Freescale BeeStack™

Application Development Guide for ZigBee 2007

Document Number: BSADGZB2007
Rev. 1.3
07/2011

Contents

Chapter 1
Introduction

Chapter 2
Building A Custom Application

Chapter 3
Designing A Custom Profile

Chapter 4
Selecting Platform Components

Chapter 5
Managing BeeStack Resources

Chapter 6
Debugging BeeStack Applications

Chapter 7
Creating a Custom BeeStack BlackBox Application

Chapter 8
Creating a ZigBee Host Application

About This Book

The *BeeStack Application Development Guide for ZigBee 2007* describes how to develop an application for the ZigBee 2007 compliant BeeStack , including discussions on major considerations for commercial applications.

Audience

This document is intended for software developers who write applications for BeeStack-based products using Freescale development tools.

It is assumed the reader is a programmer with at least rudimentary skills in the C programming language and that the reader is already familiar with the edit/compile/debug process.

Organization

This document is organized into the following chapters.

Chapter 1	Introduction – provides an overview of the BeeStack Application Development Guide, including what's included and what is not in the guide. It also describes the basic development process using both BeeKit and CodeWarrior (only in concept. This guide is not a user guide for either BeeKit or CodeWarrior).
Chapter 2	Building A Custom Application – provides a step-by-step example of creating a custom sample application.
Chapter 3	Designing A Custom Profile – describes designing a new custom-profile application, including selecting a profile, clusters, attributes and endpoints. It also describes ZigBee 2007 security options.
Chapter 4	Selecting Platform Components – describes selecting the appropriate hardware-related platform components, including the use of non-volatile memory, LEDs, the keyboard, serial port, and general hardware selection.
Chapter 5	Managing BeeStack Resources – describes using the non-hardware-related platform components appropriately, including the use of timers, messages, data queues, the task scheduler, Non-volatile-memory and low power library. It also describes how to determine how much RAM and Flash is available to the application and what to do if an application exceeds memory size.
Chapter 6	Debugging BeeStack Applications – describes how to debug an application that may not work, including use of the BDM, LEDs, ZigBee Test Client and Sensor Network Analyzers.
Chapter 7	Creating a Custom BeeStack BlackBox Application - Describes how to create a custom BeeStack BlackBox Application.
Chapter 8	Creating a ZigBee Host Application - Describes how to create a custom ZigBee Host Application.

Revision History

The following table summarizes revisions to this document since the previous release (Rev. 1.2).

Revision History

Location	Revision
Entire document	Multiple changes through entire document for CodeWarrior 10 and MC1323x.

Conventions

This *BeeStack Documentation Overview* uses the following formatting conventions when detailing commands, parameters, and sample code:

`Courier mono-space` type indicates commands, command parameters, and code examples.

Bold style indicates the command line elements, which must be entered exactly as written.

Italic type indicates command parameters that the user must type in or replace, as well as emphasizes concepts or foreign phrases and words.

Definitions, Acronyms, and Abbreviations

ACK	Acknowledgement
ADC	Analog to digital converter
AF	Application framework
AIB	Application support sub-layer information base
APDU	Application support sub-layer protocol data unit
API	Application programming interface
APL	Application layer
APS	Application support sub-layer
APSDE	APS data entity
APSDE-SAP	APS data entity - service access point
APSME	APS management entity
APSME-SAP	APS management entity - service access point
ASDU	APS service data unit
Binding	Matching ZigBee devices based on services and needs
BTR	Broadcast transaction record, the local receipt of a broadcast message
BTT	Broadcast transaction table, holds all BTRs
CBC-MAC	Cipher block chaining message authentication code
CCA	Clear channel assessment
Cluster	A collection of attributes associated with a specific cluster-identifier
Cluster identifier	An enumeration that uniquely identifies a cluster within an application profile

CSMA-CA	Carrier sense multiple access with collision avoidance
CTR	Counter
Data Transaction	Process of data transmission from the endpoint of a sending device to the endpoint of the receiving device
Device/Node	ZigBee network component containing a single IEEE 802.15.4 radio
Direct addressing	Direct data transmission including both destination and source endpoint fields
Endpoint	Component within a unit; a single IEEE 802.15.4 radio may support up to 240 independent endpoints
IB	Information base, the collection of variables configuring certain behaviors in a layer
IEEE	Institute of Electrical and Electronics Engineers, a standards body
Indirect addressing	Transmission including only the source endpoint addressing field along with the indirect addressing bit
ISO	International Standards Organization
LCD	Liquid crystal display
LED	Light-emitting diode
LQI	Link quality indicator or indication
MAC	Medium access control sub-layer
MCPS-SAP	MAC common part sub-layer - service access point
MIC	Message integrity code
MLME	MAC layer management entity
MLME-SAP	MAC sub-layer management entity service access point
NIB	Network layer information base
NLDE	Network layer data entity
NLDE-SAP	Network layer data entity - service access point
NLME	Network layer management entity
NLME-SAP	Network layer management entity - service access point
NPDU	Network protocol data unit
NSDU	Network service data unit
NVM	Non-volatile memory
NWK	Network layer
Octet	Eight bits of data, or one byte
OSI	Open System Interconnect
PAN	Personal area network
PD-SAP	Physical layer data - service access point
PDU	Protocol data unit (packet)

PHY	Physical layer
PIB	Personal area network information base
PLME-SAP	Physical layer management entity - service access point
Profile	Set of options in a stack or an application
RF	Radio frequency
SAP	Service access point
SKG	Secret key generation
SKKE	Symmetric-key key establishment protocol
SSP	Security service provider, a ZigBee stack component
Stack	ZigBee protocol stack
UART	Universal asynchronous receiver transmitter
WDA	wireless demo application
WPAN	wireless personal area network
ZDO	ZigBee device object(s)
ZDP	ZigBee device profile
802.15.4	An IEEE standard radio specification that underlies the ZigBee Specification

Reference Materials

This following served as references for this manual:

1. Document 053474r17, *ZigBee Specification*, ZigBee Alliance, October 2007
2. Document 075123r00, *ZigBee Cluster Library Specification*, ZigBee Alliance, July 2007
3. Document 053520r24, *Home Automation Profile Specification*, ZigBee Alliance, September 2007

Chapter 1
Introduction

Freescale's BeeStack is a complete, robust implementation of the ZigBee 2007 networking specification. BeeStack applications typically are used in wireless sensor and control networks.

This section provides an overview of *BeeStack Application Development Guide for ZigBee 2007*, describing what is and what is not included in the guide. It also describes the basic development process for BeeStack applications using both BeeKit and CodeWarrior.

This guide is a "how-to" guide that leads a developer through the process of developing BeeStack applications. It also includes advice on building robust networks and managing network resources.

1.1   What This Document Describes

This guide describes the following:

- How to build and customize BeeStack applications for use in wireless sensor and control applications
- A step-by-step example of modifying a BeeStack application
- How to design a custom ZigBee application profile and the intended use of application profiles, endpoints, clusters and attributes. It includes how to manage bandwidth and channels
- A suggested process for selecting the appropriate hardware platform components
- Suggestions on how best to use the Freescale task scheduler, timers, memory and other platform resources
- How to debug an application
- A step by step example of how to create a BeeStack BlackBox application

1.2   What This Document Does Not Describe

This guide does not describe the following:

- How to install BeeKit. For the BeeKit installation process, see the *BeeKit Wireless Connectivity Toolkit User's Guide*
- How to install CodeWarrior. For instructions, see the CodeWarrior documentation
- The complete BeeStack API in detail. For the BeeStack API for ZigBee 2007, see the *BeeStack Software Reference Manual for ZigBee 2007*
- How to port applications from any other stack including any implementation of the ZigBee 2004 specification.
- How to port from Freescale BeeStack for ZigBee 2006 to BeeStack for ZigBee 2007. (See the *BeeStack Software Reference Manual for ZigBee 2007)*

- How to port BeeStack to a custom board.
- ZigBee networking in general. For an overview of ZigBee, see the *BeeStack Software Reference Manual for ZigBee 2007* and the *ZigBee Specification*
- How to use the BeeStack Sample Applications. For the user interface to the sample applications, see *Freescale ZigBee Application User's Guide for ZigBee 2007*

1.3   BeeKit

BeeKit is a desktop PC graphical application that allows developers to configure Freescale networking solutions, including BeeStack, BeeStack Consumer (RF4CE), IEEE® 802.15.4 MAC, and the Freescale proprietary Simple MAC (SMAC). Figure 1-1 shows the BeeKit Wireless Connectivity start-up window.
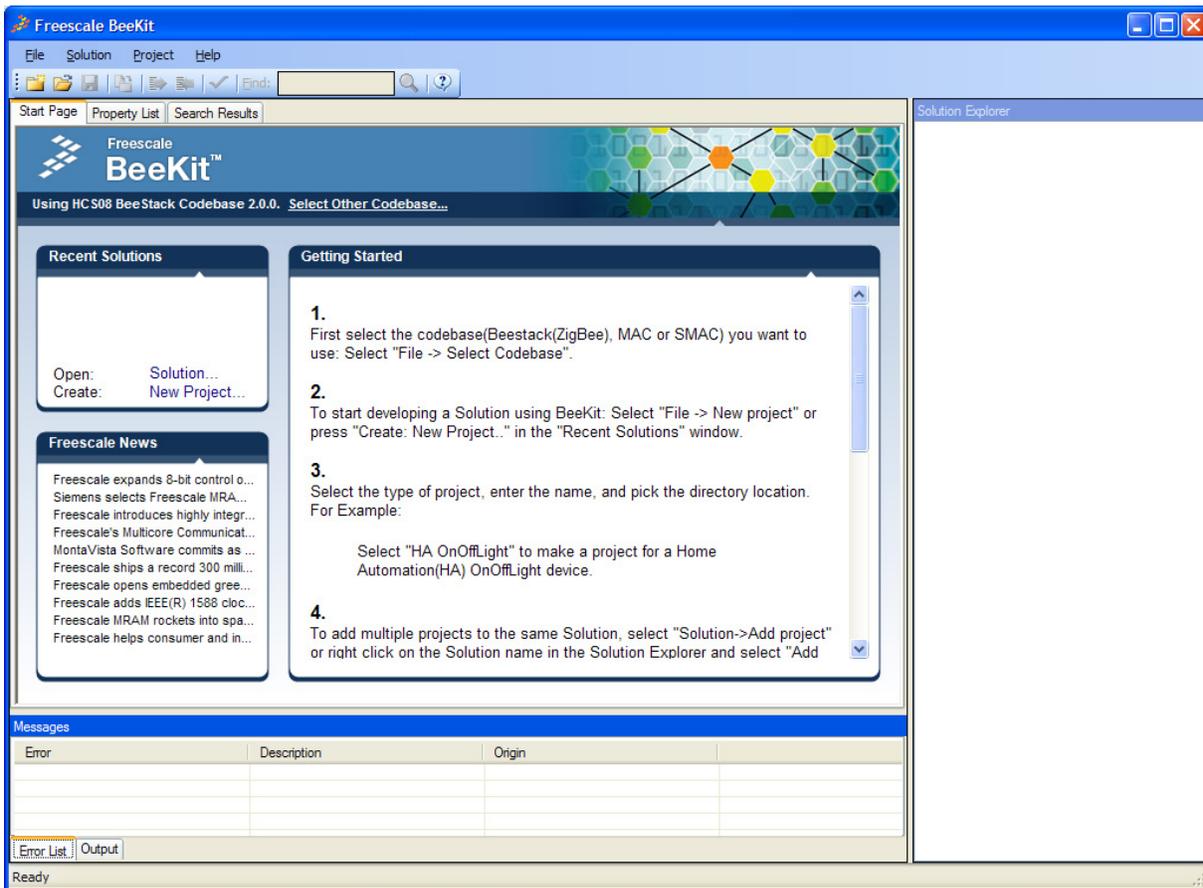
Figure 1-1. BeeKit Starting Window

BeeKit creates a sample application from templates, providing the ability to set properties (also called compile-time options) which configure the application and BeeStack. The resulting project may then be exported as an XML file to a file folder and imported into CodeWarrior or IAR Embedded WorkBench for editing, compiling and debugging. In addition, BeeKit provides a quick-start wizard that can prepare and configure sample applications in moments.

