffer or not	No space, return to Main Loop Space, transition to Flow Control Layer
Check for Packet in Receiver Buffer	Examines the receiver buffer to see if there are packets to be processed	No receive packet available, or a receive packet to be processed	No packet, return to Main Loop. Packet available, transition to Process Next Received Packet in Receiver Buffer
Process Next Received Packet in Receiver Buffer	Process and respond to the next packet in the receive buffer.	Receive packet has been processed	Return to Main Loop

Table 6.1: Main Control Loop State Transitions

The general structure of the main loop and state machine of the remote device is similar to the local device script shown in Figure 6.4. The script for the Main control loop itself in master device is:

```
#Main Loop
puts "Main Loop"
while {$state} {
  checkstate
}
puts "All Done"
puts "Number of Transmitted Packets = $tx_buffer(transmitted)"
puts "Number of Received Packets    = $rx_buffer(received)"
```

The differences in the script of the remote device compared to the local device are as follows:

- The Process Next Received Packet in Receiver Buffer state has a different functionality to the local device it responds to the commands listed in Section 6.5.
- The transmit buffer is loaded with individual packets during the Process Next Received Packet in Receiver Buffer state and not during the initialisation phase
- The Main Loop is slightly different to the local device, the reason for this is that it needs to signal to the local device when it has received the last task complete message. The script is:

```
#Main Loop
puts "Main Loop"
while {$state} {
  if {$taskcompleteflag} {
    #Flush Transmit buffer to tell the Local to finish
    set state 2
    checkstate
    set state 0
  } else {
    checkstate
  }
}
puts "All Done"
puts "Number of Transmitted Packets = $tx_buffer(transmitted)"
puts "Number of Received Packets    = $rx_buffer(received)"
```

6.6 Main Loop Functions

The subsections within this section are here to describe how significant functions found in the local device and remote device scripts operate. Some of the functions within the local device and remote device are common to both and these are highlighted and compared within the individual functions, which may be unique to either the local device or the remote device but not both. All the functions that are examined primarily make up the calls from the main loop, some of the scripts of the initialisation phase are included.

These two functions are the same in both the local device and the remote device scripts, they can be described together as the Main Loop simply calls the Check State routine if the state is non zero. If the state is zero then the Main Loop finishes and displays the message “All Done”.

The Check State routine is the state machine that is described in Section 6.5 and calls the appropriate function depending on the state reached. The Check State function script is:

```
#Check State
proc checkstate {} {
   global state
   switch $state {
      0 {puts "Task Complete"}
      1 {steadystate}
      2 {if {[checkpetxbuffer]} {flowcontrollayer 0}}
      3 {if {[receiverspace]} {flowcontrollayer 1}}
      4 {if {[checkperxbuffer]} {processrx}}
      default {puts "Unknown State"}
   }
   incr state
   #Check to see if we have done all processing
   if {$state == 5} {
     if {[checkpetxbuffer] || [checkperxbuffer]} {
       set state 1
     } else {
       set state 0
     }
   }
}
```

The initial part of the function is a C type switch statement that has the states of the state machine as its options. Depending on the state the switch calls the appropriate task. In the case of when the state is 2 a check of the transmit buffer for packets is done. If there are packets then the flow control layer is called with a parameter of value 0 passed to it, this value 0 represents the TX parameter in Figure 6.4. In the case of when the state is 3 a check of the receive buffer for space is done and if there is space then the flow control layer is called with a parameter of value 1 passed to it, this value 1 represents the RX parameter in Figure 6.4.

After the switch statement the state is incremented to the next state and the boundaries of the new state are checked. If the last state has been reached, then there is a check of the transmitter and receiver buffers to see if there are any more packets that need to be transmitted or processed. If there are, then the next state is set to 1 and the state machine carries on, otherwise the state is set to 0 and this causes the main loop to finish.

The `steadystate` function that is called in this example has no real functionality. The state that accompanies this function has been added so that if required the addition of a sleep mode can be added at this point. The use of this state becomes more apparent when you use the μBCSP engine.

6.6.1 Check Transmit Buffer for Packets to Transmit

The script below is the state for Check Transmit Buffer for Packets to Transmit in Figure 6.4 in the local device and remote device and checks to see whether there are packets in the transmit buffer to transmit. All the function does is return the number packets in the buffer:

```
#Check Transmit Buffer for Packets to Transmit
proc checkpetxbuffer {} {
   global tx_buffer
   if {$tx_buffer(noofpackets) > 0x00} {
      puts "Port Entity has $tx_buffer(noofpackets) packet(s) to transmit"
      set pckcount $tx_buffer(noofpackets)
   } else {
      puts "Port Entity has no packets to transmit"
      set pckcount 0x00
   }
   return $pckcount
}
```

6.6.2 Check Receiver Buffer for Space

A simple script that returns a 1 if there is space in the receiver buffer to receive a packet or returns 0 if no space is available. The script in the local device and remote device is :

```
#Check Receiver Buffer for Space
proc receiverspace {} {
   global rx_buffer
   if {$rx_buffer(noofpackets) < $rx_buffer(buffersize)} {
     return 1
   } else {
     return 0
   }
}
```

6.6.3 Check for Packet in Receiver Buffer

A simple script that examines if there are packets within the receiver buffer, used to see whether there are packets to be processed. The script for the local device is:

```
#Check For Packet in Receiver Buffer
proc checkperxbuffer {} {
   global rx_buffer
   if {$rx_buffer(noofpackets) > 0x00} {
      puts "Port Entity has received $rx_buffer(noofpackets) packet(s)"
      set pckcount $rx_buffer(noofpackets)
   } else {
      puts "Port Entity has no receive packets"
      set pckcount 0x00
   }
   return $pckcount
}
```

6.6.4 Flow Control Layer

As RFCLI controls the flow control layer described in Section 3.3 then the Flow Control Layer state in Figure 6.4 is a relatively simple script in the local and remote devices its as follows:

```
#Flow Control Layer
proc flowcontrollayer {fclstate} {
   global state
   switch $fclstate {
      0 {transmittx}
      1 {receiverx}
      default {puts "Unknown FCL State"}
   }
}
```

This function calls the function to transmit or receive a packet depending on whether it had a parameter of TX or RX passed into it. The Flow Control Layer function is here primarily to allow the same state transition diagram Figure 6.4 to be used for the example based on RFCLI as well as BCSP/μBCSP. In the latter example the Flow Control Layer becomes more complex.

