BeeStack Consumer Private Profile

User's Guide

Document Number: BSCONPPUG
Rev. 1.1
06/2011

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Contents

Chapter 6
Over The Air Programming

About This Book

This user's guide provides an overview of the Freescale BeeStack Consumer vendor specific profile. This document details its features and how an overlying application can access those features.

Audience

This document is intended for software developers writing applications based on the Freescale BeeStack Consumer protocol stack with intent to use the Freescale vendor specific profile to simplify application design.

Organization

This document contains the following chapters:

Chapter 1	Provides overview of the Freescale vendor specific profile and describes how an application can interface with it.
Chapter 2	Describes the Fragmented Tx/Rx functionality.
Chapter 3	Describes the polling functionality.
Chapter 4	Describes the remote pairing functionality.
Chapter 5	Describes the over the air menus functionality.
Chapter 6	Describes the over the air programmer functionality.

Revision History

The following table summarizes revisions to this manual since the previous release (Rev. 1.0).

Revision History

Doc. Version	Date / Author	Description / Location of Changes
1.1	May 2011, Dev Team	Changes for CodeWarrior 10.

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

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

Chapter 1
Freescale Vendor Specific Profile Overview

To aid in the development of applications based on the Freescale BeeStack Consumer software, Freescale has created a profile which implements functionality useful for automating specific tasks. The profile resides in the protocol stack between the BeeStack Consumer layer and the application layer. The application can still access the network layer directly.

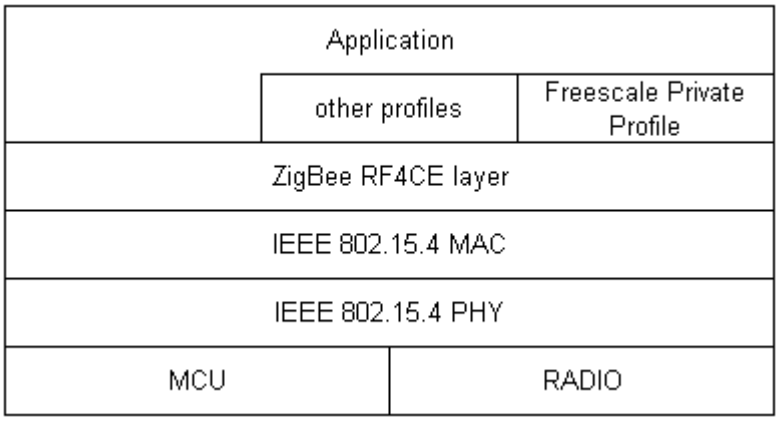

Figure 1-1. Freescale Private Profile Application Structure

1.1 Private Profile Overview

The Private Profile relies completely on the underlying network layer to perform its tasks. The profile layers of different nodes communicate with each other through the NLDE (data is sent by issuing the NLDE_DataRequest primitive with vendor specific data (utilizing Freescale's vendor Id and profile Id)). Communication is encrypted if the pairing link between the nodes is secured. The NLDE SAP must be configured to redirect messages intended for the profile layer to the profile SAP, so that these do not erroneously reach the application. The next section describes an example of how to do this.

NOTE

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

The Private Profile implements the following:

- Fragmented Transmission — This allows the application to easily transmit more data than can fit in the payload of a NLDE data request. The profile fragments the data into pieces small enough for NLDE data request and transmits them one by one to the recipient. The fragmented transmission functionality on the recipient reassembles the data and presents it to the application as a whole.

- Remote Pairing — Allows the application to request that a pairing link be created between two nodes in its pair table. (For example, a remote can request that a Video Player and a TV, both already paired with the remote, pair with each other.)
- Polling — Allows the application to periodically poll specific devices from its pair table for any data they might have to send back.
- Over the air menus — Allows the application to send menu content information to a remote device. The remote device can navigate (browse) and/or display the menu information for the device sending menu information. Intended to provide remote user interface capabilities to remote devices.

1.1.1 Interfacing with the Profile Layer

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

The profile layer runs its own task. To send messages to the application, the profile layer calls a SAP handler, much like the network layer. The SAP handler must be implemented by the application. It is a callback function which has one parameter, a pointer to the incoming message. In the Freescale sample applications using the private profile, the SAP handler simply adds the message from the profile to the profile messages queue and sends an event to the application main task:

```
void FSLProfile_App_SapHandler(fslProfileToAppMsg_t* fslProfileToAppMsg)
{
  MSG_Queue(&mFSLProfileAppInputQueue, fslProfileToAppMsg);
  TS_SendEvent(gAppTaskID, gAppEvtMsgFromFSLProfile_c);
}
```

The messages that the profile layer sends to the application have the following structure.

1.1.1.1 Message Structure

```
/* General structure of a message received by the application over FSL Profile SAP */
typedef struct fslProfileToAppMsg_tag
{
  fslProfileToAppMsgType_t                  msgType;
  union {
    /*-------------------------------------------------------*/
    fslProfileFragCnf_t                     fslProfileFragCnf;
    fslProfileFragStartInd_t                fslProfileFragStartInd;
    fslProfileFragInd_t                     fslProfileFragInd;
    /*-------------------------------------------------------*/
    fslProfilePollCnf_t                     fslProfilePollCnf;
    fslProfilePollEvent_t                   fslProfilePollEvent;
    fslProfilePollInd_t                     fslProfilePollInd;
    /*-------------------------------------------------------*/
    fslProfileRmtPairCnf_t                  fslProfileRmtPairCnf;
    fslProfileRmtPairInd_t                  fslProfileRmtPairInd;
    fslProfileRmtPairRspCnf_t               fslProfileRmtPairRspCnf;
```

```
    /*-------------------------------------------------------*/
    fslProfileMenuBrowseCnf_t               fslProfileMenuBrowseCnf;
    fslProfileMenuBrowseCompleteInd_t       fslProfileMenuBrowseCompleteInd;
    fslProfileMenuBrowseInd_t               fslProfileMenuBrowseInd;
    /*-------------------------------------------------------*/
    fslProfileDisplayMenuCnf_t              fslProfileDisplayMenuCnf;
    fslProfileDisplayMenuHeaderInd_t        fslProfileDisplayMenuHeaderInd;
    fslProfileDisplayMenuEntryInd_t         fslProfileDisplayMenuEntryInd;
    fslProfileDisplayMenuCompleteInd_t      fslProfileDisplayMenuCompleteInd;
    fslProfileDisplayMenuMessageInd_t       fslProfileDisplayMenuMessageInd;
    fslProfileDisplayMenuExitInd_t          fslProfileDisplayMenuExitInd;
  } msgData;
}fslProfileToAppMsg_t;
```

where fslProfileToAppMsgType_t is typedef for:

```
typedef enum {
  /*---------------------------*/
  /* Messages to be received on the fragTx orig device */
  gFSLProfileFragCnf_c = 0,
  /* Messages to be received on the fragTx recip device */
  gFSLProfileFragStartInd_c,
  gFSLProfileFragInd_c,
  /*---------------------------*/
  /* Messages to be received on the poll orig device */
  gFSLProfilePollCnf_c,
  gFSLProfilePollEvent_c,
  /* Messages to be received on the poll recip device */
  gFSLProfilePollInd_c,
  /*---------------------------*/
  /* Messages to be received on the remote pair orig device */
  gFSLProfileRmtPairCnf_c,
  /* Messages to be received on the remote pair recip device */
  gFSLProfileRmtPairInd_c,
  gFSLProfileRmtPairRspCnf_c,
  /*---------------------------*/
  /* Messages to be received on the menu browser device */
  gFSLProfileMenuBrowseCnf_c,
  gFSLProfileMenuBrowseCompleteInd_c,
  /* Messages to be received on the menu onwer device */
  gFSLProfileMenuBrowseInd_c,
  gFSLProfileDisplayMenuCnf_c,
  /* Messages to be received on the menu displayer device */
  gFSLProfileDisplayMenuHeaderInd_c,
  gFSLProfileDisplayMenuEntryInd_c,
  gFSLProfileDisplayMenuCompleteInd_c,
  gFSLProfileDisplayMenuMessageInd_c,
  gFSLProfileDisplayMenuExitInd_c,
  /*---------------------------*/
  gFSLProfileMax_c
}fslProfileToAppMsgType_t;
```

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

To intercept NLDE data indication and data confirm messages intended to the profile layer, the NLDE SAP handler the Freescale sample applications the SAP handler were updated as follows:

```
void NWK_NLDE_SapHandler(nwkNldeToAppMsg_t* nwkNldeToAppMsg)
{
  if(App_MessageIsForFSLProfile(nwkNldeToAppMsg))
     FSLProfile_HandleNwkNldeMsg(nwkNldeToAppMsg);
  else
  {
    MSG_Queue(&mNldeAppInputQueue, nwkNldeToAppMsg);
    TS_SendEvent(gAppTaskID, gAppEvtMsgFromNlde_c);
  }
}
```

The new SAP handler first checks whether the message must be delivered to the profile layer and calls the profile message handler if that is the case. Otherwise the message is added to the NLDE queue and an event is sent to the application task.

The function that determines the message destination, App_MessageIsForFSLProfile, simply checks whether the data is vendor specific, and was sent using Freescale's vendor Id and the Freescale profile Id:

```
static bool_t App_MessageIsForFSLProfile(nwkNldeToAppMsg_t* nwkNldeToAppMsg)
{
  uint8_t maskRxOptionsVendorSpecificData = (1 << 2);
  if(nwkNldeToAppMsg->msgType == gNwkNldeDataInd_c)
  {
    uint8_t vendorId[2] = {gFSLVendorId_c};
    if( (nwkNldeToAppMsg->msgData.nwkNldeDataInd.profileId == gFSLProfileId_c)
        &&
        (nwkNldeToAppMsg->msgData.nwkNldeDataInd.rxFlags &
              maskRxOptionsVendorSpecificData)
        &&
       (FLib_MemCmp(nwkNldeToAppMsg->msgData.nwkNldeDataInd.vendorId,
           vendorId, 2))
      )
    {
      return TRUE;
    }
  }
else if(nwkNldeToAppMsg->msgData.nwkNldeDataCnf.profileId ==
           gFSLProfileId_c)
    return TRUE;

  return FALSE;
}
```

To benefit from a particular functionality of the Freescale Private Profile, the application must first link to the relevant library (a list of the Freescale Private Profile libraries is located in the Freescale *BeeStack Consumer Private Profile Reference Manual*) in addition to the profile framework library. Each function has an initialization function which must be called once at application startup. The best place to call the initialization function is in the main application initialization function:

```
void App_Init(void)
{
...
  /* Init FSL profile procedures */
  FSLProfile_InitFragTxOrigProcedure();
  FSLProfile_InitFragTxRecipProcedure();
  FSLProfile_InitPollOrigProcedure();
  FSLProfile_InitRmtPairOrigProcedure();
...
}
```

This example uses the remote pair originator functionality, allowing it to request the remote pairing of two devices. It is also capable of polling other devices for data, as it has enabled the poll originator functionality. It can both transmit and receive fragmented data because it has enabled both the originator and the recipient fragmented transmission functionality.

NOTE

The profile framework and the functionality it provides does not need to be initialized. It is always active.

