NXP Semiconductors
User's Guide

Document Number: 802154MPDAUG
Rev. 4, 09/2016

IEEE 802.15.4 Media Access Controller (MAC) Demo Applications

Contents

1. Introduction

The scope of this document is the Connectivity Framework software used to ensure portability across the ARM®-based microcontrollers portfolio of the Freescale connectivity stacks. The Demo applications are designed for use with:

- The MKW2x System-In-Package based on the Kinetis ARM Cortex®-M4 32bit MCU.
- The split solution based on the Kinetis ARM Cortex-M4/M0+ 32bit MCU and MCR20A transceiver (simply referred as MCR20).
- The MKW01 System-In-Package based on the Kinetis ARM Cortex-M0+ 32bit MCU.
- The KW4x System-On-Chip based on the Kinetis ARM Cortex-M0+ 32bit MCU.

1.1. Audience

This document is intended for application developers building IEEE 802.15.4™/ZigBee® applications.

1.2. References

Table 1. References

Title	Reference	Version	Location
802.15.4-2006 - IEEE Standard for Information technology - Local and metropolitan area networks - Specific requirements - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)	IEEE Std 802.15.4-2006	2006	*ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11161*
IEEE Standard for Local and metropolitan area networks - Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)	IEEE Std 802.15.4-2011(Revision of IEEE Std 802.15.4-2006)	2011	*ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6012485*
IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer	IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)	2012	*ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6185525*
IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Physical Layer (PHY) Specifications for Low-Data-Rate, Wireless, Smart Metering Utility Networks	IEEE Std 802.15.4g-2012 (Amendment to IEEE Std 802.15.4-2011)	2012	*ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6190698*
KINETIS 802.15.4 MACPHY Application Programming Interface	802154MPAPIRM	2015	—
Connectivity Framework Reference Manual	CONNFWKRM	2015	—
ZigBee Security Services Specification	—	V.092	—

1.3. Acronyms and abbreviations

Table 2. Acronyms and abreviations

Acronym / term	Description
Beacon	A special data packet transmitted by an 802.15.4 coordinator. This packet provides the information necessary for a new device to join the network. When transmitted periodically (in a beacon network), beacons can be used to synchronize the attached devices, to identify the PAN and to describe the structure of the superframe.
BI	Beacon interval: a special data packet transmitted by an 802.15.4 Full Function Device. This packet provides the information necessary for a new device to join the network. When transmitted periodically (in a beacon network), beacons can be used to synchronize the attached devices, to identify the PAN and to describe the structure of the superframe.
BO	Beacon Order: one (1) byte variable used to compute BI.
CAP	Contention Access Period is a predefined period of time (in seconds), immediately following a beacon transmission, during which communication may occur between a coordinator (which transmitted the beacon) and other devices (which received the beacon). This implies that the CAP must be less than the Beacon Interval. The CAP is determined by the Superframe Order.
GUI	Graphical User Interface
MAC	Medium Access Control
MCPS	MAC common part sublayer
MCU	MicroController Unit
MLME	MAC sublayer management entity
NVM	None-Volatile Memory
OTA	Over-the-Air
PAN	Personal Area Network
PC	Personal Computer
PCB	Printed Circuit Board
PIB	PAN Information Base
SAP	Service Access Point
SO	SuperFrame Order: one (1) byte variable used to compute CAP.

2. Common Features for All Applications

2.1. System overview

Project names with ending in ‘(Coordinator)’ are demonstrating coordinator behavior and project names with ending in ‘(End Device)’ demonstrate device behavior.

The OTAP Server demo application demonstrates coordinator behavior, and the OTAP Client demo application, demonstrates device behavior.

System setup consists of one PAN Coordinator and one to four End Devices as shown in the figure below.

The serial connections with PCs are for displaying different status messages and for sending text messages to the devices in the network. The serial connections are optional because the network can function in autonomous mode.