6.6.5 Process Next Received Packet in Receiver Buffer

This function sets a pointer to the top of the receiver buffer and loads the contents of packet into `result`, it then re-adjusts the buffer by decrementing the number of received packets, moving the pointer to point at the next message and wrapping the pointers if the bottom of the buffer is reached. Once the result is loaded the first character of the message payload is stripped off to get at the command. The command will cause the function to carry out 1 of for tasks:

- Indicate that data transfer has been completed
- Call the Read Bluetooth Address Message outlined in Section 6.7.5
- Call the Read Local Name Message outlined in Section 1.1.1
- Call the Change Local Name Message outlined in Section 1.1.1

The local device and remote device script are as follows:

```
#Process Next Received Packet in Receiver Buffer
proc processrx {} {
   global rx_buffer
   puts "Process receive buffer"
   #Get the Next Received packet to process
   set rxptr $rx_buffer(topptr)
   #Decrement the number of packets
   incr rx_buffer(noofpackets) -1
   #Increment the top of rx buffer pointer
   incr rx_buffer(topptr)
   #Check if top pointer needs to wrap around
   if {$rx_buffer(topptr) == $rx_buffer(buffersize)} {
      set rx_buffer(topptr) 0
   }
   #If number of packets is 0 then reset the both rx buffer pointers
   if {$rx_buffer(noofpackets) == 0} {
     set rx_buffer(topptr) 0
     set rx_buffer(bottomptr) 0
   }
   set result $rx_buffer($rxptr)
   puts "Payload: [lindex $result 5]"
   set firstchar [lindex [lindex $result 5] 0]
   puts "First character of payload $firstchar"
   switch $firstchar {
     0x00 {puts "Data transfer complete"}
     0x01 {readbdaddr [lindex $result 5]}
     0x02 {readlocalname [lindex $result 5] [lindex $result 4]}
     0x03 {changelocalname [lindex $result 5] [lindex $result 4]}
   }
}
```

6.6.6 Steady State

At present the example script for the Steady Sate for the local and remote device does nothing but is available as a hook if functionality at a later date is required. Added functionality may include sleep modes.

6.7 Other Functions

This section documents functions or useful sections of TCL script that do not appear on the Main Control Loop diagram in Figure 6.4. The reason they may not appear is that either they are internal to a state in the Main Control Loop or they are part of the initialisation phase for the local or remote device.

6.7.1 Start Request

This state initialises a logical connection between devices with suggested parameters for the port speed and the maximum frame size of a RFCOMM packet. In the example port speed is set to `0xff`, which means that this parameter is unused and the maximum frame size is left as the default size of 127bytes. The script for both the local and remote device is:

```
#Start Request
proc start_request {port_speed max_frame_size} {
     global result_code connectionAttempts
     global psm_remote respond_phandle
     #Start Request
     puts "RFC START Request"
     set ps $port_speed
     set fs $max_frame_size
     for {set c 1} {$c < $connectionAttempts} {incr c} {
         puts "Attempt: $c"
         set port_speed $ps
         set max_frame_size $fs
         RFC_START_REQ - - - $psm_remote $port_speed $max_frame_size
$respond_phandle
         RFC_START_CFM
         #If result_code is not success we need to wait
         while {$result_code == 1} {
               RFC_START_CFM
         }
         puts "result_code: $result_code"
         if {$result_code != 6} {return $result_code}
     }
     return $result_code
}
```

The state will not transition until a RFC_START_CFM has been received and the `result_code` that is received with the RFC_START_CFM primitive is not RFC_CONNECTION_PENDING (which in this script is equivalent to `result_code` equal to 1). This state will eventually return the `result_code` that is received with the RFC_START_CFM primitive, which indicates whether the request has been successful i.e. when the return value is zero, or it has failed i.e. any value greater than one. If there is a failure the return value represents a reason code that can be found in the Section 4.4.3 in the BlueStack User Manual.

6.7.2 Register RFCOMM

This state allows the port entity to register a protocol handle (phandle) with RFCOMM. The registered phandle is used to signal to the port entity the arrival of RFC_ESTABLISH_IND message on the given server channel, see Section 6.3.10 for details on the RFC_ESTABLISH_IND. The script for the local and remote device is:

```
#Register RFCOMM
proc register_rfcomm {} {
     global phandle server_chan accept
     set state 0x00
     while {$state == 0} {
     #Register with RFCOMM
           puts "Register with RFCOMM"
           RFC_REGISTER_REQ $phandle
           RFC_REGISTER_CFM
           puts "Server Channel: $server_chan"
           puts "Accept        : $accept"
           puts "phandle       : $phandle"
           if {$accept == 1} {
              puts "RFCOMM Registration Accepted"
              incr state +1
           } else {
                 puts "RFCOMM Registration NOT Accepted"
           }
     }
}
```

The state transition occurs once RFC_REGISTER_CFM is received. A check on the `accept` variable may be required but at this stage this is not included.

6.7.3 Receive RFCOMM Data

This routine is found within the Flow Control Layer state and received data on the RFCOMM channel is stored as a packet in the receiver buffer, the script for the local device and the remote device is as follows:

```
#Receive RFCOMM Data
proc receiverx {} {
  global rx_buffer
  global credits
  puts "Wait to receive data"
  set result [RFC_DATA_IND]
  puts "Credits $credits"
  puts "result $result"
  if {$rx_buffer(noofpackets)} {
    incr rx_buffer(bottomptr)
  }
  set rxptr $rx_buffer($bottomptr)
  set rx_buffer($rxptr) $result
  #Increment number of received packets
  incr rx_buffer(noofpackets)
  #Check for bottom pointer needs to wrap around
  if {$rx_buffer(bottomptr) == $rx_buffer(buffersize)} {
     set rx_buffer(bottomptr) 0
  }
}
```

The initial part of the function waits for the data to arrive and stores it in `result`. The contents of `result` are then placed in the next free slot in the receive buffer. The number of received packets is incremented, the pointers modified and pointers wrapped around as required.

6.7.4 Transmit RFCOMM Data

This routine is found within the Flow Control Layer state and transmits data of the next packet from the top of the transmit buffer along the RFCOMM channel, the script for the local device and the remote device is as follows:

```
#Transmit RFCOMM Data
proc transmittx {} {
  global tx_buffer
  puts "Transmit RFCOMM"
  set txptr $tx_buffer(topptr)
  RFC_DATA_REQ - - - ? $tx_buffer($txptr)
  #Decrement the number of packets
  incr tx_buffer(noofpackets) -1
  #Increment the top of tx buffer pointer
  incr tx_buffer(topptr)
  #Check if top pointer needs to wrap around
  if {$tx_buffer(topptr) == $tx_buffer(buffersize)} {
     set tx_buffer(topptr) 0
  }
  #If number of packets is 0 then reset the both tx buffer pointers
  if { $tx_buffer(noofpackets) == 0} {
    set tx_buffer(topptr) 0
    set tx_buffer(bottomptr) 0
  }
  incr tx_buffer(transmitted)
}
```

The initial part of the function takes data from the top of the buffer and is sent out along the RFCOMM channel with the RFC_DATA_REQ command. The transmit buffer is then re-adjusted to remove this packet by decrementing the number of transmitted packets, the pointers modified and wrapped around as required.