1.1.2 Feature Discovery

A node in the pair table can be interrogated by the profile layer, to find out which Freescale Private Profile features it supports. To begin feature discovery, the application must call FSLProfile_GetSupportedFeatures

1.1.2.1 Prototype

```
uint8_t FSLProfile_GetSupportedFeatures(
                            uint8_t deviceId,
                            bool_t bUseSecurity
                  );
```

The application must provide the device ID of the node to be interrogated and must indicate whether to send the interrogation command encrypted or not.

As with the network layer requests, if an error which can be reported immediately is encountered, the function return value contains this error code. In this case, the process is aborted and no further confirm messages will be arriving.

A return value of *gNWSuccess_c* indicates that the interrogation process has begun. The profile layer will send an interrogation command to the target node, which should reply automatically if it supports any Freescale Private Profile features at all. When the response from the target node arrives the application will be informed via a Get Supported Features Confirm message.

1.1.2.2 Message Structure

```
typedef struct fslProfileGetSupportedFeaturesCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  uint8_t                 supportedFeaturesMap[4];
}fslProfileGetSupportedFeaturesCnf_t;
```

It has the following fields:

- status – the interrogation status
- deviceId – the device ID of the node that was interrogated
- supportedFeaturesMap – the map of supported features

If the interrogation was successful (i.e. status is *gNWSuccess_c*), the supported features map lists all the features that the target supports. The supported features map is a bit map with each set bit indicated a supported and initialized feature and each cleared bit indicating a not supported feature. Bit numbering starts with the right-most, least significant bit (which is bit 0). The following table shows the correspondence between the bits and the supported features:

Table 1-1. 5 Bit index - supported features correspondence

Bit index	Feature
0	Fragmented transmission
1	Fragmented reception
2	Poll originator
3	Poll recipient
4	Remote pair originator
5	Remote pair recipient
6	OTA menu browser
7	OTA menu owner
8	OTA menu displayer
9 – 31	Reserved

NOTE

No Freescale Profile process performs any feature interrogation on its own. For example, when initiating a fragmented transmission the originator node does not ask the recipient whether it supports fragmented reception. To ensure that the recipient node(s) support(s) the desired features the application must request the interrogation itself.

Chapter 2
Fragmented Transmission

The fragmented transmission feature allows an application to request the transmission of more data that can fit in the payload of a single NLDE Data request frame.

2.1 Configuration

To initialization functions for the fragmented transmission functionality are:

- FSLProfile_InitFragTxOrigProcedure – for the originator functionality
- FSLProfile_InitFragTxRecipProcedure – for the recipient functionality

The buffer used for intermediate storage of fragments (either received, or waiting to be transmitted) must be defined by the application. There are two variables used to define this: the actual buffer and its size in bytes:

```
CONST uint16_t fragTxRxBufferLength_c = gFSLProfileFragTxRxBufferLength_c;
uint8_t fragTxRxDataBuffer[gFSLProfileFragTxRxBufferLength_c];
```

The profile library (with either fragmented transmission or fragmented reception functionality enabled) expects these external variables at link time.

BeeKit configures these automatically, by setting an appropriate value to gFSLProfileFragTxRxBufferLength_c.

2.1.1 State of the Buffer

The buffer for fragmented transmission has an associated state variable that indicates who is using it at that particular time. The buffer can be in one of the following states:

```
/* FragTxRx buffer states */
typedef enum
{
  gFragTxRxBufferFree_c,
  gFragTxRxBufferBusyApp_c,
  gFragTxRxBufferBusyProfile_c
} fslProfileFragTxRxBufferState_t;
```

The states have the following significance:

- *gFragTxRxBufferFree_c* - The buffer is free/available
- *gFragTxRxBufferBusyApp_c* - The buffer is reserved for the application. No fragmented reception can begin when the buffer is in this state (fragmented transmission requests are discarded by the

profile layer). When the profile layer receives the last fragment of a transmission, it sets the buffer state to *gFragTxRxBufferBusyApp_c* and notifies the application that a fragmented reception has been completed and the received data is in the buffer. Also, before requesting a fragmented transmission the application should set the buffer state to *gFragTxRxBufferBufferBusyApp_c*, write the data to be transmitted in the buffer, and then pass the control to the profile layer.

- *gFragTxRxBufferBusyProfile_c* – The buffer is reserved by the profile (when profile is busy either sending or receiving fragmented data, the buffer state is set to *gFragTxRxBufferBusyProfile_c*). The application should neither read from, nor write to the buffer when it is in this state.

To find out the current state of the buffer, call

```
profileFragTxRxBufferState_t FSLProfile_GetFragTxRxBufferStateRequest(void);
```

The function call returns the current state of the buffer.

The state of the buffer can be modified by the application by calling

```
uint8_t FSLProfile_SetFragTxRxBufferStateRequest(
         profileFragTxRxBufferState_t newBufferState);
```

The function takes as its only parameter the new state of the buffer.

If the buffer is in use by the profile layer its state will be *gFragTxRxBufferBusyProfile_c*. Under these conditions new requests for fragmented transmission will fail. The application is not allowed to change the buffer state either to, or from this state.

Incoming requests for fragmented reception will be ignored if the state of the buffer is anything other than *gFragTxRxBufferFree_c*.

2.1.2 Data Transmission

The application initiates data transmission by calling FSLProfile_FragTxRequest.

2.1.2.1 Prototype

```
uint8_t FSLProfile_FragTxRequest(
                                          uint8_t  deviceId,
                                          uint16_t dataLen,
                                          uint8_t* pData,
                                          bool_t           bUseSecurity
                          );
```

The parameters are as follows:

- deviceId – the recipients pair table entry index
- dataLen – the amount of data to be transmitted
- pData – a pointer to the data to be transmitted; the data itself must not be modified by the application until the transmission process is complete

- bUseSecurity – whether to use secured transmission or not. This setting will apply to all over the air frame transmissions required to transfer the complete data from originator to recipient.

As with the network layer requests, if an error which can be reported immediately is encountered, the function return value contains this error code. In this case, the process is aborted and no further confirm messages will arrive.

A return value of *gNWSuccess_c* indicates that the transmission process has started. The state of the buffer is changed to *gFragTxRxBufferBusyProfile_c* by the profile layer.

When the transmission process is complete (whether successful or not) the application will be notified via a fragmented transmission confirm message:

```
typedef struct fslProfileFragCnf_tag
{
  uint8_t                 status;
  uint16_t                fragRxMaxAcceptedLen;
}fslProfileFragCnf_t;
```

It has the following fields:

- status – the status of the transmission, which can be either *gNWSuccess_c* or it describes the error (for a detailed description of all statuses check the Freescale Private Profile Reference Manual)
- fragRxMaxAcceptedLen – the maximum amount of data the recipient can accept in a single transmission (i.e. the size of its buffer). This is useful if status is *gNWNoRecipCapacity_c*, as the originator has requested more data be transmitted than the recipient can accept. If status is *gNWSuccess_c* the application can increase the amount of data transmitted to the recipient up to this amount. However if the status is anything other than either *gNWSuccess_c* or *gNWNoRecipCapacity_c*, this field should be ignored.

While transmission is in progress the receiver is enabled (power saving is disabled) by the profile layer. The initial receiver active period is saved in an internal variable and restored when transmission is complete. The application should not change the state of the receiver during this time period as the transmission may fail.

The demo applications provided by BeeKit shows a typical way of how to handle the fragmented transmission process:

```
if( (events & gAppEvtStateStart_c)  &&
   (appStateMachine.subState == gAppSubStateStart_c))
  {
    UartUtil_Print("\n\rSending FragTx data... ", gAllowToBlock_d);

    /* Try to send the command using the FSL Profile FragTx service */
    status = FSLProfile_FragTxRequest(
                              appStateMachine.deviceId,
                              gAppFragTxPayloadLength_c,
                              fragTxDataPayload,
                              TRUE
                             );
    /* Exit the state if this is not successful, otherwise wait confirm */
    if(gNWSuccess_c == status)
```

```
    {
      appStateMachine.subState = gAppSubStateWaitCnf_c;
    }
    else
    {
      appStateMachine.subState = gAppSubStateEnd_c;
    }
  }

  switch(appStateMachine.subState)
  {
    case gAppSubStateWaitCnf_c:
      if((events & gAppEvtMsgFromFSLProfile_c) &&
         (pMsgIn != NULL))
      {
        fslProfileToAppMsg_t* pFSLProfileMsgIn =
                  (fslProfileToAppMsg_t*)pMsgIn;
        /* Data confirm received */
        if(pFSLProfileMsgIn->msgType == gFSLProfileFragCnf_c)
        {
          status = pFSLProfileMsgIn->msgData.fslProfileFragCnf.status;
          appStateMachine.subState = gAppSubStateEnd_c;
        }
      }
      break;
    default:
      break;
  }

  if(appStateMachine.subState == gAppSubStateEnd_c)
  {
    /* Print the status */
    App_PrintResult(status);
    /* Send event to end the state */
    TS_SendEvent(gAppTaskID, gAppEvtStateEnd_c);
}
```

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

In the start sub-state (*gAppSubStateStart_c*) the transmission request is sent by calling FSLProfile_FragTxRequest. A fragmented transmission confirm message will only arrive if FSLProfile_FragTxRequest returns *gNWSuccess_c*, so the switch to sub-state gAppSubStateWaitCnf_c only in that case, otherwise the switch goes directly to the final sub-state (gAppSubStateEnd_c), which performs cleanup.

NOTE

Before writing to the buffer the application must first check the buffer state (if it is *gFragTxRxBufferBusyProfile_c* the buffer is in use and its contents should not be modified). Then the buffer state should be set to *gFragTxRxBufferBusyApp_c*. This is not absolutely necessary, but highly recommended, as in this state, the buffer cannot not be overwritten by incoming fragmented transmissions. It is now safe to fill the buffer with the needed data and then call FSLProfile_FragTxRequest.

2.1.3 Data Reception

Data reception can only occur when the fragmented transmission buffer is free (i.e. in the *gProfileBufferFree_c* state). In any other state, incoming request frames for a new fragmented transmission are discarded by the profile layer. To accept incoming fragmented transmissions set the buffer state to *gProfileBufferFree_c*. To reject incoming fragmented transmissions set the buffer state to *gProfileBufferBusyAppW_c*.

At the beginning of reception the application will receive a Fragmented Transmission Start indication message.

2.1.3.1 Message Structure

```
typedef struct fslProfileFragStartInd_tag
{
  uint8_t                 deviceId;
  uint16_t                fragDataLen;
}fslProfileFragStartInd_t;
```

It indicates the originator of the fragmented transmission and how much data will arrive in total.

At this point, the state of the buffer has changed to *gFragTxRxBufferBusyProfile_c*.

After all fragments have been received the application will receive a Fragmentation Indication message.

2.1.3.2 Message Structure

```
typedef struct fslProfileFragInd_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  bool_t                  bFragSecured;
  uint16_t                fragDataLen;
}fslProfileFragInd_t;
```

The message fields have the following significance:

- status – the status of the transmission (either *gNWSuccess_c* or the error code)
- deviceId – the originator of the transmission

- bFragSecured – whether the fragmented transmission was encrypted or not
- fragDataLen – the total amount of data received. If status is *gNWSuccess_c* this must match the value of fragDataLen from the Fragmented Transmission Start Indication. Otherwise this field should be ignored.

The received data can be found in the fragmented transmission buffer. The state of the buffer is now *gFragTxRxBufferBusyApp_c*. Further incoming requests for fragmented reception from other nodes will be ignored. In order to re-enable reception, once the application has finished processing the received data it should set the buffer state to *gFragTxRxBufferFree_c*.