BeeKit provides compile-time configuration of BeeStack and applications during the entire life of the project.

See the BeeKit documentation for more information.

1.4 CodeWarrior

CodeWarrior is a desktop PC integrated development environment (IDE) which includes a C compiler for the HCS08 core MCUs and the other tools to generate a downloadable image as well as a debugger that can download code into the MCU's flash memory.
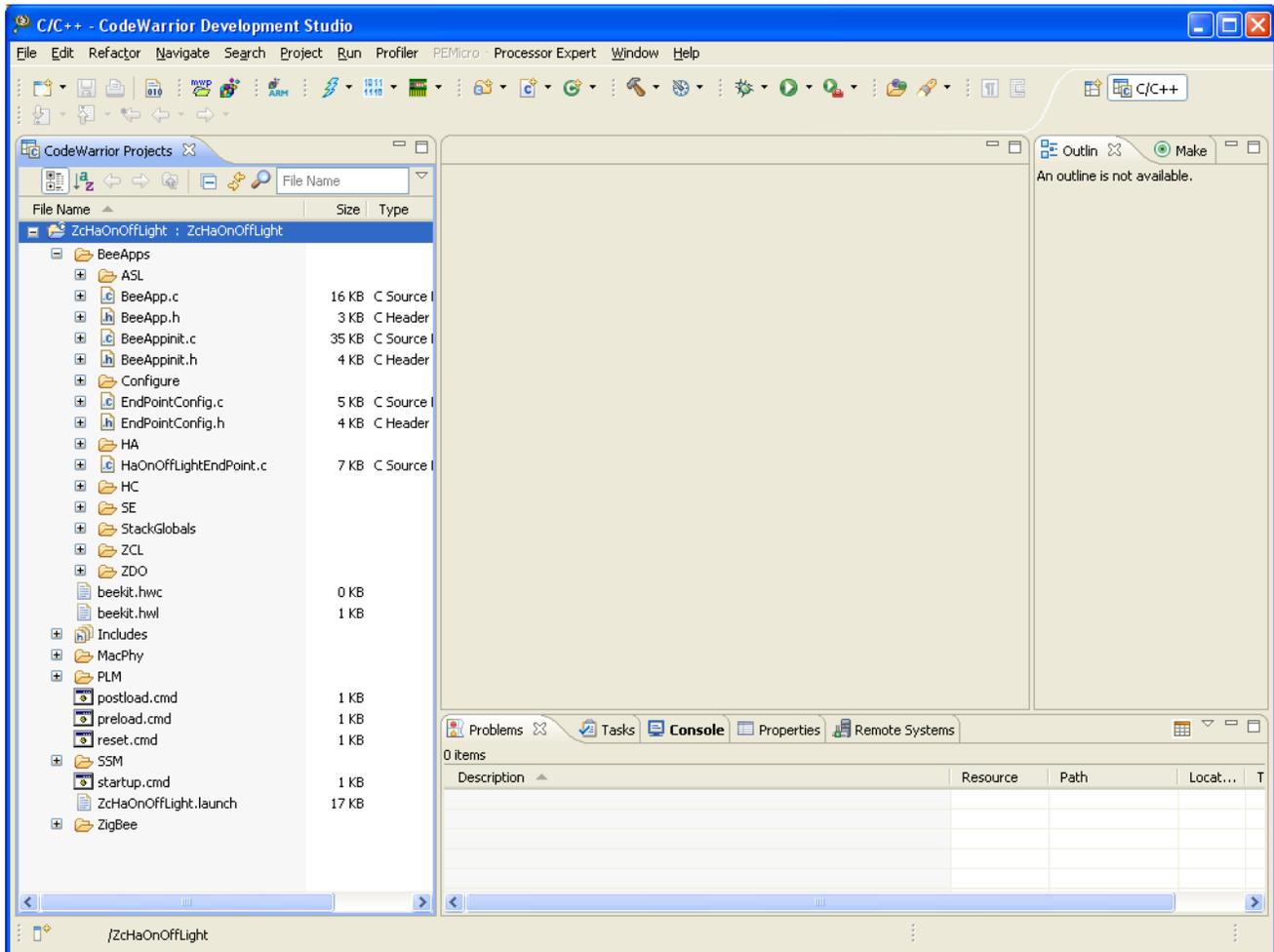
Figure 1-2. Freescale CodeWarrior

CodeWarrior takes the output of BeeKit and compiles and links the C source code and libraries into a binary image that may be downloaded into the FLASH memory of an HCS08 MCU using the background debug memory (BDM) port.

See the CodeWarrior documentation for more information.

1.5   BeeStack

BeeStack is the term used to describe all of the software placed into target boards, with the exception of the application. BeeStack is comprised of ZigBee networking components, which provide access to ZigBee networking functionality, and platform components, which provide a framework for the application to operate and access the hardware.

The components of BeeStack are shown in Figure 1-3.
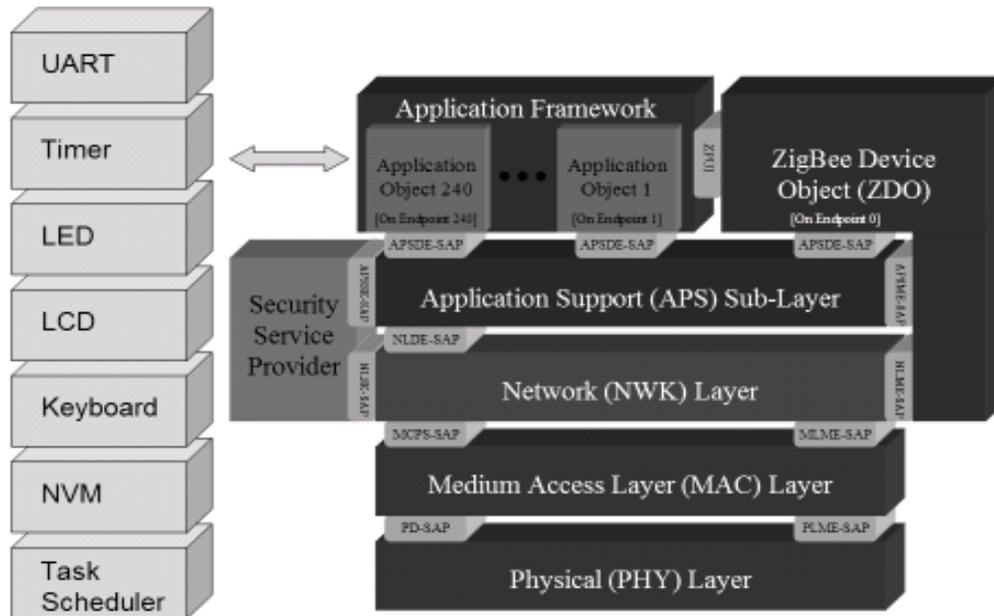
Figure 1-3. BeeStack Components

The networking (NWK) task in BeeStack is responsible for routing packets, including broadcasting, route discovery, unicasting and rejecting packets not for this node or network.

The Application Support Sub-layer (APS) task is responsible for delivering and receiving application data, including binding endpoints, and end-to-end acknowledgements. APS also contains the authentication process for secure networks, including the trust center on ZigBee Coordinator (ZC) nodes.

The Application Framework (AF) task is responsible for delivery of data indications and confirms to the application endpoints.

The ZigBee Device Object (ZDO) task is responsible for the state of the network, and it includes functions to join and leave the network.

The ZigBee Device Profile (ZDP) task handles requests and responses for a set of common over-the-air ZigBee commands for managing nodes within the network. For example, any node may ask for the IEEE (or MAC) address of any other node in the network using a ZDP command.

The various platform management (PLM) components are responsible for interacting with the hardware such as switches, LEDs, the LCD or timers. All of the PLM components may be customized for a particular application.

1.6   The Development Process

Developing applications for BeeStack is similar to any embedded development. The addition of BeeKit makes starting and configuring a new application very easy. The steps are as follows.

1. Design the application.
2. Use BeeKit to create the application framework from a template and to configure the application to include the appropriate components, property settings and endpoint settings.
3. Export the application solution from BeeKit and import it into CodeWarrior.
4. Edit the application as necessary, adding custom code.
5. Compile the application.
6. Download the application into a target board.
7. Debug the application (see Chapter 6, "Debugging BeeStack Applications").
8. Repeat steps 4-7 as necessary.
9. If creating more than one application, use BeeKit to add another application to the BeeKit solution and repeat steps 2-8 as necessary.

Chapter 2
Building A Custom Application

This chapter provides a step-by-step example of how to create a sample custom application. Chapter 3, "Designing A Custom Profile" and Chapter 4, "Selecting Platform Components" explain in more detail about how to build applications for BeeStack.

The general process for creating this custom application is as follows:

- Create the project in BeeKit from an existing application template, making custom property settings and endpoint settings
- Export the project from BeeKit
- Import the project into CodeWarrior
- Edit the application in CodeWarrior to remove unneeded functionality from the template code and add the new functionality of the application
- Compile the custom application with CodeWarrior
- Download the custom application in the target board with CodeWarrior
- Debug the custom application

When building a custom application, always start with a template application in BeeKit. In this case the example will start with the Generic Application Template and transform it into a custom application.

The Generic Application Template by default uses the accelerometer hardware available in some of the Freescale boards to determine tilt of the board and transmit this data to a remote node for display on that remote node. The same code is used for both the accelerometer and display nodes (that is, a node can assume either role).

The custom application described in this chapter will ignore the accelerometer; it will simply flash a light (LED2) on the remote display for a one-second period.

NOTE

Both the Generic Application and this Custom Application use what is called a private profile. Private profiles are useful for those application that do not need to interoperate at an ZigBee application level with other vendors' applications. For Public Profiles which do interoperate (such as a Home Automation On/Off Light and Switch), see the *ZigBee Cluster Library Reference Manual*.

The keys for the Custom Application will be as follows:

Table 2-1. Custom Application Keys

Switch	Description
SW1	Form (ZC) or join (ZR, ZED) the network
SW2	Flash remote light (LED2)
Long SW2	No action
SW3	Find a remote node with a light for sending light commands to
SW4	No action

2.1   Creating a Custom Application In BeeKit

This section describes the steps required in BeeKit to create the custom application. This is not a BeeKit tutorial and assumes users have some familiarity with BeeKit.

1. Create a new project starting from Other ZigBee Applications -> GenericApp named ZcQe128CustomApp. The solution should be named CustomApp. This example uses the Freescale QE128 board. If using another board for the ZigBee Coordinator (such as the MC1323x boards), use the name of that board in the name of the project. Using the three-part naming convention for projects, with the ZigBee node type (ZC, ZR or ZED), the Freescale board type (QE128), and the application name (CustomApp) allows any project to be easily recognized by name. The location of this project should be the folder: `C:\BeeStack`. Also users may select another path, but this path should be known.

In the BeeStack configuration wizard, make sure to select the MC1320x-S08QE128-EVB board as the hardware target (or the board already chosen), with LEDs, Keyboard, LCD module enabled in the Platform Modules Page, ZTC disabled, ZigBee Coordinator device type, no security without mesh routing network type Stack Profile 1, default BeeStack Configuration Parameters, default extended address and PAN ID and channel 25 as the default channel.

2. Modify the endpoint and simple descriptor to contain the information depicted in Figure 2-2 (endpoint number 1, profile 0xc021, application id 0x1234, an input and output OnOff cluster 0x0100).

