ZigBee Remote Control (ZRC) Application Profile

User's Guide

Document Number: ZRCAPUG
Rev. 1.2
2/2012

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Contents

About This Book

This user's guide provides an overview of the Freescale Zigbee Remote Control (ZRC) Application Profile implementation and describes how an overlying application can access the features it provides. This document replaces the *Freescale Consumer Electronics Remote Control Application Profile User's Guide* (CERCAPUG).

Audience

This document is intended for software developers writing applications based on Freescale's BeeStack Consumer stack intending to use the Freescale's ZRC application profile implementation to simplify the application design.

Organization

This document contains the following chapters:

Chapter 1	ZRC Application Profile Implementation Overview — Describes the ZRC application profile implementation and how an application can interface with it.
Chapter 2	Controller-side Push-button Pairing — Describes the controller side push-button pairing procedure.
Chapter 3	Target-side Push-button Pairing — Describes the target-side push-button pairing procedure.
Chapter 4	ZRC Command Transmit and Receive — Describes the command transmit and receive procedures.

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 1.1).

Revision History

Date / Author	Description / Location of Changes
Feb. 2012, Dev Team	Minor changes for March software release.

Conventions

This document uses the following notational conventions:

- Courier monospaced type is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.
- Italic type is used for emphasis, to identify new terms, and for replaceable command parameters.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

API	Application Programming Interface
NLDE	Network Layer Data Entity
NLME	Network Layer Management Entity
SAP	Service Access Point
NWK	Network Layer
gMaxPairTableEntries_c	The size of the RF4CE pair table
PBP	Push Button Pair

Chapter 1
ZRC Application Profile Implementation Overview

NOTE

This document refers only to HS08 platforms. For ARM platform legacy refer to documents from \ Documentation \ BeeStack Consumer Documents \ ARM Legacy Documents \.

To aid in the development of applications based on BeeStack Consumer, Freescale has developed an implementation of the ZRC profile. The profile resides in the protocol stack between the BeeStack Consumer layer and the application layer. The application can still access the network layer directly.

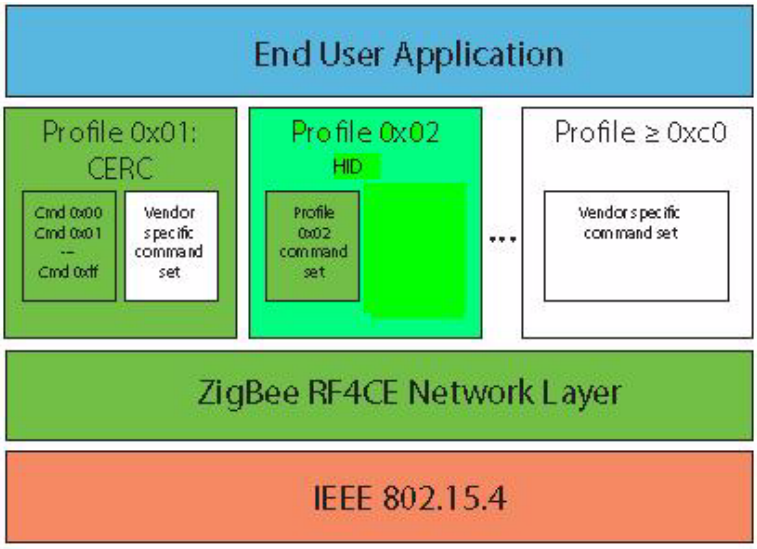

Figure 1-1. Freescale ZRC Profile Application Structure

The profile relies completely on the underlying network layer to perform its tasks. The profile layer uses BeeStack Consumer API calls to pass data to the network layer and depends on indication and confirm messages to receive data from the network layer. The NLDE and NLME SAPs must be configured to redirect messages intended for the profile layer to the profile SAPs, so that these don't erroneously reach the application.. The next section describes an example of how to do this.

NOTE

The BeeStack Consumer network layer handles one request at a time, whether it comes directly from the application or from the profile. Care must be taken to ensure that the application and the profile never make request so the network layer simultaneously.

The ZRC profile implements the following functionality, available as separate libraries:

1. Controller-side push-button pairing.
2. Target-side push-button pairing.
3. ZRC command transmission and reception.

NOTE

The Push-Button Pair (PBP) functionality is implemented as a separate framework and can be used as a stand alone layer by other profiles, such as ZID Profile. In this way, the ZRC profile layer is seen as being formed from two small sublayers: the ZRC Command Tx/Rx sublayer and the Push Button Pair (PBP) sublayer. For more details refer to ZigBee Remote Control Application Profile Reference Manual.

Note that this document describes the PBP procedures in conjunction with the ZRC profile. The ZRC applications should include the Push-Button Pair libraries and files implementing the desired functionality.

1.1 Interfacing With the ZRC Command Tx/Rx and PBP sublayers

The application communicates with the profile layer in the same manner as it communicates with the network layer. There are API calls for the application to profile communication and indication/confirm messages for the profile to application communication.