While reception is in progress the receiver is enabled (power saving is disabled) by calling NLME_RxEnableRequest(0x00FFFFFF). The initial receiver active period is saved in an internal variable and restored when reception is complete. The application must not change the state of the receiver during this time period as the reception may fail.

Chapter 3
Polling

Polling allows a node to periodically ask specific devices in its pair table whether they have any information to transmit back. The polling mechanism does not transmit any actual application data, it merely facilitates the transmission by informing an application with data to send (the poll recipient) about the precise moment when the intended destination (the poll originator) is able to receive the data.

The profile layer in the poll originator maintains a list of devices it must poll. Periodically it sends a poll request frame to each of those devices, asking them for any data that they need to send. If any of those devices replies that it has data available, the profile layer enables the receiver for a configurable amount of time, waiting for data.

A recipient application informs its profile layer when it has any data to send. When the profile layer receives a poll request frame, it sends an automatic response back to the originator and informs the application that now is good moment to send data as the originator has its receiver enabled.

3.1 Configuration

The initialization functions for the polling functionality are:

- FSLProfile_InitPollOrigProcedure – for the originator functionality
- FSLProfile_InitPollRecipProcedure – for the recipient functionality

The poll originator needs to configure the polling frequency (the polling period) and the amount of time the receiver stays on while waiting for data. To do that, call:

```
uint8_t FSLProfile_PollConfigRequest(
                    uint32_t pollInterval,
                    uint16_t rxOnInterval
                  );
```

Its parameters are:

- pollInterval – the polling period in milliseconds. This is the interval in between polling attempts.
- rxOnInterval – how long to keep the receiver enabled (in milliseconds) after a recipient has replied that it has data available; can be set to zero in which case the poll reply itself is the data.

3.1.1 Starting the Polling Mechanism

The polling mechanism on the originator periodically polls all devices in its polling list for data. To add and remove devices from that list, the originator application should use FSLProfile_PollRequest().

3.1.1.1 Prototype

```
uint8_t FSLProfile_PollRequest(uint8_t deviceId, bool_t bPollEnable);
```

Its parameters are:

- deviceId – the index of the pair table entry of the device to add to or remove from the polling list
- bPollEnable – whether to add (TRUE) or remove(FALSE) the device from the polling list

The polling automatically stops when all devices have been removed from the polling list and starts again with the first device added to the list.

When polling is active, the profile layer will periodically conduct polling attempts (the period between polling attempts is configured by calling FSLProfile_PollConfigRequest with the desired value for the pollInterval). A polling attempt consists of sending a poll request frame to all devices in the polling list once and keeping the receiver enabled for each device which responds that it has data available.

Whenever the profile layer discovers that the pair table entry of a device in the polling list has become empty (i.e. unpairing has occurred) the device is simply removed from the polling list. This means that the application need not notify the profile layer when it unpairs a device. If the polling list becomes empty because of such a removal the profile layer will send a poll confirm message to the application.

The poll confirm message has the following structure:

3.1.1.2 Message Structure

```
typedef struct fslProfilePollCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
} fslProfilePollCnf_t;
```

In this case the status will be *gNWAborted_c* and the deviceId will be 0xFF.

This is one of the only two situations when a poll confirm message is sent to the application. The other situation is when no memory buffer could be allocated to construct the poll request frame. In this case, the status of the poll confirm message will be *gNWNoMemory_c* and the deviceId will identify the device the profile layer just tried to poll. Polling is not aborted, it continues with the next device in the polling list.

Polling is never aborted while there are devices in the polling list. To stop polling, call FSLProfile_PollRequest with a deviceId of 0xFF and bPollEnable set to *FALSE*.

NOTE

The polling mechanism disables power saving during polling attempt and restores it after it is completed. The application receives no indication when a polling attempt is in progress, so it is prudent not to change power saving parameters when polling is enabled.

3.1.2 Sending Data

Whenever the application of a poll recipient has data to send to a poll originator, it should inform the profile layer by calling FSLProfile_PollDataAvailable() with bDataAvailable set to TRUE.

3.1.2.1 Prototype

```
uint8_t FSLProfile_PollDataAvailable(uint8_t deviceId, bool_t bDataAvailable);
```

It has the following parameters:

- deviceId – the pair table index of the intended destination for the data (the poll originator)
- bDataAvailable – whether data is available or not

The profile layer of a poll recipient maintains its own list of which poll originators it has data available for. FSLProfile_PollDataAvailable is used to add or remove devices to that list. Whenever a poll request frame arrives from a device in that list, the automatic poll response indicates that data is available and the application receives a poll indication message informing it that it has a transmission window. The poll indication message has the following structure:

3.1.2.2 Message Structure

```
typedef struct fslProfilePollInd_tag
{
  uint8_t                 deviceId;
} fslProfilePollInd_t;
```

Its only member informs about the device which has just sent a poll request.

If the polling device is not in the list, the automatic response indicates that no data is available and the application does not receive a poll indication message.

NOTE

Once a device has been marked by the application that there is data available for it, the profile layer will continuously reply to polls from that device that data is available. Actual data transmission does not modify that status. If data is no longer available for that particular device, the application must call FSLProfile_PollDataAvailable() with bDataAvailable set to FALSE in order to remove the device from the list.

3.1.3 Transmitting a Single Bit of Information

When a single bit of information needs to be sent (e.g. a status update about an on/off switch) the poll response frame itself can be used to convey that information. On the poll recipient side, the application should set the bDataAvailable flag according to the value of the bit. In essence, bDataAvailable carries the information.

On the poll originator side, polling should be configured with rxOnInterval set to 0. This is a special case for the profile layer. When a poll response frame arrives and rxOnInterval is 0, a special poll event message is sent to the application. The poll event message has the following structure:

3.1.3.1 Message Structure

```
typedef struct fslProfilePollEvent_tag
{
  uint8_t                deviceId;
  bool_t                 bDataAvailable;
} fslProfilePollEvent_t;
```

It has the following fields:

- deviceId – the pair table entry index of the device responding to the poll
- bDataAvailable – whether data is available or not

Chapter 4
Remote Pairing

Remote pairing allows an application to pair two devices in its pair table (for example a remote that is paired with a TV and a DVD can pair the TV with the DVD). Both devices being paired must accept the pairing for the process to be successful. If both devices can handle security, a security key is automatically generated for the link. The secured link is tested with a standard RF4CE ping frame. At least one of the devices being remotely paired must be a target device (i.e., two controllers can not be paired)

There are three participants in this exchange:

- One device initiating the remote pairing (remote pair originator). In our examples, this will be a remote control.
- Two devices being paired (remote pair recipients)

All three devices must include support for the Freescale private profile. The remote pair originator must include the RF4CE_FSLProfile_RmtPairOrig library while the recipients must include the RF4CE_FSLProfile_RmtPairRecip library.

4.1 Configuration

The initialization functions for the remote pairing functionality are:

- FSLProfile_InitRmtPairOrigProcedure for the originator functionality
- FSLProfile_InitRmtPairRecipProcedure for the recipient functionality

Apart from initialization no other configuration needs to be done.

4.1.1 Initiating the Remote Pairing

To initiate remote pairing the originator application must call FSLProfile_RmtPairRequest().

4.1.1.1 Prototype

```
uint8_t FSLProfile_RmtPairRequest(
                                  uint16_t          appRecipRspTimeOut,
                                  uint8_t           deviceId1,
                                  appCapabilities_t dev1AppCapabilities,
                                  uint8_t*          pDev1DeviceTypeList,
                                  uint8_t*          pDev1ProfileIdList,
                                  uint8_t           deviceId2,
                                  appCapabilities_t dev2AppCapabilities,
                                  uint8_t*          pDev2DeviceTypeList,
                                  uint8_t*          pDev2ProfileIdList
                                 );
```

The parameters are as follows:

- appRecipRspTimeOut – how long to wait for response from each remote pair recipient
- deviceId1 – the device ID of the first remote pair recipient
- dev1AppCapabilities – the application capabilities of the first remote pair recipient
- pDev1DeviceTypeList – the device types supported by the first remote pair recipient
- pDev1ProfileIdList – the profiles supported by the first remote pair recipient
- deviceId2 – the device ID of the second remote pair recipient
- dev2AppCapabilities – the application capabilities of the second remote pair recipient
- pDev2DeviceTypeList – the device types supported by the second remote pair recipient
- pDev2ProfileIdList – the profiles supported by the second remote pair recipient

If all parameter verification is successful, the function returns *gNWSuccess_c*. At this point the application on the originator should wait for the remote pairing process to complete.

NOTE

The parameters passed by reference (as pointers) (e.g. the device type lists and the profile ID lists) to the profile layer must not be changed until the remote pairing process is complete.

The profile layer now communicates with the two devices that are to be paired, informing them of the remote pairing request and transmitting to each of them the other device's complete pair table entry, including a new unique security key for the pairing link (if security is supported by both devices being paired and the remote pair originator). The pairing link is tested by the two devices by exchanging ping frames. If one of the devices is a controller, the target device (there must be at least one) will also allocate a short address for it. Once the new pairing link is tested the remote pairing process is complete. The originator application will be notified of this via a remote pair confirm message, which has the following structure:

4.1.1.2 Message Structure

```
typedef struct fslProfileRmtPairCnf_tag
{
  uint8_t                 status;
}fslProfileRmtPairCnf_t;
```

The message informs the application as to how the remote pairing process has ended. If status is *gNWSuccess_c* the remote pairing process has been completed successfully. Otherwise the status field will indicate the reason for the failure.

While the remote pairing process is in progress (from the call to FSLProfile_RmtPairRequest to the arrival of the remote pair confirm message) the receiver is permanently enabled. The initial receiver active period value is saved in an internal value and restored at the end. The application should not change the state of the receiver during the remote pairing or the process may fail.

4.1.2 Remote Pairing on the Recipients

Each remote pair recipient application is informed about the remote pair request via a remote pair indication message, which has the following structure:

4.1.2.1 Message Structure

```
typedef struct fslProfileRmtPairInd_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  uint16_t                appRspTimeOut;
  appCapabilities_t       devAppCapabilities;
  uint8_t*                pDeviceTypeList;
  uint8_t*                pProfilesList;
}fslProfileRmtPairInd_t;
```

It has the following fields:

- status – indicates whether pairing can be accepted (*gNWSuccess_c*) or if the pair table is full (*gNWNoRecipCapacity_c*)
- deviceId – identifies the device requesting the remote pair
- appRspTimeOut – indicates how long the application has to respond to the remote pair request; if the application doesn't respond the remote pairing times out
- devAppCapabilities – the application capabilities of the other remote pair recipient
- pDeviceTypeList – the device types supported by the other remote pair recipient
- pProfilesList – the profiles supported by the other remote pair recipient

To respond to the remote pair request, the application must call FSLProfile_RmtPairResponse().

4.1.2.2 Prototype

```
uint8_t FSLProfile_RmtPairResponse(uint8_t status);
```

Its only parameter is:

- status – *gNWSuccess_c* if pairing is accepted, *gNWNotPermitted_c* if denied, *gNWNorecipCapacity_c* if the pair table is full.

The application has a finite amount of time to respond to the remote pair request; the amount of time is given in the remote pair indication message. After appRspTimeOut milliseconds have passed the profile layer will send an automatic remote pair response frame back to the originator informing it that the application has not responded. This will cause the remote pairing process to abort.

