BeeStack Consumer Application

User's Guide

Document Number: BSCONAUG
Rev. 1.6
06/2011

freescale™
semiconductor

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2008, 2009, 201, 20110. All rights reserved.

Contents

About This Book . . . iii
Audience . . . iii
Organization . . . iii
Revision History . . . iv
Definitions, Acronyms, and Abbreviations . . . iv
References . . . iv

Chapter 1
Introduction

1.1 What This Document Describes . . . 1-2
1.2 What This Document Does Not Describe . . . 1-2
1.3 BeeKit . . . 1-3
1.4 CodeWarrior . . . 1-4

Chapter 2
Generating a Sample Application

2.1 Connecting the Boards to the PC . . . 2-1
2.2 Generating the Application in BeeKit . . . 2-1
2.3 Application Project Folder Structure . . . 2-3
2.4 Downloading the Application to a Target Board . . . 2-4

Chapter 3
Platform Resource Overview

3.1 Task Scheduler . . . 3-1
3.1.1 Default Tasks . . . 3-2
3.2 Timers . . . 3-2
3.3 NVM Component . . . 3-3
3.4 UART Component . . . 3-5
3.5 Keyboard Component . . . 3-6
3.6 Display Component . . . 3-7
3.7 LED Component . . . 3-8
3.8 Low-Power Component . . . 3-8
3.9 Message Buffers . . . 3-9
3.10 Memory Allocation . . . 3-9
3.11 Managing the C Stack . . . 3-9
3.12 System Memory Considerations . . . 3-10

Chapter 4
Creating a Custom Application

4.1 Interfacing with the BeeStack Consumer Stack . . . 4-1
4.1.1 main() Function . . . 4-1
4.1.2 Idle Task . . . 4-2

4.1.3 BeeStack Consumer API Function Calls 4-2
4.1.4 SAP Handlers and Messages from the BeeStack Consumer Layer 4-3
4.1.5 Required Global Variables 4-5
4.1.6 Configuring NiB Attributes 4-6
4.2 Application Main Task 4-6
4.2.1 Application Initialization 4-6
4.2.2 Application Task Handler 4-7
4.3 Starting the Node 4-7
4.4 Discovery Process 4-8
4.4.1 Discovery Process (Originator Device) 4-8
4.4.2 Discovery Process (Recipient) 4-10
4.5 Pairing Process 4-12
4.5.1 Pairing Process (Originator Node) 4-12
4.5.2 Pairing Process (Recipient Node) 4-13
4.5.3 Security 4-13
4.5.4 Offline Pairing 4-14
4.6 Data Transmit and Receive 4-15
4.7 ZigBee Remote Control (ZRC) Profile 4-16
4.7.1 Push Button Pairing 4-16
4.7.2 ZRC Commands 4-18

Chapter 5
Debug

5.1 P&E MultiLink BDM 5-1
5.2 LEDs and the Display 5-1
5.3 Network Protocol Analyzers 5-2
5.4 ZigBee Test Client 5-2

Chapter 6
Working With BeeStack Consumer/SynkroRF Hybrid Nodes

6.1 Introducing the Hybrid Stack 6-1
6.2 Interfacing with the Hybrid Stack 6-1
6.3 SynkroRF API Function Calls 6-1
6.4 Network Layer SAP Handlers and Messages 6-1
6.4.1 SAP Handlers 6-1
6.4.2 General Message Structure 6-2
6.5 Required Global Variables 6-3
6.6 Starting the Node 6-3
6.7 SynkroRF Pairing 6-3
6.7.1 SynkroRF Pairing on the Controller 6-4
6.7.2 SynkroRF Pairing on the Target 6-4
6.8 SynkroRF Command Transmit and Receive 6-6

About This Book

This guide provides information about the creation and maintenance of a network based on the Freescale BeeStack Consumer implementation. The generic application described in this guide is actually a set of two applications created on top of the BeeStack Consumer layer. These applications currently run on the following Freescale development boards:

- 1321x Network Control Board (NCB)
- 1321x Sensor Reference Board (SRB)
- 1320x-QE128 Evaluation Board (EVB)
- 1323x Remote Control Motherboard (RCM)
- 1323x Remote Extender Motherboard (REM)

The applications communicate with the PC via a UART emulated over a USB port. The output of the application is human readable text, while the input is ASCII characters that can be typed from a terminal emulation program.

Audience

This document is intended for software developers writing applications based on the BeeStack Consumer stack using Freescale development tools.

It is assumed that users are programmers familiar with the C programming language and have read the RF4CE v1.00 specification.

Organization

This document contains the following chapters:

Chapter 1	Introduction -- This chapter provides an introduction to the Freescale BeeStack Consumer stack.
Chapter 2	Generating a Sample Application - Details how to use BeeKit to generate a sample solution, export it as an xml file, import it into CodeWarrior, and compile and load it to a target board.
Chapter 3	Platform Resource Overview -- Describes how to use the non-hardware related platform components, including timers, the low power library, task scheduler, messages, data queues and message buffer allocation. It also describes how to determine how much RAM and Flash is available to the application and what to do if an application exceeds the target hardware memory size.
Chapter 4	Creating a Custom Application -- Provides an overview of the hardware related platform components.
Chapter 5	Debug - Describes how to debug an BeeStack Consumer networking application.
Chapter 6	Working With BeeStack Consumer/SynkroRF Hybrid Nodes - Describes the process of designing and writing a custom application over the BeeStack Consumer/SynkroRF hybrid stack.

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 1.5).

Revision History

Date / Author	Description / Location of Changes
February 2011, Dev Team	Updated MC1323x and CodeWarrior information.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

API	Application Programming Interface
LQI	Link Quality Indicator
NIB	Network Information Base
NLDE	Network Layer Data Entity
NLME	Network Layer Management Entity
NV	Non volatile
NVM	Non volatile memory
NW Layer	Network Layer
PAN	Personal Area Network
RF	Radio Frequency
SAP	Service Access Point

References

The following sources were referenced to produce this book:

1. RF4CE Specification version 1.0.0, Document 080002r04
2. IEEE 802.15.4 Standard -2003, Part 14.5: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs), The Institute of Electrical and Electronics Engineers, Inc. October 2003
3. *BeeStack Consumer Application Reference Manual* (BSCONRM)
4. *BeeStack Consumer Blackbox Interface User's Guide* (BSCONBBIUG)
5. *Freescale BeeKit Wireless Connectivity Toolkit User's Guide* (BKWCTKUG)

Chapter 1
Introduction

The BeeStack Consumer software is a networking layer that sits on top of the IEEE® 802.15.4 MAC and PHY layers. It is designed for Wireless Personal Area Networks (WPANs) and conveys information over short distances among the participants in the network. It enables small, power efficient, inexpensive solutions to be implemented for a wide range of applications. Some key characteristics of a BeeStack Consumer network are:

- An over the air data rate of 250 kbit/s in the 2.4 GHz band
- 3 independent communication channels in the 2.4 GHz band
- 2 network node types, controller node and respectively target node
- Channel Agility mechanism
- Provides robustness and ease of use
- Includes essential functionality to build and support a CE network

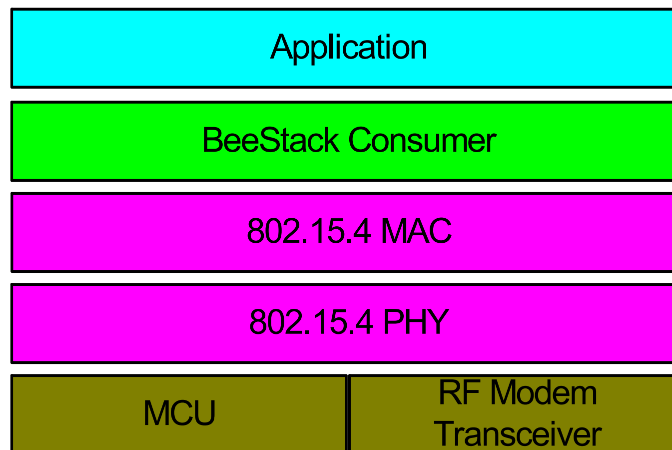

Figure 1-1. BeeStack Consumer Software Architecture

The BeeStack Consumer layer uses components from the standard HCS08 Freescale platform, which is also used by the Freescale implementations of 802.15.4. MAC or ZigBee™ layers. For more details about the platform components, see the *Freescale Platform Reference Manual*.

This section describes what is and what is not included in this guide. It also describes the basic development process for BeeStack Consumer applications using both BeeKit and CodeWarrior.

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

Currently, the BeeStack Consumer stack is written for the Freescale HCS08 microcontroller, but the concepts in the document may apply to the BeeStack Consumer stack ported to any microcontroller.

1.1 What This Document Describes

- How to generate, build and customize BeeStack Consumer applications
- The resources available to the BeeStack Consumer application designer (Freescale task scheduler, timers, dynamic memory) and suggestions on how to best use them
- An example BeeStack Consumer application illustrating the above

1.2 What This Document Does Not Describe

- How to install BeeKit. For the BeeKit installation process, see the *BeeKit Wireless Connectivity Toolkit Quick Start Guide*
- How to install CodeWarrior. For instructions, see the CodeWarrior documentation
- The Radio Frequency for Consumer Electronics (BeeStack Consumer) standard
- The complete Freescale BeeStack Consumer API. For a detailed description of the Freescale BeeStack Consumer API see the *Freescale BeeStack Consumer Reference Manual*.

1.3 BeeKit

BeeKit is a desktop PC graphical application that allows developers to configure Freescale networking solutions, including BeeStack Consumer, BeeStack, IEEE® 802.15.4 MAC, and the Freescale proprietary Simple MAC(SMAC). Figure 1-2 shows the BeeKit Wireless Connectivity start-up window.