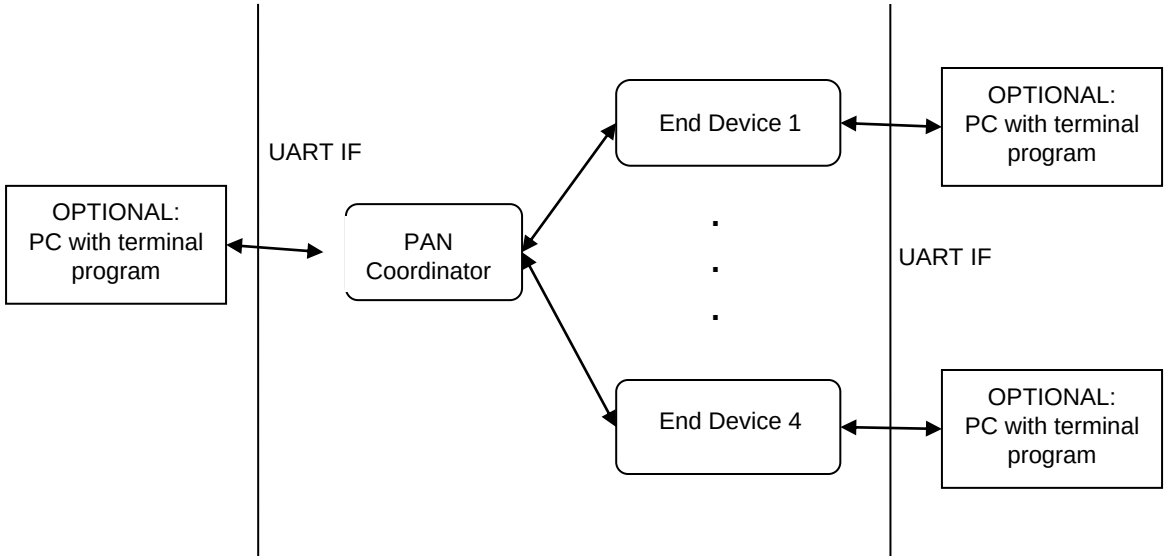


Figure 1. System overview

2.2. Creating the network

Create the network by powering on the boards. One of the boards must be running the PAN Coordinator firmware. After power-up, a switch must be pressed in order to start the PAN Coordinator. When the starting phase is finished, the PAN Coordinator prints the following messages about the characteristics of the network:

- RF channel used
- PAN ID
- Short address

The PAN Coordinator also prints a message every time an End Device associates.

NOTE

The OTA Programming Server Demo Application does not print anything on the terminal, and it should only be used with the Freescale Test Tool application.

The End Device prints the following messages when it successfully associates to the PAN Coordinator:

- PAN Coordinator address
- PAND ID
- RF channel
- Beacon information
- Link Quality Indicator (LQI)
- Short address received

Data can be exchanged after the first End Device is associated to the PAN Coordinator. The network dynamically extends when new End Devices are associated.

2.3. Monitoring messages using the serial port

All MAC demo applications except the “OTAP Server Demo Application” use the serial interface to output messages. To monitor these messages, the boards must be connected to a PC through the serial interface.

Before attempting to monitor messages, ensure that a terminal program is properly configured as follows (TeraTerm is used as an example here):

1. Launch the terminal program. Select the COM port to which the board is connected. Configure the Port Settings communication options as shown below:

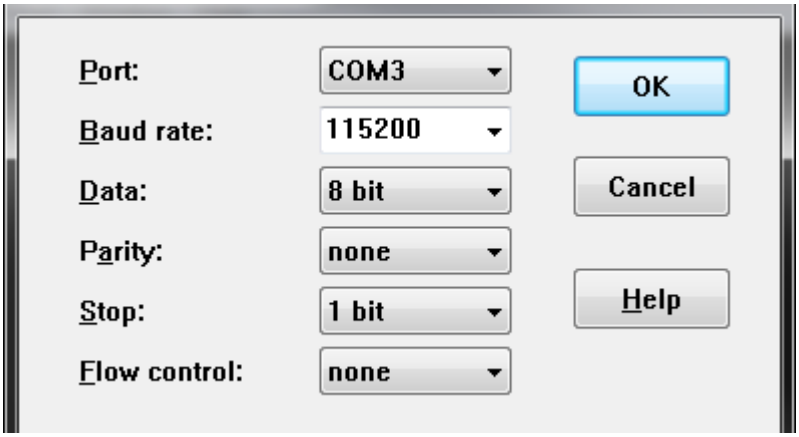


Figure 2. Properties window

NOTE

The default communication speed of the MAC demo applications is 115200 baud, except the “My Star Network App Demo” where the default communication speed is 57600 baud.