After this function returns *gNWSuccess_c*, the application should wait for the remote pairing process to complete. The application will be notified of this via a remote pair response confirm message, which has the following structure:

4.1.2.3 Message Structure

```
typedef struct fslProfileRmtPairRspCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
}fslProfileRmtPairRspCnf_t;
```

It has the following fields:

- status – indicates how the remote pairing process was completed
- deviceId – the pair table index of the entry of the new device (this value should be ignored if remote pairing was not successful)

If the application has accepted pairing and the remote pairing process has been completed successfully the deviceId field contains the index of the new pair table entry.

While the remote pairing process is in progress (from the call to FSLProfile_RmtPairResponse to the arrival of the remote pair response confirm message) the receiver is permanently enabled. The initial receiver active period value is saved in an internal value and restored at the end. The application should not change the state of the receiver during remote pairing or the process may fail.

Chapter 5
Over the Air Menus

The over the air menus functionality links up to three devices together: a menu browser, a menu owner and a menu displayer. A single device can share two functions. For example, if a remote has an LCD screen, it can be both the menu browser and the menu displayer. The menu browser is typically a remote controller or any other device that can receive user input. The menu owner is the device being controlled (a media player, a TV, a DVD, etc.). The menu displayer should be a device with a GUI capable of displaying the menu.

5.1 The Menu Browser

The menu browser's role is to react to user input and send a menu browsing command to the menu owner.

5.1.1 Configuration

The initialization function for the menu browser is FSLProfile_InitMenuBrowserProcedure. No other configuration besides initialization needs to be done.

5.1.2 Sending a Menu Browsing Command

To send a menu browsing command the application should call FSLProfile_BrowseMenuRequest.

5.1.2.1 Prototype

```
uint8_t FSLProfile_BrowseMenuRequest(
         uint8_t deviceId,
         menuBrowseDirection_t direction,
         bool_t bUseSecurity
);
```

The parameters are as follows:

- deviceId – the device ID of the menu owner
- direction – the menu browsing direction
- bUseSecurity – whether to transmit the request encrypted or not

The menu browsing direction can have any of the following values:

```
typedef enum
{
  menuBrowseUp_c = 0,
  menuBrowseDown_c,
```

```
  menuBrowseLeft_c,
  menuBrowseRight_c,
  menuBrowseOk_c,
  menuBrowseExit_c,
  menuBrowseRefresh_c,
  menuBrowseMax_c
}menuBrowseDirection_t;
```

These values have no intrinsic meaning for the profile layer, so the application is free to use them as it chooses.

As with the network layer requests, if an error which can be reported immediately are encountered, the function return value contains this error code. In this case, the process is aborted and no further confirmation messages will be arriving.

A return value of *gNWSuccess_c* indicates that the menu browse command transmission process has started. When the process is complete (whether successful or not) the application will be notified via a Menu Browse Confirm message, which has the following structure:

5.1.2.2 Message Structure

```
typedef struct fslProfileMenuBrowseCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
}fslProfileMenuBrowseCnf_t;
```

It has the following fields:

- status – informs the application how the transmission was completed; the status is copied from the NLDE Data Confirm
- deviceId – the menu owner's device ID

While transmission is in progress the receiver is enabled indefinitely (power saving is disabled). The initial receiver active period is saved in an internal variable and restored when transmission is complete. The application should take care not to change the state of the receiver during this time period as transmission may fail.

The menu owner may issue a response to the menu browsing command. The application is informed of the arrival of this response via a Menu Browse Complete Indication message, which has the following structure:

5.1.2.3 Message Structure

```
typedef struct fslProfileMenuBrowseCompleteInd_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
  uint8_t                 userString[gSizeOfUserString_c];
}fslProfileMenuBrowseCompleteInd_t;
```

It has the following fields:

- status – the menu browse command's execution status
- deviceId – the responding menu owner's device ID
- userString – the menu displayer's user string

The status field can usually have only one of two values: either *gNWSuccess_c* or *gNWNoResponse_c*. When status is *gNWSuccess_c* the menu browse command has been executed successfully. When status is *gNWNoResponse_c* it means that the menu owner was unable to transmit the display menu commands to the menu displayer. This is the recommended usage for the status field; however the menu owner is free to assign any value to it. Also, depending on the exact protocol details, the menu owner may not send menu browse complete message at all.

5.2 The Menu Owner

The menu owner is typically the device being controlled by the menu browser. Its role is to react to menu browsing commands received over the air by executing the requested commands and sending menu display commands to the menu displayer.

5.2.1 Configuration

The initialization function for the menu owner is FSLProfile_InitMenuOwnerLightProcedure. Apart from initialization, no other configuration needs to be done.

5.2.2 Sending an Entire Menu Window to the Menu Displayer

A menu window consists of a menu header, which contains the number of menu entries in the window, plus the menu entries themselves. The menu header must be sent first, followed by all the menu entries.

To send the menu header the application should call FSLProfile_DisplayMenuHeaderRequest.

5.2.2.1 Prototype

```
uint8_t FSLProfile_DisplayMenuHeaderRequest(uint8_t   deviceId,
                                            bool_t    bUseSecurity,
                                            uint8_t   idxSelectedEntry,
                                            uint8_t   nrMenuItemsInWindow,
                                            uint16_t  nrMenuItemsInMenu,
                                            uint16_t  firstEntryNumber,
                                            uint8_t   contentType,
                                            uint8_t   menuTextLength,
                                            uint8_t*  pMenuText);
```

It has the following parameters:

- deviceId – the menu displayer's device ID
- bUseSecurity – whether to use secured transmission or not

- idxSelectedEntry – which entry in the menu is the currently selected one
- nrMenuItemsInWindow – how many entries there are in the menu window
- nrMenuItemsInMenu – the total amount of entries in the menu node (can be higher than the number of entries in the window)
- firstEntryNumber – specifies the index in the menu node of the menu entry with index zero in the menu window (this is useful if the displayer numbers the entries in a menu according to their index in the node e.g. if the first tune displayed on screen is actually the fiftieth in the playlist)
- contentType – an application defined value describing the type of content in the menu
- menuTextLength – the length of the menu text
- pMenuText – application defined text describing the menu header (it is copied internally by the function call, so it need not be preserved during transmission)

As with the network layer requests, if an error which can be reported immediately is encountered, the function call return value contains this error code. In this case, the process is aborted and no further confirmation messages will be arriving. Otherwise the function will return *gNWSuccess_c* and the transmission process has been started.

When transmission is complete the application will be notified via a Display Menu Confirm message, which has the following structure:

5.2.2.2 Message Structure

```
typedef struct fslProfileDisplayMenuCnf_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
}fslProfileDisplayMenuCnf_t;
```

It has the following fields:

- status – the transmission status
- deviceId – the menu displayer's device ID

After the menu header has been successfully sent, the application should immediately start sending the menu entries. To send a menu entry the application should call FSLProfile_DisplayMenuEntryRequest.

5.2.2.3 Prototype

```
uint8_t FSLProfile_DisplayMenuEntryRequest( uint8_t   deviceId,
                                            bool_t    bUseSecurity,
                                            uint8_t   entryIndex,
                                            uint8_t   entryType,
                                            uint8_t   contentType,
                                            uint8_t   entryValueLength,
                                            uint8_t   entryTextLength,
                                            uint8_t*  pEntryValue,
                                            uint8_t*  pEntryText);
```

It has the following parameters:

- deviceId – the menu displayer's device ID
- bUseSecurity – whether to use secured transmission or not
- entryIndex – the index in the menu window of the current entry
- entryType – an application defined value describing the entry
- contentType – an application defined value describing the entry content
- entryValueLength – the length of the menu entry value
- entryTextLength – the length of the menu entry text
- pEntryValue – the menu entry value
- pEntryText – the menu entry text

Each menu entry can be described by two different strings: the entry text, which is the name of the entry and an optional additional value. The value can be displayed for example in the right end of the menu entry and may change in reaction to user input. Not all menu entries need to have a value, so the value length may be zero.

As with the network layer requests, if an error which can be reported immediately is encountered, the function call return value contains this error code. In this case, the process is aborted and no further confirmation messages will be arriving. Otherwise the function will return *gNWSuccess_c* and the transmission process has been started.

When transmission is complete the application will be notified via a Display Menu Confirm message, previously described.

The application should send the menu entries as quickly as possible, otherwise the menu displayer will time out in waiting for them. Typically, the application should handle the Display Menu Confirm message generated by the transmission of a menu entry by requesting the transmission of the next menu entry.

Also, the menu entries must be sent in order, starting with the menu entry with entry index zero and ending with the entry with index (*nrMenuItemsInWindow* – 1). Failure to do so will trigger an error on the displayer.

5.2.3 Sending a Single Menu Entry to the Menu Displayer

If the result of the menu browsing command is a change in a single entry of the currently displayed menu window, the menu owner may opt to not re-send the entire menu window. Instead, a single menu entry message may be sent, describing the updated menu entry. The menu displayer's normal reaction to receiving a menu entry separate from the menu header should be to update the particular menu entry in the menu window and move the selection to the just updated entry.

This can be used, for example, if the user selects another entry, but the change in selection does not trigger a window scroll.

5.2.4 Sending a Menu Message to the Menu Displayer

To send a simple message that the displayer should display on its screen, the application should call FSLProfile_DisplayMenuMessageRequest.

5.2.4.1 Prototype

```
uint8_t FSLProfile_DisplayMenuMessageRequest(uint8_t           deviceId,
                                             bool_t            bUseSecurity,
                                             menuMessageType_t messageType,
                                             uint8_t           messageLength,
                                             uint8_t*          pMessage);
```

Its parameters are as follows:

- deviceId – the menu displayer's device ID
- bUseSecurity – whether to send the command encrypted or not
- messageType – the type of the message
- messageLength – the length of the message text
- pMessage – the message text

The message type can have one of the following values:

```
typedef enum
{
  menuMessageType_Information_c = 0,
  menuMessageType_Warning_c,
  menuMessageType_Error_c,
  menuMessageType_Max_c
}menuMessageType_t;
```

The application is free to add any user defined values to the above list, but it must ensure that the menu displayer can handle them correctly.

Typically, on receipt of the display menu command, the menu displayer should display a message box on its screen.

When the transmission is complete, the application will be notified via a Display Menu Confirm message.

5.2.5 Deactivating the Menu

When the menu becomes inactive, for any reason, the menu owner should inform the displayer so that the menu can be removed from the screen. To do so, the menu owner's application should call FSLProfile_DisplayMenuExitRequest.

5.2.5.1 Prototype

```
uint8_t FSLProfile_DisplayMenuExitRequest(uint8_t deviceId,
                                          bool_t  bUseSecurity);
```

It has the following parameters:

- deviceId – the menu displayer's device ID
- bUseSecurity – whether to transmit the command encrypted or not

As with the network layer requests, if an error which can be reported immediately is encountered, the function call return value contains this error code. In this case, the process is aborted and no further confirmation messages will be arriving. Otherwise the function will return *gNWSuccess_c* and the transmission process has been started.

When transmission is complete the application will be notified via a Display Menu Confirm message.

5.2.6 Responding to the Menu Browser

The menu owner may reply to the menu browser with information about how the menu browsing request has been carried out. To do so, the application should call DisplayCompleteIndToBrowserRequest.