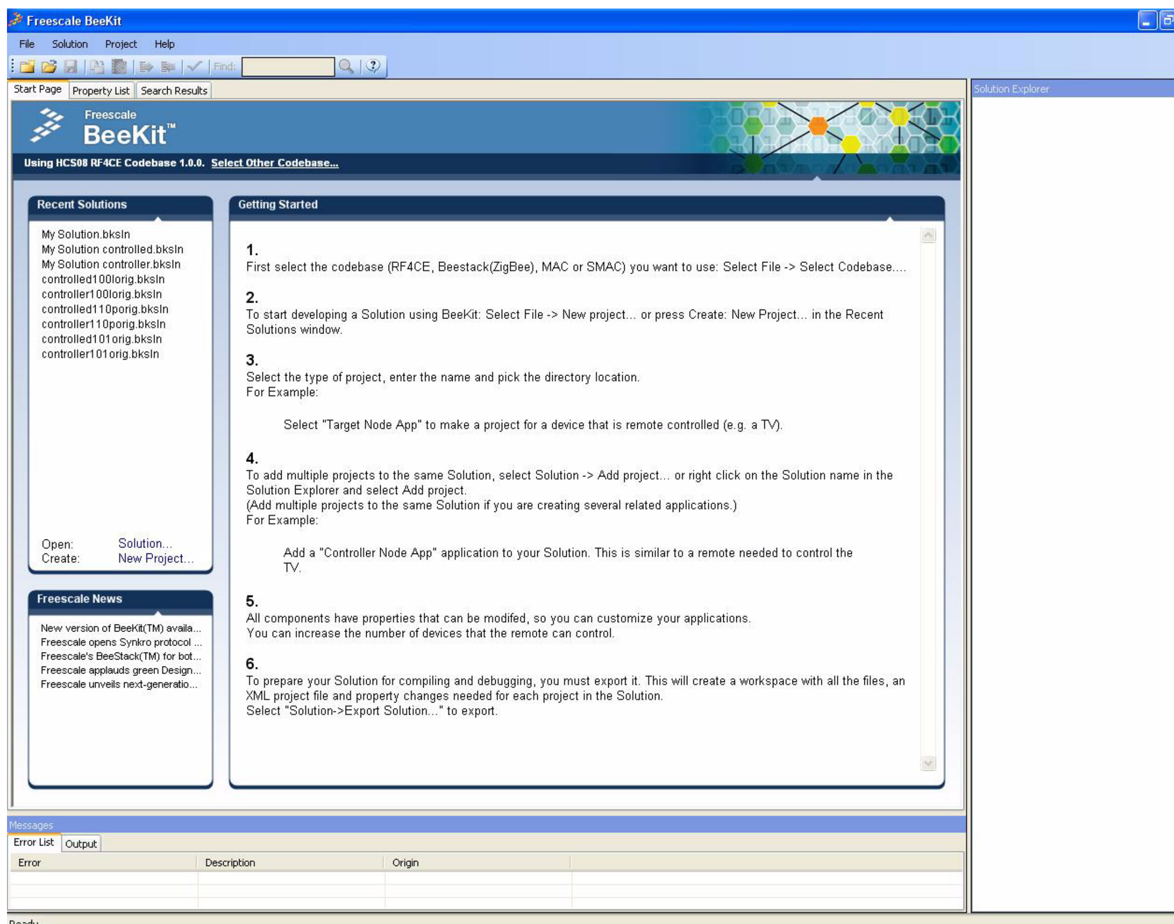

Figure 1-2. Freescale BeeKit

BeeKit creates a sample application from templates, providing the ability to set properties (i.e. compile-time options) which configure the application and BeeStack Consumer. The resulting project may then be exported as an XML file to a file folder and imported into CodeWarrior for editing, compiling and debugging.

See the *BeeKit Wireless Connectivity Toolkit User's Guide* for more information.

1.4 CodeWarrior

CodeWarrior is a desktop PC integrated development environment (IDE) which includes a C compiler for the HCS08 MCU and the other tools to generate a downloadable image as well as a debugger that can download code into the HCS08 MCU FLASH memory.

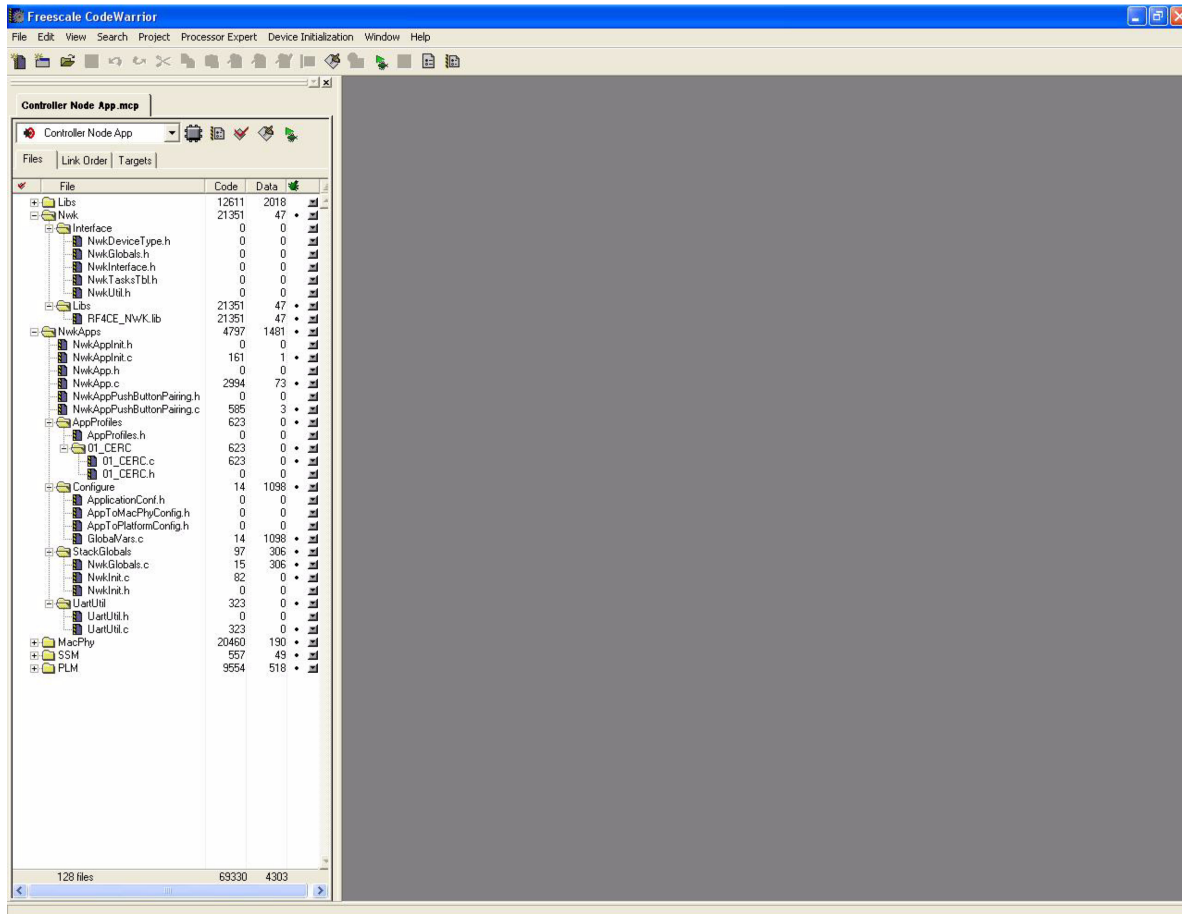

Figure 1-3. Freescale CodeWarrior

CodeWarrior takes the output of BeeKit (an XML file that it can import and convert into a project file and a source directory tree) 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.

Chapter 2
Generating a Sample Application

This chapter describes how to generate a sample generic application from BeeKit. Freescale assumes that users are familiar with BeeKit and have installed the most recent HCS08 BeeStack Consumer Codebase. The sample application described in this chapter works on the Freescale development boards listed in Preface Section, “About This Book”. This example uses the NCB and the SRB, but the boards can be substituted as follows:

- RCM = NCB
- REM = SRB

There are two versions of the application:

- One for target BeeStack Consumer nodes
- One for controller nodes

This chapter describes the steps needed to generate the application, export the solution, import it to CodeWarrior as a project and download it to a target board.

2.1 Connecting the Boards to the PC

The target boards are connected to the PC through a USB port which emulates a UART. For this to work correctly, drivers must be installed on the PC. BeeKit comes with the necessary drivers. The default location is:

```
C:\Program Files\Freescale\Drivers
```

Connect the boards to the PC with the USB cable. When the Found New Hardware wizard appears, point Windows to that location to search for drivers. After installation is complete, each device will have a COM port associated with it.

2.2 Generating the Application in BeeKit

To generate the BeeStack Consumer application in BeeKit, perform the following tasks:

1. Launch BeeKit.
2. Select the appropriate HCS08 BeeStack Consumer Codebase.
3. Create a new project and choose a suitable destination folder.

1. Generate the controller application by choosing the Controller Node App template.

1. Enter the parameters as needed. This project contains several customizable parameters as follows:

- Hardware target board – Allows users to choose from all currently defined target boards. The application is functionally identical on all the boards. However, among other things, the NCB and RCM have an LCD where the application can display status messages.
- Platform modules to enable -- Optional platform modules are used to enhance the application and can be selected here. The available modules are:
  - LEDs -- Optional output for the application. LEDs can flash or be turned on/off depending on the state of the application
  - Keyboard -- Optional input for the application
  - Low power module -- Enables the whole platform to enter sleep mode when the application allows it.
  - LCD – This is only for the NCB and is an optional output for the application
- UART Communication Settings -- Configure the UART used to communicate from the PC with the target board. Select the UART baud rate and whether UART communication is to be emulated over USB.
- BeeStack Consumer device identifiers -- Several BeeStack Consumer specific parameters can be configured here as follows:
  - The device identification string -- The BeeStack Consumer user string. Enter any ASCII string at most 14 characters long in the text box (it will be zero terminated by the application). Each device should have its own unique user string.
  - First supported device type -- The BeeStack Consumer device type that this device supports initially. Select one from the list. Remote controller is recommended for controller nodes. Additional device types can be added in the application code.
  - First supported profile ID -- The BeeStack Consumer profile ID that this device supports initially. Additional profile IDs can be added in the application code.
  - Extended address of the device -- The 64 bit IEEE 802.15.4 extended address (MAC address) of the device. Enter any number in hex format. Make sure that no devices in close proximity have the same extended address.
  - Vendor ID -- The BeeStack Consumer vendor ID of the device manufacturer. See the *BeeStack Consumer Vendor ID list* for all assigned vendor IDs.
  - Device vendor name -- The BeeStack Consumer vendor string, a name that is used to identify the vendor of a device. Enter any 6 ASCII characters here, the application will zero terminate the string.
- BeeStack Consumer Data Request options -- Some parameters for the data request primitive that is exercised by the application can be configured here. The parameters that can be configured are:
  - The profile ID -- The profile field of the data request primitive
  - Application Command ID -- The ID of the command to be sent with the data request. Profile specific. The description next to the selection box specifies the significance of the ID in the ZRC profile.
  - The transmission options to be used with the data request
- BeeStack Consumer Discovery Options -- Several parameters relevant to pairing and discovery can be configured as follows:

- — The maximum number of entries in the pairing table
- — The device type to search for during discovery
- — The discovery duration in MAC symbols (a 802.15.4 MAC symbol when transmitting in the 2.4GHz band is 16 s)
- — The maximum number of discovery attempts
- — The discovery repetition interval in MAC symbols

After the solution is exported, the code for the application is generated and placed in the specified destination folder. A CodeWarrior project xml file is also generated to allow CodeWarrior to import the project. Users now have a complete application project at their disposal which can be compiled to a binary image suitable to be downloaded into a target board.

The steps needed to generate a target application are nearly identical to that of a Controller Node. The only difference is users must select a Target Node application template (Step 4) and different parameters can be customized during the BeeKit solution configuration, such as BeeStack Consumer discovery options.

The following BeeStack Consumer parameters can be configured for the Target Node application template as follows:

- The maximum number of pairing table entries
- The discovery Link Quality Indication value

2.3 Application Project Folder Structure

Figure 2-1 shows a Controller Node demonstration application project folder structure.