3. To edit the endpoint, click on the button in the BeeKit window as circled in Figure 2-1. This button will only be available if the "Generic Endpoint" is selected in the Solution Explorer window.
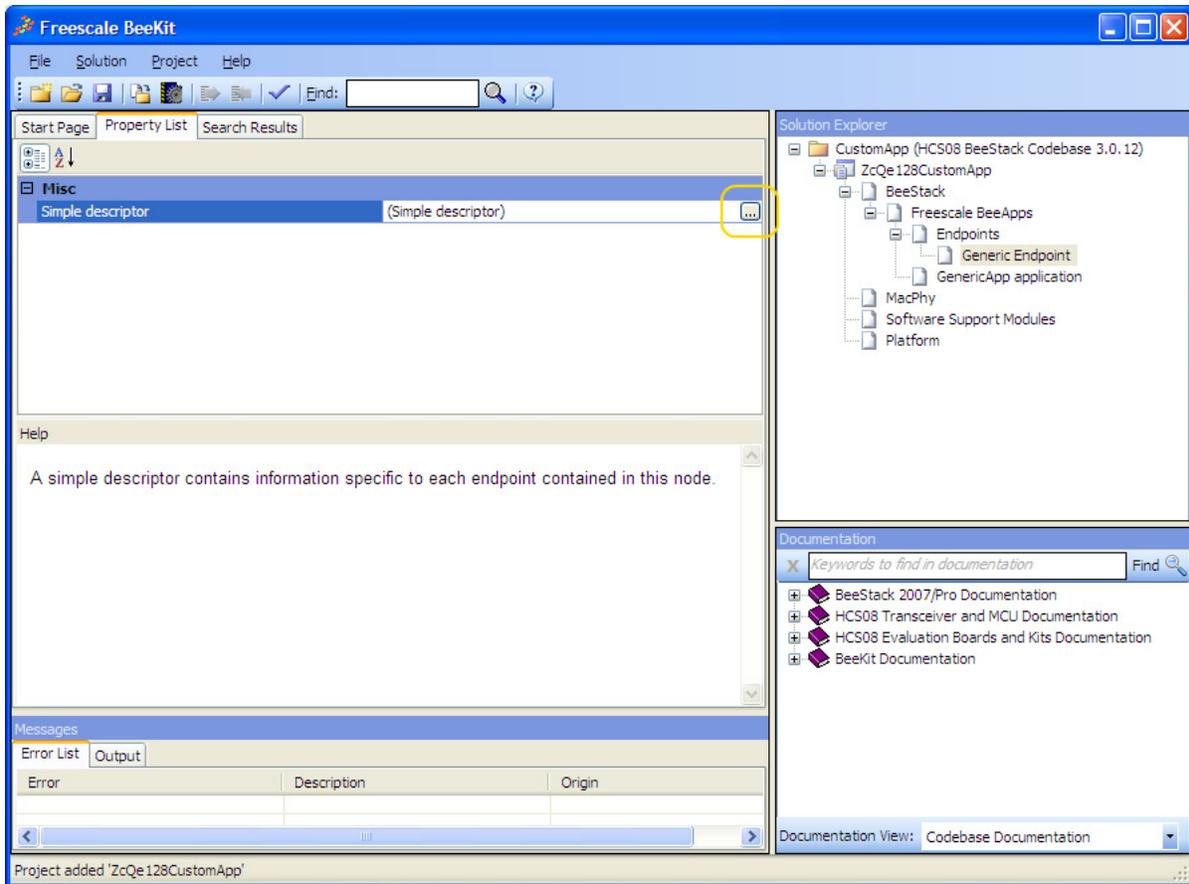
Figure 2-1. Modifying Endpoints in BeeKit

After clicking this button, the Simple Descriptor Editor window appears as shown in Figure 2-2. Modify the endpoint's simple descriptor to contain the same information as shown in Figure 2-2.
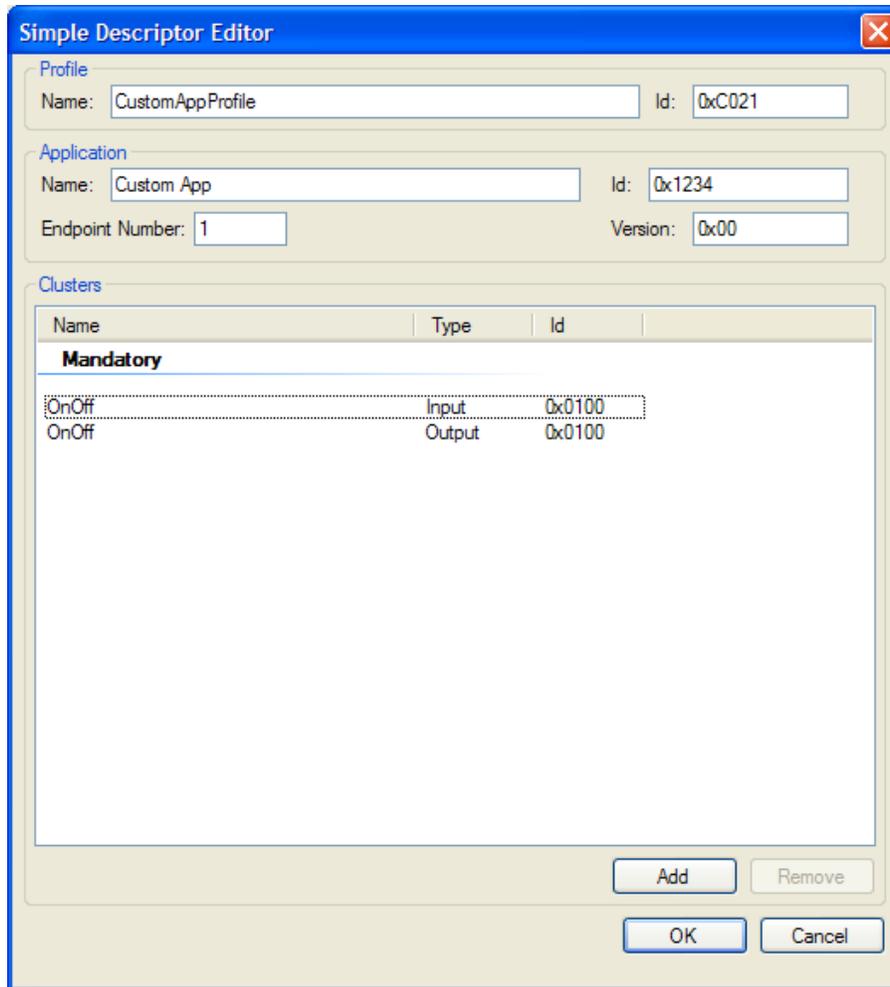
Figure 2-2. Custom Application Simple Descriptor

4. Add another project in the same solution starting from GenericApp named ZedQe128CustomApp (from the menu, use Solution ->Add New Project…). This project will be used with another Freescale QE128 board. Again, if selecting a different Freescale board, change the project name accordingly. This time, make sure to select ZigBee node type of ZigBee End Device in the BeeStack configuration wizard on the "Select ZigBee Device Type" page. Keep default low power setting. All other settings are the same as in Step 1.

5. Change the endpoint's simple descriptor as described in Step 2.

6. Export the solution to the folder `C:\BeeStack\CustomApp` (from the menu, use Solution ->Export Solution…). Users can employ another path; BeeKit and CodeWarrior place no restriction on using another path.

Once the solution (containing two projects) is exported, the following two directories will exist:

```
C:\BeeStack\CustomApp\ZcQe128CustomApp
C:\BeeStack\CustomApp\ZedQe128CustomApp
```

At this point, the application is ready for importing into CodeWarrior and editing to contain custom code.

2.2   Editing the Custom Application in CodeWarrior

This section describes the steps involved to transform the source code from the Generic Application Template to the custom application.

1. Start CodeWarrior 10.1 and choose an workspace.

2. Import the project that was exported by BeeKit into CodeWarrior. From the menu, choose File ->Import -> Existing Projects into Workspace… and navigate to `BeeStack\CustomApp`. Select both Zc and Zed projects to be imported into CodeWarrior.

3. Compile the project to make sure everything in BeeKit and CodeWarrior worked. The projects should compile without warnings or errors. To compile, click on the build icon as shown in Figure 2-3 after.

Figure 2-3. Build Icon

4. Edit the `BeeApp.c` file of the ZcQe128CustomApp to contain the custom code. This is the longest step and requires a number of edits as outlined below. `BeeApp.c` can be found in the BeeApps folder.

In `BeeApp.c`, find the BeeAppInit() function. The key combination Ctrl-L in the CodeWarrior editor will go to a specified line. Ctrl-F will find text.

In BeeAppInit(), change the name of the application. Change the following lines from:

```
/* indicate the app on the LCD */
LCD_WriteString(2, "Accelerometer");
```

To:

```
/* indicate the app on the LCD */
LCD_WriteString(2, "CustomApp");
```

In BeeAppInit(), remove the accelerometer timer and initialization code. Change the following lines from:

```
/* allocate timers for use by this application */
appTimerId = TMR_AllocateTimer();
accelModeTimerId = TMR_AllocateTimer();

/* initialize accelerometer */
AccelerometerInit();
```

To:

```
/* allocate timers for use by this application */
appTimerId = TMR_AllocateTimer();
```

Next, modify the defines for the Accelerometer to be those for the custom application. Find these defines:

```
/* BeeAppTask events */
#define accelEventReport_c  (1 << 0)  /* send a report */
#define accelEventState_c   (1 << 1)  /* move on to next state */
#define accelEventDisplay_c (1 << 2)  /* display data */
```

Add an event, so the lines read:

```
/* BeeAppTask events */
```

```
#define accelEventReport_c  (1 << 0)  /* send a report */
#define accelEventState_c   (1 << 1)  /* move on to next state */
#define accelEventDisplay_c (1 << 2)  /* display data */
#define customAppTurnOffLed2_c  (1 << 3)  /* turn off light */
```

Find the function, BeeAppTask(). Find the lines that read:

```
/* display the accelerometer data */
if(events & accelEventDisplay_c) {
AccelerometerDisplayData();
}

/* report accelerometer data */
if(events & accelEventReport_c) {

/* report data over-the-air (assumes SW3 has been pressed to find display) */
AccelerometerReportData();

/* start up state machine again */
giAccelDemoState = accelStateStart_c;
TS_SendEvent(gAppTaskID, accelEventState_c);
}

/* handle accelerometer events */
if(events & accelEventState_c) {
AccelerometerStateMachine(giAccelDemoState);
}
```

And replace these lines in BeeAppTask() with the following lines of code:

```
if(events & customAppTurnOffLed2_c)
LED_SetLed(LED2, gLedOff_c);
```

Next, find the function BeeAppHandleKeys(). Find the lines that read as follows, and remove them:

```
uint8_t led;
uint8_t accelData;
```

Next find the lines in BeeAppHandleKeys() that read:

```
case gKBD_EventSW2_c:
    /* walk through value of accelerometer */
    accelData = gaAccelDemoXYZ[giAccelIndex];
    if(accelData < accelDemo1Led_c)
      accelData = accelDemo1Led_c;
    else if(accelData < accelDemo2Leds_c)
      accelData = accelDemo2Leds_c;
    else if(accelData < accelDemo3Leds_c)
      accelData = accelDemo3Leds_c;
    else if(accelData < accelDemo4Leds_c)
      accelData = accelDemo4Leds_c;
    else
      accelData = 0;
    gaAccelDemoXYZ[giAccelIndex] = accelData;

    /* display on LEDs, LCD */
    TS_SendEvent(gAppTaskID, accelEventDisplay_c);
    break;
```

And replace them with:

```
case gKBD_EventSW2_c:
     CustomAppToggleLED2();
     break;
```

Remove the code from the case statements for gKBD_EventSW4_c and gKBD_EventLongSW2_c so they have no action, as shown below:

```
case gKBD_EventSW4_c:
     break;
```