5.2.6.1 Prototype

```
uint8_t FSLProfile_DisplayCompleteIndToBrowserRequest(
                          uint8_t browserDeviceId,
                          uint8_t displayerDeviceId,
                          bool_t  bUseSecurity,
                          uint8_t status
                 );
```

Its parameters are as follows:

- browserDeviceId – the menu browser's device Id
- displayerDeviceId – the menu displayer's deviceId; the displayer's user string is included in the Display Complete Indication message on the assumption that the displaying capabilities (size of the display, etc) are included in the user string
- bUseSecurity – whether to transmit the response secured or not
- status – the menu browsing command's execution status

The application should use one of two values for the status parameter:

- *gNWSuccess_c* if all the display menu commands have reached the menu displayer (i.e. all the Display Menu Confirm message have had a status of *gNWSuccess_c*)
- *gNWNoResponse_c* if at least one display menu command has not reached the menu displayer; in this case the menu browser, which is normally the remote can take specific action (e.g. inform the user, change the displayer device, etc.)

These values however are not mandatory (they have no intrinsic meaning to the profile layer) and the application can use any value that is deemed appropriate, provided that the menu browser can understand its meaning.

When the transmission is complete, the application will be notified via a Display Menu Confirm message.

5.3 The Menu Displayer

The menu displayer is a passive device from the point of view of the OTA menus protocol. It receives menu headers, entries and messages from the menu owner and displays them on the GUI. It does not send any menu command over the air.

The details of displaying a menu are implementation and hardware dependant and are beyond the scope of this manual.

5.3.1 Configuration

The initialization function for the menu browser is FSLProfile_InitMenuDisplayerProcedure.

5.3.1.1 Prototype

```
uint8_t FSLProfile_InitMenuDisplayerProcedure(uint16_t waitMenuEntryTimeout);
```

In addition to initializing the menu displayer functionality in the profile layer, the function also configures the maximum amount of time to wait between two successive menu entries before timing out.

5.3.2 Receiving an Entire Menu Window

The process of receiving an entire menu window begins with the arrival of Display Menu Header Indication, which has the following structure:

5.3.2.1 Message Structure

```
typedef struct fslProfileDisplayMenuHeaderInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 idxSelectedEntry;
  uint8_t                 nrMenuItemsInWindow;
  uint16_t                nrMenuItemsInMenu;
  uint16_t                firstEntryNumber;
  uint8_t                 contentType;
  uint8_t                 menuTextLength;
```

```
  uint8_t*                pMenuText;
}fslProfileDisplayMenuHeaderInd_t;
```

It has the following fields:

- deviceId – the menu owner's device ID
- idxSelectedEntry – which entry in the menu is currently selected
- nrMenuItemsInWindow – how many menu entries fit in the current menu window to be displayed
- nrMenuItemsIn – how many menu entries there are in the entire menu node (there can be more entries that can fit in the entire window)
- firstEntryNumber – which is the first entry in the menu
- contentType – the type of content in the menu
- menuTextLength – the length of the text describing the menu
- pMenuText – a pointer to the text string describing the menu; the actual text resides in the same memory buffer as the indication message and will be freed along with it

After the Display Menu Header Indication message is received the application should expect the arrival of all the menu entries that fit in the menu window. Unless an error occurs there should be exactly *nrMenuItemsInWindow* menu entries. Each menu entry is described by a Display Menu Entry Indication message, which has the following structure:

5.3.2.2 Message Structure

```
typedef struct fslProfileDisplayMenuEntryInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 entryIndex;
  uint8_t                 entryType;
  uint8_t                 contentType;
  uint8_t                 entryValueLength;
  uint8_t                 entryTextLength;
  uint8_t*                pEntryValue;
  uint8_t*                pEntryText;
}fslProfileDisplayMenuEntryInd_t;
```

It has the following fields:

- deviceId – the menu owner's device ID
- entryIndex – the current index of the entry; this is a value from zero to (*nrMenuItemsInWindow* – 1)
- entryType – the application defined type of the menu entry
- contentType – the application defined type of the content in the menu entry
- entryValueLength – the length of the menu entry value
- entryTextLength – the length of the menu entry text
- pEntryValue – the menu entry value
- pEntryText – the menu entry text

Each menu entry can be described by two different strings: the entry text, which is the name of the entry and an optional additional value. The value can be displayed for example in the right end of the menu entry and may change in reaction to user input. Not all menu entries must have a value, so the value length may be zero.

The profile layer expects all menu entries to arrive in order; otherwise the reception process ends with an error. Also, the maximum amount of time the profile layer will wait between two successive menu entries is *waitMenuEntryTimeout*, configured with FSLProfile_InitMenuDisplayerProcedure. It is allowed to call the FSLProfile_InitMenuDisplayerProcedure function every time *waitMenuEntryTimeout* needs to be changed.

The menu entry text and value are stored inside the same memory buffer that also contains the Display Menu Entry Indication. The menu entry and value should be copied to application internal variables and the memory buffer freed, otherwise the profile may not have memory to receive all the menu entries.

When the menu window reception is complete (either because all menu entries have been received or the process was aborted) the application will receive a Display Menu Complete Indication message, which has the following structure:

5.3.2.3 Message Structure

```
typedef struct fslProfileDisplayMenuCompleteInd_tag
{
  uint8_t                 status;
  uint8_t                 deviceId;
}fslProfileDisplayMenuCompleteInd_t;
```

It has the following fields:

- status – how well the menu window was received
- deviceId – the menu owner's device Id

The Display Menu Complete Indication always arrives after a Display Menu Header Indication, no matter how many menu entries are received.

If the entire menu window was successfully received the application now has all the necessary information to display the menu window.

5.3.3 Receiving a Single Menu Entry

The displayer may also receive a separate Display Menu Entry Indication message, not preceded by a Display Menu Header Indication. The displayer should react to the arrival of such a message by updating the entry referred to by entryIndex field and switching selection to the just updated entry. No Display Menu Complete Indication will arrive after such a Display Menu Entry Indication.

5.3.4 Receiving a Menu Message

The menu owner may send a menu message to the displayer that should normally be displayed on the screen. The application is notified via a Display Menu Message Indication, which has the following structure:

5.3.4.1 Message Structure

```
typedef struct fslProfileDisplayMenuMessageInd_tag
{
  uint8_t                 deviceId;
  menuMessageType_t       messageType;
  uint8_t                 messageLength;
  uint8_t*                pMessage;
}fslProfileDisplayMenuMessageInd_t;
```

It has the following fields:

- deviceId – the menu owner's device Id
- messageType – the type of message
- messageLength – the length of the message
- pMessage – a pointer to the message text

The message type can have any one of the following values:

```
typedef enum
{
  menuMessageType_Information_c = 0,
  menuMessageType_Warning_c,
  menuMessageType_Error_c,
  menuMessageType_Max_c
}menuMessageType_t;
```

The actual message text string is contained within the same memory buffer as the indication message.

5.3.5 Removing the Menu From the Screen

When the menu becomes inactive on the owner, for any reason, it will inform the displayer by sending a display menu exit frame. When such a frame is received the application will be notified via a Display Menu Exit Indication message, which has the following structure:

5.3.5.1 Message Structure

```
typedef struct fslProfileDisplayMenuExitInd_tag
{
  uint8_t                 deviceId;
}fslProfileDisplayMenuExitInd_t;
```

Its only field contains the device ID of the menu owner. When receiving this message, the displayer should remove the menu from the screen.

5.4 Embedded Menu Owner

The embedded menu owner automatically handles received menu browsing requests by generating display menu request frames and transmitting them to the displayer. The application designer's task is to provide the profile layer with a static array containing all the menu entries in sorted order (named *menu*) plus three arrays of function pointers to callback functions (named *menuFuncList*, *menuFuncValue* and *menuFuncExe*), used to execute actions in case of dynamic menu entries.

When the profile layer receives a menu browse request from a menu browser device, it parses the menu entry array and calls the necessary callback functions to generate the display menu frames that are then sent automatically to the displayer.

When entering a submenu or activating an item, the menu item is pushed on a menu stack. When the user leaves the submenu the parent is popped off the stack. The stack size is equal to the maximum depth of the menu.

The display request frames are constructed in a display buffer, which is accessible to the application. Typically, the callback functions modify the contents of the display buffer in order to update the display frames with run-time information.

The application has very little direct run-time interaction with the embedded owner functionality, besides the callback functions.

This functionality cannot be exposed over a Black Box Interface.

5.4.1 Configuration

The initialization function for the embedded menu owner is FSLProfile_InitMenuOwnerProcedure.

5.4.1.1 Prototype

```
uint8_t FSLProfile_InitMenuOwnerProcedure(uint16_t noActivityExitMenuTimeout);
```

It takes as its parameter a time-out value. Whenever the time-out expires with no menu browsing commands arriving over the air the menu becomes inactive and will ignore further menu browsing commands. At start-up the menu is inactive. The application can activate or inactivate the menu by calling FSLProfile_SetMenuActiveMode.

5.4.1.2 Prototype

```
void FSLProfile_SetMenuActiveMode(bool_t bActive)
```

Its only parameters informs the profile layer whether to activate or shut down the menu.

The application must also configure the device ID of the menu displayer and the maximum number items in the displayer's menu window. To configure the displayer the application should call FSLProfile_SetDisplayerRequest.

5.4.1.3 Prototype

```
uint8_t FSLProfile_SetDisplayerRequest(uint8_t deviceId)
```

It takes as its only parameter the device ID of the menu displayer. The profile layer needs to know the device ID of the menu displayer since display menu frames are sent automatically, without the application requesting it.

The profile layer also needs to know the size of the menu window that the displayer can accept (e.g. fit on its screen). To configure it, the application needs to call FSLProfile_SetDisplaySizeRequest.

5.4.1.4 Prototype

```
uint8_t FSLProfile_SetDisplaySizeRequest(
                  uint8_t deviceId,
                  uint8_t nrOfDisplayableMenuEntries
        );
```

It takes as its parameters the displayer's device ID and its menu window size. A menu owner can have several displayers configured (the size of their menu windows known). However, the profile layer will send the display menu frames to only one of them, the one configured with FSLProfile_SetDisplayerRequest.

NOTE

Discovery of a displayer's maximum menu window size is the applications responsibility and beyond the scope of this guide.

5.4.2 Interpretation of the Menu Browsing Commands

The menu browsing commands described in Section 5.1.2, "Sending a Menu Browsing Command", are interpreted as follows:

- *menuBrowseUp_c* – scroll the menu upwards
- *menuBrowseDown_c* – scroll the menu downwards
- *menuBrowseLeft_c* – go up in the menu hierarchy, popping the parent menu item off the stack.
- *menuBrowseRight_c* and *menuBrowseOk_c* – go down in the menu hierarchy if the selected menu entry is a parent with children (i.e. enter the submenu) or activate the menu entry otherwise. The parent will be pushed on the menu stack
- *menuBrowseExit_c* – exit the menu and deactivate it. The menu stack will be cleared.