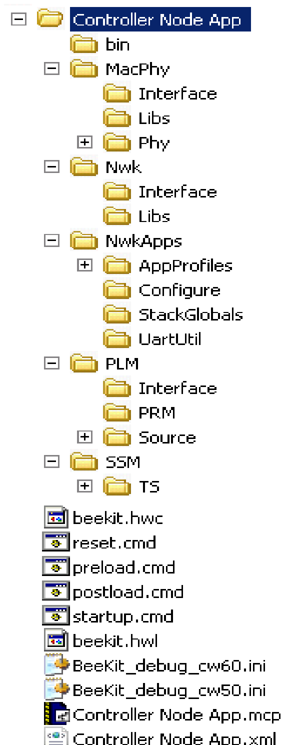

Figure 2-1. Folder Structure (Controller Node)

2.4 Downloading the Application to a Target Board

To download the application into a target board, the BeeStack Consumer application project must be open in CodeWarrior. From the CodeWarrior main window select the Debug option. This rebuilds the project if it is not up to date and launches the True-Time Simulator.

The True-Time Simulator is already configured by BeeKit. To proceed with downloading, ensure that the P&E Multilink USB debugger is connected to the BDM port on the target board and press Connect (the Reset button on the board).

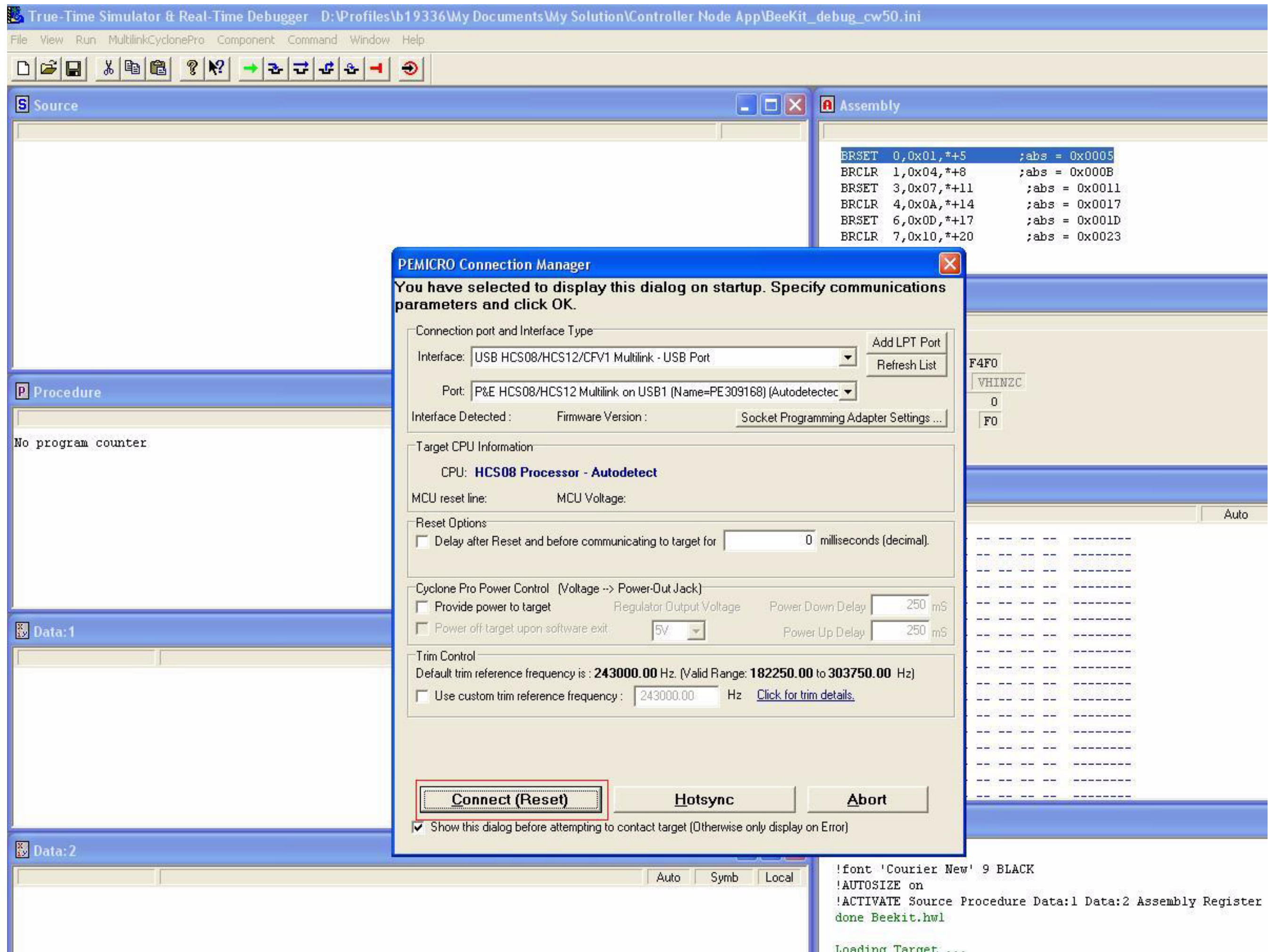

Figure 2-2. True-Time Simulator

After the download is complete the controller application is ready. Open a terminal emulator on the COM port of the target board and set the UART baud rate the same as in the BeeKit configuration. Press the reset button on the board.

At this point the application is ready. The open terminal window will display the application menu. Press the corresponding key of each menu item (the first letter in the row) to start the appropriate action.

Chapter 3
Platform Resource Overview

This chapter describes various platform resources, including the use of messages, timers, data queues, and the task scheduler. It also describes how to determine how much RAM and FLASH are available to the application and what to do if an application exceeds memory size.

3.1 Task Scheduler

The entire BeeStack Consumer protocol stack runs under the control of a priority based non-preemptive, event-driven task scheduler. It is essentially a forever loop that checks all registered tasks in order of their priority for outstanding events and calls their event handlers them if they do.

Tasks are created using TS_CreateTask(), which accepts two parameters:

- The priority of the task to be registered
- The event handler of the task, which is a pointer to a function with the following prototype:

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

  `event_t` is an unsigned 16 bit integer used as a bit mask for events. TS_CreateTask returns the task ID of the newly created task. Events are sent to a task using TS_SendEvent(), which has two parameters:

- The ID of the task to send an event to
- The event to be sent

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.

A task cannot have the same event being transmitted more than once during a task run, as sending the same event to a task more than once will set the same bit in the event mask. Also, be sure to handle all events when task handler is called because the task scheduler does not save unhandled events.

For more detailed information about the task scheduler see the *Freescale Platform Reference Manual*.

3.1.1 Default Tasks

The tasks created by default are briefly described in their ascending priority order:

- The idle task has the lowest priority and runs when all the other tasks are do not have any pending events. The idle task event handler function must be implemented in the application
- The BeeStack Consumer task executes the BeeStack Consumer state machine
- The MAC task executes the MAC state machine
- The timer task is responsible for maintaining the software timers. This is the highest priority task

In this example, the application code is contained in the application task, which is assigned priority 0x80.

Applications may use any task priority in the range 0x40 – 0xbf. If needed, the application code may be contained in several tasks.

3.2 Timers

Use software timers whenever some event must be timed (e.g. blinking LEDs or for application timing purposes). The software timers 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.

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

The number of timers available to the application is defined at compile-time by modifying a define macro called gTmrApplicationTimers_c, in `TMR_Interface.h`. The default number of application timers is 6. Each timer requires 7 bytes of RAM.

Timers are started 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) can be forcefully stopped using the following:

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

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

Timers provide information about the expiration of the programmed time interval via a callback. The callback is a function with the following prototype (it can have any name the application chooses)

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

When a timer expires, it calls the callback function of the application's choice, as given to the start timer functions. Typically, the callback function should set an event, but it could do any work required. The callback is in the timer task context.

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

When one or more software timer is active (currently counting down), low-power mode will not enter deep sleep, but will use light sleep only.

For more detailed information about timers see the *Freescale Platform Reference Manual*.

3.3 NVM Component

Non-Volatile Memory (NVM) provides a permanent storage mechanism that can survive system resets and power outages. Parts of the HCS08 flash memory are used to provide three 512 byte pages for storage. Of these, only two are in use at any time, the third being kept on standby for writing new data. In each page 8 bytes are used for housekeeping, leaving only 504 bytes available for actual storage.

The NVM writes data to flash through data sets. A data set is an array of type NvDataItemDescription_t

```
typedef struct NvDataItemDescription_tag
{
void *pointer;
NvDataItemLength_t length;
} NvDataItemDescription_t;
```

It contains several {array, length} pairs which point to the data that needs to be saved in flash. When the NVM component saves a dataset, it parses the dataset array and reads from every pointer the amount of data specified in the length field. The last element in the array must be a {NULL, 0} pair.

The NVM handles two data sets. One is reserved for the BeeStack Consumer layer (who stores the pair table and the NIB table in it), the other is available for the application.

To save data to FLASH, first construct the dataset array, like this:

```
NvDataItemDescription_t const gaNvAppDataSet[] =
{
{array1, length1},
{array2, length2},
...
{NULL, 0}       /* Required end-of-table marker. */
};
```

The `gaNvAppDataSet` array must be defined by the application even if it is not actually used. In this case, just insert the {NULL, 0} pair.

Second, whenever data needs to be saved, mark the dataset as pending to be updated in FLASH, by calling the following:

```
NV_SaveOnIdle(gNvDataSet_App_ID_c);
```

This does not actually save the data, instead it is only marked as dirty. Data is written to FLASH next time NV_Idle() is called from the idle task.

To restore the data set from FLASH, call:

```
NV_RestoreDataSet(gNvDataSet_App_ID_c);
```

This updates the arrays specified in the data set with the data from the FLASH.

The application also has some degree of control over how and when the BeeStack Consumer layer saves its dataset to FLASH. Specifically, some data can be marked as dirty:

- the entire BeeStack Consumer layer dataset – this includes the pair table (nodeData) and the NIB table (gNwkNib); to mark it as dirty call

```
NWK_SavePersistentData();
```

- just the BeeStack Consumer frame counter; to mark it as dirty call

```
NWK_SaveFrameCounter();
```

Just as with the application dataset, the data is written to FLASH only next time NV_Idle() is called from the idle task.

For more detailed information about the NVM component see the *Freescale Platform Reference Manual*.

3.4 UART Component

One platform component that is often useful in BeeStack Consumer applications is the UART component. This allows one or both of the HCS08 Serial Communications Interface (SCI) 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. The SRB and NCB emulate UART over USB, but he 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.

UART initialization is performed by calling the following functions:

```
Uart_ModuleInit(void);
UartX_SetBaud(uint16_t baudRate);
UartX_SetRxCallBack(void (*pfCallBack)(void));
```

Uart_ModuleInit() initializes the UART module, UartX_SetBaud() sets the baud rate, UartX_SetRxCallBack registers a callback which will be called whenever a byte is received over the serial line. The “UartX_...” functions are macros which evaluate to the corresponding functions for either the UART1 module or the UART2 module, depending on the value of gUart_PortDefault_d, which is set to 2 by BeeKit.