6.7.5 Read Bluetooth Address Message

In the local device and remote device script the Read Bluetooth Address Message is internal to the state Process Next Received Packet in Receiver Buffer as outlined in Section 6.6.5. The actual functions differ slightly between the local device and the remote device scripts. In the local device the function just strips out the lap, uap and nap from the data passed to it and formats this to be displayed as a hex number. The local device script is as follows:

```
#Read Bluetooth Address Message
proc readbdaddr {remotebdaddr} {
     puts "Bluetooth Address:"
     set nap0 [lindex $remotebdaddr 1]
     set nap1 [lindex $remotebdaddr 2]
     set uap [lindex $remotebdaddr 3]
     set lap0 [lindex $remotebdaddr 4]
     set lap1 [lindex $remotebdaddr 5]
     set lap2 [lindex $remotebdaddr 6]
     set nap [expr (($nap0*0x100)+$nap1)]
     set lap [expr (($lap0*0x10000)+($lap1*0x100)+$lap2)]
     puts [format "NAP: 0x%x" $nap]
     puts [format "UAP: 0x%x" $uap]
     puts [format "LAP: 0x%x" $lap]
}
```

The remote device script for Read Bluetooth Address Message sends a primitive to the device manager that is the HCI command Read_BD_ADDR command. The script waits for the HCI command to be returned from the device manager and the complete data returned from this completed message is stored into `result`. The Bluetooth lap, uap and nap are then stripped out of `result` and these are loaded as a response into the transmit buffer. The remote device script is:

```
#Read Bluetooth Address Message
proc readbdaddr {} {
     puts "Read Bluetooth Address Message"
     DM_HCI_READ_BD_ADDR
     set result [DM_HCI_READ_BD_ADDR_COMPLETE]
     puts "Bluetooth Address: "
     set lap [lindex $result 2]
     #Create correct number of bytes for lap
     set lap0 [expr $lap/0x10000]
     set laptemp [expr $lap%0x10000]
     set lap1 [expr $laptemp/0x100]
     set lap2 [expr $laptemp%0x100]
     set uap [lindex $result 3]
     set nap [lindex $result 4]
     puts [format "NAP: 0x%x" $nap]
     puts [format "UAP: 0x%x" $uap]
     puts [format "LAP: 0x%x" $lap]
     #Create correct number of bytes for nap
     set nap0 [expr $nap/0x100]
     set nap1 [expr $nap%0x100]
     #Create response character
     set resp 0x01
     #Bind Message together
     set bd "$resp $nap0 $nap1 $uap $lap0 $lap1 $lap2"
     #Put Message in the transmit buffer
     if {[putpktpetxbuffer $bd]} {
       puts "Read Local Bluetooth Address Message failed to load in PE Tx
Buffer"
     }
}
```

6.7.6 Read Local Name Message

In the local device and remote device script the Read Local Name Message is internal to the state Process Next Received Packet in Receiver Buffer as outlined in Section 6.6.5. The actual functions differ slightly between the local device and the remote device scripts. In the local device the function just strips out the local name portion from the data passed to it and formats this to be displayed. The local device script is as follows:

```
#Read Local Name Message
proc readlocalname {remotename remotenamelength} {
     puts "READ LOCAL NAME"
     puts "Length = [expr $remotenamelength-1]"
     set rln0 [format "%s" $remotename]
     set rln [lreplace $rln0 0 0]
     puts "Local Name ASCII: $rln"
     set fln {}
     foreach el $rln {
         lappend fln [format "%1c" $el]
     }
     puts "Local Name: $fln"
     set lfln [llength $fln]
     puts "Length of Name: $lfln"
}
```

The remote device script for Read Local Name Message sends a primitive to the device manager that is the HCI command Read_Local_Name command. The script waits for the HCI command Read_Local_Name_Complete to be returned from the device manager and the complete data returned from this completed message is stored into `result`. The Local name is then stripped out of the out of `result` and formatted and loaded into the transmit buffer. The remote device script is:

```
#Read Local Name Message
proc readlocalname {} {
     puts "Read Local Name Message"
     DM_HCI_READ_LOCAL_NAME
     set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
     puts "Local Name: [lindex $result 2]"
     set rln0 0x02
     set rln1 [lindex $result 2]
     set rln2 [split $rln1 {}]
     set strl [llength $rln2]
     for {set c 0} {$c < $strl} {incr c} {
         scan [lindex $rln2 $c] %c rln3
         lappend rln0 $rln3
     }
     if {[putpktpetxbuffer $rln0]} {
       puts "Read Local Name Message failed to load in PE Tx Buffer"
     }
}
```

Note:

On the RFCLI screen for the local device the local name is displayed formatted as below. This format is due to the way TCL displays a variable that contains a list of elements. Each character in the local name is an element of the variable; the spaces are denoted by the curly brackets "`{ }`" and the spaces between each character are shown as an element separator and therefore are not true elements:

```
Local Name: 2 : { } N a m e { } c h a n g e d { } b y { } R F C L I
```

6.7.7 Change Local Name Message

In the local device and remote device script the Change Local Name Message is internal to the state Process Next Received Packet in Receiver Buffer as outlined in Section 6.6.5. The actual functions differ slightly between the local device and the remote device scripts. In the local device the function just strips out the new local name portion from the data passed to it and formats this to be displayed. The local device script is as follows:

```
#Change Local Name Message
proc changelocalname {changename changenamelength} {
     puts "CHANGE LOCAL NAME"
     puts "Length = [expr $changenamelength-1]"
     puts "$changename"
     set cln0 [format "%s" $changename]
     set cln [lreplace $cln0 0 0]
     puts "Local Name ASCII: $cln"
     set fln {}
     foreach el $cln {
         lappend fln [format "%1c" $el]
     }
     puts "Local Name: $fln"
     set lfln [llength $fln]
     puts "Length of Name: $lfln"
}
```