Find the `BeeAppDataIndication()` function. Find the text that reads:

```
if(pIndication->aClusterId[0] == appDataCluster[0]) {

    /* indicate we're the display */
    gfAccelIsDisplay = TRUE;

    /* get the new accelerometer readings */
    FLib_MemCpy(gaAccelDemoXYZ, pIndication->pAsdu, sizeof(gaAccelDemoXYZ));

    /* update display with new data */
    TS_SendEvent(gAppTaskID, accelEventDisplay_c);
  }
```

And change that text to read:

```
if(pIndication->aClusterId[0] == appDataCluster[0]) {
      LED_SetLed(LED2, gLedOn_c);
      TMR_StartSingleShotTimer(appTimerId, 1000, CustomAppTimerCallBack);

      /* update display with new data */
      TS_SendEvent(gAppTaskID, accelEventDisplay_c);
  }
```

Add the following two functions at the end of the file:

```
void CustomAppTimerCallBack
  (
  tmrTimerID_t timerId  /* IN: */
  )
{
  (void)timerId;  /* to prevent compiler warnings */

  TS_SendEvent(gAppTaskID, customAppTurnOffLed2_c);
}

void CustomAppToggleLED2
  (
  void
  )
{
  afAddrInfo_t  addrInfo;

  /* don't have a place to send data to, give up */
  if(!gfAccelFoundDst)
    return;

  /* set up address information */
      addrInfo.dstAddrMode = gZbAddrMode16Bit_c;
  Copy2Bytes(addrInfo.dstAddr.aNwkAddr, gaAccelDstAddr);
```

```
addrInfo.dstEndPoint = gAccelDstEndPoint;
addrInfo.srcEndPoint = appEndPoint;
addrInfo.txOptions = gApsTxOptionNone_c;
addrInfo.radiusCounter = afDefaultRadius_c;

/* set up cluster */
Copy2Bytes(addrInfo.aClusterId, appDataCluster);

/* send the data request */
(void)AF_DataRequest(&addrInfo, 10, "ToggleLed2", NULL);
}
```

Finally, add prototypes for those functions in the "Private Prototypes" section of the file.

```
void CustomAppTimerCallBack ( tmrTimerID_t timerId );
void CustomAppToggleLED2(void);
```

At this point, all changes are made to the code. Press Ctrl-S to save the file.

5. Make sure the code compiles without errors or warnings (if users cut and paste from this document, it should). Resolve any compiler warnings or errors before running the custom application. (See Section 2.3, "Installing and Running The Custom Application")

6. Copy the `BeeApp.c` file created in the `BeeStack\CustomApp\ZcQe128CustomApp\BeeApps` directory to the `BeeStack\CustomApp\ZedQe128CustomApp\BeeApps` directory. This overwrites the previous `BeeApp.c` in that directory.

7. Import and compile the ZedSrbCustomApp application. CodeWarrior allows multiple projects to be open at the same time.

2.3 Installing and Running The Custom Application

This section describes how to download the custom application created in the previous section.

1. Connect the P&E USB Multilink pod to the BDM port on the QE128 Zc board. Click the debug icon in the ZcQe128CustomApp project to download the code to the QE128 board. Note: the red portion of the ribbon cable should be toward the edge of the board. The 6-pin connector is labelled BDM.

2. Connect the P&E USB Multilink pod to the BDM port on the QE128 Zed board. Click the green debug icon in the ZedQe128CustomApp project to download the code to the QE128 board. Note: the red portion of the ribbon cable should be toward the edge of the board. The 6-pin connector is labelled BDM. Disconnect the BDM pod.

3. Reset each board. Press SW1 on each board. The LEDs should chase each other for a few seconds while BeeStack forms a ZigBee network.

4. Tell each board to find the other in the network by pressing SW3 on each board. LED3 should light indicating a remote custom application was found.

5. Press SW2 on the device that has LED3 on to toggle the remote LED2 on for 1 second.

2.4 Examining the Custom Application

In addition to using the tools, this example demonstrates a number of concepts.

- All application initialization takes place in BeeAppInit().
- Events for the application task come into the function BeeAppTask().
- Incoming ZigBee messages come into the function BeeAppDataIndication().
- Keyboard events come into the function BeeAppHandleKeys().

Examine the BeeAppDataIndication() function. Notice the newly added code both starts a timer and sends an event to the application task. The timer is used to turn off the LED that was turned on in the data indication handler.

Notice also the data indication handler didn't need to worry about the application profile or endpoints, because these are taken care of when the application registered the endpoint in BeeAppInit(). The lower layers will filter any incoming data not for that registered endpoint or on the wrong application profile ID. The application needs only to concern itself with clusters.

The cluster ID itself was retrieved from the endpoint's simple descriptor, as found in BeeAppDataIndication().

Examine the function BeeAppHandleKeys(). Note how SW1 starts the network with a single call to ZDO. Note how SW3 finds the other node using the ASL_MatchDescriptor_req() function. The results of that function come back to the callback registered in BeeAppInit(), in the line that reads:

    Zdp_AppRegisterCallBack(BeeAppZdpCallBack);

The function BeeAppZdpCallBack() stores the results of a successful match descriptor so that LED3 can be lit and the application can now know which node to send its commands to. Note that match descriptor will return ALL nodes that match, so it's only useful if the application knows there will be no or only a few nodes in the network with the same profile ID and cluster list as described by their endpoint's simple descriptor.

This same set of nodes will work in any sized ZigBee network, filled with many devices on many application profiles, and they can still communicate with each other.

Chapter 3
Designing A Custom Profile

This chapter describes some issues to consider when designing a new custom-profile application, including selecting a profile ID, clusters, attributes and endpoints. It also describes ZigBee 2006 security options and includes a discussion on channel and bandwidth use.

3.1   Application Profiles

Application profiles are a collection of related services designed to be interoperable. In the case of public application profiles, the ZigBee Alliance specifies these services to allow for interoperability between OEM vendors' products. An Application Profile ID is a 16-bit number assigned by the ZigBee Alliance. Private Application Profile IDs must also be obtained from the ZigBee Alliance. The Freescale Private Profile ID 0xc021 is used for the example code provided by Freescale.

ZigBee Alliance public profiles include

- Home Automation
- Smart Energy
- Health Care
- Commercial Building Automation
- Industrial Plant Monitoring

Most profiles require the use of the ZigBee Cluster Library, a common library of services shared among profiles.

Every ZigBee node contains one or more application profiles. As an OEM, the only decision to make is whether to use a public ZigBee Alliance profile or a private profile. Private profiles have the advantage of being simple to implement and flexible for the project. Public profiles have the advantage of being interoperable among vendors, but at the expense of extra code size and complexity.

Public profiles are given an ID by the ZigBee alliance, for example 0x0104 for Home Automation. Contact the ZigBee Alliance (http://www.zigbee.org) for a private profile ID.

One public profile is available in every ZigBee node: the ZigBee Device Profile (profile ID 0x0000). This profile provides common services to all ZigBee nodes.

As an OEM, users should choose a public profile ID that matches their particular application or request a private profile from the ZigBee Alliance.

3.2   Endpoints, Clusters and Attributes

Think of endpoints as a virtual wire. Endpoints serve three purposes in ZigBee:

- To provide a location within a node to connect two services. For example, an endpoint on an on/off switch connects to an endpoint on an on/off light
- To provide addressing within the node. Separate endpoints could control separate lights within a single node, for example
- To support multiple application profiles within a node (every endpoint supports exactly one profile)

Application endpoints are numbered 1-240. Endpoint 0 is for the ZigBee Device Profile (a set of common ZigBee services available in all nodes). Endpoint 255 is the broadcast endpoint; a message to endpoint 255 is delivered to all endpoints in the receiving node with the same application profile as the sender.

Clusters are the services on that endpoint. For example, a single home automation endpoint which supports an on/off light, supports an on/off cluster for turning the light on and off. In addition, that endpoint will contain a groups cluster for grouping a set of lights together and a scenes cluster so that the light can be set up to go to various scenes (movie viewing or gone on vacation, for example).

In the ZigBee Cluster Library, a cluster can support zero or more attributes. Where clusters are commands, attributes define the state of the application on that endpoint. For example, an on/off light has an attribute that describes whether the light is on or off.

3.3   Customizing A Public Profile

Public profiles using the ZigBee Cluster Library can be augmented with OEM specific extensions. For example, an HaOnOffLight, which normally can only turn a light on and off, could be augmented with a feature of adjusting the hue of the light in addition to turning it on and off. Check with the latest ZigBee Alliance profile specifications, however, because the feature may exist in the profile already.

To augment a cluster, a payload must be created starting with the zclMfgFrame_t type. This frame contains a manufacturer specific ID which must be obtained from the ZigBee Alliance.

3.4   Stack Profiles

ZigBee networking, in addition to supporting multiple application profiles within a network, must reside on a single stack profile.The ZigBee feature set uses the Home Controls Stack Profile. Its profile ID is 0x01. Although the stack profile is called Home Controls, it supports both Home and Commercial applications and is used by Industrial Plant Monitoring.

This stack profile has the following characteristics:

- Supports up to 31,101 nodes in the network, theoretically.
- Uses both tree and mesh routing. Tree routing allows reduced routing tables.
- Supports full AES 128-bit encryption with a network-wide key. The network key may be predetermined, or it may be sent over the air to nodes as they join. Note: it is sent over the air in the clear, so there is a small time window when the network is open for joining that would allow a rogue node to obtain the network key.

- Supports a maximum of 10 hops (maxdepth 5 * 2) across the network.
- Supports 20 children per router, 14 of which may be ZigBee End Devices.
- End Devices can sleep for up to 1 hour.
- End Devices will be compatible with upcoming ZigBee specifications.

It is possible to make a custom stack profile with a different number of maximum hops across the network or number of children, but it is not recommended, except under very rare circumstances.

See also gNwkMaximumDepth_c, gNwkMaximumChildren_c and gNwkMaximumRouters_c in `BeeStackConfiguration.h`.

Chapter 4
Selecting Platform Components

This chapter describes how to select the appropriate hardware-related platform components, including the use of non-volatile memory, LEDs, the keyboard, serial port, and general hardware.

Platform components are optional for any given ZigBee application. Platform components are enabled in BeeKit using the Platform Property List in the Solution Explorer pane. They include:

- Display (LCD, not available on all standard Freescale platforms)
- Keyboard
- LED
- NVM
- Power
- Timer
- UART

4.1 The Display Component

The display component can be used to support an LCD controller. There is a LCD display built into the some Freescale boards. Enable this in BeeKit to support an LCD. The file `Display.c` (found in the PLM folder) will also need to be modified, unless using the same controller as found on Freescale development boards.

4.2 The Keyboard Component

The keyboard component supports 8 key inputs using only 4 physical keys. Each keypress can be detected as either a short or long press. This component can be disabled if not needed. The keyboard interface is also a way to wake low power units on interrupt. For example, a keyboard pin could be used to detect that a window or door was opened. This could wake a low power node which would then inform a security monitor of a breach. From the application's standpoint, it received a key-press and can act accordingly.

4.3 The LED Component

The LED interface allows for LEDs to be independently controlled. The LED interface simply sets a GPIO pin to high or low. This interface can also be used to control any sort of device, such as starting a motor, or communicating data on the GPIO pins. See `Led.h` for a definition of the pins used.

The function LED_SetLed() is used for almost all of the LED interaction. The states a particular LED can be set to are:

- gLedFlashing_c — flash at a fixed rate

- gLedBlip_c — just like flashing, but blinks only once
- gLedOn_c — on solid
- gLedOff_c — off solid
- gLedToggle_c — toggle state

The LEDs can be combined using bitwise OR. This save code space. For example:

```
LED_SetLed(LED2 | LED3, gLedOn_c);
```

4.4 The NVM Component

Non-volatile Memory (NVM) is used to preserve the state of the ZigBee network across reboots and power outages. For example, it is critical for a light switch to remember which lights it is controlling after a power outage is over and power returns.