The ZRC Command Tx/Rx sublayer runs its own task. To send messages to the application, the ZRC Command Tx/Rx sublayer calls a ZRC SAP handler. The ZRC SAP handler must be implemented by the application. It is a callback function which has one parameter, a pointer to the incoming message. In our demo application the profile ZRC SAP handler simply adds the message from the profile to the profile messages queue and sends an event to the application main task:

```
void ZRCProfile_App_SapHandler(zrcProfileToAppMsg_t* zrcProfileToAppMsg)
{
  /* Put the incoming ZRC profile message in the applications input queue. */
  MSG_Queue(&mZRCProfileAppInputQueue, zrcProfileToAppMsg);
  TS_SendEvent(gAppTaskID, gAppEvtMsgFromZRCProfile_c);
}
```

The ZRC Command Tx/Rx sublayer sends messages to the application. These messages have the following structure:

1.1.0.1 Message Structure

```
/* General structure of a message received by the application over ZRC Profile SAP */
typedef struct zrcProfileToAppMsg_tag
{
  zrcProfileToAppMsgType_t              msgType;
  union {
    /*-------------------------------------------------------*/
```

```
    zrcProfileCommandInd_t                  zrcProfileCommandInd;
    zrcProfileCommandCnf_t                  zrcProfileCommandCnf;
    zrcProfileDiscoveryCmdCnf_t             zrcProfileDiscoveryCmdCnf;
  } msgData;
}zrcProfileToAppMsg_t;
```

where zrcProfileToAppMsgType_t is typedef for:

```
typedef enum {
  gZRCProfileCommandInd_c          = 0x10,
  gZRCProfileCommandCnf_c,
  gZRCProfileDiscoveryCmdCnf,
  /*---------------------------*/
  gZRCProfileMax_c
}zrcProfileToAppMsgType_t;
```

The application should determine the message type from the msgType field and use that information to access the correct member of the msgData union.

To allow the ZRC profile to handle transmission and reception of ZRC commands it must receive the data indication and data confirm messages the have a ZRC profile ID (0x01).

To intercept NLDE data indication and data confirm messages intended to the ZRC Command Tx/Rx sublayer , the Freescale sample applications the SAP handler was updated as shown below :

```
void NWK_NLDE_SapHandler(nwkNldeToAppMsg_t* nwkNldeToAppMsg)
{
  /* Put the incoming NLDE message in the right queue. */
    if(((nwkNldeToAppMsg->msgType == gNwkNldeDataInd_c) &&
(nwkNldeToAppMsg->msgData.nwkNldeDataInd.profileId == gZRCProfileId_c)) ||
       ((nwkNldeToAppMsg->msgType == gNwkNldeDataCnf_c) &&
(nwkNldeToAppMsg->msgData.nwkNldeDataCnf.profileId == gZRCProfileId_c)))
    {
      /* NLDE message is for ZRC profile */
      ZRCProfile_HandleNwkNldeMsg(nwkNldeToAppMsg);
    }
    else
    {
      /* Put the incoming NLDE message in the applications input queue. */
      MSG_Queue(&mNldeAppInputQueue, nwkNldeToAppMsg);
      TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlde_c);
    }
}
```

The modified NLDE SAP handler first checks whether the message has the ZRC profile ID and calls the ZRC profile message handler if that is the case. Otherwise the message is added to the NLDE queue and an event is sent to the application task. All data indication and data confirm messages that have the profile ID set to 0x01 (ZRC profile) must be redirected to the ZRC Command Tx/Rx sublayer. The application should abstain from issuing NLDE Data Requests with ZRC profile ID directly, as all NLDE Data Confirm messages with a ZRC profile ID is redirected to the ZRC Command Tx/Rx sublayer .

In the same manner as the ZRC Command Tx/Rx sublayer, the PBP sublayer runs its own task. To send messages to the application, the PBP sublayer calls a PBP SAP handler. The PBP SAP handler must be

implemented by the application. This handler simply adds the incoming message from the PBP sublayer to the PBP messages queue and sends an event to the application main task:

```
void PBP_APP_SapHandler(pushButtonToAppMsg_t* pbpToAppMsg)
{
  MSG_Queue(&mPushButtonAppInputQueue, pbpToAppMsg);
  TS_SendEvent(gAppTaskID, gAppEvtMsgFromPushButton_c);
}
```

The incoming messages structure and messages type are:

```
typedef struct pushButtonToAppMsg_tag
{
  pushButtonToAppMsgType_t               msgType;
  union {
    /*---------------------------------------------------------*/
    pushButtonPairOrigCnf_t          pushButtonPairOrigCnf;
    pushButtonPairRecipCnf_t         pushButtonPairRecipCnf;
    pushButtonPairOrigContinueInd_t  pushButtonPairOrigContinueInd;
    pushButtonPairRecipContinueInd_t pushButtonPairRecipContinueInd;
  } msgData;
}pushButtonToAppMsg_t;

typedef enum {
  /*---------------------------*/
  gPushButtonPairOrigCnf_c = 0,
  gPushButtonPairRecipCnf_c,
  gPushButtonPairOrigContinueInd_c,
  gPushButtonPairRecipContinueInd_c
  /*---------------------------*/
}pushButtonToAppMsgType_t;
```

To allow the PBP sublayer to perform the push-button pairing process, it must have access to the NLME indication and confirm messages (specifically discovery confirm/auto-discovery confirm, pair confirm/pair indication and commStatus indication messages). The Freescale sample applications accomplish this as shown in the following example.

```
void NWK_NLME_SapHandler(nwkNlmeToAppMsg_t* nwkNlmeToAppMsg)
{
  if(appStateMachine.state == gAppStatePushButtonPairing_c)
  {
    /* NLME message is for Push Button Pair */
    PBP_HandleNwkNlmeMsg(nwkNlmeToAppMsg);
  }
  else
  {
    /* Put the incoming NLME message in the applications input queue. */
    MSG_Queue(&mNlmeAppInputQueue, nwkNlmeToAppMsg);
    TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlme_c);
  }
}
```

When doing push-button pairing the sample application's state machine is in a special state called *gAppStatePushButtonPairing_c*. Whenever the application is in that state, all NLME messages are redirected to the PBP sublayer; otherwise they are put in the NLME message queue, as usual. Any

application utilizing the PBP sublayer must ensure that NLME messages are forwarded to the PBP sublayer during the push button pairing process.