5.4.3 Menu Items Array

The main content of the menu is inside a structure of type menu_t, as follows:

5.4.3.1 Message Structure

```
typedef struct
{
  const uint16_t  size;
  menuItem_t*     pMenuItems;
  const uint8_t*  title;
}menu_t;
```

The menu variable resides in application space and must be called *menu*. The profile layer accesses this variable and expects that name at link time. Any other name will cause a link error. The menu structure has the following members:

- size – the size of the menu items array
- pMenuItems – a pointer to the first entry in the menu items array
- title – a pointer to a string containing the application defined name of the array (e.g. "Main menu")

A typical menu variable declaration is as follows:

```
const menu_t menu =
{
  sizeof(menuMainItems)/sizeof(menuItem_t),
  (menuItem_t*)menuMainItems,
  "Main menu",
};
```

An entry in the menu items array has the following structure:

5.4.3.2 Message Structure

```
typedef struct
{
  /* Each menu entry in a menu tree is uniquely identified by
  an array of identifiers. The fact that each identifier is
  16 bit allows one menu entry to have up to 65535 direct children */
  /* Each menu entry has its own properties */
  uint16_t                menuId[gMaxMenuDepth_c];


  menuEntryProperties_t   menuEntryProperties;
}menuItem_t;
```

It has two members, the menu item ID and its properties described in the following sections.

5.4.3.3 Menu ID

The menu item ID is an array of identifiers uniquely describing the path through menu tree to reach the leaf that is represented by the current menu item. The first entry in the menuId array identifies the root parent, the second identifies the root parent's child, and so on. The menu tree has a maximum depth of *gMaxMenuDepth_c*, which is configured by the application so a menuId array can identify any menu item.

For example, considering a maximum menu depth of 5, the entry with a menuId of {2, 3, 0, 0, 0} is the third child of the second child of the root. The children of this entry will start from {2, 3, 1, 0, 0}. An example menu tree and the numbering of the nodes is shown in the following figure:

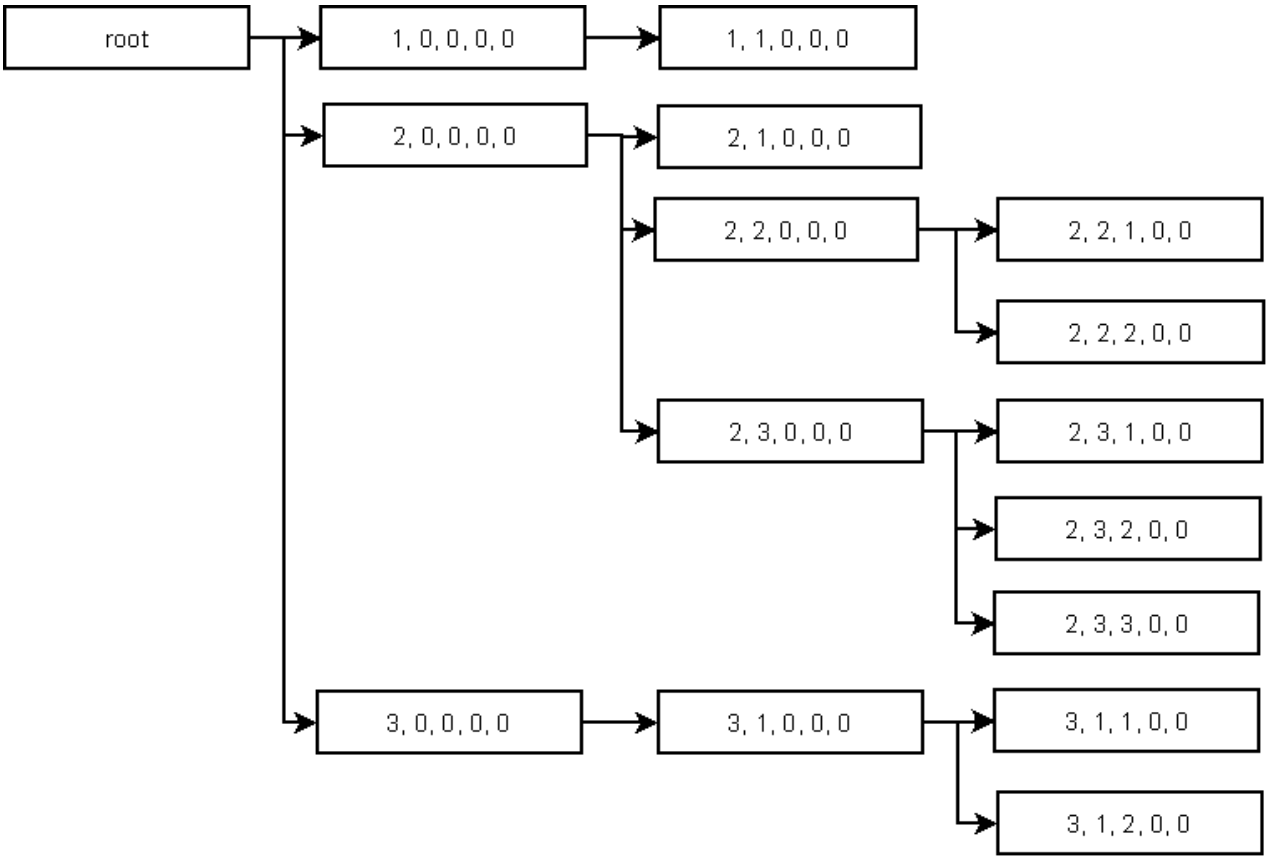

Figure 5-1. An Example Menu Tree

The array of menu entries must be sorted after the menu ID in a very specific order. In essence the order is recursive. For each menu tree node (starting with the root), add all the node's direct children to the array, then recalculate the algorithm for the children nodes. The root node is symbolic and does not appear in the menu item array.

The menu items array for the menu tree shown in Figure 5-1 is as follows:

```
static const menuItem_t  menuMainItems[]= {
/* D1  D2  D3  D4  D5    menu properties */
  { 1 , 0 , 0 , 0 , 0 , ...
  { 2 , 0 , 0 , 0 , 0 , ...
  { 3 , 0 , 0 , 0 , 0 , ...
  { 1 , 1 , 0 , 0 , 0 , ...
  { 2 , 1 , 0 , 0 , 0 , ...
  { 2 , 2 , 0 , 0 , 0 , ...
  { 2 , 3 , 0 , 0 , 0 , ...
  { 2 , 2 , 1 , 0 , 0 , ...
  { 2 , 2 , 2 , 0 , 0 , ...
  { 2 , 3 , 1 , 0 , 0 , ...
  { 2 , 3 , 2 , 0 , 0 , ...
  { 2 , 3 , 3 , 0 , 0 , ...
  { 3 , 1 , 0 , 0 , 0 , ...
  { 3 , 1 , 1 , 0 , 0 , ...
  { 3 , 1 , 2 , 0 , 0 , ...
}
```

5.4.3.4 Menu Item Properties

Each menu item is described by a set of properties grouped in the following structure:

5.4.3.5 Message Structure

```
typedef struct
{
  entryType_t     entryType;
  uint8_t         contentType;

  /* Must be set to a valid value only when the sub-menu is generated
     dynamically or the menu executes some action.
     In both cases, idxOfpfAction indicates the position in the array of
     pointers to functions that create sub-menus or execute
     actions where the pointer to this menu function resides */
  uint8_t         idxOfpfAction;

  /* Useful field to store menu characteristic information */
  uint16_t        tag;

  /* Pointer to a null terminated string that is the menu's text */
  uint8_t*        text;

  /* Pointer to a null terminated string that is the title of the menu's
     submenu
     Usually it is the same as the text */
  uint8_t*        title;
}menuEntryProperties_t;
```

It has the following members:

- entryType – describes the type of entry. It is sent over the air with the menu entry and also has significance for the embedded menu parser (The possible values and their significance are described shortly)
- contentType – the type of content in the menu entry; application defined, it is sent over the air with the display menu entry frame
- idxOfpfAction – if the menu item triggers an action when activated, this is the index in the function pointer tables of the function that will get called to execute the action. Only the function from one of the tables gets called, depending on the entryType.
- tag – an application defined number that is passed to the function executing the action
- text – a pointer to an application defined string describing the menu item; it will be sent to the displayer
- title – a pointer to an application defined string containing the menu item’s title (can be the same as text); it will be sent to the displayer

Each menu item has an entry type, which can have one of the following values:

```
typedef enum
{
```

```
  gEntryType_GenericLeaf = 0,
  gEntryType_GenericStaticParent,
  gEntryType_GenericDynamicParent,
  gEntryType_RoValue,
  gEntryType_RwValue,
  gEntryType_RwValueEditable,
  gEntryType_Action,
  gEntryType_ActionBack1
}entryType_t;
```

They have the following significance:

- *gEntryType_GenericLeaf* – a menu item without submenus and which triggers no action, useful only for listing or descriptive purposes. Its properties are completely known at compile time. Activating this entry will have no effect.
- *gEntryType_GenericStaticParent* – a menu item with a static submenu. The children of this node are known at compile time. Activating this entry will enter the static submenu and the parent entry is pushed on the menu stack.
- *gEntryType_GenericDynamicParent* – a menu item with a dynamic submenu, generated at runtime. It has only one child in the menu items array that serves as a blueprint. When activated, the callback function at position idxOfpfAction in the *menuFuncList* array will be triggered. The single child of a dynamic parent must have as the last nonzero value of the menu ID the value *ANY* (which is 0xFFFF). For example if the parent's menu ID is {2, 3, 1, 0, 0} the child ID must be {2, 3, 1, ANY, 0}. The child can have children of its own. If the child is itself a dynamic parent the grandchild's ID will be {2, 3, 1, ANY, ANY}. The profile layer does not send the menu window to the displayer automatically, this must be requested by the application (See Section 5.4.5.1, "Dynamic Submenus".)
- *gEntryType_RoValue* – a menu item which provides access to a read-only value and has no children. When parsed in order to be sent over the air, the callback function at position idxOfpfAction in the *menuFuncValue* array will be triggered. When activated, the value will be refreshed in the display buffer but no updates will be sent over the air to the menu displayer.
- *gEntryType_RwValue* – a menu item which provides access to a readable and writable value and has no children. When parsed in order to be sent over the air, the callback function at position idxOfpfAction in the menuFuncValue array will be triggered. When activated, the menu entry's type in the display buffer will be changed to *gEntryType_RwValueEditable* and further menu up or down browsing commands will be used to increase or decrease the value rather than scroll the menu (i.e. the menu item is in edit mode). The callback function at position idxOfpfAction in the *menuFuncValue* array will be triggered for this. Every time the value is modified a display menu entry frame is sent to the displayer to update the screen (See Section 5.4.5.4, "Value Menu Entries".)
- *gEntryType_RwValueEditable* – should not be used in the menu item array. When the top item in the menu stack is of this type, activating it or going up in the hierarchy will revert it to *gEntryType_RwValue* type and menu up or down browsing commands will be used to scroll the menu again. When the menu item is in edit mode, the Display Menu Entry Indications received by the displayer will have an entryType of *gEntryType_RwValueEditable*, allowing it to highlight the entry in a particular way of its choosing, so that the user is aware that the item is in edit mode.

- *gEntryType_Action* – a menu item which triggers an action. When activated, the callback function at position idxOfpfAction in the *menuFuncExe* array will be triggered. The menu will be resent to the displayer
- *gEntryType_ActionBack1* – a menu item which triggers an action and then goes up in the menu hierarchy. When activated, the callback function at position idxOfpfAction in the *menuFuncExe* array will be triggered followed by the effects of *menuBrowseLeft_c* command.