The application can also use NVM to store critical application data that should be preserved across resets. The following join modes are available from ZDO:

- gStartWithOutNvm_c
- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c
- gStartSilentRejoinWithOutNvm_c
- gStartSilentNwkRejoin_c

To start the network using NVM (or not), use the `ZDO_Start()` function. For example:

```
ZDO_Start(gStartSilentRejoinWithNvm_c);
```

NVM uses flash pages of 512 bytes in size to store the non-volatile data. Due to the limitations of flash, the entire page is erased before rewriting to it. The NVM engine keeps a spare page, so when new data is written, it is written to the spare before the old page is erased.

One page is devoted entirely to network structures, such as the neighbor and routing tables. A second page is partially available for the application to use, and is partly in use by BeeStack.

See `NV_Data.c`. Each page is called a "data set", which contains a collection of pointers and sizes of the items to store in that page when the data is "dirty" or has been changed. Data is marked "dirty" when one of the following functions is used:

```
void NvSaveOnIdle(NvDataSetID_t dataSetID);
void NvSaveOnInterval(NvDataSetID_t dataSetID);
void NvSaveOnCount(NvDataSetID_t dataSetID);
```

Notice that only the data set ID is used. The entire data set is saved, even if one byte in the data set has changed.

Make sure not to save data to NVM too often. For a 20 year product life, the system should not save to NVM more than once every 1.8 hours (This is calculated assuming 100,000 erase cycles). Typically,

applications will save often when first forming or joining the network and during the commissioning process. From then on, saving should occur rarely.

See the *Freescale Platform Reference Manual* for more information on non-volatile memory.

4.5   The Low-Power Component

BeeStack allows for low power devices on ZigBee End Devices (ZEDs) only. ZigBee Routers (ZR) and the ZigBee Coordinator (ZC) do not have this capability.

To enable low power, disable the `MAC Capability: Rx On When Idle` property and enable the `ZDO: Low Power Mode Enabled` property in BeeKit in a ZigBee End Device. The low-power component is shared by other Freescale networking solutions, such as the IEEE 802.15.4 MAC and the SMAC. The power library options can be found in `PWR_Configuration.h`.

Keep in mind also that the power library will not enter deep sleep unless all of the BeeStack timers are off. Timers may still be active if, for example, BeeStack is still attempting to deliver an acknowledged message or broadcast. Also, the power library will not enter sleep at all unless every task is idle. See the *Freescale Platform Reference Manual* for more information on power management.

4.6   The Timer Component

BeeStack timers are used to gain control within a task after a certain period of time has elapsed. Timers can be one-time events (single-shot) or repeating (interval), and can range in duration from 4 to 262,143 milliseconds (about 4 minutes). Interval timers will continue to repeat until stopped.

Use timers to blink LEDs, pace ZigBee data requests, or for application timing purposes.

BeeStack timers are implemented using a single hardware timer (TPM1 on the HCS08), leaving any other hardware timer resources available to the application.

In BeeStack, the number of timers available to the application is defined at compile-time through a property in BeeKit called gTmrApplicationTimers_c, in `TMR_Interface.h`. The default number of application timers is 4. Each timer requires 7 bytes of RAM.

Like all BeeStack platform (PLM) components, timers retrieve control via a callback. The callback is a function with the following prototype (it can have any name the application chooses)

```
void BeeAppTimerCallBack
  (
  tmrTimerID_t timerId  /* IN: */
  );
```

Timers are initiated through the use of one of the following functions

```
void TMR_StartSingleShotTimer
  (
  tmrTimerID_t timerID,
  tmrTimeInMilliseconds_t timeInMilliseconds,
  void (*pfTimerCallBack)(tmrTimerID_t)
  );

void TMR_StartIntervalTimer
  (
```

```
tmrTimerID_t timerID,
tmrTimeInMilliseconds_t timeInMilliseconds,
void (*pfTimerCallBack)(tmrTimerID_t)
);

void TMR_StartMinuteTimer
  (
tmrTimerID_t timerId,
tmrTimeInMilliseconds_t timeInMinutes,
void (*pfTmrCallBack)(tmrTimerID_t)
);
```

Timers (both interval and single-shot) are stopped through the use of

```
void TMR_StopTimer
  (
tmrTimerID_t timerID
);
```

4.7   The UART Component

One platform component that is often useful in ZigBee applications is the UART component. This allows one or both of the HCS08 SCI (serial communications interface) ports to be used to connect to a PC or another host processor.

All standard Freescale development boards provide a hardware connection to one or more UARTs. Some boards offer a USB connection, and others offer RS-232. The UART software component does not need to distinguish between these two port types because the differences are handled by external hardware.

The UART can be used to communicate all networking traffic. The ZigBee Test Client (ZTC) interface is very useful for this function, or a custom serial protocol can be developed. See the *Freescale BeeStack BlackBox ZigBee Test Client (ZTC) Reference Manual* for more information on the ZigBee Test Client.

Chapter 5
Managing BeeStack Resources

This chapter describes using the non-hardware related platform components appropriately, including the use of messages, timers, data queues, and the task scheduler. It also describes how to determine how much RAM and Flash is available to the application and what to do if an application exceeds memory size. It also describes managing ZigBee channels and bandwidth.

5.1 BeeStack Start-up Sequence

BeeStack begins control (at least from a C language point of view) in a module called `BeeAppInit.c`, at the function main(). From there, main() initializes the radio, MAC, platform components and ZigBee networking stack. After all of BeeStack is initialized, the application is initialized when BeeAppInit() is called in the application.

The typical application BeeAppInit() (which is found in `BeeApp.c`, not `BeeAppInit.c`) registers endpoints with the stack (to be able to receive ZigBee networking communications), registers with the keyboard to receive key presses, and initializes the Application Support Layer (ASL) and possibly the ZigBee Cluster Library (ZCL).

Most applications will not need to modify the start-up sequence, however it is provided in full source code, so that it can be modified as appropriate for any given application.

5.2 Managing Tasks

BeeStack relies on a platform component called the task scheduler to accomplish ZigBee networking. This scheduler is a non-pre-emptive priority based scheduler. It can have up to 255 tasks.

BeeStack contains a task for each of the ZigBee functional modules (NWK, APS, AF, ZDP, ZDO, ZCL), and for some platform components (such as UART).

The application is contained in one task by default but can be split up into multiple tasks for a particularly complex application.

NOTE

MCU interrupts operate independently of tasks, and may often pass control to a task through the use of the TS_SendEvent() function.

The maximum number of tasks in a BeeStack system is allocated at compile-time using the `TS: Number of tasks` BeeKit property, which defaults to 14 tasks. Each task requires 6 bytes of RAM. Depending on BeeKit property settings, BeeStack uses up to 11 tasks.

Each task is comprised of at least two (2) functions:

- Task initialization – this function is run once on start-up to initialize the task

- Task event handler – this function is run every time there is an event for the task

The task prototypes are as follows:

```
void TaskInit( void  );
void TaskEventHandler ( event_t events  );
```

Each task may define up to 16 distinct events, each of which is an event of the event_t type. Events are unique per task (that is, the event mask for one task is distinct from the event masks of all other tasks). Each event is a single bit in the events bit mask.

In the application task, defined by the functions BeeAppInit() and BeeAppTask(), the events are used as in Table 5-1.

Table 5-1. Application Events

Event	Bit Mask	Description
gAppEvtDataConfirm_c	0x8000	Sent by APS when a data confirm is received on an endpoint
gAppEvtDataIndication_c	0x4000	Sent by APS when a data indication is received on an endpoint
gAppEvtSyncReq_c	0x2000	Sent by ZDO when it's time to poll for data from a parent (ZigBee End Devices only)
gAppEvtAddGroup_c	0x1000	Sent by the ZigBee Cluster Library (ZCL) when it's time to add a group.
gAppEvtStoreScene_c	0x0800	Sent by ZCL when it's time to store a scene
Available	variable	Bits 0-10 are available for the application. If the ZigBee Cluster Library is not used, bits 11 and 12 are available also. If the device is a ZigBee Router or Coordinator, bit 13 is also available.

The following compile-time tasks are defined in BeeStack:

- idle task – gains control when there is nothing else to do
- MAC task – services MAC layer primitives
- NWK task – services network layer primitives
- ZDO task – services ZigBee Device Object primitives
- APS task – services Application Support Sub-layer primitives
- AF task – services application framework primitives
- PLM task – services platform management primitives
- Application Task – services the application

BeeStack tasks are non-pre-emptive. Once a task gains control, it has full control until the task completes (returns from the task event handler function). Tasks should complete quickly (less than 2ms) to avoid starving other tasks of processing time. If the task takes too long to process it will prevent the lower layers of the Stack to process incoming packets.A task should under no circumstance take longer than 10 milliseconds.

BeeStack tasks are priority based, with the idle task being the lowest priority. The highest priority task is the MAC task, to service network data traffic. Applications may use task priority 0x40 – 0xbf. The default priority for the BeeAppTask() is 0x80, as defined by gTsAppTaskPriority_c.

The decision of whether to make more than one application task is up to the application designer. Generally, a single application task is sufficient, but if an application is particularly complex, or if it contains an independent hardware component, adding a task can simplify the coding.

Call TS_TaskCreate() to add a task, usually in BeeAppInit(), and make sure to call an initialization function for the task.

5.3   Managing Timers

Use timers whenever some event must be timed. When a timer expires, it calls a callback function of the application's choice, as given to the start timer functions, TMR_StartSingleShotTimer() and TMR_StartIntervalTimer(). Typically, the callback function should set an event, but it could do any work required. The callback is in the timer task context.

The stop timer function, TMR_StopTimer() is safe to call at any time, even if a timer is already stopped.

Timers must be allocated before they can be started or stopped. Use the TMR_AllocateTimer() function for this purpose.

The total number of timers for the application is defined by gTmrApplicationTimers_c.

When one or more timer is active (currently counting down), low-power mode will not enter deep sleep, but will use light sleep only. Make sure all application timers are stopped to enter deep sleep on ZigBee End Devices (Routers and Coordinators do not sleep).

5.4   Managing Message Buffers

BeeStack uses message buffers to transmit data over the ZigBee network (data requests) and to receive data from other nodes (data indications). This section describes some tips and techniques to manage these buffers so the network runs smoothly and to reduce or eliminate packet loss due to insufficient buffers.

The number of BeeStack message buffers is determined at compile-time through a set of BeeStack properties, one called gTotalBigMsgs_d, and another called gTotalSmallMsgs_d. The defaults for these are 5 each.

Big messages are 161 bytes in size and are large enough to hold the largest ZigBee packet, including all over-the-air frame headers plus some additional information for housekeeping. Big messages are used for both data transmit and receive. If the network is expected to be very busy, transmitting or routing many packets over a short period of time, the number of big buffers should be increased as much as RAM allows. The value 7 is reasonable for a modestly busy network.

Small messages are used for temporary data within BeeStack and the number of these buffers should not be changed.

When the application initiates a ZigBee data request (to transmit data) using AF_DataRequest(), it allocates a big buffer. AF_DataRequest() will fail if no big buffers are available. ZDP commands also allocate big buffers to do their work.

Freeing buffers once they are no longer needed is of course critical to system operation. BeeStack follows very specific rules for freeing buffers, as follows:

When the higher layer allocates a message buffer (e.g., a data request) and passes it to a lower layer, the lower layer is responsible for freeing the message buffer.

When a lower layer allocates a message buffer (e.g. a data indication) and passes it to a higher layer, the higher layer is responsible for freeing the message buffer.

`AF_FreeDataIndicationMsg()` must be used to free Data indication messages and `MSG_Free()` should be used to free any other message.

`AF_MsgAlloc()` must be used to allocate a message buffer for a Data Request and the `AF_MsgAllocFragment()` must be used for allocation of message buffers for Data Request that uses fragmentation.. `MSG_Alloc()` should be used in any other case.

Lower layers may retain the message buffer for up to a number of seconds, depending on the operation. For example, if an application initiates an AF_DataRequest() to transmit data and has the gApsTxOptionAckTx_c flag set in the txOptions field, the big message buffer will be retained until the acknowledgement (ACK) is received from the remote node or a time-out of 4.5 seconds occurs, whichever comes first. If required, the packet will be resent by the APS layer up to 3 times to ensure reliable transmission of the packet.