2. Click the “OK” button and the terminal main window appears as shown below:

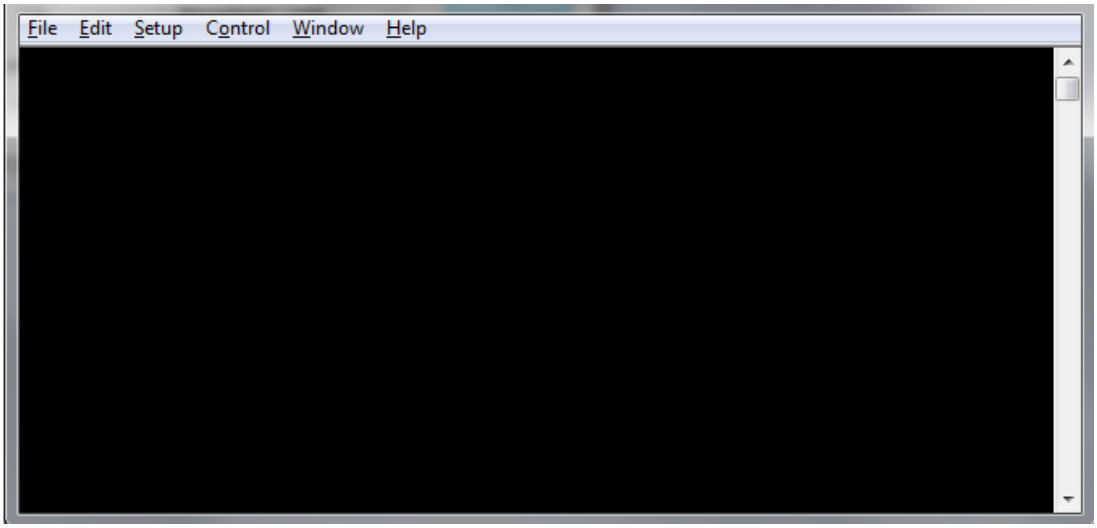


Figure 3. Terminal main window

3. Select the following terminal options as shown in Figure 4:
   - Send line ends with line feeds (CR+LF)
   - Echo typed characters locally
   - Leave the other options set as shown in Figure 4.

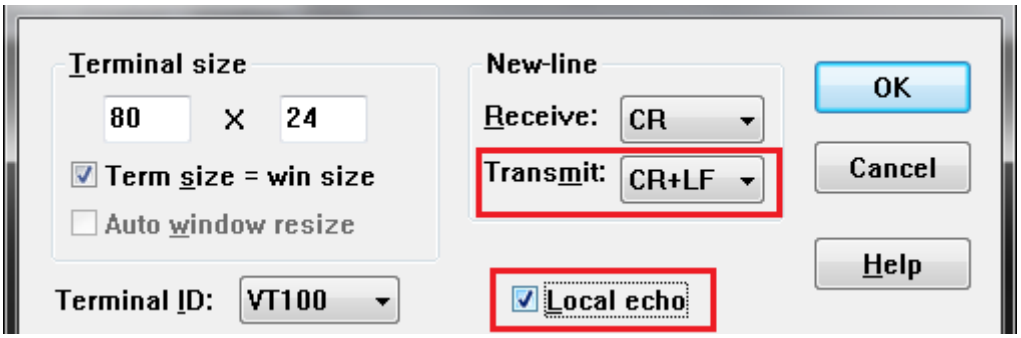


Figure 4. ASCII setup window

For example, the Coordinator is linked to COM3 and the End Device is linked to COM4. Perform the following steps:

1. Start two terminal instances, one for each of the ports used.
2. Turn on the Coordinator. The terminal assigned to COM3 outputs the following message:

```
Press any switch on board to start running the application.
```

3. After pressing a switch on the Coordinator board, terminal will output the following messages, assuming that the PAN uses the first channel in the 2.4 GHz band (channel no. 11):

```
The MyStarNetworkDemo application is initialized and ready.
Initiating the Energy Detection Scan.
Sending the MLME-Scan Request message to the MAC... Done.
Received the MLME-Scan Confirm message from the MAC.
ED scan returned the following results:
[00 00 00 00 00 00 00 00 2C B0 00 10 10 10 00 00].
Based on the ED scan the logical channel 0x0B was selected.
Starting as PAN coordinator on channel 0x0B.
PAN Coordinator started.
Sending the MLME-Start Request message to the MAC... Done.
Started the coordinator with PAN ID 0xBEEF, and short address 0xCAFE.
Ready to send and receive data over the UART.
```

4. Turn on the End Device. The appropriate terminal window outputs the following message:

```
Press any switch on board to start running the application.
```

5. After pressing a switch on the End Device board, and after it has connected to the PAN (the short address 1 has been assigned by the Coordinator), the appropriate termial window shows a message similar to the following (some values may be different):

```
Found a coordinator with the following properties:
Address. . . . . 0xCAFE
```

```
PAN ID. . . . . 0xBEEF
Logical Channel. . . .0x0B
Beacon Spec. . . . . .0xCFFF
Link Quality. . . . . 0x52
```

6. Go to the terminal session assigned to COM3 (the one connected to the Coordinator) and type some text in the window and press Enter. The typed in text is sent through the UART to the Coordinator. The Coordinator forwards it through the PAN to the End Device. The text received from the Coordinator is displayed by the End Device to the UART (check the output of COM4 in the corresponding terminal window). If there are more devices associated to the PAN, all of them receive the same message.