To be able to benefit from the functionality of the PBP sublayer the application must first link to the relevant library (a list of the profile libraries can be found in the Freescale ZRC reference manual) in addition to the profile task framework library. Each functionality also has an initialization function which must be called once at application startup. The best place to call the initialization function is in the main application initialization function:

```
void App_Init(void)
{
...
  /* Init ZRC profile and PBP procedures */
PBP_InitPushButtonPairOrig();
ZRCProfile_InitCommandTxRx();
...
}
```

The sample application code above is using the originator push-button pairing functionality, allowing it to conduct controller-side push-button pairing. It is also transmitting and receiving ZRC commands through the ZRC Command Tx/Rx sublayer .

1.2 Power Saving

The application Idle task needs to now when the entire layer stack is idle to enter a low power state. To check whether the ZRC Command Tx/Rx and PBP sublayers are idle the application can use the ZRCProfile_IsIdle() and PBP_IsIdle() functions.

1.2.0.1 Prototypes

```
bool_t ZRCProfile_IsIdle(void);
bool_t PBP_IsIdle(void);
```

It returns FALSE if the layer is currently busy and TRUE otherwise.

The idle task should test the layers for idleness before entering low power. The Freescale sample application’s idle task shows a correct way how to do this:

```
void IdleTask(event_t events)
{

  (void)events; /* remove compiler warning */

  /* There are some UART errors that are hard to clear in the UART */
  /* ISRs, and the UART driver does not have a task to clear them */
  /* in non-interrupt context. */
  Uart_ClearErrors();

if((NWK_IsIdle() == TRUE)
#if gZRCProfileCommandTxRx_d
     && (ZRCProfile_IsIdle())
```

```
#endif
#if gPBPTask_d
     && (PBP_IsIdle())
#endif
     )
  {
#if gNvStorageIncluded_d
               /* Process NV Storage save-on-idle and save-on-count requests. */
               NvIdle();
#endif /* gNvStorageIncluded_d */

#if gLpmIncluded_d
        HandleLowPower();
#endif /* gLpmIncluded_d *
}
```

Chapter 2
Controller-side Push-button Pairing

Controller-side push-button pairing is made easy with the PBP sublayer. Although it was designed for controllers, this functionality can also be used on target nodes to pair two targets. A node using this feature will start the discovery process followed by the pair request. The push-button pairing functionality can be used to pair with any RF4CE ZRC compliant device.

2.1 Configuration

The initialization function for the controller-side push-button pairing functionality is PBP_InitPushButtonPairOrig.

Aside from initialization no other configuration needs to be done. The application must only ensure that the RF4CE_PushButtonTask and RF4CE_PushButtonOrig libraries are linked.

2.1.1 The Push-button Pairing Process

The application initiates the push-button pairing process by calling PBP_PushButtonPairOrigRequest.

2.1.1.1 Prototype

```
uint8_t PBP_PushButtonPairOrigRequest(
                                        uint16_t          recipPanId,
                                        uint16_t          recipShortAddress,
                                        uint8_t           recipDeviceType,
                                        appCapabilities_t origAppCapabilities,
                                        uint8_t*          origDeviceTypeList,
                                        uint8_t*          origProfileIdList,
                                        uint8_t           discProfileIdListSize,
                                        uint8_t*          discProfileIdList,
                                        uint8_t           keyExTransferCount,
                                        bool_t            bRequestAppAcceptToPair,
                                        uint16_t          timeToWaitAppAcceptToPair
                                        )
```

The parameters are as follows:

- recipPanId – the recipient's PAN Id, this is the destination PAN Id of the discovery request frame (usually this is 0xFFFF unless the application knows the PAN Id of the device it intends to pair with)
- recipShortAddress – the recipient's short address, this is the destination address of the discovery request frame (usually 0xFFFF)
- recipDeviceType – the required device type of the device it wants to pair with
- origAppCapabilities – the current node's application capabilities

- origDeviceTypeList – the list of device types the current node supports
- origProfileIdList – the list of profiles the current node supports
- discProfileIdListSize – the size of the discovery profile list
- discProfileIdList – the discovery profile list (the profiles the current node is looking for)
- keyExTransferCount – the number of frames that should be used for exchanging the security key (this is required when both devices support security as a secured pairing link is established under this condition).
- bRequestAppAcceptToPair - the application's option to request whether or not to accept the push-button pair procedure by pairing with the successfully discovered device
- timeToWaitAppAcceptToPair - time (in ms) the PBP sublayer waits for the application to respond if it pairs with the successfully discovered device

If an error which can be reported immediately has been encountered, the function call return value contains the error code. In this case, the process is aborted and no further confirmation messages will arrive.

A return value of *gNWSuccess_c* indicates that the push-button pairing process has begun.