Transmission over UART is done with UartX_Transmit:

```
bool_t UartX_Transmit
(
unsigned char const *pBuf,
index_t bufLen,
void (*pfCallBack)(unsigned char const *pBuf)
);
```

The UART driver has no buffer for the characters to be transmitted (neither does the hardware). UartX_Transmit stores only the pointer to the data. The driver can remember three pointers at most until actual transmission over the serial line must occur. UartXTransmit returns FALSE if the pointer could not be stored. The callback is called when transmission over the serial line is complete.
Reception is done with:

```
bool_t UartX_GetByteFromRxBuffer(unsigned char *pDst)
```

This reads a character from the RX buffer and places it at the address provided by pDest. Returns FALSE if nothing was read.

The typical procedure to read from the UART is to wait for pfCallBack to be called then read all the bytes in the buffer in a while loop:

```
while(UartX_GetByteFromRxBuffer(&pressedKey))
{
...
}
```

In the example application, the UART Rx callback simply sends an event to the application task informing it of received bytes over the serial line.

The sample application includes a utility library which provides a simple printf() function over UART, which can be used to print data in a hyper terminal window:

```
void UartUtil_Print(uint8_t* pString, uartUtilBlock_t allowToBlock)
```

Where pString is a pointer to the string to be printed and allowToBlock can have one of two values:

- gNoBlock_d – the function returns immediately. At most three such UartUtil_Print calls can be issued until the UART driver must be allowed to transmit. The advantage is that the function call is asynchronous.
- gAllowToBlock_d – the function only returns after transmission over the serial line is complete. This is the recommended value for normal use. The function call is basically synchronous.

For more detailed information about the UART component see the *Freescale Platform Reference Manual*.

3.5 Keyboard Component

The keyboard component supports 8 key inputs using only 4 physical keys. Each key press can be detected as either a short or long press. The keyboard interface is also a way to wake low power units on interrupt (this is how we use it in this sample application). From the application's standpoint, activity on a keyboard pin will be interpreted as receiving a key press and can be acted on accordingly.

To be used, the keyboard module must first be initialized by calling:

```
KBD_Init(void (*kbd_handler)()(key_event_t key_event));
```

Where kbd_handler is a pointer to the keyboard handler callback function.

The keyboard handler is called every time a key is pressed. The key_event parameter indicates the key pressed. It can have one of the following values:

```
enum {
  gKBD_EventSW1_c = 1,
  gKBD_EventSW2_c,
  gKBD_EventSW3_c,
  gKBD_EventSW4_c,
  gKBD_EventLongSW1_c,
  gKBD_EventLongSW2_c,
  gKBD_EventLongSW3_c,
  gKBD_EventLongSW4_c
};
```

The keyboard brings the device out of low-power mode. As such, the keyboard handler looks like this:

```
void App_HandleKeyboard(key_event_t keyEvent)
{
  switch(keyEvent)
  {
    case gKBD_EventSW1_c:
    case gKBD_EventSW2_c:
    case gKBD_EventSW3_c:
    case gKBD_EventSW4_c:
    case gKBD_EventLongSW1_c:
    case gKBD_EventLongSW2_c:
    case gKBD_EventLongSW3_c:
    case gKBD_EventLongSW4_c:
      if(PWR_CheckIfDeviceCanGoToSleep())
      {
        UartUtil_Print("\n\rEntering in low power mode is not allowed by the 
application.",gAllowToBlock_d);
      /* Do not allow device to enter the low power mode. The low power mode is configured 
in PWR_Configuration.h */
        PWR_DisallowDeviceToSleep();
      }
      break;
  }
}
```

Alternatively, an event could be sent to the application task, requiring it to react to the key press.

For more detailed information about the keyboard component see the *Freescale Platform Reference Manual*.

3.6 Display Component

The display component can be used to support an LCD controller. There is a LCD built into the Freescale NCB. The display software is specific to this controller. Unless using the same controller as found on the NCB, the file `Display.c` (found in the PLM folder) will need to be modified.

The LCD on the NCB has two lines of 16 characters each. To be used, the display module must first be initialized by calling

```
LCD_Init();
```

Two function are most commonly used:

```
LCD_ClearDisplay() - erases the display
```

`LCD_WriteString(line, pstr)` – writes the given string to the given line on the display, automatically truncating strings which are too long (over 16 characters).

For more detailed information about the display component see the *Freescale Platform Reference Manual*.

3.7 LED Component

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

To be used, the LED module must first be initialized by calling

```
LED_Init();
```

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 — the LED blinks once and then turns off
- gLedOn_c — on solid
- gLedOff_c — off solid
- gLedToggle_c — toggle state

More than one LED can be controlled at once. LEDs can be combined in the LED_SetLet() function call using bitwise OR. For example:

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

For more detailed information about the LED component see the *Freescale Platform Reference Manual*.

3.8 Low-Power Component

The low power component allows the application to request the MCU and transceiver to enter low power modes and to handle the entering and exiting of these modes in a safely manner. Both MCU and transceiver have multiple low power modes. For example, STOP1, STOP2, STOP3 in the case of the MCU, doze, acoma, hibernate in the case of the transceiver. The options for configuring what are the low power modes the MCU and the transceiver should use can be found in `PWR_Configuration.h` file.

Keep in mind also that the power library will not enter deep sleep unless all of the software timers are off. Also, the power library will not enter sleep at all unless every task is idle.

For more detailed information about the Low-Power component see the *Freescale Platform Reference Manual*.

3.9 Message Buffers

Message buffers are a form of dynamic memory allocation in the BeeStack Consumer stack. The BeeStack Consumer stack uses message buffers to pass information between the layers in the stack, thus decoupling the execution context which ensures that the call stack does not build up between layers when communicating.

There are two types of buffers, small buffers and big buffers. Small buffers can store up to 22 bytes of information and are generally used for control messages. Big messages can store up to 144 bytes of information and are large enough to hold the largest BeeStack Consumer packet, including all over-the-air frame headers plus some additional information for housekeeping. Big messages are used for both data transmit and receive. The default configuration includes a total of 5 small messages and 5 big messages available to the stack and application (plus one extra big message reserved for the MAC layer).

In order to use a message buffer, it must first be allocated. The function to do that is MSG_Alloc() which takes as a parameter the size of the memory required and returns a pointer to the smallest available buffer that can hold the data of the given size, or NULL if there are no available buffers. When buffers are no longer needed they must be freed by using MSG_Free(), which takes as a parameter a pointer to the buffer to be freed.

The BeeStack Consumer layer sends messages to the application layer in message buffers. The application is responsible for freeing these. The BeeStack Consumer API function calls do not expect payloads to be in contained in message buffers. If buffers are allocated to construct these payloads, they must be free in the application.

3.10 Memory Allocation

In the BeeStack Consumer stack, memory is generally allocated statically at compile-time. There is no concept in the BeeStack Consumer stack of a heap in the general sense, and there is no malloc(). Be careful when allocating message buffers for the application allocation as they are also needed by the BeeStack Consumer stack to properly handle networking (incoming frames will be lost without available buffers to store them in).

3.11 Managing the C Stack

The C Stack is 352 bytes by default. When a task gains control, it is at the top of the 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 does not 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 `Linker.prm`.

3.12 System Memory Considerations

In deeply embedded systems, both RAM and ROM are a limited resource. The MC1321x MCU contains 4 KB of RAM and approximately 60 KB of FLASH.

Depending on the implemented functionality, a given application could exceed either RAM or FLASH.

The following is a list of tips to help reduce FLASH usage:

- Do not use UART driver unless the application needs it.
- Look in the `.map` file 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.

The following is a list of tips to help reduce RAM usage:

- Make tables as small as possible.
- Reduce the C stack size if the application doesn't use the full stack.
- Reduce the number of software timers to the minimum needed by the application

Chapter 4
Creating a Custom Application

This chapter describes how to design and write a custom application over the BeeStack Consumer stack. This chapter also shows how to properly interface with the BeeStack Consumer stack and develop an application that can participate in an BeeStack Consumer network.

4.1 Interfacing with the BeeStack Consumer Stack

There are several global variables and function calls that the BeeStack Consumer stack expects to find in the application.

4.1.1 main() Function

The main() function must be implemented by the application. A typical main() function can be found in the sample generic application generated by BeeKit:

Structure

```
void main(void) {
      /* Disable all interrupts */
      IrqControlLib_DisableAllIrqs();
      /* Initialize 802.15.4 MAC and PHY */
      Init_802_15_4();
      /* Initialize the operation system kernel */
      TS_Init();
      /* Initialize the platform timer module */
      TMR_Init();
      /* Initialize BeeStack Consumer layer */
      NwkInit();
      /* Create the application task */
      gAppTaskID = TS_CreateTask(gTsAppTaskPriority_c, App_MainTask);
      /* Initialize the application over BeeStack Consumer layer */
      App_Init();
      /* Start the task scheduler. Does not return. */
      TS_Scheduler();
}
```

Description

It turns off interrupts, runs the initialization functions for all BeeStack Consumer stack tasks (which also create the tasks), creates the application task and initializes it and then passes control to the task scheduler. The interrupts are then enabled by the App_Init function.

4.1.2 Idle Task

The application also needs to implement the idle task handler. The idle task is the lowest priority task in the entire application and also the first one created (it is created at task scheduler initialization). The idle task is the best place to implement low priority operations or operations which require that the rest of the system is idle, such as entering low power or saving data to flash.

This example idle task does exactly that:

Structure

```
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 gNvStorageIncluded_d
     /* Process NV Storage save-on-idle and save-on-count requests. */
         NvIdle();
     #endif
     #if gLpmIncluded_d
     HandleLowPower();
     #endif
}
}
```

Description

First, some possible error flags are cleared form the HCS08 serial communication status register. Secondly, any data that needs saving (marked as dirty) is written to flash. Third, the system enters low power if both the application and the network allow this.

4.1.3 BeeStack Consumer API Function Calls

To communicate with the BeeStack Consumer layer, that is, access request and response primitives, several API functions are available to the application. The prototypes for the functions can be found in NwkInterface.h. A function exists for each BeeStack Consumer request or response primitive. Some of the functions are synchronous, meaning that the service they execute will be completed when the function will return, while others are asynchronous, meaning that the service they execute will be started during the call and run even after the function has returned. The completion of their action will be signaled using confirm messages sent from the network to the application through Service Access Points.

The function calls are mostly asynchronous. The asynchronous functions will always return a status. Some errors that appear during the call, like parameter validation, can be reported immediately by the error code. In such cases the request is terminated immediately. If an asynchronous function call returns gNWSuccess_c, the application must expect the arrival of a confirm message, containing information

about the completion of the requested action. For a detailed list of BeeStack Consumer API calls and their possible return codes see the Freescale BeeStack Consumer Reference Manual.