2.4. Software implementation

2.4.1. Initialization

The application initialization is done in App_init function shown below:

```
void App_init( void )
{
    OSA_EventCreate(&mAppEvent, kEventAutoClear);
    /* The initial application state */
    gState = stateInit;
    /* Reset number of pending packets */
    mcPendingPackets = 0;

    /* Prepare input queues.*/
    MSG_InitQueue(&mMlmeNwkInputQueue);
    MSG_InitQueue(&mMcpsNwkInputQueue);

    /* Initialize the MAC 802.15.4 extended address */
    Mac_SetExtendedAddress( (uint8_t*)&mExtendedAddress, macInstance );

    /* register keyboard callback function */
    KBD_Init(App_HandleKeys);

    /* Initialize the UART so that we can print out status messages */
    Serial_InitInterface(&interfaceId, gSerialMgrUart_c, BOARD_DEBUG_UART_INSTANCE);
    Serial_SetBaudRate(interfaceId, gUARTBaudRate115200_c);
    Serial_SetRxCallBack(interfaceId, UartRxCallBack, NULL);
```

```
    /*signal app ready*/
    Led1Flashing();
    Led2Flashing();
    Led3Flashing();
    Led4Flashing();

    Serial_Print(interfaceId, "\n\rPress any switch on board to start running the
application.\n\r", gAllowToBlock_d);
}
```

The *App_init()* function is called from the *main_task()* function, after the PHY, MAC and the entire framework was initialized:

```
        hardware_init();
        MEM_Init();
        TMR_Init();
        LED_Init();
        SerialManager_Init();
        Phy_Init();
        RNG_Init(); /* RNG must be initialized after the PHY is Initialized */
        MAC_Init();
        /* Bind to MAC layer */
        macInstance = BindToMAC( (instanceId_t)0 );
        Mac_RegisterSapHandlers( MCPS_NWK_SapHandler, MLME_NWK_SapHandler, macInstance );
        App_init();
```

After initialization and setup, the application waits for the user to press a switch on the Coordinator board. The keyboard handler, App_HandleKeys, will send the application task an gAppEvtDummyEvent_c event which will start up the application.

2.4.2. Resetting

After this point, it is always safe to reset the MAC (and thereby also the PHY) layer by using the service MLME-RESET.request as shown in the following code.

```
mlmeMessage_t mlmeReset;
/* Create and execute the Reset request */
mlmeReset.msgType = gMlmeResetReq_c;
mlmeReset.msgData.resetReq.setDefaultPib = TRUE;
(void)NWK_MLME_SapHandler( &mlmeReset, macInstance );
```

By calling this service, the MAC layer is reset and brought into the same state as it was right after having called *MAC_Init()*. The `setDefaultPib` parameter tells the MAC layer whether its PIB attributes should be set to their default value (as specified in the IEEE 802.15.4 Standard) or if they are to stay unchanged after the reset. Notice that when requesting the reset primitive, the reset call is

synchronous. That is, there is no confirmation message on the reset request and the function *NWK_MLME_SapHandler()* simply returns `gSuccess_c`. Because the call is synchronous, the message structure need not be allocated through *MEM_BufferAlloc()* but can be allocated on the stack. In all circumstances, it is the responsibility of the calling entity to de-allocate the message. Notice that it is not necessary to reset the MAC right after having called *MAC_Init()*.

The application source code from MyWirelessApp Demo Framework project does not contain any example of code for resetting a device using the MLME-RESET.request. However, it is a well-defined way of bringing back the MAC and PHY layers to a known state.

2.4.3. Implementing SAPs

Before being able to compile the project, it is necessary to create the SAP handlers that process the messages from the MAC layer to the next higher layer. For now, create empty functions and implement the functionality later as shown in this code.

```
/* The following functions are called by the MAC
   to put messages into the Application's queue.
   They need to be defined even if they are not
   used in order to avoid linker errors. */
resultType_t MLME_NWK_SapHandler(nwkMessage_t *pMsg , instanceId_t instanceId)
{
    /* This only serves as a temporary way of avoiding a compiler warning.
    Later this will be changed so that we return gSuccess_c instead. */
    return gSuccess_c;
}
resultType_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg , instanceId_t instanceId)
{
    /* This only serves as a temporary way of avoiding a compiler warning.
    Later this will be changed so that we return gSuccess_c instead. */
    return gSuccess_c;
}
```

Typically, the SAP handlers buffer the messages in a queue and then send an event to the application task to process them.

2.4.4. Starting and joining a PAN

Once the MAC, PHY and framework layers are initialized it is now safe to access the MCPS, and MLME services of the MAC layer. Start by accessing the MLME. The example code for this is found in the MyWirelessApp Demo Non Beacon project, the `App.c` source file.

The units are now ready to start up a PAN. First, set up a PAN coordinator because all IEEE 802.15.4 Standard PANs must have a PAN coordinator.