The remote device script for Change Local Name Message sends a primitive to the device manager that is the HCI command Change_Local_Name command. The new local name supplied to the primitive is taken from the data supplied by the local device. The script removes the command and the rest of the payload is the new local name to be used. The script waits for the HCI command Read_Local_Name_Complete to be returned from the device manager and the complete data returned from this completed message is stored into `result`. The Local name is then stripped out of `result` and formatted and loaded into the transmit buffer. The remote device script is:

```
#Change Local Name Message
proc changelocalname {changename changenamelength} {
     set cln [lreplace $changename 0 0]
     set changename $cln
     puts "Change Local Name Message"
     DM_HCI_READ_LOCAL_NAME
     set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
     puts "Old Local Name = [lindex $result 2]"
     puts "Requested Name = $changename"
     DM_HCI_CHANGE_LOCAL_NAME $changename
     set result [DM_HCI_CHANGE_LOCAL_NAME_COMPLETE]
     #Read the local name and send back to the master
     DM_HCI_READ_LOCAL_NAME
     set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
     puts "Local Name: [lindex $result 2]"
     set rln0 0x03
     set rln1 [lindex $result 2]
     set rln2 [split $rln1 {}]
     set strl [llength $rln2]
     for {set c 0} {$c < $strl} {incr c} {
         scan [lindex $rln2 $c] %c rln3
         lappend rln0 $rln3
     }
     if {[putpktpetxbuffer $rln0]} {
       puts "Change Local Name Message failed to load in PE Tx Buffer"
     }
}
```

Note:

On the RFCLI screen for the local device the local name is displayed in same format as described Section 6.7.6.

6.7.8 Load Packet in Transmit Buffer

The Load Packet in Transmit Buffer script is available in the local device and the remote device its function is to check for free space within the transmit buffer and then load the command packet into the transmit buffer. If there is no space then an error code of 1 is returned to the calling routine. If there is no error the new packet is placed at the bottom of the queue inside the buffer, number of packets is incremented and the pointers are adjusted and wrapped if necessary. The script is:

```
#Load Packet in Tranmit Buffer
proc putpktpetxbuffer {petxpacket} {
   global tx_buffer
   #Check for space in the buffer
   if {$tx_buffer(noofpackets) < $tx_buffer(buffersize)} {
      set errcode 0x00
      #Put tx packet at bottom of the tx buffer
      set bptr $tx_buffer(bottomptr)
      set tx_buffer($bptr) $petxpacket
      #Calculate Next available position in tx buffer
      incr tx_buffer(noofpackets)
      incr tx_buffer(bottomptr)
      #Check if bottom pointer needs to wrap around
      if {$tx_buffer(bottomptr) == $tx_buffer(buffersize)} {
         set tx_buffer(bottomptr) 0x00
      }
   } else {
      set errcode 0x01
   }
   return $errcode
}
```

7 Document References

Document:	Reference, Date:
Bluetooth Specification - Core	V1.1, v1.1, 22 February 2001
BlueStack User Manual	C6066-UM-001, v1.6
RFCLI User Manual	bcore-ug-003Pa, a, September 2002
Bluetooth Connect Without Cables – Jennifer Bray and Charles F Sturman	ISBN 0-13-089840-6, Prentice Hall PTR, 2001
Tcl and the Tk Toolkit – John K Ousterhout	ISBN 0-201-63337-X, Addison-Wesley, 1994
RFCOMM Packing Rules Application Note	bcore-an-004Pa, a, August 2002
Frontline Sniffer Quick Start User Guide	bcore-ug-004Pa, a, September 2002

Appendix A RFCLI Section 5 Example Script Source Files

A1 RFCLI Source Script file for `exampleslave.tcl`

```
#Slave script
#Connect to Casira
puts "Connect to Casira"
BC_connect com2 bcsp 115200
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set max_frame_size 0x7f
set initial_credits 0x07
#Register with RFCOMM
puts "Register with RFCOMM"
RFC_REGISTER_REQ $phandle
RFC_REGISTER_CFM
#Initialize RFCOMM
puts "Initialise RFCOMM"
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type $fc_threshold
$fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
#Register with Device Manager
puts "Register with Device Manager"
DM_AM_REGISTER_REQ $phandle
DM_AM_REGISTER_CFM
#Enabling page scanning so that the Master can connect to us
DM_HCI_WRITE_PAGESCAN_ACTIVITY 0x800 0x700
DM_HCI_WRITE_PAGESCAN_ACTIVITY_COMPLETE
DM_HCI_WRITE_SCAN_ENABLE 3
DM_HCI_WRITE_SCAN_ENABLE_COMPLETE
#Wait for a connection and respond appropriately
puts "Waiting for a connection"
RFC_START_IND
RFC_START_RES
RFC_START_IND
RFC_START_RES
RFC_STARTCMP_IND
RFC_PARNEG_IND
RFC_PARNEG_RES
RFC_ESTABLISH_IND
RFC_ESTABLISH_RES
#Wait for an incoming data primitive
puts "Connection made, starting transfer"
set result [RFC_DATA_IND]
#Wait for another incoming data primitive
puts "Received payload: [lindex $result 5]"
set result [RFC_DATA_IND]
puts "Received payload: [lindex $result 5]"
puts "All done"
```

A2 RFCLI Source Script file for `examplemaster.tcl`

```
#Master script
#Connect to Casira
puts "Connect to Casira"
BC_connect com1 bcsp 115200
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set bd_addr.lap 0x10e46
set bd_addr.uap 0x5b
set bd_addr.nap 0x02
set max_frame_size 0x7f
set initial_credits 0x07
#Register with RFCOMM
puts "Register with RFCOMM"
RFC_REGISTER_REQ $phandle
RFC_REGISTER_CFM
#Initialize RFCOMM
puts "Initialise RFCOMM"
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type $fc_threshold
$fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
#Register with Device Manager
puts "Register with Device Manager"
DM_AM_REGISTER_REQ $phandle
DM_AM_REGISTER_CFM
#Request RFCOMM Start
puts "Request RFCOMM Start"
RFC_START_REQ - - - $psm_remote ${sys_pars.port_speed}
${sys_pars.max_frame_size} $respond_phandle
RFC_START_CFM
#If result_code is not success we need to wait
while {$result_code == 1} {
        RFC_START_CFM
}
#RFCOMM Parameter Negotiation
puts "RFCOMM Parameter Negotiation"
RFC_PARNEG_REQ $mux_id - - $max_frame_size $use_flow_control
$initial_credits
RFC_PARNEG_CFM
puts "Credits: ${dlc_pars.initial_credits}"
#RFCOMM Establish
puts "RFCOMM Establish"
RFC_ESTABLISH_REQ $mux_id $loc_server_chan $rem_server_chan
RFC_ESTABLISH_CFM
#Send a data primitive
puts "Connection made, starting transfer"
RFC_DATA_REQ - - 0 ? {1 2 3 4 5 6 7 8 9}
#Send another primitive
RFC_DATA_REQ - - 0 ? {"Hello World"}
puts "All done"
```