4.1.4 SAP Handlers and Messages from the BeeStack Consumer Layer

4.1.4.1 SAP Handlers

Whenever the BeeStack Consumer layer has information to send to the application, it will call one of two SAP handlers:

```
void NWK_NLME_SapHandler(nwkNlmeToAppMsg_t* nwkNlmeToAppMsg)
```

or

```
void NWK_NLDE_SapHandler(nwkNldeToAppMsg_t* nwkNldeToAppMsg)
```

The NLDE SAP handler is called for NLDE messages (data indication and confirm primitives) while the NLME SAP handler is called for NLME messages (all other indication and confirm primitives). The SAP handlers must be implemented by the application in order to receive messages from the BeeStack Consumer layer.

The parameter to the function calls is a pointer to an allocated message buffer which contain the message. It is the responsibility of the application to free the buffer when processing is complete.

In this example, two messages queues have been implemented in the application. The SAP handlers add the message pointers to the queues and send events to the application main task, indicating that new information is available from the network to be processed when convenient.

```
void NWK_NLME_SapHandler(nwkNlmeToAppMsg_t* nwkNlmeToAppMsg)
{
/* Put the incoming NLME message in the applications input queue.*/
MSG_Queue(&mNlmeAppInputQueue, nwkNlmeToAppMsg);
TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlme_c);
}
```

4.1.4.2 Message Queues

Message queues hold pointers to messages and implement the basic functionality of FIFO queues, having service calls to add a message, remove a message and check if the queue is empty.

The queue itself is a variable of type anchor_t. To use a queue, it must first be initialized. Define a variable of type anchor_t, than call MSG_InitQueue() with the address of the variable as a parameter:

```
static anchor_t mNlmeAppInputQueue;
MSG_InitQueue(&mNlmeAppInputQueue);
```

To add a message to a queue use MSG_Queue() which takes two parameters:

- the address of the queue variable
- the pointer to the message

```
MSG_Queue(&mNlmeAppInputQueue, nwkNlmeToAppMsg);
```

To remove a message from a queue use MSG_DeQueue() which takes the address of the queue variable as a parameter and returns a pointer to the message:

```
pMsgIn = MSG_DeQueue(&mNlmeAppInputQueue);
```

The pointer is NULL if the queue is empty.

To check if there are messages in the queue use MSG_Pending() which takes the address of the queue variable as a parameter and returns TRUE if there are messages in the queue, FALSE if it is empty.

4.1.4.3 General Message Structure

NLME messages have the following structure:

```
typedef struct nwkNlmeToAppMsg_tag
{
nwkNlmeToAppMsgType_t           msgType;
union {
        /*-------------------------------------------------------*/
        nwkNlmeStartCnf_t                                       nwkNlmeStartCnf;
        /*-------------------------------------------------------*/
        nwkNlmeAutoDiscoveryCnf_t     nwkNlmeAutoDiscoveryCnf;
        /*-------------------------------------------------------*/
        nwkNlmeDiscoveryCnf_t         nwkNlmeDiscoveryCnf;
        nwkNlmeDiscoveryInd_t         nwkNlmeDiscoveryInd;
        /*-------------------------------------------------------*/
        nwkNlmePairCnf_t              nwkNlmePairCnf;
        nwkNlmePairInd_t              nwkNlmePairInd;
        /*-------------------------------------------------------*/
        nwkNlmeUnpairCnf_t            nwkNlmeUnpairCnf;
        nwkNlmeUnpairInd_t            nwkNlmeUnpairInd;
        /*-------------------------------------------------------*/
        nwkNlmeCommStatusInd_t        nwkNlmeCommStatusInd;
        /*-------------------------------------------------------*/
      } msgData;
 }nwkNlmeToAppMsg_t;
```

where nwkNlmeToAppMsgType_t is a typedef for:

```
typedef enum {
    /*---------------------------*/
    gNwkNlmeStartCnf_c = 0,
    /*---------------------------*/
    gNwkNlmeAutoDiscoveryCnf_c,
    /*---------------------------*/
    gNwkNlmeDiscoveryCnf_c,
    gNwkNlmeDiscoveryInd_c,
    /*---------------------------*/
    gNwkNlmePairCnf_c,
    gNwkNlmePairInd_c,
    /*---------------------------*/
    gNwkNlmeUnpairCnf_c,
    gNwkNlmeUnpairInd_c,
    /*---------------------------*/
    gNwkNlmeCommStatusInd_c,
    /*---------------------------*/
    gNwkNlmeMax_c
}nwkNlmeToAppMsgType_t
```

NLDE messages have the following structure:

```
typedef struct nwkNldeToAppMsg_tag
{
    nwkNldeToAppMsgType_t    msgType;
    union
    {
    nwkNldeDataCnf_t       nwkNldeDataCnf;
    nwkNldeDataInd_t       nwkNldeDataInd;
    /*-----------------------------------------------------*/
    } msgData;
}nwkNldeToAppMsg_t;
```

where nwkNldeToAppMsgType_t is a typedef for:

```
typedef enum {
        gNwkNldeDataCnf_c       = 0,
        gNwkNldeDataInd_c,
        /*---------------------------*/
        gNwkNldeMax_c
}nwkNldeToAppMsgType_t
```

Determine the message type from the msgType field and use that information to access the correct member of the msgData union.

```
if(gNwkNlmeDiscoveryInd_c == pMsgIn->msgType)
{
        // process pMsgIn->msgData.nwkNlmeDiscoveryInd here
}
```

4.1.5 Required Global Variables

The BeeStack Consumer stack accesses several global variables that must be defined by the application. These are:

- const uint8_t nwkcMaxPairingTableEntries – The maximum number of entries in the pair table. On target nodes, Freescale recommends setting this value to at least 5 so the application can use the network according to the BeeStack Consumer standard.
  — On the MC1321x and MC1322x platforms the value can range between 1 and 8.
- const uint8_t nwkcNodeCapabilities — The capabilities of the local node
- const uint8_t nwkcVendorId[] — An array containing the vendor ID
- uint8_t nwkcVendorString[] — An array containing the vendor string
- uint32_t nwkcFrameCounterWindow — The window of acceptable frame counters
- nodeData_t nodeData — The pair table plus some IEEE 802.15.4 identifying information of the local node like PanId and ShortAddress
- nwkNib_t gNwkNib — The NIB table

All these values can be configured in the NwkGlobals.c file.

4.1.6 Configuring NiB Attributes

To view and modify NIB attributes two functions are available to the application: NLME_SetRequest() and NLME_GetRequest()

```
uint8_t NLME_GetRequest
(
    uint8_t  nibAttribute,
    uint8_t  nibAttributeIndex,
    uint8_t* nibAttributeValue
);
```

NLME_GetRequest fetches the NiB attribute identified by the nibAtrribute and nibAttributeIndex parameters and places it at the address indicated by nibAttributeValue. The function call is synchronous, the value is present at the specified address as soon as the function returns. No confirm message will be sent by the BeeStack Consumer layer.

```
uint8_t NLME_SetRequest
(
    uint8_t  nibAttribute,
    uint8_t  nibAttributeIndex,
    uint8_t* nibAttributeValue
);
```

NLME_SetRequest sets fetches the NiB attribute identified by the nibAtrribute and nibAttributeIndex parameters to the value pointed at by nib nibAttributeValue. The function call is synchronous, the NiB attribute will have the new value as soon as the function returns. No confirm message will be sent by the BeeStack Consumer layer.

For a list of all NiB attributes and their values see the Freescale BeeStack Consumer Reference Manual.

4.2 Application Main Task

The main task of the application is constructed around a state machine. This is helpful in dealing with send request – wait for confirms interactions that are described by the BeeStack Consumer protocol specification. For example, when sending a data request, the application enters the data request state, with substates for sending the request, waiting for the data confirm and returning to the idle state. While waiting for the data confirm, all other messages and inputs are ignored.

4.2.1 Application Initialization

All actions that must be undertaken before the application task handler runs the first time should be placed in an initialization function which is called by main() before the task scheduler runs. The initialization function of this sample application, AppInit(), does the following:

- Initialize the LED, Keyboard, LCD, UART, Keyboard and Low-Power modules
- Enables all the MCU interrupts
- Disallow the device to enter low power
- Initialize the message queues
- Send a reset request to the BeeStack Consumer layer
- Print some welcome messages on the LCD and over UART

- Initialize the application state machine and send a gAppEvtStateStart_c event to the application task, in order for it to begin the node start procedure when it gains control

Also, the MAC address of the board is set here, by calling NWK_SetMacAddress(). The function takes as a parameter a pointer to an array containing the desired value of the MAC address and writes the value to an internal variable of the MAC task. It is not written to flash.

4.2.2 Application Task Handler

The application task handler is the main entry point for the application code. It is designed to handle events that signal either an external or a change in the state of the state machine. Events may be handled differently depending on the application state.

The arrival of a message from the BeeStack Consumer layer is signaled by the arrival of the gAppEvtMsgFromNlme_c and the gAppEvtMsgFromNlde_c events. If they are present in the ”event_t events” parameter, the messages are extracted from the queues and handled according to the application state.

It is possible that two or more messages arrive on a SAP before the application task is allowed to process the first one by the task handler. Because an event may not be outstanding more than once, several messages may be present in the queue. To handle this possibility, the application task checks whether the queues are empty at the end, and resends the gAppEvtMsgFromNlme_c or gAppEvtMsgFromNlde_c events to itself in order to process them next time it is run:

```
if(MSG_Pending(&mNlmeAppInputQueue))
        TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlme_c);

if(MSG_Pending(&mNldeAppInputQueue))
        TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlde_c);
```

4.3 Starting the Node

The node start procedure involves calling the NLME_StartRequest() function, then waiting for the start confirm message.

```
uint8_t NLME_StartRequest
(
    void
);
```

The NLME_StartRequest() returns gNWSuccess_c if the node start process has been initiated. Otherwise, the return code indicates the error. If NLME_StartRequest() does not return success, a start confirm message will not be sent by the BeeStack Consumer layer.

The start confirm message has the following structure:

```
typedef struct nwkNlmeStartCnf_tag
{
        uint8_t                 status;
}nwkNlmeStartCnf_t;
```

The node start procedure is not complete until the arrival of the start confirm message with a status of gNWSuccess_c.

4.4 Discovery Process

The discovery process takes place on one or more devices. On the originating device, the application initiates a discovery request and than waits for the discovery confirm which will list the devices (if any) that responded to the request. The recipients react to the arrival of a discovery indication message by deciding whether to reply with discovery response.