2.4.5. Running energy detection scan

The first task that a PAN coordinator must perform is to choose which radio frequency to use for its PAN. This is called choosing the logical channel. The logical channel can have a predefined value, but a better method is to choose a channel that is not used by other units. For that purpose, there are primitives that the PAN coordinator can use for scanning all (or selected) channels. This is called the energy detection scan (ED scan). The following code shows this task.

```
static uint8_t App_StartScan(macScanType_t scanType, uint8_t appInstance)
{
  mlmeMessage_t *pMsg;
  mlmeScanReq_t *pScanReq;

  Serial_Print(interfaceId,"Sending the MLME-Scan Request message to the MAC...",
gAllowToBlock_d);
  /* Allocate a message for the MLME (We should check for NULL). */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* This is a MLME-SCAN.req command */
    pMsg->msgType = gMlmeScanReq_c;
    /* Create the Scan request message data. */
    pScanReq = &pMsg->msgData.scanReq;
    /* gScanModeED_c, gScanModeActive_c, gScanModePassive_c, or gScanModeOrphan_c */
    pScanReq->scanType = scanType;
    /* ChannelsToScan */
#ifdef gPHY_802_15_4g_d
    pScanReq->channelPage = gChannelPageId9_c;
    pScanReq->scanChannels[0] = mDefaultValueOfChannel_c;
#else
    pScanReq->scanChannels = mDefaultValueOfChannel_c;
#endif

    /* Duration per channel 0-14 (dc). T[sec] = (16*960*((2^dc)+1))/1000000.
       A scan duration of 3 on 16 channels approximately takes 2 secs. */
    pScanReq->scanDuration = 3;
    /* Don't use security */
    pScanReq->securityLevel = gMacSecurityNone_c;
```

```
    /* Send the Scan request to the MLME. */
    if( NWK_MLME_SapHandler( pMsg, macInstance ) == gSuccess_c )
    {
      Serial_Print(interfaceId,"Done\n\r", gAllowToBlock_d);
      return errorNoError;
    }
    else
    {
      Serial_Print(interfaceId,"Invalid parameter!\n\r", gAllowToBlock_d);
      return errorInvalidParameter;
    }
  }
  else
  {
    /* Allocation of a message buffer failed. */
    Serial_Print(interfaceId,"Message allocation failed!\n\r", gAllowToBlock_d);
    return errorAllocFailed;
  }
}
```

In this case, the `scanType` parameter for `App_StartScan` must be set to `gScanModeED_c`. Other types of scanning are explained later in this chapter. If there is any activity on a channel at the time the PAN coordinator scans that channel, this shows up in the result of the scan. A channel that showed no sign of activity at the time of the scan, shows an energy level of approximately 0x00. The higher the number, the more activity was detected on that channel.

The previous code example scans all 16 available channels in the 2.4 GHz band. The parameter `scanChannels` is a bit mask where each bit that is set indicates that this channel must be scanned. Because the 2.4 GHz band contains channel numbers 11 to 26, bits 11 to 26 are set in the `scanChannels` bit mask. Lower channel numbers are ignored. Also, the `scanDuration` parameter tells the MAC how long to scan each channel. The numbers 0 to 14 are valid entries for this parameter. The higher the number, the longer the time spent scanning. The exact scan duration for each channel can be calculated using this equation:

$$\textbf{Scan duration} = 15.36\ \text{ms} \cdot (2^{\text{scanDuration}} + 1)$$

The `scanDuration` parameter in the example requests scanning for approximately 0.5 seconds on each of the 16 channels. Once the scan request message has successfully been sent to the MLME, the scanning commences and a scan confirmation (an `nwkMessage_t` struct with `msgType == gNwkScanCnf_c`) is received asynchronously in the SAP handler for the messages from the MLME to the NWK. The following code processes the scan confirmation message.

```
static void App_HandleScanEdConfirm(nwkMessage_t *pMsg)
{
```

```
  uint8_t n, minEnergy;
  uint8_t *pEdList;
  uint8_t Channel;
  uint8_t idx;
  uint32_t chMask = mDefaultValueOfChannel_c;

  Serial_Print(interfaceId,"Received the MLME-Scan Confirm message from the MAC\n\r",
gAllowToBlock_d);

  /* Get a pointer to the energy detect results */
  pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList;

  /* Set the minimum energy to a large value */
  minEnergy = 0xFF;

  /* Select default channel */
  mLogicalChannel = 11;

  /* Search for the channel with least energy */
  for(idx=0, n=0; n<16; n++)
  {
      /* Channel numbering is 11 to 26 both inclusive */
      Channel = n + 11;
      if( (chMask & (1 << Channel)) )
      {
          if( pEdList[idx] < minEnergy )
          {
              minEnergy = pEdList[idx];
              mLogicalChannel = Channel;
          }
          idx++;
      }
  }
  chMask &= ~(1 << mLogicalChannel);

  /* Print out the result of the ED scan */
  Serial_Print(interfaceId,"ED scan returned the following results:\n\r  [",
gAllowToBlock_d);
  Serial_PrintHex(interfaceId,pEdList, 16, gPrtHexBigEndian_c | gPrtHexSpaces_c);
```

```
  Serial_Print(interfaceId,"]\n\r\n\r", gAllowToBlock_d);

  /* Print out the selected logical channel */
  Serial_Print(interfaceId,"Based on the ED scan the logical channel 0x", gAllowToBlock_d);
  Serial_PrintHex(interfaceId,&mLogicalChannel, 1, gPrtHexNoFormat_c);
  Serial_Print(interfaceId," was selected\n\r", gAllowToBlock_d);

  /* The list of detected energies must be freed. */
  MSG_Free(pEdList);
}
```