Appendix BRFCLI Section 6 Example Script Source Files

B1 Local Device Script File

```
#Local Script

proc start_request {port_speed max_frame_size} {
     global result_code connectionAttempts
     global psm_remote respond_phandle
     #Start Request
     puts "RFC START Request"
     set ps $port_speed
     set fs $max_frame_size
     for {set c 1} {$c < $connectionAttempts} {incr c} {
         puts "Attempt: $c"
         set port_speed $ps
         set max_frame_size $fs
         RFC_START_REQ - - - $psm_remote $port_speed $max_frame_size
$respond_phandle
         RFC_START_CFM
         #If result_code is not success we need to wait
         while {$result_code == 1} {
               RFC_START_CFM
         }
         puts "result_code: $result_code"
         if {$result_code != 6} {return $result_code}
     }
     return $result_code
}

proc register_rfcomm {} {
     global phandle server_chan accept
     set state 0x00
     while {$state == 0} {
     #Register with RFCOMM
           puts "Register with RFCOMM"
           RFC_REGISTER_REQ $phandle
           RFC_REGISTER_CFM
           puts "Server Channel: $server_chan"
           puts "Accept        : $accept"
           puts "phandle       : $phandle"
           if {$accept == 1} {
              puts "DM Registration Accepted"
              incr state +1
           } else {
                 puts "DM Registration NOT Accepted"
           }
     }
}

#Check Transmit Buffer for Packets to Transmit
proc checkpetxbuffer {} {
   global tx_buffer
   puts "Check Transmit Buffer for Packets to Transmit"
   if {$tx_buffer(noofpackets) > 0x00} {
      puts "Port Entity has $tx_buffer(noofpackets) packet(s) to transmit"
      set pckcount $tx_buffer(noofpackets)
   } else {
      puts "Port Entity has no packets to transmit"
      set pckcount 0x00
   }
   return $pckcount
}

#Check For Packet in Receiver Buffer
```

```
proc checkperxbuffer {} {
   global rx_buffer
   if {$rx_buffer(noofpackets) > 0x00} {
#      puts "Port Entity has received $rx_buffer(noofpackets) packet(s)"
       set pckcount $rx_buffer(noofpackets)
   } else {
       puts "Port Entity has no receive packets"
       set pckcount 0x00
   }
   return $pckcount
}

#Load Packet in Tranmit Buffer
proc putpktpetxbuffer {petxpacket} {
   global tx_buffer
   #Check for space in the buffer
   if {$tx_buffer(noofpackets) < $tx_buffer(buffersize)} {
       set errcode 0x00
       #Put tx packet at bottom of the tx buffer
       set bptr $tx_buffer(bottomptr)
       set tx_buffer($bptr) $petxpacket
       #Calculate Next available position in tx buffer
       incr tx_buffer(noofpackets)
       incr tx_buffer(bottomptr)
       #Check if bottom pointer needs to wrap around
       if {$tx_buffer(bottomptr) == $tx_buffer(buffersize)} {
          set tx_buffer(bottomptr) 0x00
       }
   } else {
       set errcode 0x01
   }
   return $errcode
}

#Transmit RFCOMM Data
proc transmittx {} {
  global tx_buffer
  puts "Transmit RFCOMM"
  set txptr $tx_buffer(topptr)
  RFC_DATA_REQ - - - ? $tx_buffer($txptr)
  #Decrement the number of packets
  incr tx_buffer(noofpackets) -1
  #Increment the top of tx buffer pointer
  incr tx_buffer(topptr)
  #Check if top pointer needs to wrap around
  if {$tx_buffer(topptr) == $tx_buffer(buffersize)} {
     set tx_buffer(topptr) 0
  }
  #If number of packets is 0 then reset the both tx buffer pointers
  if { $tx_buffer(noofpackets) == 0} {
    set tx_buffer(topptr) 0
    set tx_buffer(bottomptr) 0
  }
  incr tx_buffer(transmitted)
}

#Receive RFCOMM Data
proc receiverx {} {
  global rx_buffer
  global credits
  puts "Wait to receive data"
  set result [RFC_DATA_IND]
  puts "Credits $credits"
  if {$rx_buffer(noofpackets)} {
    incr rx_buffer(bottomptr)
  }
  set rxptr $rx_buffer(bottomptr)
  set rx_buffer($rxptr) $result
```

```
  #Increment number of received packets
  incr rx_buffer(noofpackets)
  #Check for bottom pointer needs to wrap around
  if {$rx_buffer(bottomptr) == $rx_buffer(buffersize)} {
     set rx_buffer(bottomptr) 0
  }
  incr rx_buffer(received)
}

#Flow Control Layer
proc flowcontrollayer {fclstate} {
   global state
   switch $fclstate {
      0 {transmittx}
      1 {receiverx}
      default {puts "Unknown FCL State"}
   }
}

#Read Bluetooth Address Message
proc readbdaddr {remotebdaddr} {
     puts "Bluetooth Address:"
     set nap0 [lindex $remotebdaddr 1]
     set nap1 [lindex $remotebdaddr 2]
     set uap [lindex $remotebdaddr 3]
     set lap0 [lindex $remotebdaddr 4]
     set lap1 [lindex $remotebdaddr 5]
     set lap2 [lindex $remotebdaddr 6]
     set nap [expr (($nap0*0x100)+$nap1)]
     set lap [expr (($lap0*0x10000)+($lap1*0x100)+$lap2)]
     puts [format "NAP: 0x%x" $nap]
     puts [format "UAP: 0x%x" $uap]
     puts [format "LAP: 0x%x" $lap]
}

#Read Local Name Message
proc readlocalname {remotename remotenamelength} {
     puts "READ LOCAL NAME"
     puts "Length = [expr $remotenamelength-1]"
     set rln0 [format "%s" $remotename]
     set rln [lreplace $rln0 0 0]
     puts "Local Name ASCII: $rln"
     set fln {}
     foreach el $rln {
         lappend fln [format "%1c" $el]
     }
     puts "Local Name: $fln"
     set lfln [llength $fln]
     puts "Length of Name: $lfln"
}

#Change Local Name Message
proc changelocalname {changename changenamelength} {
     puts "CHANGE LOCAL NAME"
     puts "Length = [expr $changenamelength-1]"
     puts "$changename"
     set cln0 [format "%s" $changename]
     set cln [lreplace $cln0 0 0]
     puts "Local Name ASCII: $cln"
     set fln {}
     foreach el $cln {
         lappend fln [format "%1c" $el]
     }
     puts "Local Name: $fln"
     set lfln [llength $fln]
     puts "Length of Name: $lfln"
}
```

```
#Process Next Received Packet in Receiver Buffer
proc processrx {} {
   global rx_buffer
   puts "Process receive buffer"
   #Get the Next Received packet to process
   set rxptr $rx_buffer(topptr)
   #Decrement the number of packets
   incr rx_buffer(noofpackets) -1
   #Increment the top of rx buffer pointer
   incr rx_buffer(topptr)
   #Check if top pointer needs to wrap around
   if {$rx_buffer(topptr) == $rx_buffer(buffersize)} {
      set rx_buffer(topptr) 0
   }
   #If number of packets is 0 then reset the both rx buffer pointers
   if {$rx_buffer(noofpackets) == 0} {
     set rx_buffer(topptr) 0
     set rx_buffer(bottomptr) 0
   }
   set result $rx_buffer($rxptr)
   puts "Payload: [lindex $result 5]"
   set firstchar [lindex [lindex $result 5] 0]
   puts "First character of payload $firstchar"
   switch $firstchar {
     0x00 {puts "Data transfer complete"}
     0x01 {readbdaddr [lindex $result 5]}
     0x02 {readlocalname [lindex $result 5] [lindex $result 4]}
     0x03 {changelocalname [lindex $result 5] [lindex $result 4]}
   }
}

proc steadystate {} {
   puts "Steady State"
}

#Check Receiver Buffer for Space
proc receiverspace {} {
   global rx_buffer
   if {$rx_buffer(noofpackets) < $rx_buffer(buffersize)} {
     return 1
   } else {
     return 0
   }
}

#Check State
proc checkstate {} {
   global state
   switch $state {
      0 {puts "Task Complete"}
      1 {steadystate}
      2 {if {[checkpetxbuffer]} {flowcontrollayer 0}}
      3 {if {[receiverspace]} {flowcontrollayer 1}}
      4 {if {[checkperxbuffer]} {processrx}}
      default {puts "Unknown State"}
   }
   incr state
   #Check to see if we have done all processing
   if {$state == 5} {
     if {[checkpetxbuffer] || [checkperxbuffer]} {
       set state 1
     } else {
       set state 0
     }
   }
}

#Connect to Casira
```

```
puts "Connect to Casira"
BC_connect com1 bcsp 115200
#Initialise Port Entity Transmit Buffer
puts "Initialise Port Entity Transmit Buffer"
set tx_buffer(noofpackets) 0
set tx_buffer(txcredits) 0
set tx_buffer(pointer) 0
set tx_buffer(buffersize) 10
set tx_buffer(topptr) 0
set tx_buffer(bottomptr) 0
set tx_buffer(transmitted) 0
for {set cnt 0} {$cnt < $tx_buffer(buffersize)} {incr cnt} {
  set tx_buffer($cnt) 0xc0ffee
}
#Initialise Port Entity Receive Buffer
puts "Initialise Port Entity Receive Buffer"
set rx_buffer(noofpackets) 0
set rx_buffer(txcredits) 0
set rx_buffer(pointer) 0
set rx_buffer(topptr) 0
set rx_buffer(bottomptr) 0
set rx_buffer(buffersize) 10
set rx_buffer(received) 0
for {set cnt 0} {$cnt < $rx_buffer(buffersize)} {incr cnt} {
  set rx_buffer($cnt) 0xc0ffee
}
#Put Messages in Transmit Buffer
puts "Put Messages in Transmit Buffer"
for {set cnt 0} {$cnt < $tx_buffer(buffersize)} {incr cnt} {
   switch $cnt {
      0 {set txmsg {1}}
      1 {set txmsg {2}}
      2 {set txmsg {3 "1: Name changed by RFCLI"}}
      3 {set txmsg {1}}
      4 {set txmsg {2}}
      5 {set txmsg {3 "2: Name changed by RFCLI"}}
      6 {set txmsg {1}}
      7 {set txmsg {2}}
      8 {set txmsg {3 "3: Name changed by RFCLI"}}
      9 {set txmsg {0}}
      default {set txmsg {0}}
   }
   if {[putpktpetxbuffer $txmsg]} {
      puts "Message failed to load in PE Tx Buffer"
   }
}
#printtxbuffer
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set bd_addr.lap 0x10e46
set bd_addr.uap 0x5b
set bd_addr.nap 0x02
set max_frame_size 0x7f
set initial_credits 0x07
puts "Connection Attempts: $connectionAttempts"
#Register with RFCOMM
puts "Register with RFCOMM"
register_rfcomm
#Initialise RFCOMM
puts "Initialise RFCOMM"
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type $fc_threshold
$fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
#Register with Device Manager
puts "Register with Device Manager"
puts "DM phandle: $phandle"
DM_AM_REGISTER_REQ $phandle
```

```
DM_AM_REGISTER_CFM
#Request RFCOMM Start
puts "Request RFCOMM Start"
start_request ${sys_pars.port_speed} ${sys_pars.max_frame_size}
#RFCOMM Parameter Negotiation
puts "RFCOMM Parameter Negotiation"
RFC_PARNEG_REQ $mux_id - - $max_frame_size $use_flow_control
$initial_credits
RFC_PARNEG_CFM
puts "Credits: ${dlc_pars.initial_credits}"
#RFCOMM Establish
puts "RFCOMM Establish"
RFC_ESTABLISH_REQ $mux_id $loc_server_chan $rem_server_chan
RFC_ESTABLISH_CFM
set state 1
#Main Loop
puts "Main Loop"
while {$state} {
  checkstate
}
puts "All Done"
puts "Number of Transmitted Packets = $tx_buffer(transmitted)"
puts "Number of Received Packets    = $rx_buffer(received)"
```