When the discovery process is successfully completed, if the bRequestAppAcceptToPair parameter was set to TRUE, the application is notified through a push-button pairing originator continue indication message. This message has the same structure as the network discovery confirm node descriptor message:

```
typedef nodeDescriptor_t pushButtonPairOrigContinueInd_t;
typedef struct nodeDescriptor_tag
{
    uint8_t                 status;
    uint8_t                 recipChannel;
    uint8_t                 recipPanId[2];
    uint8_t                 recipMacAddress[8];
    uint8_t                 recipCapabilities;
    uint8_t                 recipVendorId[2];
    uint8_t                 recipVendorString[gSizeOfVendorString_c];
    appCapabilities_t       recipAppCapabilities;
    uint8_t                 recipUserString[gSizeOfUserString_c];
    uint8_t                 recipDeviceTypeList[gMaxNrOfNodeDeviceTypes_c];
    uint8_t                 recipProfilesList[gMaxNrOfNodeProfiles_c];
    uint8_t                 requestLQI;
}nodeDescriptor_t;
```

It has the following fields:

- status – this field is always set to gNWSuccess_c.
- recipChannel – the logical channel of the discovered device
- recipPanId – the PAN identifier of the discovered device
- recipMacAddress – the IEEE address of the discovered device
- recipCapabilities – the capabilities of the discovered node
- recipVendorId – the vendor identifier of the discovered node
- recipVendorString – the vendor string of the discovered node
- recipAppCapabilities – the application capabilities of the discovered node

- recipUserString – the user defined identification string of the discovered node; this field is present only if the user string specified sub-fields of the recipAppCapabilities is set to one
- recipDeviceTypeList – the list of device types supported by the discovered node
- recipProfilesList – the list of profile identifiers supported by the discovered node
- requestLQI - the LQI of the discovery request command frame reported by the discovered device

When the application receives the push-button pairing originator continue indication message, it must respond by calling PBP_PushButtonPairOrigContinueResponse. If the application does not respond in *timeToWaitAppAcceptToPair* ms from the moment it received the push-button pairing originator continue indication message, the push-button pair process will complete with the *gNWNoResponse_c* status. The PBP_PushButtonPairOrigContinueResponse function has the following prototype:

```
uint8_t PBP_PushButtonPairOrigContinueResponse(
                                                bool_t bContinue
                                                );
```

The bContinue parameter shows the option of the application to accept or not the pair process from the push-button pairing procedure on controller-side.

When the process is complete (whether successful or not) the application is notified through a push-button pairing originator confirm message, which has the following structure:

```
typedef nwkNlmePairCnf_t pushButtonPairOrigCnf_t;
typedef struct nwkNlmePairCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  uint8_t                 recipVendorId[2];
  uint8_t*                recipVendorString;
  appCapabilities_t       recipAppCapabilities;
  uint8_t*                recipUserString;
  uint8_t*                recipDeviceTypeList;
  uint8_t*                recipProfilesList;
}nwkNlmePairCnf_t;
```

It has the following fields:

- status – the status of the push-button pairing process, which can be either *gNWSuccess_c* or it describes the error. See the *ZRC Applications Profile Reference Manual* (ZRCAPRM) for a detailed description of all status checks.
- deviceId – the pair table entry index of the new pairing link
- recipVendorId[2] – the recipient's vendor Id
- recipAppCapabilities – the recipient's application capabilities
- recipUserString – the recipient's user string, actually a pointer to the user string stored the pair table
- recipDeviceTypeList – the list of device types the recipient supports
- recipProfilesList – the list of profiles the recipient supports

If the status field has any value other than *gNWSuccess_c* all the other fields should be ignored.

While push-button pairing is in progress, the receiver is enabled (power saving is disabled) by calling NLME_RxEnableRequest(0x00FFFFFF). The initial value of the activePeriod NIB is saved in an internal variable and restored when push-button pairing is complete. The application should not change the state of the receiver during this time period as pairing may fail. Additionally, the application should not change the value of the parameters that are passed by pointer (the device type list, the profile list and the discovery profile list) as the PBP sublayer will reference them throughout the push-button pairing process.

The Controller Node Demo application in the BeeStack Consumer codebase shows an example of how to handle the push-button pairing process:

```
if((events & gAppEvtStateStart_c) && (appStateMachine.subState == gAppSubStateStart_c))
  {

    UartUtil_Print("\n\rPush Button Pairing... ", gAllowToBlock_d);

   /* Make a push button pairing request.*/
    status = PBP_PushButtonPairOrigRequest(
                                    gAppZRCDiscoverySearchedPanId_c,
                                    gAppZRCDiscoverySearchedShortAddress_c,
                                    gAppZRCDiscoverySearchedDeviceType_c,
                                    localAppCapabilities,
                                    localDeviceTypesList,
                                    localProfilesList,
                                    localAppCapabilities.nrSupportedProfiles,
                                    localProfilesList,
                                    gAppZRCKeyExTransferCount_c,
                                    gAppZRCRequestAppAcceptToPair_c,
                                    gAppZRCTimeToWaitAppAcceptToPair_c
                                  );

    /* Exit the state if this is not successful, otherwise wait confirm */
    if(gNWSuccess_c == status)
    {
      appStateMachine.subState = gAppSubStateWaitCnf_c;
    }
    else
    {
      appStateMachine.subState = gAppSubStateEnd_c;
    }
  }

  switch(appStateMachine.subState)
  {
    case gAppSubStateWaitCnf_c:
      if((events & gAppEvtMsgFromPushButton_c) &&
         (pMsgIn != NULL))
      {
        /* This substate handles both the messages:
            -  gPushButtonPairOrigCnf_c - informing the application that
        the push button pair process has completed
            - gPushButtonPairOrigContinueInd_c - asking for application's aproval
       to continue the push button pair procedure by pairing with the just discovered device */
        if(pPBPMsgIn->msgType == gPushButtonPairOrigCnf_c)
        {
          status = pPBPMsgIn->msgData.pushButtonPairOrigCnf.status;
```

```
          appStateMachine.subState = gAppSubStateEnd_c;
        }
        else if(pPBPMsgIn->msgType == gPushButtonPairOrigContinueInd_c)
        {
          /* The application can decide here whether it wants to continue the push button pair
          process or not, after consulting the information about the recipient provided by the
profile
          in the pushButtonPairOrigContinueInd_t message.
             This demo app always choose to continue with the pair process */
          (void)PBP_PushButtonPairOrigContinueResponse(TRUE);
        }
      }
      break;

    default:
      break;
  }

  if(appStateMachine.subState == gAppSubStateEnd_c)
  {

    /* Print the status */
    App_PrintResult(status);

    if(status == gNWSuccess_c)
    {
      pushButtonPairOrigCnf_t* pPBPOrigCnf = &pPBPMsgIn->msgData.pushButtonPairOrigCnf;

      UartUtil_Print("Successfully paired with device ", gAllowToBlock_d);
      UartUtil_Print(pPBPOrigCnf->recipUserString, gAllowToBlock_d);
      UartUtil_Print(".\n\r", gAllowToBlock_d);
}

    /* Send event to end the state */
    TS_SendEvent(gAppTaskID, gAppEvtStateEnd_c);
  }
```