Assuming that the scanning was successful, (`status == gSuccess_c`) and that the `nwkMessage_t` struct is pointed to by a pointer `pMsg` the `pEdList = pMsg->msgData.scanCnf.resList.pEnergyDetectList` points to a byte array of 16 bytes. One byte for each channel. `pEdList[0]` contains the result for channel 11, `pEdList[1]` contains the result for channel 12 and so on.

Once the results of the energy detection scan are received, look at the results received from all the channels. The logical channel will be the one with the minimum energy detection level from all the channels selected for scanning. Store this channel number in a global variable called m`LogicalChannel`.

Do not forget to free not only the scan confirm message, but also the data structures pointed to by `pEdList` and free `pEdList` first (unless users have a local copy of `pEdList` that they can use for freeing the list of energy levels).

NOTE

Just because the ED scan returned a result that showed no activity on a channel it does not mean that another PAN coordinator is not using this channel. It only means that no transaction(s) took place between this PAN coordinator and any of its devices while performing the ED scan. Scanning for a longer time increases the possibility that another PAN coordinator is on the channel and is detected, but it is not guaranteed.

Notice that in the application source file from project MyWirelessApp Demo Non Beacon there has been added code, as shown in the following code example, for queuing incoming MLME messages and decoupling the MLME from the application.

```
/* Application input queues */
static anchor_t mMlmeNwkInputQueue;

resultType_t MLME_NWK_SapHandler (nwkMessage_t* pMsg, instanceId_t instanceId)
{
  /* Put the incoming MLME message in the applications input queue. */
  MSG_Queue(&mMlmeNwkInputQueue, pMsg);
  OSA_EventSet(&mAppEvent, gAppEvtMessageFromMLME_c);
```

```
  return gSuccess_c;
}
```

Also, in the application task there is added code for processing incoming messages from that queue as shown in this code example.

```
/* We have an event from MLME_NWK_SapHandler */
if (events & gAppEvtMessageFromMLME_c)
{
  pMsgIn = MSG_DeQueue(&mMlmeNwkInputQueue);
  … /* Process message if pMsgIn!=NULL */
  /* Messages from the MLME must always be freed. */
  MEM_BufferFree(pMsgIn);
}
```

By implementing the MLME to NWK SAP handler this way, the MLME and NWK/APP are completely decoupled. That is, the SAP handler only queues the message but does not do any processing of the message. In this way, the MLME returns from the call as fast as possible and the call stack of the MCU is exercised as little as possible reducing the risk of getting a call stack overflow.

2.4.6. Choosing short address

Now the coordinator must assign itself a short address. All Freescale development boards come with a pre-assigned extended address, but a short address must be assigned before starting the PAN. Otherwise, the start request will fail. Because the PAN coordinator is the first unit to participate in its own PAN, it can choose any short address for itself. Once the short address is chosen, it is usually hard-coded, it must be set by setting the macShortAddress PIB attribute. This is done in `App_StartCoordinator()`, shown in the following example code.

```
    /* We must always set the short address to something
       else than 0xFFFF before starting a PAN. */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
    pMsg->msgData.setReq.pibAttributeValue = (uint8_t *)&mShortAddress;
    (void)NWK_MLME_SapHandler( pMsg, macInstance );
```

In the above code example, the short address of the PAN coordinator is set to the value of `maShortAddress`. As the reset request, the MLME-SET.request is also completed synchronously. So, if `ret == gSuccess_c` after calling *NWK_MLME_SapHandler()*, the PIB attribute was set successfully. The `pibAttributeValue` parameter is not freed by the MLME.

The short address must be different from 0xFFFF. A short address of 0xFFFE tells the coordinator to use its long address in all transactions. If the short address is set to anything different from 0xFFFF and 0xFFFE, the PAN coordinator uses this address instead of its extended address and thus sends shorter packets. An extended address is 8 bytes long and a short address is 2 bytes long.

2.4.7. Choosing PAN ID

After selecting the logical channel, the last thing required before starting a PAN, is for the coordinator to select an identification number which is called PAN ID. Assuming that there is no other PAN using the same logical channel, the new PAN coordinator can freely choose a PAN ID. However, users may want to perform an active scan (see section Running Active Scan) of the channel to see if there really are no other PAN coordinators using the same logical channel. If so, the PAN ID used must be different from the ones used by other PAN coordinators on the same logical channel. In the application source file from MyWirelessApp Demo Non Beacon (Coordinator) project it is assumed that no other IEEE 802.15.4 PAN is present.