5.4.4 Display Buffer

The display buffer can contain all the information necessary to construct the frames detailing an entire menu window (the display menu header frame plus all the necessary display menu entry frames). It has the following structure:

5.4.4.1 Message Structure

```
typedef struct
{
  uint8_t               nrMenuItemsInWindow;
  uint16_t              nrMenuItemsInMenu;
  uint16_t              firstEntryNumber;
  uint8_t               selectedItem;
  menuEntryProperties_t menuEntryProperties[gMaxEntriesInMenuTxBuffer_c];
  uint8_t
menuEntryValue[gMaxEntriesInMenuTxBuffer_c][gMaxEntryValueLengthInMenuTxBuffer_c];
}displayMenuTxBuffer_t;
```

It has the following members:

- nrMenuItemsInWindow – the number of items in the menu window
- nrMenuItemsInMenu – the number of items in the entire menu node
- firstEntryNumber – the index in the menu node of the first item in the menu window
- selectedItem – the index in the menu window of the currently selected entry
- menuEntryProperties – the properties of each menu entry in the menu window (Described in Section 5.4.3.4, “Menu Item Properties”.)
- menuEntryValue – the value of each menu entry in the menu window; it is included in the display menu entry frame only if the entryType in the menuEntryType has a value of either *gEntryType_RoValue* or *gEntryType_RwValue* or *gEntryType_RwValueEditable*

When a menu browsing command arrives, the profile layer calculates the new position of the menu window and fills in the fields of the buffer. The menu array is parsed and the properties of the entries in the updated menu window are copied to the menuEntryProperties array. The menuEntryValue array is left unmodified by the profile layer.

The display buffer resides in application space and must be called *displayMenuTxBuffer*. The profile layer accesses the buffer at run-time to construct the display menu frames.

The display buffer is accessible by the application. Most importantly, the callback functions can, and should, modify the buffer with run-time information. When the parent menu item is dynamic, the profile layer copies the menu properties of the child entry at every position in the menuEntryProperties to serve as a blueprint that the callback function can modify.

5.4.5 Callback Functions

The callback functions create a dynamic menu with information only available at runtime. They are used either to create entire dynamic submenus, to read and/or modify certain values or to trigger actions. There are three arrays of pointers to callback functions, each array accessed depending on the menu entry type (See Section 5.4.3.4, “Menu Item Properties”.) The function pointer arrays reside in application space and must have the names *menuFuncList*, *menuFuncValue* and *menuFuncExe*.

5.4.5.1 Dynamic Submenus

Dynamic submenus have parents with an entryType of *gEntryType_GenericDynamicParent*. Whenever the parent is activated, or the user scrolls through the submenu itself, the profile layer fills the menuEntryProperties array in the display buffer with the child’s entry properties and then calls the callback function pointed to by the location in the menuFuncList array indicated by the idxOfpfAction of the parent. The callback function must have the following prototype:

5.4.5.2 Prototype

```
void func(uint16_t firstMenuItemId, uint8_t nrMenuItemsToGet, uint16_t tag);
```

It is passed as parameters the index of first menu item and the number of items in the menu window (i.e. the starting position and size of the menu window) plus the parent’s tag. The tag is an application defined value written to the menu array of the parent and passed to callback function. It is useful if the same function is called by several menu entries.

For example, when the menu owner is a media player of some type, the profile layer might request the list of song names starting from song 20 until song 26 in the play list identified by the tag.

The function should update the display buffer with the requested information (e.g. write the names of the songs in the text fields of each item of the menuEntriesProperty). The function is free to modify any field in the display buffer, including the size of the menu window.

Going back to the above example, an iPod might have a menu entry of type *gEntryType_GenericDynamicParent* with a callback function that reads the available play lists and places the description of the requested ones in the display buffer. The index of each play list will be placed in the tag field. The single child will be a blueprint for play list descriptions and will also be of type *gEntryType_GenericDynamicParent*. When a play list is selected, the menu entry properties of the selected menu entry (i.e. the play list) will be pushed on the stack, including the tag which identifies the play list. The child’s callback function reads the songs from a given play list and writes their descriptions to the display buffer.

Because the act of gathering the requested information can be lengthy (e.g. the information may need to be requested over the serial interface) the profile layer does not trigger the menu window transmission immediately after the call to the callback function. Thus, the application is free to gather the information asynchronously e.g. the callback function may send a message to the application task, instructing it to gather the information, rather than doing it directly. When the requested information has been placed in the display buffer the profile layer needs to be told to initiate the menu window transmission. To do this, the application must call FSLProfile_DisplayMenuRequest, which has the following prototype:

5.4.5.3 Prototype

```
uint8_t FSLProfile_DisplayMenuRequest(void);
```

On the embedded menu owner this function takes no parameters. It simply triggers the transmission of the menu window described by the contents of the menu buffer. This function should be called either at the end of the callback function, if the dynamic menu information can be gathered synchronously, or from the application task when the information gathering is complete, otherwise.

When transmission is complete the application will receive a Display Menu Confirm message. SeeSection 5.2.2, “Sending an Entire Menu Window to the Menu Displayer”.

5.4.5.4 Value Menu Entries

Value menu entries are useful to display or change the values of an associated variable. The Value menu entries have a type of either *gEntryType_RoValue*, *gEntryType_RwValue* or *gEntryType_RwValueEditable*. When read or activated, the profile layer will call the callback function from the menuFuncValue table at position indicated by the idxOfpfFunction menu entry property. The callback function should maintain a static internal variable for temporary storage and perform modifications on that variable, only writing to the associated variable when the user confirms changes (i.e. when a *menuBrowseRight_c* command or a *menuBrowseOk_c* command is received). The callback has the following prototype:

5.4.5.5 Prototype

```
void func(
        uint8_t menuLine,
        gMnuValAction_t action,
        uint16_t numericVal,
        uint16_t tag
        );
```

Its parameters are as follows:

- menuLine – the position in the display buffer where the associated value should be written (i.e. the first index into the menuEntryValue array)
- action – the type of action requested
- numericVal – a numeric value to write into the associated variable

- tag – the tag of the menu entry

The menu action can have one of the following values:

```
typedef enum
{
  gMnuValAction_GetValue_c = 0,
  gMnuValAction_SetValue_c,
  gMnuValAction_ChangeOneUp_c,
  gMnuValAction_ChangeOneDown_c,
  gMnuValAction_ChangeNumeric_c
}gMnuValAction_t;
```

The callback function is called with each value under the following circumstances:

- The callback function will be called with an action of *gMnuValAction_GetValue_c* when the profile layer is copying menu entry properties from the menu entry array to the display buffer. Whenever a menu entry with a entryType property of either *gEntryType_RoValue*, *gEntryType_RwValue* or *gEntryType_RwValueEditable* is encountered, the callback function is called after copying the menu entry property to the display buffer. This action will also be used when the user cancels editing a value (i.e. when a *menuBrowseLeft_c* command is received and the currently selected menu entry is of type *gEntryType_RwValueEditable*). The callback function's reaction should be to read the value from the associated variable and write it to the display buffer in the menuEntryValue array.
- The action will be *gMnuValAction_ChangeOneUp_c* when a user is increasing the value in editing mode (i.e. a *menuBrowseUp_c* command is received and the currently selected menu entry is of type *gEntryType_RwValueEditable*). The callback function's reaction should be to increment a temporary variable and write the new value to the display buffer in the menuEntryValue array.
- The action will be *gMnuValAction_ChangeOneDown_c* when a user is decreasing the value in editing mode (i.e. a *menuBrowseDown_c* command is received and the currently selected menu entry is of type *gEntryType_RwValueEditable*). The callback function's reaction should be to decrement a temporary variable and write the new value to the display buffer in the menuEntryValue array.
- The action will be *gMnuValAction_ChangeNumeric_c* when a numeric command is received from the application and the currently selected menu entry is of type *gEntryType_RwValueEditable*. The callback function's reaction should be to set the temporary variable to the value of the numericVal parameter and write the new value to the display buffer in the menuEntryValue array. See Section 5.4.6, "Passing Numeric Commands to the Embedded Menu Owner".
- This action will be *gMnuValAction_GetValue_c* when the user confirms editing a value (i.e. when a *menuBrowseRight_c* command or a *menuBrowseOk_c* command is received and the currently selected menu entry is of type *gEntryType_RwValueEditable*). The callback function's reaction should be to set the associated variable to the value of the temporary variable and write the value to the display buffer in the menuEntryValue array.

5.4.5.6 Action Menu Entries

Menu entries can also trigger specific actions, such as starting to play a song, turning on a light, etc. A menu entry that triggers an action has a type of either *gEntryType_Action* or *gEntryType_ActionBack1*. Whenever such an entry is activated, the profile layer will call the callback function from the menuFuncExe table at position indicated by the idxOfpfFunction menu entry property. The callback function must have the following prototype:

5.4.5.7 Prototype

```
void func(uint16_t tag);
```

Its only parameter is the tag of the menu entry which triggers it.

The difference between and entry of type *gEntryType_Action* and an entry of type *gEntryType_ActionBack1* is that in the latter case, after the callback function is called, the profile layer will also move up one level in the menu hierarchy, effectively triggering the actions of a *menuBrowseLeft_c* command. In both cases, at the end, the menu window is redrawn (i.e. display menu frames are generated from the contents of the display buffer and sent to the menu displayer.

5.4.6 Passing Numeric Commands to the Embedded Menu Owner

The embedded owner can also handled numeric commands. However there are no browsing commands with numeric content, so these must be received by the application (e.g. as CERC commands) and passed to profile layer. The details of the protocol for receiving numeric commands from the menu browser are the application designer’s responsibility and beyond the scope of this manual.

The profile accepts numeric commands as unsigned 16 bit numbers. To pass a numeric command to the profile layer the application should call FSLProfile_ProcessMenuNumericCommandRequest, which has the following prototype:

5.4.6.1 Prototype

```
uint8_t  FSLProfile_ProcessMenuNumericCommandRequest(uint16_t number);
```

It takes as its only parameter the numeric command.

The effects of passing a numeric command to the profile layer depend on the state of the menu and the currently selected menu item.

If the menu is currently deactivated the function will exit with *gNWNotPermitted_c*.

If the currently selected item has a type of *gEntryType_RwValueEditable* the callback function for changing the value will be called and the updated menu entry is sent to the displayer.

Otherwise the effect is to move the selection to the entry in the menu node with the index equal to the numeric command (scrolling the menu window as necessary). The updated menu entry is sent to the displayer if no scrolling was done, otherwise the entire updated menu window is sent. Numbering of the entries in a menu node begins with 1. Numeric commands with a value of 0 will have no effect when moving the selection (i.e. when the selected item is not of type *gEntryType_RwValueEditable*).

Chapter 6
Over The Air Programming

The over the air programming allows devices to transmit firmware upgrade images within the network. The device that distributes new firmware images is called the OTAP server. The device that downloads a new firmware image and reboots with it is called the OTAP client. OTAP functionality is not mutually exclusive, that is, a device can be both an OTAP server and an OTAP client.

6.1 OTAP Process Flow

Figure 6-1 shows the OTAP process flow and the frames exchanged over the air. The process proceeds in several steps as follows:

1. The server sends out Image Notify messages, announcing the availability of a new image for download. This step is optional.
2. The client requests a new image and includes its hardware version and its current firmware version in the request frame.The server analyzes the request, decides whether is has a suitable image for the client or not and sends an appropriate response.
3. If the server has indicated in Step 2 that an upgrade image is available, the client requests transmission of the image, block by block. The server is stateless and the client always requests a block of a maximum size from a specified offset and the server responds with that particular block. The client writes each received block into external memory.
4. Once the client has received the entire image it notifies the server and the server provides and acknowledgment. The server's response includes a delay value in milliseconds. The client should not reboot with the new firmware image until the specified delay time has passed.