B2 Remote Device Script File

```
#Remote script

proc start_request {port_speed max_frame_size} {
     global result_code connectionAttempts
     global psm_remote respond_phandle
     #Start Request
     puts "RFC START Request"
     set ps $port_speed
     set fs $max_frame_size
     for {set c 1} {$c < $connectionAttempts} {incr c} {
         puts "Attempt: $c"
         set port_speed $ps
         set max_frame_size $fs
         RFC_START_REQ - - - $psm_remote $port_speed $max_frame_size
$respond_phandle
         RFC_START_CFM
         #If result_code is not success we need to wait
         while {$result_code == 1} {
               RFC_START_CFM
         }
         puts "result_code: $result_code"
         if {$result_code != 6} {return $result_code}
     }
     return $result_code
}

proc register_rfcomm {} {
     global phandle server_chan accept
     set state 0x00
     while {$state == 0} {
     #Register with RFCOMM
           puts "Register with RFCOMM"
           RFC_REGISTER_REQ $phandle
           RFC_REGISTER_CFM
           puts "Server Channel: $server_chan"
           puts "Accept        : $accept"
           puts "phandle       : $phandle"
           if {$accept == 1} {
              puts "RFCOMM Registration Accepted"
              incr state +1
```

```
            } else {
                  puts "RFCOMM Registration NOT Accepted"
            }
      }
}

#Check Transmit Buffer for Packets to Transmit
proc checkpetxbuffer {} {
   global tx_buffer
   puts "Check Transmit Buffer for Packets to Transmit"
   if {$tx_buffer(noofpackets) > 0x00} {
      puts "Port Entity has $tx_buffer(noofpackets) packet(s) to transmit"
      set pckcount $tx_buffer(noofpackets)
   } else {
      puts "Port Entity has no packets to transmit"
      set pckcount 0x00
   }
   return $pckcount
}

#Check For Packet in Receiver Buffer
proc checkperxbuffer {} {
   global rx_buffer
   puts "Check For Packet in Receiver Buffer"
   if {$rx_buffer(noofpackets) > 0x00} {
      puts "Port Entity has received $rx_buffer(noofpackets) packet(s)"
      set pckcount $rx_buffer(noofpackets)
   } else {
      puts "Port Entity has no receive packets"
      set pckcount 0x00
   }
   return $pckcount
}

#Load Packet in Tranmit Buffer
proc putpktpetxbuffer {petxpacket} {
   global tx_buffer
   puts "Load Packet in Tranmit Buffer"
   #Check for space in the buffer
   if {$tx_buffer(noofpackets) < $tx_buffer(buffersize)} {
      set errcode 0x00
      #Put tx packet at bottom of the tx buffer
      set bptr $tx_buffer(bottomptr)
      set tx_buffer($bptr) $petxpacket
      #Calculate Next available position in tx buffer
      incr tx_buffer(noofpackets)
      incr tx_buffer(bottomptr)
      #Check if bottom pointer needs to wrap around
      if {$tx_buffer(bottomptr) == $tx_buffer(buffersize)} {
         set tx_buffer(bottomptr) 0x00
      }
   } else {
      set errcode 0x01
   }
   return $errcode
}

#Transmit RFCOMM Data
proc transmittx {} {
  global tx_buffer
  puts "Transmit RFCOMM Data"
  set txptr $tx_buffer(topptr)
  RFC_DATA_REQ - - - ? $tx_buffer($txptr)
  #Decrement the number of packets
  incr tx_buffer(noofpackets) -1
  #Increment the top of tx buffer pointer
  incr tx_buffer(topptr)
  #Check if top pointer needs to wrap around
```

```
  if {$tx_buffer(topptr) == $tx_buffer(buffersize)} {
     set tx_buffer(topptr) 0
  }
  #If number of packets is 0 then reset the both tx buffer pointers
  if { $tx_buffer(noofpackets) == 0} {
    set tx_buffer(topptr) 0
    set tx_buffer(bottomptr) 0
  }
  incr tx_buffer(transmitted)
}

#Receive RFCOMM Data
proc receiverx {} {
  global rx_buffer
  global credits
  puts "Receive RFCOMM Data"
  set result [RFC_DATA_IND]
  puts "Credits $credits"
  if {$rx_buffer(noofpackets)} {
    incr rx_buffer(bottomptr)
  }
  set rxptr $rx_buffer(bottomptr)
  set rx_buffer($rxptr) $result
  #Increment number of received packets
  incr rx_buffer(noofpackets)
  #Check for bottom pointer needs to wrap around
  if {$rx_buffer(bottomptr) == $rx_buffer(buffersize)} {
     set rx_buffer(bottomptr) 0
  }
  incr rx_buffer(received)
}

#Flow Control Layer
proc flowcontrollayer {fclstate} {
   global state
   puts "Flow Control Layer"
   switch $fclstate {
      0 {transmittx}
      1 {receiverx}
      default {puts "Unknown FCL State"}
   }
}

#Read Bluetooth Address Message
proc readbdaddr {} {
     puts "Read Bluetooth Address Message"
     DM_HCI_READ_BD_ADDR
     set result [DM_HCI_READ_BD_ADDR_COMPLETE]
     puts "Bluetooth Address: "
     set lap [lindex $result 2]
     #Create correct number of bytes for lap
     set lap0 [expr $lap/0x10000]
     set laptemp [expr $lap%0x10000]
     set lap1 [expr $laptemp/0x100]
     set lap2 [expr $laptemp%0x100]
     set uap [lindex $result 3]
     set nap [lindex $result 4]
     puts [format "NAP: 0x%x" $nap]
     puts [format "UAP: 0x%x" $uap]
     puts [format "LAP: 0x%x" $lap]
     #Create correct number of bytes for nap
     set nap0 [expr $nap/0x100]
     set nap1 [expr $nap%0x100]
     #Create response character
     set resp 0x01
     #Bind Message together
     set bd "$resp $nap0 $nap1 $uap $lap0 $lap1 $lap2"
     #Put Message in the transmit buffer
```

```
      if {[putpktpetxbuffer $bd]} {
        puts "Read Local Bluetooth Address Message failed to load in PE Tx
Buffer"
      }
}

#Read Local Name Message
proc readlocalname {} {