2.4.8. Starting a PAN

After choosing the logical channel, PAN ID and short address, it is time to start up the PAN using the MLME_START.request primitive.

The `panCoordinator` parameter indicates whether the start request is to start up a PAN coordinator or a coordinator. For an explanation of the difference, see the IEEE 802.15.4 Standard. In this example a PAN coordinator is started.

The `beaconOrder` and `superFrameOrder` parameters are set to 0x0F because a non-beacon network is started. See the 802.15.4 Standard about how to start a beacon network. Any combination of `beaconOrder` and `superFrameOrder` where `beaconOrder` is set to 0x0F creates a non-beacon network.

The `batteryLifeExt` parameter is ignored for now and is set to 0x00.

The `coordRealignment` parameter tells the MLME whether it should send out a coordinator realignment command prior to starting up the PAN. For now, users should set this to 0x00 (no coordinator realignment command) because it is not relevant for this example.

Finally, the `beaconSecurityLevel` parameter tells the MLME if it should apply any security to the transactions taking place over the air. For now, users should leave this set to *gMacSecurityNone_c*.

App_StartCoordinator() in MyWirelessApp Demo (Coordinator) application contains the code for starting a PAN.

```
/* Message for the MLME will be allocated and attached to this pointer */
  mlmeMessage_t *pMsg;
  /* Value that will be written to the MAC PIB */
  uint8_t value;
  /* Pointer which is used for easy access inside the allocated message */
  mlmeStartReq_t *pStartReq;


  Serial_Print(interfaceId,"Sending the MLME-Start Request message to the MAC...",
gAllowToBlock_d);


  /* Allocate a message for the MLME (check for NULL). */
```

```
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {

    /* Set-up MAC PIB attributes. Note that Set, Get,
       and Reset messages are not freed by the MLME. */

    /* Initialize the MAC 802.15.4 extended address */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMacPibExtendedAddress_c;
    pMsg->msgData.setReq.pibAttributeValue = (uint8_t *)&mExtendedAddress;
    (void)NWK_MLME_SapHandler( pMsg, macInstance );

    /* We must always set the short address to something
       else than 0xFFFF before starting a PAN. */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
    pMsg->msgData.setReq.pibAttributeValue = (uint8_t *)&mShortAddress;
    (void)NWK_MLME_SapHandler( pMsg, macInstance );

    /* We must set the Association Permit flag to TRUE
       in order to allow devices to associate to us. */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMPibAssociationPermit_c;
    value = TRUE;
    pMsg->msgData.setReq.pibAttributeValue = &value;
    (void)NWK_MLME_SapHandler( pMsg, macInstance );

    /* This is a MLME-START.req command */
    pMsg->msgType = gMlmeStartReq_c;

    /* Create the Start request message data. */
    pStartReq = &pMsg->msgData.startReq;
    /* PAN ID - LSB, MSB. The example shows a PAN ID of 0xBEEF. */
    FLib_MemCpy(&pStartReq->panId, (void *)&mPanId, 2);
    /* Logical Channel - the default of 11 will be overridden */
    pStartReq->logicalChannel = mLogicalChannel;
#ifdef gPHY_802_15_4g_d
```

```
    pStartReq->channelPage = gChannelPageId9_c;
#endif
    /* Beacon Order - 0xF = turn off beacons */
    pStartReq->beaconOrder = 0x0F;
    /* Superframe Order - 0xF = turn off beacons */
    pStartReq->superframeOrder = 0x0F;
    /* Be a PAN coordinator */
    pStartReq->panCoordinator = TRUE;
    /* Dont use battery life extension */
    pStartReq->batteryLifeExtension = FALSE;
    /* This is not a Realignment command */
    pStartReq->coordRealignment = FALSE;
    pStartReq->startTime = 0;

    /* Don't use security */
    pStartReq->coordRealignSecurityLevel = gMacSecurityNone_c;
    pStartReq->beaconSecurityLevel       = gMacSecurityNone_c;

    /* Send the Start request to the MLME. */
    if(NWK_MLME_SapHandler( pMsg, macInstance ) != gSuccess_c)
    {
      /* One or more parameters in the Start Request message were invalid. */
      Serial_Print(interfaceId,"Invalid parameter!\n\r", gAllowToBlock_d);
      return errorInvalidParameter;
    }
  }
  else
  {
    /* Allocation of a message buffer failed. */
    Serial_Print(interfaceId,"Message allocation failed!\n\r", gAllowToBlock_d);
    return errorAllocFailed;
  }
```