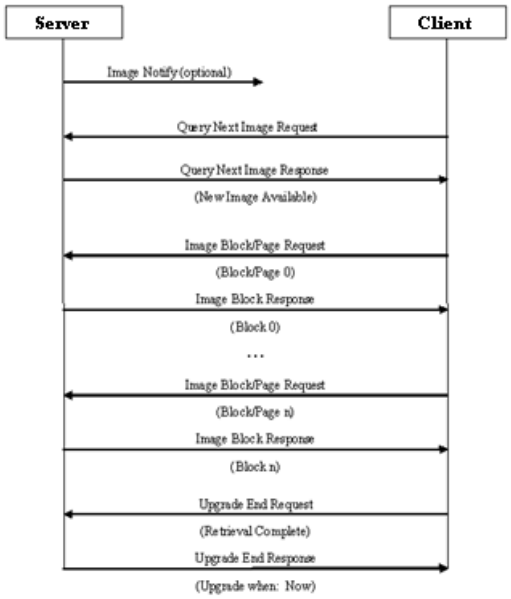

Figure 6-1. OTAP Process flow

6.2 OTAP Server Overview

The OTAP server transmits available firmware images to clients upon request. Each request for an image is forwarded by the profile layer to the application, which inspects the request and decides whether or not it will upload an image to the client. If the download process continues, the profile layer forwards all requests for specific blocks to the application. The application must obtain the appropriate block (from an external EEPROM, over UART, etc.) and pass it to the profile layer for transmission to the client. Finally, when the client completes the download, the server application can instruct the client to wait a specified period of time until it reboots with the new firmware.

Optionally, the server may send out notifications that it has new images available for download.

6.2.1 OTAP Server Configuration

The OTAP server functionality is contained in the *RF4CE_FSLProfile_OtapServer* library. Before use, the server functionality must be initialized by a call to *FSLProfile_InitOtapServerProcedure*.

6.2.2 Transmitting a Message to a Client

All transmissions to a client are initiated by a call to *FSLProfile_OtapServerSend*, which has the following prototype:

```
uint8_t FSLProfile_OtapServerSend(
                  uint8_t deviceId,
                  fslProfileApptoOtapCmd_t* otapCmdMsg
);
```

Its parameters are the device Id of the recipient client and a pointer to a structure detailing the message to send. The structure may be allocated on the stack (i.e. the function will not free this address before exiting)

There are four types of messages that can be sent:

1. Image notifications - when new images are available for download
2. Query next image responses - when a client requests a new firmware image
3. Image block responses - blocks of the firmware image requested by a client
4. Upgrade end responses - when a client reports that it has downloaded an entire image

The structure has the following format:

```
typedef struct fslProfileApptoOtapCmd_tag
{
  fslProfileOTAPServerCommand_t               cmdType;
  union {

    imageNotify_t           imageNotify;
    querryNextImgResp_t     querryNextImgResp;
    imageBlockResp_t        imageBlockResp;
    upgradeEndResp_t        upgradeEndResp;
  } cmdData;
}fslProfileApptoOtapCmd_t;
```

There is an associated enumeration value for *cmdType* and an associated *cmdData* union member for each of the message types described.

When the function exits with an error status (different from *gNWSuccess_c*) the transmission process aborts because of the error. When the function returns *gNWSuccess_c*, the message was passed to the network layer for transmission. After transmission is complete, the application receives an OTAP Server Confirm message, which contains the status of the transmission (either *gNWSuccess_c* or the network layer error code).

All messages, message usage and how to fill out the *cmdType* and *cmdData* parameters are described in detail in the following sections.

6.2.2.1 Sending Image Notifications

Whenever a server has a new image available for download, it can send out a notification. The application should set the *cmdType* parameter to *gFslProfileOtapImageNotifyCmd* and fill out the fields of the *imageNofity* structure, which has the following format:

```
typedef struct imageNotify_tag
{
  uint8_t                jitterVal;
  uint8_t                imageType[2];
  uint8_t                fileVersion[4];
}imageNotify_t;
```

The available fields are the image type (2 bytes, application specific), the new firmware file version (4 bytes, also application specific) and a jitter value.

The jitter is a value between 0x01 and 0x64, which prevents flooding the server with simultaneous requests for new images from several clients. When a the profile layer of a client receives an image notify message, it computes a random value between 0x01 and 0x64 and compares it to the received jitter value. If the generated value is greater than the received jitter value, the image notify message is silently discarded. This is how the server controls the fraction of clients that will request a new image after the notify message.

For example, by setting the jitter to 0x32, approximately half the interested clients will request a new image. By setting the jitter to 0x16, approximately one quarter of the interested clients will request a new image. Setting the jitter to 0x64 will allow all interested clients to request a new image.

6.2.2.2 Responding to Queries for the Next Image

When a client requests a new image it sends a Query Next Image message to the server. The application layer on the server receives an OTAP Server Query Next Image Indication message, which has the following format:

```
typedef struct fslProfileOtapServerQueryNextImageInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 fileVersion[4];
  uint8_t                 hardwareVersion[2];
}fslProfileOtapServerQueryNextImageInd_t;
```

The indication message contains the client's deviceId, the client's current firmware version and the client's hardware version. Based on this information, the server application must decide whether there is a suitable image for this client and whether the client is authorized to receive it. To respond to the client, the application should set the *cmdType* parameter to *gFslProfileOtapQueryNextImageRespCmd* and fill out the fields of the *queryNextImgResp* structure, which has the following format:

```
typedef struct queryNextImgResp_tag
{
  queryCmdStatusCode_t    status;
  uint8_t                 fileVersion[4];
  uint8_t                 imageSize[4];
  uint8_t                 crc[4];
  uint8_t                 bitmapLength;
  #if (defined(PROCESSOR_QE128) || defined(PROCESSOR_MC1323X))
  uint8_t                 bitmap[32];
  #else
  uint8_t                 bitmap[15];
  #endif
}queryNextImgResp_t;
```

The *status* field indicates whether the application can upload the requested image to the client. It may have one of the three following values:

1. *gNWSuccess_c* if the download process can proceed.
2. *gFSLProfile_NotImageAvailable* if no suitable firmware image is available for the client
3. *gFSLProfile_NotAuthorized* if the request is being denied for security reasons

The rest of the fields are only relevant if *status* is *gNWSuccess_c*.

They are the version of firmware image to be downloaded, the total size of the new firmware image, the 16 bit CRC-CCITT of the firmware image and a bitmap, describing which flash page needs to be write-protected when flashing the new firmware (i.e. which flash page must not be overwritten).

6.2.2.3 Sending Blocks of The Image to The Client

Transmission of the firmware image is transmitted one block at the time. Clients request a specific block with a specified offset in the file and length. The server application is stateless, because each request contains all the information required to handle it.

The server application is notified of a client request for a block via a Query Next Image Request Indication message, which has the following format:

```
typedef struct fslProfileOtapServerNextBlockReqInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 fileVersion[4];
  uint8_t                 fileOffset[4];
  uint8_t                 maxDataSize;
}fslProfileOtapOrigNextBlockReqInd_t;
```

The indication message contains the client's device Id, the version of the requested firmware file version, the offset of the requested block and a maximum block size it can accept.

To respond to the client, the application should set the *cmdType* parameter to *gFslProfileOtapImageBlockRespCmd* and fill out the fields of the *ImageBlockResp* structure, which has the following format:

```
typedef struct imageBlockResp_tag
{
  nextBlockStatusCode_t   status;
  uint8_t                 fileOffset[4];
  uint8_t                 dataSize;
  uint32_t                addressImage;
  uint8_t*                pData;
}imageBlockResp_t;
```

The *status* may have one of the two following values:

1. *gNWSuccess_c* if the application has the requested image block available
2. *gFSLProfile_BlockStateAbort* if the application wishes to abort the download process

If *status* is *gNWSuccess_c* the following fields describe the file offset of the block (must be copied from the Image Block Request Indication message), the size of the block (must not be greater than the maximum data size from the indication, may be less) and a pointer to the actual block.

6.2.2.4 Finishing The Upgrade Process

After a client has downloaded an entire image it reports this to the server. The application is informed via an OTAP Server Upgrade End Request Indication message, which has the following format:

```
typedef struct fslProfileOtapServerUpgradeEndReqInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 status;
  uint8_t                 fileVersion[4];
}fslProfileOtapServerUpgradeEndReqInd_t;
```

The fields are the device Id of the client, a status which can only be *gNWSuccess_c* (reserved for future use) and the new firmware file version the client is using. The server application must respond with an upgrade end response message by setting *cmdType* to *gFslProfileOtapUpgradeEndRespCmd* and filling out the fields of the *upgradeEndResp* union member, which has the following format:

```
typedef struct upgradeEndResp_tag
{
  uint8_t                 delayUntilUpgrade[4];
}upgradeEndResp_t;
```

The server can delay the client from booting with the new firmware by specifying a delay time until upgrade other than zero. The *delayUntilUpgrade* is in milliseconds and is in little endian format.

6.3 OTAP Client Overview

The OTAP client downloads a new firmware image from an OTAP server and writes it to external memory. A special bootloader is required to FLASH the new firmware. The entire download process is highly automated. A single function call initiates it and a single confirmation message is received upon completion.

6.3.1 Receiving New Image Notifications

Whenever the server announces a new image via an image notification frame, the recipient OTAP client profile layer inform the application via an OTAP Client Image Notify Indication message, provided the jitter check described in Section 6.2.2.1 is passed. The OTAP Client Image Notify Indication message has the following format:

```
typedef struct fslProfileOtapClientImageNotifyInd_tag
{
  uint8_t                 deviceId;
  uint8_t                 fileVersion[4];
} fslProfileOtapClientImageNotifyInd_t;
```

The client application is informed of the OTAP server's device Id and of the new firmware version.

6.3.2 Downloading a New Image

The client application may decide to request downloading a new image from the server at any time. A good time to do this is after receiving an OTAP Client Image Notify Indication message.

To start the download process, the application should call *FSLProfile_OtapQueryNextImageRequest*, which has the following prototype:

```
uint8_t FSLProfile_OtapQueryNextImageRequest(
                    uint8_t deviceId,
                    const uint8_t fileVersion[4],
                    const uint8_t hardwareVersion[2]
);
```

Its parameters are the device Id of the OTAP server, the client's current firmware version and the client's hardware version. If the function call returns an error, the process is aborted. If the function call returns *gNWSuccess_c* the application must expect reception of an OTAP Client Query Next Image Confirm message, which has the following structure:

```
typedef struct fslProfileOtapClientQueryNextImgCnf_tag
{
  uint8_t                 status;
  uint8_t                 delayUntilUpgrade[4];
}fslProfileOtapClientQueryNextImgCnf_t;
```

The *status* fields informs the application about the result of the download process. It can have the following values:

1. *gNWSuccess_c* - the new image has been downloaded and is ready for booting
2. *gFSLProfile_NotImageAvailable* - the server has reported that no suitable image is available for download
3. *gFSLProfile_NotAuthorized* - the server has refused image transfer on security grounds
4. *gFSLProfile_BlockStateAbort* - the server has aborted the transfer process
5. *gNWAborted_c* - the downloaded image has an invalid CRC
6. any network layer error code - an error occured at the network layer

If the status is *gNWSuccess_c* the application should delay restarting for the duration specified in the *delayUntilUpgrade* parameter. The duration is in milliseconds and is in little endian format.