4.4.1 Discovery Process (Originator Device)

Before starting a discovery request configure the number of discovery trials that the BeeStack Consumer stack should conduct by setting the nwkMaxDiscoveryRepetitions NiB attribute, as well as the repetition interval by setting the nwkDiscoveryRepetitionInterval NiB attribute. The following code sets the number of discovery repetitions to 16 and the repetition interval to 1000 MAC symbols (16ms)

```
tmp8Bit = 16;
status = NLME_SetRequest(
     gNwkNib_ MaxDiscoveryRepetitions_c,
     0,
     (uint8_t*)&tmp8Bit
         );

tmp32Bit = 1000;
status = NLME_SetRequest(
     gNwkNib_ MaxDiscoveryRepetitions_c,
     0,
     (uint8_t*)&tmp8Bit
         );
```

Another thing that needs to be configured is the maximum number of node descriptors in the discovery node descriptor list. Each node descriptor contains information about each unique device that responds to the discovery request. Extra buffer space is needed to contain the node descriptor list. A large message buffer can contain only two node descriptors, so for every two node descriptors another large message buffer must be allocated. The following code example sets a maximum of three node descriptors, which will require two large buffers to hold.

```
tmp8Bit = 3;
status = NLME_SetRequest(
     gNwkNib_MaxReportedNodeDescriptors_c,
     0,
     (uint8_t*)&tmp8Bit
         );
```

To start a discovery process call NLME_DiscoveryRequest(), which has the following prototype:

```
uint8_t NLME_DiscoveryRequest
(
     uint8_t*          recipPanId,
     uint8_t*          recipShortAddress,
     uint8_t           recipDeviceType,
     appCapabilities_t origAppCapabilities,
     uint8_t*          origDeviceTypeList,
     uint8_t*          origProfileIdList,
     uint8_t           discProfileIdListSize,
     uint8_t*          discProfileIdList,
     uint32_t          discDuration
```

```
);
```

Each parameter corresponds to the equivalent parameter described in the BeeStack Consumer specification at the discovery request primitive. For convenience, the originator appcapabilities bitfield is passed in a bitfield structure:

```
typedef struct appCapabilities_tag
{
    uint8_t                 bUserStringSpecified   :1;
    uint8_t                 nrSupportedDeviceTypes :2;
    uint8_t                 reserved1              :1;
    uint8_t                 nrSupportedProfiles    :3;
    uint8_t                 reserved2              :1;
}appCapabilities_t;
```

If the NLME_DiscoveryRequest() call returns gNWSuccess_c the discovery process has been started. The BeeStack Consumer stack will now proceed with constructing an BeeStack Consumer discovery command frame, broadcasting it on all three BeeStack Consumer channels and listening for responses. Only those responses are accepted/recorded in the descriptor list which meet the following conditions:

- At least one device type from the list in the responding frame matches one device type provided by the application in origDeviceTypeList
- At least one profile ID from the list in the responding frame matches one profile ID in the list provide by the application in origProfileIdList

After the discovery request duration expires or sufficient responses have arrived to fill the discovery node descriptor list the BeeStack Consumer layer will send a discovery confirm message to the application indicating the results. The discovery message has the following structure:

```
typedef struct nwkNlmeDiscoveryCnf_tag
{
        uint8_t                                                     status;
        uint8_t                                           nrDiscoveredNodes;
        nodeDescriptorBlock_t*    pNodeDescriptorBlocks;
}nwkNlmeDiscoveryCnf_t;
```

It informs the application of the request status, the number of responding devices and also forwards the list of node descriptors to the application. The node descriptor list is organized in node descriptor blocks, with two node descriptors per block. Each node descriptor block is contained in an allocated large message buffer. The node descriptor block structure is the following:

```
struct nodeDescriptorBlock_tag {
    nodeDescriptor_t      nodeDescriptorList[aNodeDescriptorsPerBlock];
    uint8_t               nodeDescriptorCount;
    struct                nodeDescriptorBlock_tag* pNext;
};
```

It contains an array of node descriptors of size aNodeDescriptorsPerBlock (which is two), the number of node descriptors in the array (which may be less than the maximum in the last block) and a pointer to the next block. This pointer is NULL in the last block.

A node descriptor has the following structure:

```
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;
```

The descriptor list is best parsed in two loops, the outer loop goes over the descriptor blocks while the inner loop iterates over the descriptors inside a block:

```
pNodeDescriptor =
pMsgIn->msgData.nwkNlmeDiscoveryCnf.pNodeDescriptorBlocks;
while(pNodeDescriptor)
{
  for(index = 0; index < nodeDescriptorCount)
  {
             // parse the information in
             // pNodeDescriptor->nodeDescriptorList[index] here
  }
  pNodeDescriptor = pNodeDescriptor->pNext;
}
```

The application is responsible for freeing the node descriptor blocks. This can be done in the following manner:

```
pNodeDescriptor =
pMsgIn->msgData.nwkNlmeDiscoveryCnf.pNodeDescriptorBlocks;
while(pNodeDescriptor)
{
         pOldNodeDescriptor = pNodeDescriptor;
         pNodeDescriptor = pNodeDescriptor->pNext;
         MSG_Free(pOldNodeDescriptor);
}
```

4.4.2 Discovery Process (Recipient)

The discovery process on a recipient device begins with the arrival of a discovery request frame over the air. Upon reception, the BeeStack Consumer stack will construct a discovery indication message and send it to the application. The discovery indication message has the following structure:

```
typedef struct nwkNlmeDiscoveryInd_tag
{
         uint8_t                 status;
         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                 requestedDeviceType;
```

```
        uint8_t                rxLinkQuality;
}nwkNlmeDiscoveryInd_t;
```

Discovery indication messages will be sent to the application only when the following conditions are met:

1. The discovery request frame LQI is higher than the LQI threshold indicated by the nwkDiscoveryLQIThreshold NiB attribute.
2. The application has allowed the sending of discovery indication messages by setting the setting the nwkIndicateDiscoveryRequests NiB attribute to TRUE. Keep in mind that the default value is FALSE according the BeeStack Consumer specification.

Upon receipt of a discovery indication message the application must first decide whether a discovery response should be sent. Discovery requests are normally sent to facilitate a later pairing, send a response only if it is required to pair with the originator device.

To send a response call NLME_DiscoveryResponse(), which has the following prototype:

```
uint8_t NLME_DiscoveryResponse
(
    uint8_t            status,
    uint8_t*           recipMacAddress,
    appCapabilities_t  origAppCapabilities,
    uint8_t*           origDeviceTypeList,
    uint8_t*           origProfileIdList,
    uint8_t            discoveryReqLQI
);
```

The recipMacAddress and discoveryReqLQI parameters should have the same values as the origMacAddress and rxLinkQuality fields, respectively, from the discovery indication message

A discovery response frame will now be constructed and sent to the originating device. The discovery process is complete with the arrival of a comm status indication message with a MAC address equal to that of the originating device, indicating the status of the transmission.

4.4.2.1 Autodiscovery

Autodiscovery allows a node to instruct its BeeStack Consumer layer to respond automatically to incoming discovery requests. To enter autodiscovery mode call NLME_AutoDiscoveryRequest(), which has the following prototype:

```
uint8_t NLME_AutoDiscoveryRequest
(
    appCapabilities_t recipAppCapabilities,
    uint8_t*          recipDeviceTypeList,
    uint8_t*          recipProfileIdList,
    uint32_t          autoDiscDuration
);
```

While in autodiscovery mode the BeeStack Consumer layer will automatically parse incoming discovery request frames. If an incoming discovery request frame has a LQI higher than nwkDiscoveryLQIThreshold, the requested device type matches one of the device types in recipDeviceTypeList and there is at least one common profile in the profile ID list in the discovery request frame and in recipProfileIdList, the BeeStack Consumer layer will wait for a second, identical discovery request frame to arrive. If one arrives, a discovery response frame will be sent automatically and the

application will receive an autodiscovery confirm message with the IEEE address and PAN ID of the discovery request originator and a status of gNWSuccess_c. Otherwise, the autodiscovery message will have a status of gNWTimeout_c. The autodiscovery confirm message has the following structure:

```
typedef struct nwkNlmeAutoDiscoveryCnf_tag
{
        uint8_t                                                         status;
        uint8_t          origMacAddress[8];
        uint8_t          origPanId[2];
}nwkNlmeAutoDiscoveryCnf_t;
```

4.5 Pairing Process

Pairing allows two devices to associate in order to transmit data from one to the other. Pairing can be initiated from either a controller or a target node but the recipient must be a target node. When a controller node wants to pair with a target node, the purpose of this process is for the controller node to obtain a PAN ID and a Short Address from the target node to use in future communications with this target device. When a target node wants to pair with another target node, the purpose of this process is that both target nodes to obtain the PAN ID and the Short Address of the other device.

4.5.1 Pairing Process (Originator Node)

To start a paring process call NLME_PairRequest(), which has the following prototype:

```
uint8_t NLME_PairRequest
(
    uint8_t           recipChannel,
    uint8_t*          recipPanId,
    uint8_t*                                                recipMacAddress,
    appCapabilities_t origAppCapabilities,
    uint8_t*          origDeviceTypeList,
    uint8_t*          origProfileIdList,
    uint8_t           keyExTransferCount
);
```

The recipient's channel, PAN ID and MAC address are normally obtained from a previous discovery process.

A pair request frame is now constructed and sent to the recipient. When the recipient replies with a pair response frame the BeeStack Consumer layer collects the assigned short address from the frame and completes an entry in the pair table for the new device. A pair confirm message is then sent to the application with the pair table index of the new entry, which completes the pairing process. The application should remember the pair table index for future data exchanges with the corresponding device. The pair response message has the following message structure:

```
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;
```

4.5.2 Pairing Process (Recipient Node)

The pairing process on the recipient node begins with the arrival of a pair request frame. The BeeStack Consumer layer constructs a pair indication message and sends it to the application. The pair indication message has the following structure:

```
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;
```

Upon receipt of a pair indication message, the application must first decide whether to accept the pairing. If the pairing is not accepted, a PairResponse with the status set to gNWNotPermitted_c should be issued. If the pairing is accepted, it should call NLME_PairResponse() with the status set to gNWSuccess_c. Also, the application should remember the deviceId provided, as it is the index in the pair table where the new entry corresponding to originating device will reside. NLME_PairResponse() has the following prototype:

```
uint8_t NLME_PairResponse
(
    uint8_t           status,
    uint8_t*          recipPanId,
    uint8_t*                                              recipMacAddress,
    appCapabilities_t origAppCapabilities,
    uint8_t*          origDeviceTypeList,
    uint8_t*          origProfileIdList,
    uint8_t           deviceId
);
```

A short address will now be generated for the originator node and inserted into a pair response frame, which will be sent to the originator. The pairing process is complete with the arrival of a comm status indication message with the targetAddress field set to the MAC address of the originating device, indicating the status of the transmission.

4.5.3 Security

The pairing process will automatically generate a security key for the pairing link if both nodes declare that they support security (the corresponding bit in nwkcNodeCapabilities is set). The keyExTransferCount parameter of the NLME_PairRequest() function will determine the number of key

seed frames exchanged. To disable security for the pairing link even if both nodes support it, set this parameter to zero.