The process begins when the user selects push-button pairing from the menu. The recipient's device Id is stored and the application enters push-button pairing state with a *gAppEvtStateStart_c* event. The push-button pairing state has three sub-states: start, wait for confirm and end.

In the start sub-state (*gAppSubStateStart_c*) the push-button pairing process is initiated by calling PBP_PushButtonPairOrigRequest. Depending on the return value we change to the next state. A push-button pairing originator confirm message will only arrive if PBP_PushButtonPairOrigRequest returns *gNWSuccess_c*, so we switch to sub-state gAppSubStateWaitCnf_c only in that case, otherwise we switch directly to the end sub-state, which performs some cleanup.

Chapter 3
Target-side Push-button Pairing

Target-side push-button pairing is made easy with the PBP sublayer. The functionality can only be used on target nodes. A node using this feature will start the auto-discovery process followed by the waiting of a pair request frame. The push-button pairing functionality can be used to pair with any RF4CE ZRC compliant device.

3.1 Configuration

The initialization function for the target-side push-button pairing functionality is PBP_InitPushButtonPairRecip.

Aside from initialization no other configuration needs to be done. The application must only ensure that the RF4CE_PushButtonTask and RF4CE_PushButtonRecip libraries are linked.

3.1.1 The Push-button Pairing Process

The application initiates the push-button pairing process by calling PBP_PushButtonPairRecipRequest.

3.1.1.1 Prototype

```
uint8_t PBP_PushButtonPairRecipRequest(
                                        appCapabilities_t origAppCapabilities,
                                        uint8_t*          origDeviceTypeList,
                                        uint8_t*          origProfileIdList,
                                        uint8_t           discLQIThreshold,
                                        bool_t            bRequestAppAcceptToPair,
                                        uint16_t          timeToWaitPairInd,
                                        uint16_t          timeToWaitAppAcceptToPair
                                        )
```

The parameters are as follows:

- origAppCapabilities – the current node's application capabilities
- origDeviceTypeList – the list of device types the current node supports
- origProfileIdList – the list of profiles the current node supports
- discLQIThreshold – the discovery request link quality threshold that must be met by an incoming discovery request frame
- bRequestAppAcceptToPair - the application's option to be asked (or not) if it accepts to continue the push-button pair process by pairing with the device who successfully discovered it and initiated the pair request
- timeToWaitPairInd - time (in ms) the PBP sublayer will wait for a pair indication from the device which successfully discovered the current node. If the value of the timeToWaitPairInd parameter

is less than aplcMaxPairIndicationWaitTime (1000ms) this function sets the timeToWaitPairInd parameter to aplcMaxPairIndicationWaitTime. The parameter offers the possibility to wait for the Pair Indication longer than aplcMaxPairIndicationWaitTime (1000 ms).

- timeToWaitAppAcceptToPair - time (in ms) the PBP sublayer will wait the application to respond if it accepts the pair process with the device which has sent the pair request

If an error which can be reported immediately has been encountered, the function call return value contains the error code. In this case, the process is aborted and no further confirmation messages will arrive.

A return value of *gNWSuccess_c* indicates that the push-button pairing process has begun.

After the auto-discovery process is successfully ended, a pair request from the device which has successfully discovered the current node is expected for *timeToWaitPairInd* ms. If the pair request is not received in this amount of time, *gNWNoResponse_c* status is returned and the push-button pairing process fails.

If a pair request is received from the same device that has successfully discovered the current node and if the bRequestAppAcceptToPair parameter was set to TRUE, the application is notified through a push-button pairing recipient continue indication message that a pair request has arrived. The message has the following structure:

```
typedef nwkNlmePairInd_t pushButtonPairRecipContinueInd_t;
typedef struct nwkNlmePairInd_tag
{
        uint8_t                 status;
        uint8_t                 origPanId[2];
        uint8_t                 origMacAddress[8];
        uint8_t                 origCapabilities;
        uint8_t                 origVendorId[2];
        uint8_t*                origVendorString;
        appCapabilities_t       origAppCapabilities;
        uint8_t*                origUserString;
        uint8_t*                origDeviceTypeList;
        uint8_t*                origProfilesList;
        uint8_t                    keyExTransferCount;
        uint8_t                 deviceId;
}nwkNlmePairInd_t;
```

It has the following fields:

- status – this field will always be set to *gNWSuccess_c* or *gNWDuplicatePairing_c*
- origPanId[2] – the pair's originator PAN Id
- origMacAddress – the pair's originator MAC address
- origCapabilities – the pair's originator node capabilities
- origVendorId[2] – the pair's originator vendor Id
- origVendorString – the pair's originator vendor string
- origAppCapabilities – the pair's originator application capabilities
- origUserString – the pair's originator user string, actually a pointer to the user string stored the pair table