BeeStack issues a confirm on each data request, received through the BeeAppDataConfirm() callback. This confirm can be used to regulate the pace of data requests, and thus regulate the use of big buffers.

Follow these rules to effectively manage message buffers:

- Do not issue broadcasts more often than once every 2 seconds. Be aware that other nodes may issues broadcasts. Broadcasts must retain big buffers for up to 2 seconds
- Do not issue another unicast until the confirm has been received
- Check the confirm value. If the confirm is gZbBusy_c, the request was not sent due to a busy system (many routing packets). Try again after waiting 100 milliseconds or so

Two message buffers are always reserved by the MAC for data indications.

5.5 Managing Memory

In BeeStack, memory is generally allocated statically at compile-time. There is no concept in BeeStack of a heap, and there is no malloc(). Do not use the message buffers for application allocation as they are needed by BeeStack to operate ZigBee networking.

5.6 Managing The C Stack

When a task gains control, it is at the top of the stack. When a callback (such as a timer or keyboard callback) is made, it is not at the top of the stack, because callbacks are not in the context of some stack.

The number of bytes used in the stack is dependant on the application. To detect how many bytes are used, look at offset 0x0100 in memory (the bottom of the stack) and count the number of bytes that are equal to 0x55 (the stack initialization value).

Make sure the stack doesn't overflow in an application during its testing phase. A rule of thumb is to make sure there are at least 40 bytes of stack unused after worst case testing.

If the stack is largely unused in a particular application, some RAM can be saved by adjusting the size of the stack. The size of the stack can be adjusted in the linker file `*.prm`.

5.7 Managing RAM and FLASH

RAM and FLASH in embedded systems can be a scarce resource. The Freescale development platforms have the following RAM and Flash characteristics:

MC13213 (HCS08GT60)   4KB of RAM and approximately 60KB of FLASH

HCSQE128   8KB of RAM and approximately 128KB of FLASH

MC13233   5KB of RAM and approximately 82KB of FLASH

MC1322x   96KB of RAM with FLASH mirroring and an 80KB ROM containing code for IEEE 802.15.4 MAC (MC13224/MC13225) and BeeStack (MC13226).

Depending on the BeeStack options chosen, any application could exceed either RAM or FLASH.

Tips to help reduce FLASH:

- Use a ZigBee End Device rather than a ZigBee Router. ZigBee End Devices can be about 10KB smaller in FLASH. ZigBee End Devices can also be set to RxOnIdle. This means they do not need to poll and can respond immediately.
- Only include those ZDP functions actually used by the application. These can be selected in BeeKit.
- Do not use ZTC or the UART driver unless the application needs them.
- Do not use a secure network unless required. ZigBee security is approximately 8 KB.
- Look in the `*.map` file generated by the linker for the largest functions and modules. Try to eliminate functions that are not required by the application.
- Remove debugging code (if any).
- If possible, reduce NVM storage needs or disable the NVM module.

Tips to help reduce RAM:

- Make tables as small as possible. For example, the routing table, neighbor table or group table can be reduced.
- Reduce the C stack size if the application does not use the full stack.

5.8 Managing ZigBee Channels

ZigBee communicates on the IEEE® 802.15.4 MAC and PHY standard. In the 2.4 GHz band, this standard allows for 16 channels, numbered 11-26 (channels 0-11 are used for sub 1GHz bands). The 2.4 GHz channels are physically separated by 5 MHz, so they cannot hear each other. ZigBee operates on one channel at a time.

Some application profiles require a specific channel selection. For example, Home Automation requires a node to be able to operate in a network on any channel.

Private profiles can restrict the application to a particular set of channels. Channels 11 and 26 (the edge channels) are often good choices for a private profile.

Channels 15, 20, 25 and 26 tend to be clear of WiFi channels. In practice, ZigBee tends to co-exist with WiFi and other 2.4 GHz technologies. ZigBee uses CSMA-CA, so it takes advantage of silence on the channels to communicate.

5.9   Managing ZigBee Bandwidth

On any given channel, in a given location, only one radio may be transmitting at the same time. That means that a dense network with lots of traffic could end up interfering with itself.

Keep the following items in mind when planning or deploying a network:

- ZigBee End Devices (ZEDs) that sleep, poll their parent to receive messages. Keep this polling rate to 5 seconds or longer if possible. Most ZEDs can wait for responses
- Use a "push" rather than a "pull" method of communicating. That is, have a sensor report a change, rather than querying the sensor constantly for change
- Keep the broadcast radius small (1-3) for broadcasts that are expected to be serviced by nearby nodes. Do not use up bandwidth on other parts of the network
- If gateways are used, do not create a bottleneck by sending all traffic to a single data aggregator or gateway. Instead, aggregate the data in intermediate nodes, which in turn send combined packets back to the gateway to reduce network traffic
- If the network is large (200+ nodes) consider multiple gateways
- Do not use ACK on data requests unless the application will use the data confirm results
- Do not use security unless the application needs it. Security makes larger packets
- ZigBee is a low-speed network. Use it as such
- Always keep bandwidth in mind. Bandwidth is finite

Chapter 6
Debugging BeeStack Applications

This section describes how to debug a networking application, including use of the BDM, LEDs, ZigBee Test Client, and Sensor Network Analyzers.

6.1 The P&E MultiLink BDM

One of the most powerful tools for debugging a BeeStack application is the use of the P&E MultiLink Background Debug Mode (BDM) pod. This device plugs into a 6-pin connector on each development board and not only allows code download into the on-board flash of the HCS08 MCU, but it also allows stepping through the source code.

When using the CodeWarrior TrueTime debugger, the following tips can be helpful

- Only 2 breakpoints at any given time are allowed
- Only set or clear breakpoints when the debugger is stopped
- When single stepping through the code, if the debugger ends up in an interrupt handler instead of the next C source line, use single step (F10) again (in the interrupt handler) and then step out (Shift-F11). It may take several iterations if the interrupts are particularly active
- If the BDM will not download the code, disconnect the BDM, reset the board and try again
- Multiple BDMs (and debuggers) can be used simultaneously
- Only keep at most one debugger window open per BDM

For details on the CodeWarrior TrueTime debugger, see the CodeWarrior documentation.

6.2 LEDs and the Display

The BeeStack LED component contains a function, LED_SetHex(), which allows a hex nibble (the lower 4 bits of a byte) to be displayed on the 4 LEDs on the Freescale reference boards. This function can be used to show the latest state of the application.

Another technique is to toggle the LED every time the application task gets control. Use LED_SetLED() with gLedToggle_c as the state parameter.

LEDs can also be useful to see when the board enters low power (see the idle task in BeeAppInit.c) or when the board is communicating over ZigBee on BeeAppDataIndication(), for example.

The LCD display on those boards that support them, such as the NCB and Axiom, is also a very useful debugging tool. LCD_WriteString() and LCD_WriteStringValue() can be used to great effect, indicating where the problem lies.

6.3   Network Protocol Analyzers

A network protocol analyzer is a tool that captures over-the-air data for later examination to aid in debugging network activity. Freescale offers "sniffer" hardware that can passively monitor an 802.15.4 channel for activity, and reports each packet through a USB port to the desktop PC. Communication problems that are extremely difficult to identify in the code are frequently very easy to understand from the over-the-air behavior.

Third parties offer protocol analyzers. Figure 6-1 shows one of the many network analyzers available. Notice that the network is shown as a graphic and with packet decode. This includes time stamps on the packets.
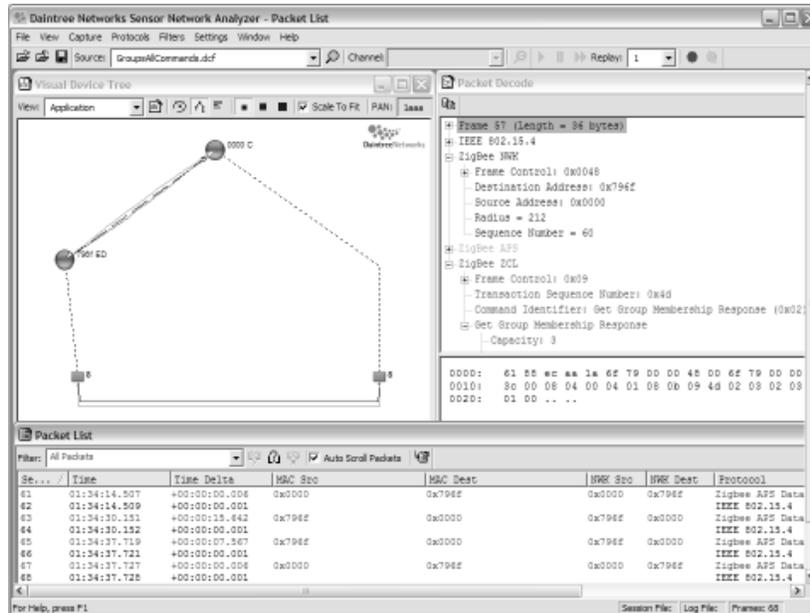

Figure 6-1. Sensor Network Analyzer

6.4   ZigBee Test Client

Another method for debugging a network is the Freescale Test Tool and ZigBee Test Client combination. Test Tool, a desktop PC tool from Freescale, uses a serial (USB) port to communicate to one or more boards. An XML file describes the commands Test Tool uses, which in turn communicates to Freescale ZigBee development boards. A small network of 2-10 nodes can easily be set up and controlled by Test Tool for manual testing of application commands.

ZTC and Test Tool can be extended to support any commands over the serial link allowing a very flexible tool for debugging. The standard ZTC configuration supports all BeeStack ZigBee commands.

ZTC also allows for automated testing. Freescale uses this technique to test BeeStack itself, with a large test suite covering the BeeStack API and ZigBee commands.

For a complete list of ZTC commands, see the Freescale *BeeStack BlackBox ZigBee Test Client Reference Manual*. For more information about Test Tool, see the Freescale *Test Tool User's Guide*.

Chapter 7
Creating a Custom BeeStack BlackBox Application

This chapter describes how to create and extend a Custom BeeStack BlackBox application.

7.1   Generating a BeeStack BlackBox Project with BeeKit

This section describes how to generate a BeeStack Development project in Beekit.

1. Start BeeKit. The BeeKit main window appears as shown in Figure 1-1.
2. If another Codebase (MAC, SMAC or SynkroRF) is selected, perform the following:
    a) Select File -> Select Codebase… or click the "Select Other Codebase..." link.
    b) Select the BeeStack Codebase version to use from the codebase list.
    c) Click Set Active.
3. To create a new project to configure a new device, select File -> New Project...
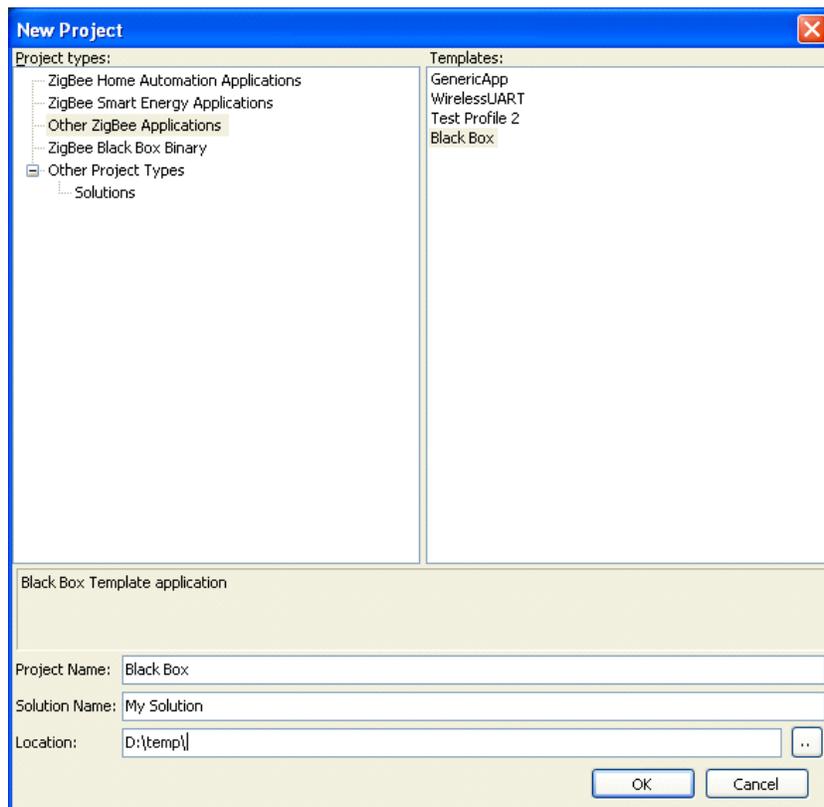
The New Project window appears as shown in Figure 7-1.