4.5.4 Offline Pairing

An entry can be created into the pair table artificially, without going through the pairing process, provided information about the other device required to complete a pairing table entry is available to the application:

- The default channel the other device listens on
- The MAC address of the other device
- The PAN ID and short address
- The application capabilities
- The user string
- The security key

If this information is available, call NWK_AddNewPairTableEntry() to create the new pair table entry. It has the following prototype:

```
uint8_t NWK_AddNewPairTableEntry
(
    uint8_t*            localShortAddress,
    uint8_t             recipChannel,
    uint8_t*            recipMacAddress,
    uint8_t*            recipPanId,
    uint8_t*            recipShortAddress,
    uint8_t             recipCapabilities,
    uint8_t*            securityKey,
    uint8_t*            recipUserString
);
```

The call returns the pair table index of the new entry, or either `gNWNoOrigCapacity_c` (if the pair table is full) or `gNWInvalidParam_c` (if a parameter is invalid, normally the recipient channel). Two utility functions are available for generating some of the necessary information:

```
void NWK_GenerateShortAddress
(
    uint8_t* shortAddress
);
```

NWK_GenerateShortAddress() takes as a parameter a pointer to a two byte array. A random short address will be generated and written to the array.

```
void NWK_GenerateSecurityKey
(
    uint8_t* securityKey
);
```

NWK_GenerateSecurityKey() takes as a parameter a pointer to a 16 byte array. A random security key will be generated and written to the array. The security key is guaranteed to be different from any other security key in the pair table.

4.6 Data Transmit and Receive

Construct the data request payload according to the chosen application profile, decide upon the transmission options, then call NLDE_DataRequest(), which has the following prototype:

```
uint8_t NLDE_DataRequest
(
    uint8_t   deviceId,
    uint8_t   profileId,
    uint8_t*  vendorId,
    uint8_t   nsduLength,
    uint8_t*  nsdu,
    uint8_t   txOptions
);
```

The BeeStack Consumer stack will not free the pointer. If users have allocated a message buffer for the data request payload they must free it. This can be done immediately after NLDE_DataRequest() returns as the payload has already been copied by the network.

A data request frame will be constructed and transmitted over the air. When transmission is complete, or when the recipient has acknowledged if an acknowledgement was requested, the application will be informed via a data confirm message, which has the following structure:

```
typedef struct nwkNldeDataCnf_tag
{
        uint8_t                                                         status;
        uint8_t                                                       deviceId;
}nwkNldeDataCnf_t;
```

When the data frame arrives at the recipient, a data indication message will be constructed and sent to the recipient application. The data indication message has the following structure:

```
typedef struct nwkNldeDataInd_tag
{
        uint8_t                 deviceId;
        uint8_t                 profileId;
        uint8_t                 vendorId[2];
        uint8_t                 LQI;
        uint8_t                 rxFlags;
        uint8_t*                pData;
        uint8_t                 dataLength;
}nwkNldeDataInd_t;
```

The data request payload is in the same buffer as the data indication message itself. A buffer has not been allocated for it. As such, pData must not be freed.

4.7 ZigBee Remote Control (ZRC) Profile

This section describes the ZigBee Remote Control (ZRC) profile including the currently available functions and how to transmit and receive ZRC commands.

4.7.1 Push Button Pairing

Push-button pairing is a specific type of pairing process that must be used by ZRC compliant applications. It involves a discovery process with auto-discovery on the recipient, immediately followed by pairing.

4.7.1.1 Push Button Pairing Process (Originator)

When doing push-button pairing the discovery process on an originator must be configured in the following way:

- nwkDiscoveryRepetitionInterval — Must be set to 0x00f424 MAC symbols (1 second)
- nwkMaxDiscoveryRepetitions — Must be set to 0x1e
- nwkMaxReportedNodeDescriptors — Must be set to 1

This means that the discovery process ends with the first arriving discovery response. Also, when calling NLME_DiscoveryRequest() some parameters must have specific values: the local profile ID list must have only one element, the value of the ZRC profile ID (0x01). The same holds true for the list of profile identifiers against which the profile ID lists from incoming discovery response frames will be matched (In this example we just provided a pointer to the same list). The discovery duration must be set to 0x00186a (100 ms):

```
localAppCapabilities.nrSupportedProfiles = 1;
discDuration = 0x001861;
localProfilesList[0] = 0x01;
status = NLME_DiscoveryRequest(
            discoverySearchedPanId,
            discoverySearchedShortAddress,
            gNlmeDiscoverySearchedDeviceType_c,
            localAppCapabilities,
            localDeviceTypesList,
            localProfilesList,
            localAppCapabilities.nrSupportedProfiles,
            localProfilesList,
            discDuration
        );
```

When the discovery confirm message arrives, we use the data from the first node descriptor (should be only one) to initiate pairing with the responding device.

```
nodeDescriptorBlock_t* pDiscoveryTable =
pNlmeMsgIn->msgData.nwkNlmeDiscoveryCnf.pNodeDescriptorBlocks;
/* Send pair request to the discovered device */
status = NLME_PairRequest(
            pDiscoveryTable->nodeDescriptorList[0].recipChannel,
            pDiscoveryTable->nodeDescriptorList[0].recipPanId,
            pDiscoveryTable->nodeDescriptorList[0].recipMacAddress,
            localAppCapabilities,
            localDeviceTypesList,
            localProfilesList,
```

```
                gNlmeKeyExTransferCount
            );
```

Pairing is complete when the pair confirm message arrives.

4.7.1.2 Push Button Pairing Process (Recipient)

The push-button pairing process on the recipient begins when the application initiates an auto-discovery process. No special parameters are required.

When the auto-discovery confirm message arrives, the MAC address and PAN ID of the requesting device must be saved, as a pair request will normally follow. They should be compared with the MAC Address and PAN ID of the pair request originator and the pair request rejected in case of mismatch:

```
FLib_MemCpy(
(void*)discoveryOrigPanId,
     (void*)pNlmeMsgIn->msgData.nwkNlmeAutoDiscoveryCnf.origPanId,
     2
           );
FLib_MemCpy((void*)discoveryOrigMacAddress,
     (void*)pNlmeMsgIn->msgData.nwkNlmeAutoDiscoveryCnf.origMacAddress,
     8
           );

if(FLib_MemCmp(
(void*)pNwkNlmePairInd->origPanId,
(void*)discoveryOrigPanId,
2
)
   && FLib_MemCmp(
(void*)pNwkNlmePairIndrigMacAddress,
(void*)discoveryOrigMacAddress,
8
)
  )
{
                status =
          NLME_PairResponse(
             status,
             pNwkNlmePairInd->msgData.nwkNlmePairInd.origPanId[0],
             pNwkNlmePairInd->msgData.nwkNlmePairInd.origMacAddress[0],
             localAppCapabilities,
             localDeviceTypesList,
             localProfilesList,
             pMsgIn->msgData.nwkNlmePairInd.deviceId
          );
}
```

The push-button pairing process is complete with the arrival of the Comm Status indication message informing the application of the result of the transmission of the pair response frame.

In this example we also start a single shot timer at the beginning of the push-button pairing process (before calling NLME_AutoDiscoveryRequest()). The timeout is 30 seconds and the callback aborts the push-button pairing process. This is to ensure the application does not remain blocked, e.g. waiting for the pair indication message.

4.7.2 ZRC Commands

The ZRC profile provides the structure of the data request payload. The payload is a ZRC command frame and is usually two bytes long with the first byte, the frame control, describing how a button on a remote control was pressed (single press, repeated press, etc.) and the second indicating the button that was pressed (the RC command). Some RC commands may have additional payloads.

4.7.2.1 Transmitting ZRC Commands

To transmit a ZRC command construct the ZRC command frame and transmit it using NLDE_DataRequest():

```
uint8_t appCmdPayload[] = “abc”;
uint8_t vendorId[]      = { gDefaultValueOfVendorId_c };
uint8_t nldeDataPayload[2 + sizeof(appCmdPayload)];

/* Build the payload for the NLDE Data request. */
nldeDataPayload[0] = gCmdZRC_UserCtrlPressed_c;
nldeDataPayload[1] = gCmdRC_Up_c;
FLib_MemCpy(
(void*)&nldeDataPayload[2],
     (void*)appCmdPayload,
     gAppCmdPayloadLength_c
           );

UartUtil_Print("\n\rSending data... ", gAllowToBlock_d);

/* Try to send the command using the nwk NLDE Data service */
status = NLDE_DataRequest(
     deviceId,
     0x01,
     vendorId,
     2 + sizeof(appCmdPayload),
     nldeDataPayload,
     gTxOptions_c
           );
```

4.7.2.2 Receiving ZRC commands

Receiving ZRC commands means parsing the payloads of data indication messages:

```
void ZRC_HandleCommandInd(nwkNldeDataInd_t* pNwkNldeDataInd)
{
       /* Handle the profile commands */
       switch(pNwkNldeDataInd->pData[0])
       {
             case gCmdZRC_UserCtrlPressed_c:
         UartUtil_Print("\n\rUser control pressed:", gAllowToBlock_d);
               switch(pNwkNldeDataInd->pData[1])
               {
             case gCmdRC_Select_c:
         UartUtil_Print("'Select'", gAllowToBlock_d);
               break;

             case gCmdRC_Up_c:
         UartUtil_Print("'Up'", gAllowToBlock_d);
```

```
            break;

          case gCmdRC_Down_c:
      UartUtil_Print("'Down'", gAllowToBlock_d);
            break;

          case gCmdRC_Left_c:
      UartUtil_Print("'Left'", gAllowToBlock_d);
            break;

          case gCmdRC_Right_c:
      UartUtil_Print("'Rigth'", gAllowToBlock_d);
            break;


        /* ...
           Application should implement here the code for handling the supported
commands
  ... */


        default:
          UartUtil_PrintHex(&pNwkNldeDataInd->pData[1], 1, gPrtHexSpaces_c);

           }
         break;

          case gCmdZRC_UserCtrlRepetead_c:
       UartUtil_Print("\n\rUser control repetead", gAllowToBlock_d);
            break;

          case gCmdZRC_UserCtrlReleased_c:
       UartUtil_Print("\n\rUser control released", gAllowToBlock_d);
            break;

  }
```

Chapter 5
Debug

This chapter describes how to debug an BeeStack Consumer networking application, including use of the BDM, LEDs, ZigBee Test Client, and Network Packet Analyzers.

5.1 P&E MultiLink BDM

One of the most powerful tools for debugging an BeeStack Consumer 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, keep the following in mind

- 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.

5.2 LEDs and the Display

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

Alternatively, if the board is connected to the PC via a USB cable, users can print data over UART in a hyper terminal window.

5.3 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 any single 802.15.4 channel for activity, and reports each packet through a USB port. Communication problems that are extremely difficult to identify in the code are frequently very easy to understand from the over-the-air behavior.

5.4 ZigBee Test Client

Another method for debugging a network is the Freescale Test Tool and ZigBee Test Client combination. The Freescale Test Tool is a desktop PC tool that 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 BeeStack Consumer 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 the BeeStack Consumer stack itself, with a large test suite covering the BeeStack API and ZigBee commands.

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

Chapter 6
Working With BeeStack Consumer/SynkroRF Hybrid Nodes

This chapter describes how to design and write a custom application over the ZigBee RF4CE (BeeStack Consumer)/SynkroRF hybrid stack. This chapter details how to properly interface with the stack and how to develop an application that can participate in both an BeeStack Consumer network and a SynkroRF network.

6.1 Introducing the Hybrid Stack

On a basic level, working with the BeeStack Consumer/SynkroRF hybrid stack is no different than working with a pure BeeStack Consumer stack. The hybrid node provides additional functionality that allows it to communicate (pair and exchange commands with) other SynkroRF nodes. A BeeStack Consumer/SynkroRF hybrid node does not contain all the SynkroRF functionality of a legacy SynkroRF node. For example, polling, cloning and bulk data transfers are not supported.