- origDeviceTypeList – the list of device types the pair's originator supports
- origProfilesList – the list of profiles the pair's originator supports
- keyExTransferCount – The number of transfers to use for exchanging the encryption key
- deviceId – the pair table entry index of the new provisional pairing link

When the application receives push-button pairing recipient continue indication message, it must respond by calling PBP_PushButtonPairRecipContinueResponse whether it wants to continue the push-button pair procedure by accepting the pair or not. If the application does not respond in *timeToWaitAppAcceptToPair* ms from the moment it received the push-button pairing recipient continue indication message, the push-button pair process completes with the *gNWNoResponse_c* status. The PBP_PushButtonPairRecipContinueResponse function has the following prototype:

```
uint8_t PBP_PushButtonPairRecipContinueResponse(
                                                bool_t bContinue
                                                );
```

The bContinue parameter shows the option of the application to accept or not the pair request inside the push-button pairing procedure on target-side.

When the process is complete (whether successful or not) the application is notified through a push-button pairing recipient confirm message, which has the following structure (the PBP sublayer reuses the network layer pair indication message, but alters the meaning of the fields):

```
typedef nwkNlmePairInd_t pushButtonPairRecipCnf_t;
typedef struct nwkNlmePairInd_tag
{
        uint8_t                 status;
        uint8_t                 origPanId[2];
        uint8_t                 origMacAddress[8];
        uint8_t                 origCapabilities;
        uint8_t                 origVendorId[2];
        uint8_t*                origVendorString;
        appCapabilities_t       origAppCapabilities;
        uint8_t*                origUserString;
        uint8_t*                origDeviceTypeList;
        uint8_t*                origProfilesList;
        uint8_t                    keyExTransferCount;
        uint8_t                 deviceId;
}nwkNlmePairInd_t;
```

It has the following fields:

- status – the status of the push-button pairing process, which can be either *gNWSuccess_c, gNWDuplicatePairing_c* or it describes the error (for a detailed description of all statuses check the ZRC Reference Manual).
- origPanId[2] – the originator's PAN Id
- origMacAddress – the originator's MAC address
- origCapabilities – the originator's node capabilities
- origVendorId[2] – the originator's vendor Id
- origVendorString – the originator's vendor string

- origAppCapabilities – the originator's application capabilities
- origUserString – the originator's user string, actually a pointer to the user string stored the pair table
- origDeviceTypeList – the list of device types the originator supports
- origProfilesList – the list of profiles the originator supports
- keyExTransferCount – The number of transfers to use for exchanging the encryption key
- deviceId – the pair table entry index of the new pairing link

If the status field has any value other than *gNWSuccess_c* or *gNWDuplicatePairing_c*, all the other fields should be ignored.

While push-button pairing is in progress, the receiver is enabled (power saving is disabled) by calling NLME_RxEnableRequest(0x00FFFFFF). The initial value of the activePeriod NIB is saved in an internal variable and restored when push-button pairing is complete. The application should not change the state of the receiver during this time period as pairing may fail. Additionally, the application should not change the value of the parameters that are passed by pointer (the device type list and the profile list) as the PBP sublayer will reference them throughout the push-button pairing process.

The Target Node Demo application provided with the BeeStack Consumer Codebase shows an example of how to handle the push-button pairing process:

```
if((events & gAppEvtStateStart_c) && (appStateMachine.subState == gAppSubStateStart_c))
  {

    UartUtil_Print("\n\rPush Button Pairing... ", gAllowToBlock_d);

   /* Make a push button pairing request.*/
    status = PBP_PushButtonPairRecipRequest(
                                    localAppCapabilities,
                                    localDeviceTypesList,
                                    localProfilesList,
                                    gAppZRCDiscoveryLQIThreshold_c,
                                    gAppZRCRequestAppAcceptToPair_c,
                                    gAppZRCTimeToWaitPairInd_c,
                                    gAppZRCTimeToWaitAppAcceptToPair_c
                                  );

    /* Exit the state if this is not successful, otherwise wait confirm */
    if(gNWSuccess_c == status)
    {
      appStateMachine.subState = gAppSubStateWaitCnf_c;
    }
    else
    {
      appStateMachine.subState = gAppSubStateEnd_c;
    }
  }

  switch(appStateMachine.subState)
  {
    case gAppSubStateWaitCnf_c:
      if((events & gAppEvtMsgFromPushButton_c) &&
```

```
         (pMsgIn != NULL))
      {
        /* This substate handles both the messages:
            -  gPushButtonPairRecipCnf_c - informing the application that
        the push button pair process has completed
            - gPushButtonPairRecipContinueInd_c - asking for application's approval
        to continue the push button pair procedure by accepting the pair from the originator
device */
        if(pPBPMsgIn->msgType == gPushButtonPairRecipCnf_c)
        {
          status = pPBPMsgIn->msgData.pushButtonPairRecipCnf.status;
          appStateMachine.subState = gAppSubStateEnd_c;
        }
        else if(pPBPMsgIn->msgType == gPushButtonPairRecipContinueInd_c)
        {
          /* The application can decide here whether it wants to continue the push button pair
         process or not, after consulting the information about the originator provided by the
profile
           in the pushButtonPairRecipContinueInd_t message.
              This demo app always choose to continue with the pair process */
          (void)PBP_PushButtonPairRecipContinueResponse(TRUE);
        }
      }
      break;

    default:
      break;
  }

  if(appStateMachine.subState == gAppSubStateEnd_c)
  {

    /* Print the status */
    App_PrintResult(status);

    if(status == gNWSuccess_c)
    {
      pushButtonPairRecipCnf_t* pPBPRecipCnf = &pPBPMsgIn->msgData.pushButtonPairRecipCnf;
      UartUtil_Print("Successfully paired with device ", gAllowToBlock_d);
      UartUtil_Print(pPBPRecipCnf->origUserString, gAllowToBlock_d);
      UartUtil_Print(".\n\r", gAllowToBlock_d);
      NvSaveOnIdle(gNvDataSet_App_ID_c);
    }

    /* Send event to end the state */
    TS_SendEvent(gAppTaskID, gAppEvtStateEnd_c);
  }
```

The process begins when the user selects push-button pairing from the menu. The recipient's device Id is stored and the application enters push-button pairing state with a *gAppEvtStateStart_c* event. The push-button pairing state has three sub-states: start, wait for confirm and end.