      puts "Read Local Name Message"
      DM_HCI_READ_LOCAL_NAME
      set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
      puts "Local Name: [lindex $result 2]"
      set rln0 0x02
      set rln1 [lindex $result 2]
      set rln2 [split $rln1 {}]
      set strl [llength $rln2]
      for {set c 0} {$c < $strl} {incr c} {
          scan [lindex $rln2 $c] %c rln3
          lappend rln0 $rln3
      }
      if {[putpktpetxbuffer $rln0]} {
        puts "Read Local Name Message failed to load in PE Tx Buffer"
      }
}

#Change Local Name Message
proc changelocalname {changename changenamelength} {
      set cln [lreplace $changename 0 0]
      set changename $cln
      puts "Change Local Name Message"
      DM_HCI_READ_LOCAL_NAME
      set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
      puts "Old Local Name = [lindex $result 2]"
      puts "Requested Name = $changename"
      DM_HCI_CHANGE_LOCAL_NAME $changename
      set result [DM_HCI_CHANGE_LOCAL_NAME_COMPLETE]
      #Read the local name and send back to the master
      DM_HCI_READ_LOCAL_NAME
      set result [DM_HCI_READ_LOCAL_NAME_COMPLETE]
      puts "Local Name: [lindex $result 2]"
      set rln0 0x03
      set rln1 [lindex $result 2]
      set rln2 [split $rln1 {}]
      set strl [llength $rln2]
      for {set c 0} {$c < $strl} {incr c} {
          scan [lindex $rln2 $c] %c rln3
          lappend rln0 $rln3
      }
      if {[putpktpetxbuffer $rln0]} {
        puts "Change Local Name Message failed to load in PE Tx Buffer"
      }
}

#Data Transfer Complete
proc datacomplete {} {
      global taskcompleteflag
      puts "Data transfer complete"
      if {[putpktpetxbuffer 0]} {
        puts "Change Local Name Message failed to load in PE Tx Buffer"
      }
      set taskcompleteflag 1
}

#Process Next Received Packet in Receiver Buffer
proc processrx {} {
   global rx_buffer
   puts "Process receive buffer"
   #Get the Next Received packet to process
```

```
   set rxptr $rx_buffer(topptr)
   #Decrement the number of packets
   incr rx_buffer(noofpackets) -1
   #Increment the top of rx buffer pointer
   incr rx_buffer(topptr)
   #Check if top pointer needs to wrap around
   if {$rx_buffer(topptr) == $rx_buffer(buffersize)} {
      set rx_buffer(topptr) 0
   }
   #If number of packets is 0 then reset the both rx buffer pointers
   if {$rx_buffer(noofpackets) == 0} {
     set rx_buffer(topptr) 0
     set rx_buffer(bottomptr) 0
   }
   set result $rx_buffer($rxptr)
   puts "Payload: [lindex $result 5]"
   set firstchar [lindex [lindex $result 5] 0]
   puts "First character of payload $firstchar"
   switch $firstchar {
     0x00 {datacomplete}
     0x01 {readbdaddr}
     0x02 {readlocalname}
     0x03 {changelocalname [lindex $result 5] [lindex $result 4]}
   }
}

#Steady State
proc steadystate {} {
   puts "Steady State"
}

#Check Receiver Buffer for Space
proc receiverspace {} {
   global rx_buffer
   if {$rx_buffer(noofpackets) < $rx_buffer(buffersize)} {
     return 1
   } else {
     return 0
   }
}

#Check State
proc checkstate {} {
   global state
   switch $state {
      0 {puts "Task Complete"}
      1 {steadystate}
      2 {if {[checkpetxbuffer]} {flowcontrollayer 0}}
      3 {if {[receiverspace]} {flowcontrollayer 1}}
      4 {if {[checkperxbuffer]} {processrx}}
      default {puts "Unknown State"}
   }
   incr state
   #Check to see if we have done all processing
   if {$state == 5} {
     if {[checkpetxbuffer] || [checkperxbuffer]} {
       set state 1
     } else {
       set state 0
     }
   }
}

#Make a connection to the Casira on comport2
BC_connect com2 bcsp 115200
#Initialise Port Entity Transmit Buffer
puts "Initialise Port Entity Transmit Buffer"
set tx_buffer(noofpackets) 0
```

```
set tx_buffer(txcredits) 0
set tx_buffer(pointer) 0
set tx_buffer(buffersize) 10
set tx_buffer(topptr) 0
set tx_buffer(bottomptr) 0
set tx_buffer(transmitted) 0
for {set cnt 0} {$cnt < $tx_buffer(buffersize)} {incr cnt} {
  set tx_buffer($cnt) 0xc0ffee
}
#Initialise Port Entity Receive Buffer
puts "Initialise Port Entity Receive Buffer"
set rx_buffer(noofpackets) 0
set rx_buffer(txcredits) 0
set rx_buffer(pointer) 0
set rx_buffer(topptr) 0
set rx_buffer(bottomptr) 0
set rx_buffer(buffersize) 10
set rx_buffer(received) 0
for {set cnt 0} {$cnt < $rx_buffer(buffersize)} {incr cnt} {
  set rx_buffer($cnt) 0xc0ffee
}
#Initialise System Variables
puts "Initialise System Variables"
set use_flow_control 0x01
set max_frame_size 0x7f
set initial_credits 0x07
set state 1
set taskcompleteflag 0
#Register with RFCOMM
register_rfcomm
#Initialise RFCOMM
RFC_INIT_REQ $phandle $psm_local $use_flow_control $fc_type $fc_threshold
$fc_timer $rsvd_4 $rsvd_5
RFC_INIT_CFM
#Register with the Device Manager
puts "Register with Device Manager"
puts "DM phandle: $phandle"
DM_AM_REGISTER_REQ $phandle
DM_AM_REGISTER_CFM
#Listen
listen
#Output Waiting For Connection Message
puts "Waiting For Connection Message"
#Connect as Slave
rfc_connect_slv
#Wait for an incoming data primitive
puts "Connection made, starting transfer"
#Main Loop
puts "Main Loop"
while {$state} {
  if {$taskcompleteflag} {
    #Flush Transmit buffer to tell the Master to finish
    set state 2
    checkstate
    set state 0
  } else {
    checkstate
  }
}
puts "All Done"
puts "Number of Transmitted Packets = $tx_buffer(transmitted)"
puts "Number of Received Packets    = $rx_buffer(received)"
```