6.2 Interfacing with the Hybrid Stack

Interfacing with the BeeStack Consumer/SynkroRF hybrid stack is almost identical to interfacing with the standard BeeStack Consumer stack. There are no new network tasks, no new network layer entities and no new SynkroRF specific NIBs.

6.3 SynkroRF API Function Calls

Several API functions were added to allow for SynkroRF functionality. They work in a similar manner to the BeeStack Consumer API function calls. No new return codes were added.

6.4 Network Layer SAP Handlers and Messages

The following sections outline the network layer SAP handlers and describe the general message structure.

6.4.1 SAP Handlers

The hybrid network stack requires no new SAP handler functions. When the SynkroRF part of the stack needs to communicate with the application, a message is sent through either the NLME SAP or the NLDE SAP.

6.4.2 General Message Structure

The network layer entity message structure of a hybrid stack is enhanced with the addition of new SynkroRF related messages.

6.4.2.1 NLME Message Structure

```
typedef struct nwkNlmeToAppMsg_tag
{
  nwkNlmeToAppMsgType_t           msgType;
  union {
    /*------------------------------------------------------*/
    nwkNlmeStartCnf_t             nwkNlmeStartCnf;
    /*------------------------------------------------------*/
    nwkNlmeAutoDiscoveryCnf_t     nwkNlmeAutoDiscoveryCnf;
    /*------------------------------------------------------*/
    nwkNlmeDiscoveryCnf_t         nwkNlmeDiscoveryCnf;
    nwkNlmeDiscoveryInd_t         nwkNlmeDiscoveryInd;
    /*------------------------------------------------------*/
    nwkNlmePairCnf_t              nwkNlmePairCnf;
    nwkNlmePairInd_t              nwkNlmePairInd;
    /*------------------------------------------------------*/
    nwkNlmeUnpairCnf_t            nwkNlmeUnpairCnf;
    nwkNlmeUnpairInd_t            nwkNlmeUnpairInd;
    /*------------------------------------------------------*/
    nwkNlmeCommStatusInd_t        nwkNlmeCommStatusInd;
    /*------------------------------------------------------*/
    nwkSynkroRFPairCnf_t            nwkSynkroRFPairCnf;
    nwkSynkroRFPairInd_t            nwkSynkroRFPairInd;
    /*------------------------------------------------------*/
  } msgData;
}nwkNlmeToAppMsg_t;
```

Where: nwkNlmeToAppMsgType_t is defined as follows:

```
typedef enum {
  /*---------------------------*/
  gNwkNlmeStartCnf_c = 0,
  /*---------------------------*/
  gNwkNlmeAutoDiscoveryCnf_c,
  /*---------------------------*/
  gNwkNlmeDiscoveryCnf_c,
  gNwkNlmeDiscoveryInd_c,
  /*---------------------------*/
  gNwkNlmePairCnf_c,
  gNwkNlmePairInd_c,
  /*---------------------------*/
  gNwkNlmeUnpairCnf_c,
  gNwkNlmeUnpairInd_c,
  /*---------------------------*/
  gNwkNlmeCommStatusInd_c,
  /*---------------------------*/
  gNwkSynkroRFPairCnf_c = 0xF0,
  gNwkSynkroRFPairInd_c,
  /*---------------------------*/
  gNwkNlmeMax_c
}nwkNlmeToAppMsgType_t;
```

6.4.2.2 NLDE Messages Structure

```
typedef struct nwkNldeToAppMsg_tag
{
  nwkNldeToAppMsgType_t    msgType;
  union
  {
    nwkNldeDataCnf_t            nwkNldeDataCnf;
    nwkNldeDataInd_t            nwkNldeDataInd;
    /*-----------------------------------------------------*/
    nwkSynkroRFCommandCnf_t       nwkSynkroRFCommandCnf;
    nwkSynkroRFCommandInd_t       nwkSynkroRFCommandInd;
    /*-----------------------------------------------------*/
  } msgData;
}nwkNldeToAppMsg_t;
```

Where: nwkNldeToAppMsg_t is defined as follows:

```
typedef enum {
  gNwkNldeDataCnf_c       = 0,
  gNwkNldeDataInd_c,
  /*---------------------------*/
  gNwkSynkroRFCommandCnf_c  = 0xF0,
  gNwkSynkroRFCommandInd_c,
  /*---------------------------*/
  gNwkNldeMax_c
}nwkNldeToAppMsgType_t;
```

Additions to the hybrid stack are in bold font.

6.5 Required Global Variables

A BeeStack Consumer/SynkroRF hybrid network stack requires one additional global variable; the SynkroRF node descriptor. This variable is exchanged during pairing and contains SynkroRF related information about the capabilities of the node, including its SynkroRF device type.

```
const SynkroRFNodeDescriptor_t  SynkroRFNodeDescriptor;
```

6.6 Starting the Node

No special activity is required to start a BeeStack Consumer/SynkroRF hybrid node. The BeeStack Consumer NLME_Start process also initializes the SynkroRF functionality. If a hybrid node is started as a BeeStack Consumer Controller, it will also be a SynkroRF controller. If a hybrid node is started as a BeeStack Consumer target, it will also be a SynkroRF target (controlled device).

6.7 SynkroRF Pairing

Unlike a legacy SynkroRF stack, callbacks are not needed for pairing. Instead, an approach similar to that used in the standard BeeStack Consumer networks is used for the hybrid node.

6.7.1 SynkroRF Pairing on the Controller

The controller is always the device that initiates SynkroRF pairing. To initiate pairing, the application should call the SynkroRF_PairRequest.

Prototype

```
uint8_t SynkroRF_PairRequest(
                            uint8_t     deviceType,
                            uint8_t*    pPairingData,
                            uint8_t     length,
                            uint16_t    timeOut
                          );
```

The parameters for the function call include the SynkroRF device type to search for during pairing, some additional data to include in the SynkroRF pair request frame and a maximum duration for the pairing process. The device type can be 0xFF, which means the node is willing to pair with any type of device. If the application needs to add some additional data to the pair request frame, it should pass a pointer to it in *pPairingData* and its length in the *length* parameter.

The function call initiates the controller side SynkroRF pairing process. The node broadcasts SynkroRF pair request frames on all three SynkroRF channels (15, 20, 25, same as the BeeStack Consumer channels) until either a suitable response is received or the pairing process times out.

When the pairing process is complete (no matter the reason) the application is notified by a SynkroRF pair confirm message received through the NLME SAP.

Message Structure

```
typedef struct nwkSynkroRFPairCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  uint8_t                 length;
  uint8_t*                pPairingData;
}nwkSynkroRFPairCnf_t;
```

The *status* field tells the application how the pairing process has completed. If it is gNWSuccess_c, the pairing process has completed successfully and the *deviceId* field contains the reference into the pair table for the new pairing link. Any other status means pairing has failed. In this case, the *deviceId* field should be ignored. If pairing is successful, any additional pairing data received in the SynkroRF pair response can be accessed through the *pPairingData* pointer. A maximum of 15 bytes of it is also stored in the pair table in the *recipUserString* field.

6.7.2 SynkroRF Pairing on the Target

SynkroRF pairing on the target begins with the arrival of SynkroRF pair request frame. The frame is first filtered. The requested device type must match the device type in the local SynkroRF node descriptor (unless the requested device type is 0xFF). Also, the LQI of the frame must be greater than the value of the discoveryLQIThreshold NIB. The BeeStack Consumer discovery LQI threshold acts as the SynkroRF pairing threshold.

If the frame passes filtering, the application receives a SynkroRF pair indication message.

Message Structure

```
typedef struct nwkSynkroRFPairInd_tag
{
  uint8_t                   status;
  uint8_t                   deviceId;
  uint8_t                   LQI;
  uint8_t                   length;
  uint8_t*                  pPairingData;
  SynkroRFNodeDescriptor_t*   pNodeDescriptor;
}nwkSynkroRFPairInd_t;
```

The *status* field can have three values:

gNWSuccess_c If pairing can be accepted.

gNWDuplicatePairing_c If the requesting device is already in the pair table.

gNWNoRecipCapacity_c If the pair table is full.

The deviceId points to the location of the provisional (or actual, in the case of duplicate pairing) pair table entry of the requesting device. If the pair table is full, the value is 0xFF. The LQI has the link quality indicator of the pair request frame. The requesters node descriptor can be inspected through the *pNodeDescriptor* field. And additional pairing data present in the pair request frame can be accessed through the *pPairingData* pointer.

Based on the information in the pair indication message, the application should decide whether to accept pairing or not and inform the network layer by calling the SynkroRF_PairResponse.

Prototype

```
uint8_t SynkroRF_PairResponse(
                             uint8_t          deviceId,
                             uint8_t          status,
                             uint8_t          length,
                             uint8_t*         pPairingData
                           );
```

The *deviceId* parameter must be the same as the *deviceId* field in the pair indication message. The *status* parameter can be either gNWSuccess_c if the application accepts pairing or gNWNotPermitted_c if the application rejects pairing. Pairing cannot be accepted when the pair table is full. If pairing is rejected in the case of a duplicate pairing, the old pairing link is also deleted. If the application needs to add some additional data to the pair response frame it should pass a pointer to it in *pPairingData* and its length in the *length* parameter.

Once the pairing process is complete, the application receives a NLME_CommStatus indication message informing it of the outcome.

6.8 SynkroRF Command Transmit and Receive

To transmit a SynkroRF command to a SynkroRF device in the pair table, the application should call the SynkroRF_SendCommand.

Prototype

```
uint8_t SynkroRF_SendCommand(
                            uint8_t             deviceId,
                            uint16_t            cmdId,
                            uint8_t*            pData,
                            uint8_t             length,
                            bool_t              bAckRequired
                          );
```

SynkroRF_SendCommand has the following parameters:

deviceId	The pair table entry index of the command recipient.
cmdId	The ID of the SynkroRF command to be transmitted.
pData, length	Any payload and it’s length, that may be included in the command.
bAckRequired	Whether or not to request a MAC layer acknowledgement.

As with the NLDE_DataRequest, the network layer will not free the pData pointer.

A SynkroRF command frame is constructed and transmitted over the air. When transmission is complete, or when the recipient has acknowledged (if an acknowledgement was requested) the application is informed through a SynkroRF command confirm message.

Message Structure

```
typedef struct nwkSynkroRFCommandCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
} nwkSynkroRFCommandCnf_t;
```

When the SynkroRF command frame arrives at the recipient, a SynkroRF command indication message is constructed and sent to the recipient application.

Message Structure

```
typedef struct nwkSynkroRFCommandInd_tag
{
  uint8_t                 deviceId;
  uint16_t                cmdId;
  uint8_t                 LQI;
  uint8_t                 dataLength;
  uint8_t*                pData;
}nwkSynkroRFCommandInd_t;
```

The command payload is in the same buffer as the data indication message itself. A buffer has not been allocated for it so pData must not be freed.