In the start sub-state (*gAppSubStateStart_c*) the push-button pairing process is initiated by calling PBP_PushButtonPairRecipRequest. Depending on the return value we change to the next state. A push button pairing confirm message will only arrive if PBP_PushButtonPairRecipRequest returns

gNWSuccess_c, so we switch to sub-state gAppSubStateWaitCnf_c only in that case, otherwise we switch directly to the end sub-state, which performs general cleanup activities.

Chapter 4
ZRC Command Transmit and Receive

The command transmit and receive feature of the ZRC profile automatically constructs the NLDE payload for ZRC commands relieving the application of this burden. It also parses a received NLDE data indication with a ZRC profile ID and presents the individual ZRC command parameters to the application. A device with the Freescale ZRC Profile implementation can exchange commands with any RF4CE ZRC compliant devices.

4.1 Configuration

The initialization function for the ZRC command transmission and reception feature is ZRCProfile_InitCommandTxRx. Aside from initialization no other configuration needs to be done. The application must only ensure that the RF4CE_ZRCProfile_CommandTxRx library is linked.

4.1.1 Command Transmit

To transmit a ZRC command the application needs to call the ZRCProfile_CommandRequest function with the necessary parameters. The ZRCProfile_CommandRequest prototype is as follows:

4.1.1.1 Prototype

```
uint8_t ZRCProfile_CommandRequest(
                                    uint8_t   deviceId,
                                    uint8_t   commandCode,
                                    uint8_t   command,
                                    uint8_t*  vendorId,
                                    uint8_t   payloadLength,
                                    uint8_t*  payload,
                                    uint8_t   txOptions
                                    );
```

The parameters are as follows:

- deviceId – the command recipient
- commandCode – the ZRC command code; will only be used if txOptions does not indicate vendor specific data and can take the following values: gZRC_CmdCode_UserCtrlPressed_c, gZRC_CmdCode_UserCtrlReleased_c, gZRC_CmdCode_DiscoveryRequest_c, gZRC_CmdCode_UserCtrlPressedAndRepeat_c.
- command – the command Id; its meaning depends on the ZRC command code
- vendorId – the vendor Id to be placed in the RF4CE data frame; will only be used if txOptions indicates vendor specific data

- payloadLength – the length of the command payload
- payload – the command payload; its meaning depends on the ZRC command code
- txOptions – the RF4CE NLDE Data service transmission options are passed to the network layer; if data is vendor specific the vendorId is included in the data frame. A detailed description of the RF4CE transmission options can be found in the ZigBee RF4CE specification.

If an error which can be reported immediately has been encountered, the function call return value contains the error code. In this case, the process is aborted and no further confirmation messages will arrive.

A return value of *gNWSuccess_c* indicates that the command transmission process has begun. The ZRC Command Tx/Rx sublayer constructs the NLDE data request payload from the provided parameters and transmits it to the recipient through the network layer. If the txOptions parameter indicates a vendor specific transmission the ZRC command code and ZRC command ID are ignored and the NLDE Data Request payload consists entirely of the ZRC command payload. If the data is not vendor specific the NLDE Data Request payload depends on the ZRC command code, according to the following table:

Table 4-1. NLDE Data Request Payload (Depending on the ZRC Command Code)

ZRC command code	NLDE Data Request payload (in addition to the ZRC command code)
gZRC_CmdCode_UserCtrlPressed_c	The command ID, interpreted as the RC command code + the ZRC command payload. The ZRC profile sends over the air only the User Control Pressed command frame.
gZRC_CmdCode_UserCtrlReleased_c	The command ID; the rest of the parameters are ignored. The command Id should match the command Id of a previous sent gZRC_CmdCode_UserCtrlPressedAndRepeat_c request. If the command Id is successfully matched, the ZRC profile sends over the air the User Control Released command frame.
gZRC_CmdCode_ DiscoveryRequest_c	All the parameters are ignored. The ZRC profile sends over the air the discovery request command frame and waits the response.
gZRC_CmdCode_UserCtrlPressedAndRepeat_c	The command ID, interpreted as the RC command code + the ZRC command payload. The profile sends the User Control Pressed command frame and starts to repeat the RC command ID (sending a User Control Repeated command frames) at an interval specified by the aplKeyRepeatInterval attribute (gZrcAttr. keyRepeatInterval). Each transmission of the user control repeated command frame is signaled to the application only if the Freescale specific attribute receiveKeyRepeatCnf is set TRUE. Otherwise, the profile repeats the RC command ID without notifying the application layer.The RC command is repeated until the application sends a release command (the commandCode parameter in the function should be set to gZRC_CmdCode_UserCtrlReleased_c). The release command has to specify the right RC command ID to end the repetitions. If the ZRC command ID to be released is not valid, the profile will continue to repeat the RC command ID.

NOTE

The ZRC profile can simultaneously process a maximum of two requests (requests for sending vendor specific data and ZRC commands) at a time. If more than two requests are sent at a time, the profile will only process the first two requests and the rest will be denied.On receipt, the ZRC profile can simultaneously handle the repetitions for a maximum of two RC commands (e.g. two RC buttons are simultaneously held down).