Acronyms and Definitions

ABM	Asynchronous Balanced Mode
API	Application Programming Interface
BCSP	BlueCore Serial port Protocol
BIST	Built In Self Test
BlueCore	Group term for CSR's range of Bluetooth chips.
Bluetooth	A set of technologies providing audio and data transfer over short-range radio connections
CD	Carrier Detect
CID	Channel IDentifier
CTS	Clear To Send
DCE	Data Communications Equipment
DISC	DISconnected Command
DLC	Data Link Connection
DLCI	Data Link Connection Identifier
DM	Disconnect Mode
DSR	Data Set Ready
DTE	Data Terminal Equipment
DTR	Data Terminal Ready
DV	Data Valid
ETSI	European Communications Standards Institute
FCL	Flow Control Layer
FCT	Flow Control Token
GSM	Global System for Mobile communications
IC	Incoming Call indicator
L2CAP	Logical Link Control and Adaptation Protocol (protocol layer)
MSC	Modem Status Commands
PC	Personal Computer
PEE	Port Emulation Entity
PICS	Protocol Implementation Confirmation Statement
PN	Parameter Negotiation
PPE	Port Proxy Entity
RAM	Random Access Memory
RD	Receive Data
RFCLI	RFCOMM Command Line Interface
RFCOMM	Protocol layer providing serial port emulation over L2CAP
RI	Ring Indicator
RTC	Ready To Communicate
RTS	Ready To Send
RTR	Ready To Receive
SABM	Start Asynchronous Balanced Mode frame
SDD	Service Discovery Database
SDP	Service Discovery Protocol
SPP	Serial Port Profile

TCL	Tool kit Command Language
TD	Transmit Data
UA	Unnumbered Acknowledgement frame
UIH	Unnumbered Information with Header frame

Record of Changes

Date:	Revision	Reason for Change:
25 SEP 02	a	Original publication of this document. (CSR reference: bcore-an-006Pa)

Accessing RFCOMM Using RFCLI and TCL Application Note

bcore-an-006Pa

September 2002