Figure 7-1. New Project Window

4. Select the "Other ZigBee Applications" project type from the left side of the window.

5. As shown in Figure 7-1, select the "BlackBox" template and fill in the Project name, Solution name and location fields.

6. Click the OK button to create the project.

7. After the New Project window closes, the BeeKit Project Wizard Welcome window appears as shown in Figure 7-2.
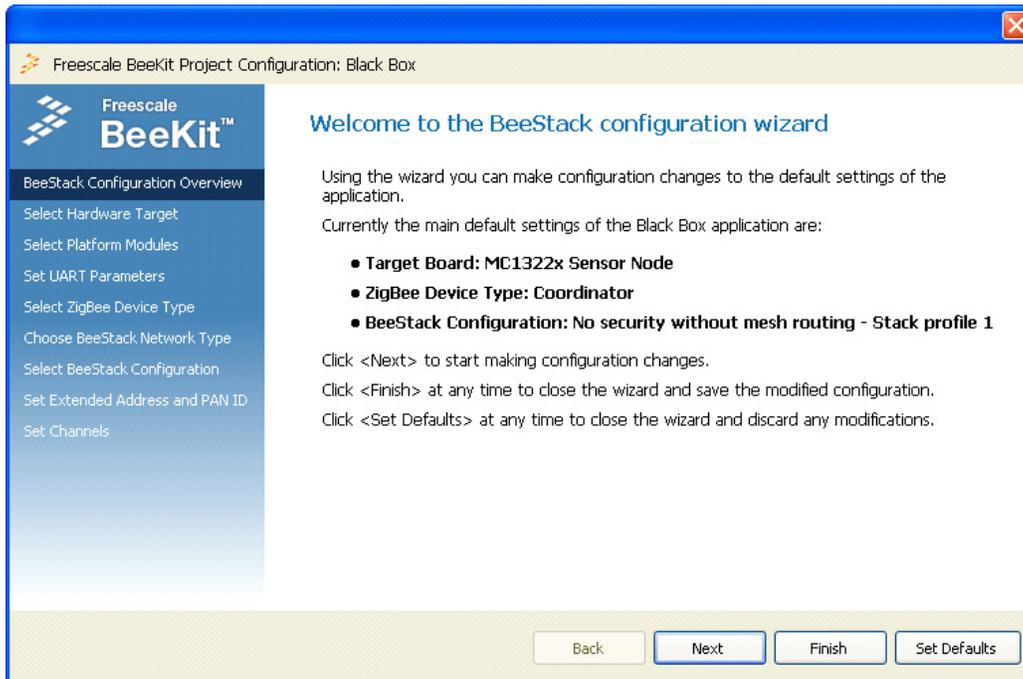

Figure 7-2. Project Configuration Wizard Window

Review the current project settings. The default settings used in this example are shown in Figure 7-2.

There are four ways to proceed from the project configuration wizard window.

a) Accept all the current settings by clicking on the Finish button. (No more configuration.)

b) If users are sure of the settings that need changing, go directly to them and make the appropriate selections from the menu choices on the left side of the project configuration wizard window.

c) If users are not sure of the settings that need changing, click on the Next button. This is the choice described in Section 2.1, "Creating a Custom Application In BeeKit".

d) If users need to discard the modifications already made to the default configuration and close the wizard, click the Set Defaults button.

7.1.1 ZigBee Black Box Device Type Configuration

As shown in Figure 7-3, users can now choose one of the following ZigBee BlackBox device types from the Freescale BeeKit Project Configuration Black Box window:

- Coordinator
- Router
- End device
- Combo device.

NOTE

Selecting the proper device type has a significant impact on the size of the memory available for the application. Choosing the Combo device type allows the target to function as a Coordinator, a Router or an End device, but the Combo device type uses the most memory.

For example, if the target is only intended to function as an End device, then select the End device type instead of a Combo device type. This leaves more memory space for adding other useful ZigBee features such as fragmentation or for defining a larger memory pool for buffer allocation.
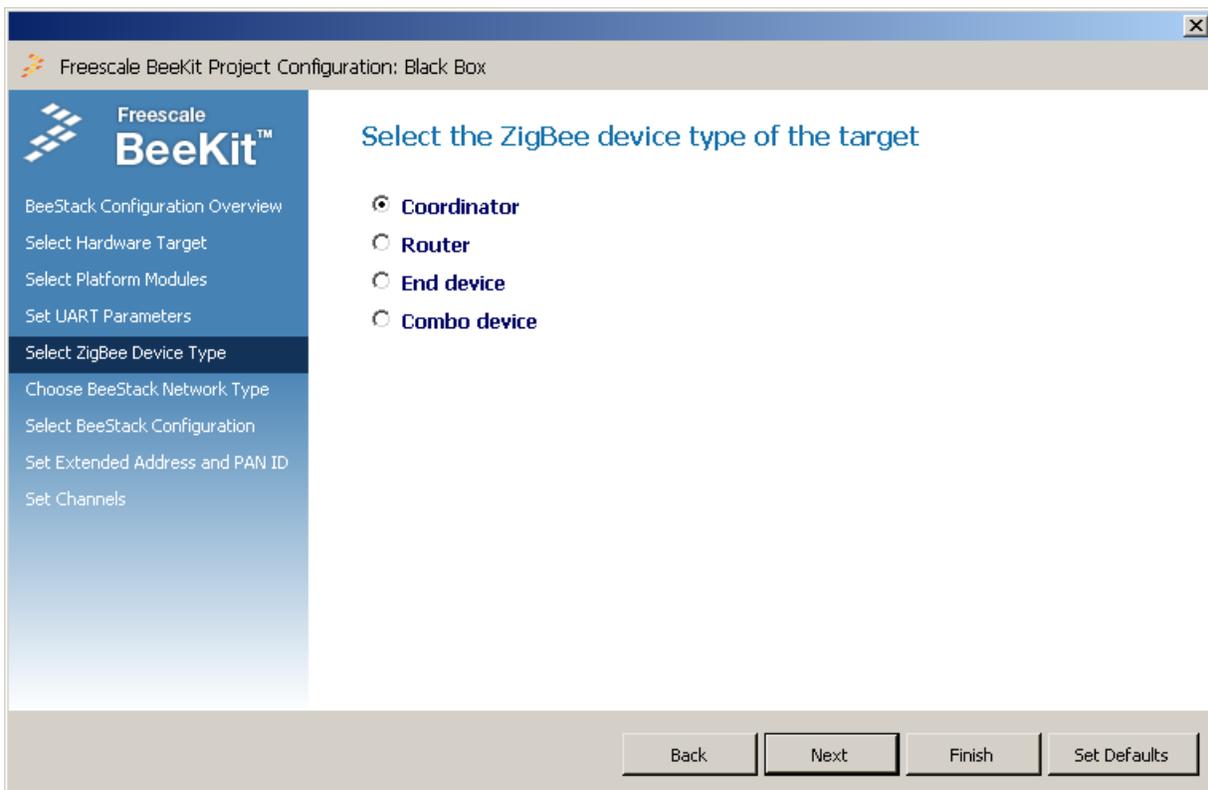

Figure 7-3. Device type selection window.

8. After configuring the devices as shown in Figure 7-4 and after completing the tasks in the project configuration wizard window, configure and match the project with a feature set.
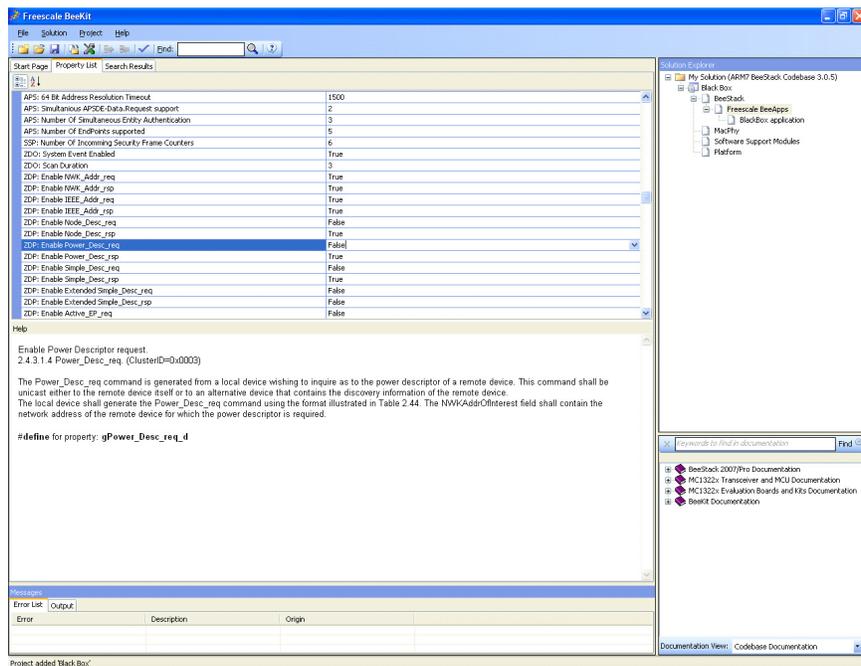

Figure 7-4. Project Configuration

NOTE

The BeeStack BlackBox ZTC client, UART or IIC module must be enabled to be able to communicate with a host CPU.

9. As shown in Figure 7-5, the project can now be exported and opened in the Integrated Development Environment (IDE).
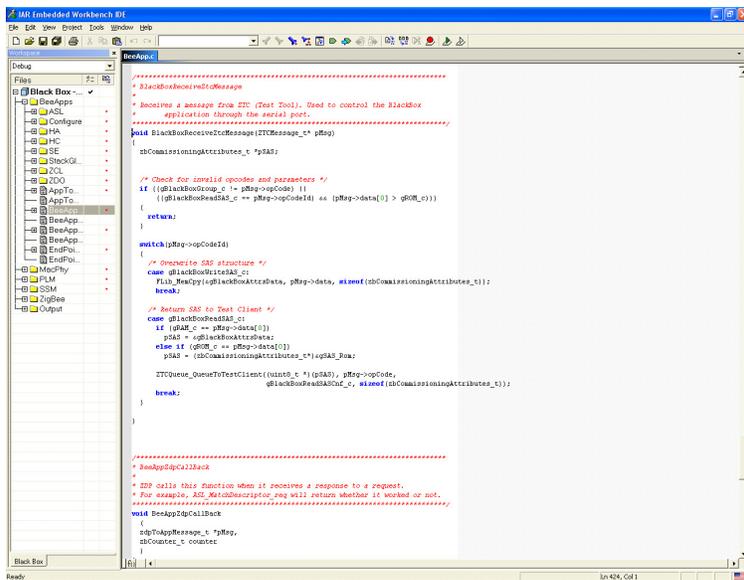

Figure 7-5. Integrated Development Environment

BeeStack™ Application Development Guide for ZigBee 2007, Rev. 1.3

To learn more about BeeKit and project configuration see the *Freescale ZigBee Applications User's Guide* (ZAUGZB2007).