When the transmission of the ZRC command Discovery Request (using gZRC_CmdCode_ DiscoveryRequest_c command code) is completed (whether successful or not), the application is notified through a ZRC Discovery Command Confirm message, which has the following structure:

```
typedef struct zrcProfileDiscoveryCmdCnf_tag
{
  uint8_t     status;
  uint8_t     deviceId;
  uint8_t     cmdSupportedBitMap[gCmdsSupportedFieldLength_c];
}zrcProfileDiscoveryCmdCnf_t;
```

It has the following fields:

status – how the transmission request was completed, this is the status from the NLDE data confirm message

deviceId – identifies the command recipient.

```
cmdSupportedBitMap - the bitmap containing the supported commands of the remote note.
```

NOTE

Usually, the command Discovery Request is sent to the remote node when the pairing process is successfully ended. On the remote node, the application should keep the receiver open for a while, so that to receive the Discovery Request command frame. When Discovery Request frame is received, the remote node will respond by sending a Discovery Response frame which contains the ZRC supported commands. The supported ZRC commands are kept in gaZRCCmdSupportedBitMap bitmap which can be configured from `ZRCProfileCommands.h` file.

When any other command transmission (except ZRC command Discovery Request) is completed, the application is notified by the profile through a ZRC command Confirm message, which has the following structure:

4.1.1.2 Message Structure

```
typedef struct zrcProfileCommandCnf_tag
{
  uint8_t     status;
  uint8_t     deviceId;
  uint8_t     commandCode;
  uint8_t     command;
}zrcProfileCommandCnf_t;
```

It has the following fields:

status – how the transmission request was completed, this is the status from the NLDE data confirm message

deviceId – identifies the command recipient.

commandcode - the ZRC command code; will only be used if txOptions does not indicate vendor specific data.

command - the command Id; its meaning depends on the ZRC command code

4.1.2 Command Receive

Whenever a NLDE data indication message with a ZRC profile ID arrives at the profile layer, a ZRC Command Indication message is sent to the application. It has the following structure:

4.1.2.1 Message Structure

```
typedef struct zrcProfileCommandInd_tag
{
  uint8_t     deviceId;
  uint8_t     dataLength;
  uint8_t     vendorId[2];
  uint8_t     LQI;
  uint8_t     rxFlags;
  uint8_t     commandCode;
  uint8_t     command;
  uint8_t*    pData;
}zrcProfileCommandInd_t;
```

The fields have the following significance:

- deviceId – identifies the command originator
- dataLength – the length of the command payload
- vendorId – the originator's vendor ID; should be ignored if the rxFlags parameter does not indicate vendor specific data
- LQI – the link quality of the received data frame
- rxFlags – the NLDE Data Indication rxFlags, see the ZigBee RF4CE specification for details
- commandCode – the command action; should be ignored if the rxFlags parameter indicates vendor specific data
- command – the command ID; its significance depends on the ZRC command code
- pData – the command payload; consists of the entire NLDE Data Indication payload if rxFlags indicates vendor specific data; otherwise its significance depends on the ZRC command code.

If the data is vendor specific (i.e. the vendor specific data bit in the rxFlags field is set), the *commandCode* and *command* fields should be ignored and the ZRC Command payload contains the full NLDE Data Indication payload. If the data is not vendor specific, then the *command* and *pData* parameters have the

following significance, depending on the ZRC command code (i.e. depending on the value of the *commandCode* field).

Table 4-2 lists command significance.

Table 4-2. Significance of the Command and pData parameters (Depending on the ZRC Command Code)

ZRC command code	NLDE Data Request payload (in addition to the ZRC command code)
gZRC_CmdCode_UserCtrlPressed_c gZRC_CmdCode_UserCtrlRepeated_c	command contains the RC command code and pData contains the RC command payload (if any)
gZRC_CmdCode_UserCtrlReleased_c	the pData parameter should be ignored

4.2 ZRC Attributes

The ZRC attributes are declared and initialized in the `ZRCProfileGlobals.c` file as follows:

```
zrcAttrData_t gZrcAttr ={
                         gDefaultKeyRepeatInterval_c,
                         gDefaultKeyRepeatWaitTime_c,
                         gDefaultExTransferCount_c,
                         gDefaultReceiveKeyRepeatCnf_c
                        };

typedef struct zrcAttrData_tag{
  uint8_t  keyRepeatInterval;
  uint16_t keyRepeatWaitTime;
  uint8_t  keyExTransferCount;
  uint8_t  receiveKeyRepeatCnf;
}zrcAttrData_t;
```

Each attribute has the following meaning:

keyRepeatInterval - The interval in milliseconds at which user command repeat frames will be transmitted (a key pressed is followed by key repetitions – using gZRC_CmdCode_UserCtrlPressedAndRepeat_c command code).

keyRepeatWaitTime - The duration that a recipient of a user control repeated command frame waits before terminating a repeated operation.

keyExTransferCount - The value of the KeyExTransferCount parameter passed to the pair request primitive during the push button pairing procedure.

gDefaultReceiveKeyRepeatCnf_c – is a Freescale-specific attribute and when it is set (TRUE) the profile signals the application (via a ZRC Command Confirm message) that an user control repeated command frame was sent (whether successful or not) over the air.

These attributes can be configured at the initialization by modifying the macros from the ZRCProfileGlobals.h file.

To set or read these attributes you can use the ZRCProfile_SetRequest() and ZRCProfile_GetRequest() functions which are described in the ZRC Application Profile Reference Manual.