10. Modify and debug the project using the IDE.

To learn more about how to add commands to the BlackBox application, see the Freescale *BeeStack BlackBox ZTC Reference Manual* (BSBBZTCRM).

NOTE

The BlackBoxReceiveZtcMessage() function in `BeeApp.c` contains a framework for adding new commands to simplify the process.

11. After downloading the code with the debugger, the BeeStack BlackBox can be used as described in the *Freescale ZigBee Applications User's Guide* (ZAUGZB2007).

Chapter 8
Creating a ZigBee Host Application

This chapter describes how to create and extend a ZigBee Host application.

8.1   Hardware Setup

A ZigBee Host application requires that the target hardware be connected via the serial port to the hardware on which the ZigBee BlackBox runs. This section describes how to setup and connect boards as Hosts and BlackBoxes to each other. For more information on the boards, see the appropriate Freescale development kit documentation delivered with BeeKit.

8.1.1   Setting Up a MC1320x QE128 Board as a Host

The MC1320x QE128 board uses the UART1 interface to communicate with the ZigBee BlackBox. The UART2 interface communicates with an external client such as the Freescale Test Tool. The MC1320x QE128 Host board needs to use its own clock source.

Perform the following tasks on the MC1320x QE128 Host board:

- Remove jumpers J13 (TX) and J15 (RX)
- Set jumper J5 to the 32 KHz position
- Connect the TX pin to the appropriate UART RX pin of the ZigBee MC1320x QE128 BlackBox board
- Connect the RX pin to the appropriate UART TX pin of the ZigBee MC1320x QE128 BlackBox board
- Connect the GPIO connector GND pin to the appropriate system ground pin of the ZigBee MC1320x QE128 BlackBox board

8.1.2   Setting Up a MC1322x Board as a Host

The MC1322x NCB and SRB boards use the UART2 interface to communicate with a ZigBee MC1322x BlackBox board. The UART1 interface communicates with an external client, such as the Freescale Test Tool. Connect the UART2 interface pins as follows:

- Connect GPIO connector UART2_TX pin (17) to the appropriate UART RX pin of the ZigBee MC1322x BlackBox board
- Connect GPIO connector UART2_RX pin (18) to the appropriate UART TX pin of the ZigBee MC1322x BlackBox board
- Connect GPIO connector GND (4) to the appropriate system ground pin on the ZigBee MC1322x BlackBox board

8.1.3 Setting Up a MC1320x QE128 Board as a BlackBox

The MC1320x QE128 board uses the UART1 interface to communicate with a ZigBee MC1320x QE128 Host board.

Perform the following tasks on the MC1320x QE128 BlackBox board:

- Remove jumpers J13 (TX) and J15 (RX)
- Set jumper J5 to the CLKO position
- Connect the TX pin to the appropriate UART RX pin of the ZigBee MC1320x QE128 Host board
- Connect the RX pin to the appropriate UART TX pin of the ZigBee MC1320x QE128 Host board
- Connect the GPIO connector GND pin to the appropriate system ground pin of the ZigBee MC1320x QE128 Host board

8.1.4 Setting Up a MC1322x Board as a BlackBox

The MC1322x NCB and SRB boards use the UART2 interface to communicate with a ZigBee MC1322x Host board. Connect the UART2 interface pins as follows:

- Connect GPIO connector UART2_TX pin (17) to the appropriate UART RX pin of the ZigBee MC1322x Host board.
- Connect GPIO connector UART2_RX pin (18) to the appropriate UART TX pin of the ZigBee MC1322x Host board.
- Connect GPIO connector GND (4) to the appropriate system ground pin on the ZigBee MC1322x Host board.

8.2 Software Setup

This section describes the steps needed to generate a BlackBox Host solution after the two boards are connected.

8.2.1 Software Setup on the BlackBox Board

Flash the appropriate ZigBee BlackBox image to the BlackBox board. The following image files are available in the HCS08 Codebases and ARM7 Codebases (for MC13224 and MC13226):

- `HCS08BeeStackBlackBox_SP1_UART1.s19`
- `HCS08BeeStackBlackBox_SP2_UART1.s19`
- `ARM7BeeStackBlackBox_SP1_UART2_MC13224.bin`
- `ARM7BeeStackBlackBox_SP2_UART2_MC13224.bin`
- `ARM7BeeStackBlackBox_SP1_UART2_MC13226.bin`
- `ARM7BeeStackBlackBox_SP2_UART2_MC13226.bin`

8.2.2   Software Setup on the Host Board

This section describes how to configure the software on the Host board.

1. Ensure that the appropriate image is flashed to the BlackBox board as described in Section 8.2.1, "Software Setup on the BlackBox Board".

2. Depending on the target board, from the BeeKit main window, select the appropriate HCS08 (MC1320x) or ARM7 (MC1322x) BeeStack codebase.

3. In the New Project window shown in Figure 8-1, review the available ZigBee Host templates located in the Project types display area. Be sure to review and select only those types that have titles beginning with "BlackBox Host". For example, BlackBox Host Automation Applications.
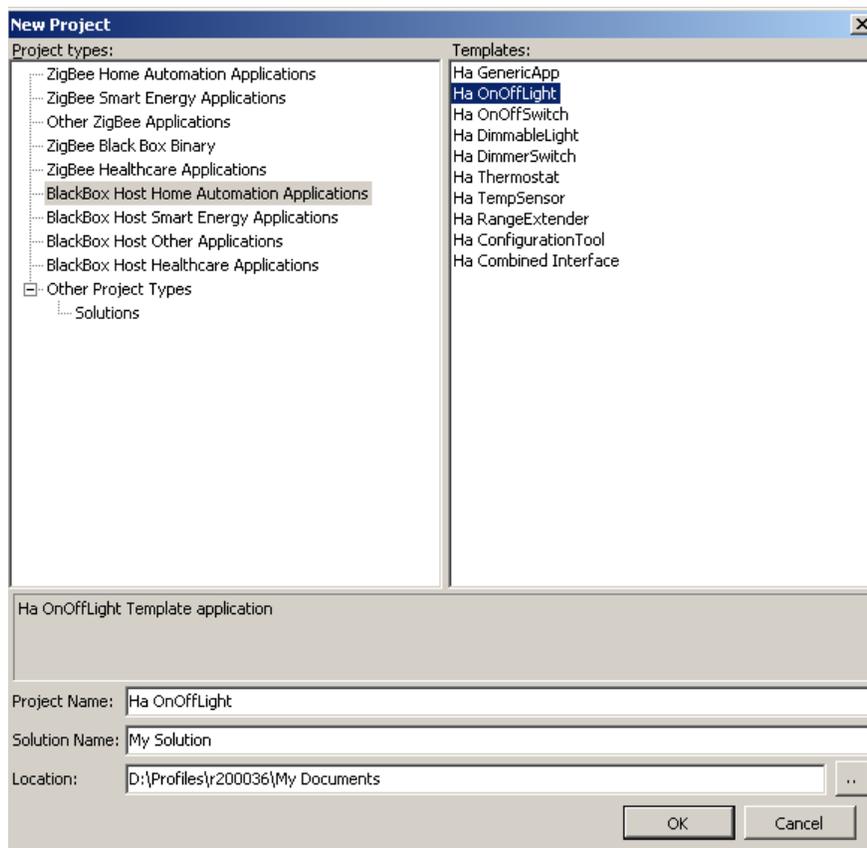

Figure 8-1. New Project Window

NOTE

Because the 802.15.4 Mac/Phy software is configured within the BlackBox, the Host project does not contain this component and it is not shown in the Solution Explorer display area in BeeKit (Figure 8-2). Instead, a BlackBox Host BeeStack Wrapper is included. Choosing this wrapper allows users to select the Host type (Coordinator, Router, End Device or Combo device) and whether to include the security library (Figure 8-2).
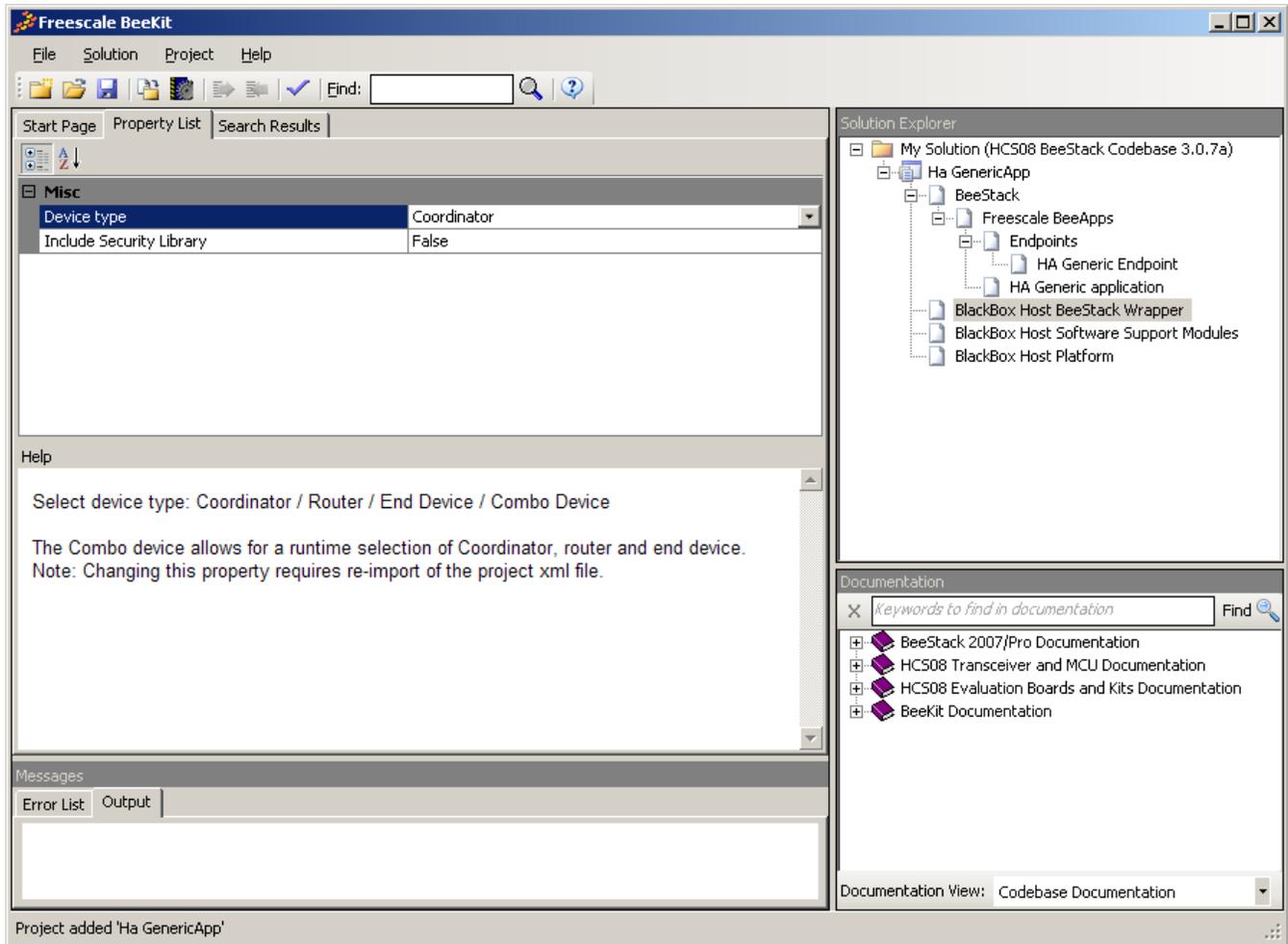
BeeStack™ Application Development Guide for ZigBee 2007, Rev. 1.3

Figure 8-2. Solution Explorer Display Area

NOTE

Ensure that the selected Host type matches the BlackBox device type. For example, if the BlackBox is set to function as a Coordinator, then the Host should also function as a Coordinator. For more details, see Section 7.1, "Generating a BeeStack BlackBox Project with BeeKit".