A start confirmation is received asynchronously on the SAP that handles the messages from the MLME to the NWK. The application task is notified by sending to it the *gAppEvtMessageFromMLME_c* event. If the PAN was started successfully, a status code of `gSuccess_c` is returned. The MLME-START.confirm message is processed in the state `stateStartCoordinatorWaitConfirm`, as shown in the following code:

```
case stateStartCoordinatorWaitConfirm:
```

```
  /* Stay in this state until the Start confirm message
  arrives, and then goto the Listen state. */
  if (ev & gAppEvtMessageFromMLME_c)
  {
      if (pMsgIn)
      {
          ret = App_WaitMsg(pMsgIn, gMlmeStartCnf_c);
          if(ret == errorNoError)
          {
              Serial_Print(interfaceId,"Started the coordinator with PAN ID 0x",
gAllowToBlock_d);
              Serial_PrintHex(interfaceId,(uint8_t *)&mPanId, 2, gPrtHexNoFormat_c);
              Serial_Print(interfaceId,", and short address 0x", gAllowToBlock_d);
              Serial_PrintHex(interfaceId,(uint8_t *)&mShortAddress, 2, gPrtHexNoFormat_c);
              Serial_Print(interfaceId,".\n\rReady to send and receive data over the
UART.\n\r\n\r", gAllowToBlock_d);

              gState = stateListen;
              OSA_EventSet(&mAppEvent, gAppEvtDummyEvent_c);
          }
      }
  }
  break;
```

After successfully starting a PAN, the macRxOnWhenIdle PIB is set to 0x01. This means that whenever the PAN coordinator is not transmitting a packet, it is listening for incoming packets. This is implemented this way because it is not logical to start a PAN and then not start listening for incoming association requests or beacon requests from devices performing active scans. However, if this behavior is unwanted, the macRxOnWhenIdle PIB can be set back to its default value of 0x00.

All that needs to be done on the PAN coordinator is to set the PIB attribute macAssociationPermit to 0x01 in order to ensure that devices are allowed to associate to the PAN coordinator. If this PIB is left untouched, or at any time set to 0x00, any incoming association requests from a device will be ignored by the MAC (see `App_StartCoordinator()` as shown in the following code:

```
    /* We must set the Association Permit flag to TRUE
       in order to allow devices to associate to us. */
    pMsg->msgType = gMlmeSetReq_c;
    pMsg->msgData.setReq.pibAttribute = gMPibAssociationPermit_c;
    value = TRUE;
    pMsg->msgData.setReq.pibAttributeValue = &value;
    (void)NWK_MLME_SapHandler( pMsg, macInstance );
```

As was the case when setting the macShortAddress PIB, the `ret` variable contains `gSuccess_c` if the PIB was successfully set.

In the above source code example, the allocated message `pMsg` was used repeatedly for setting PIB attributes and for starting the PAN. When setting PIB attributes the message is not freed by the MLME so it can be reused. But as soon as `pMsg` is used for requesting the startup of a PAN, the message is no longer valid in the application context. Now the message is owned and freed by the MLME. So, in `App_StartCoordinator(), pMsg` is never freed because this is done by the MLME.

2.4.9. Understanding active scan

Before associating to a non-beacon PAN coordinator, the device first needs to find a PAN coordinator that is accepting incoming association requests. For this purpose, the device must use the active scan feature. This feature is initiated much the same way as the energy detection (ED) scan and users can reuse the function `App_StartScan()` as described in "Running Energy Detection Scan". The only difference is that the `scanType` parameter must now be set to `gScanModeActive_c`.

2.4.10. Running active scan

The device could also do an ED scan prior to doing the active scan in order to estimate which logical channels would be worth doing active scan on. However, because the network that was set up by the PAN coordinator is non-beacon and thus no air-traffic is available to detect in the ED scan. Because there is no other data in the air, users should not do an ED scan on the device.

After receiving the active scan request, the MLME starts sending out beacon requests on the requested channels (in the `scanChannels` parameter) and starts listening for beacons from (PAN) coordinators for as long as indicated in the `scanDuration` parameter. Basically, sending a beacon request is like asking, *"Are there any PAN coordinators out there that have a PAN on this channel?"* and a PAN coordinator sending back a beacon means, *"Yes, I have a PAN, here is my PAN information"*.

Because the device does not know which channel the PAN coordinator chose after performing the ED scan, the device will be scanning all channels. Assuming that only the PAN coordinator that was just started is in the vicinity of the device and it did see the beacon request, a scan confirm with the *status == gSuccess_c* and *resultListSize == 1* is returned. If no answer was received, then status is *gNoBeacon_c*.

After receiving the scan confirmation, now look at the pDescBlock pointer in the scan confirmation message and to the data it contains. The `descriptorList` is an array of structures of type `panDescriptor_t` as many as are indicated in the `descriptorCount` parameter as shown in the following code example.

```
typedef MAC_STRUCT panDescriptorBlock_tag
{
    panDescriptor_t               panDescriptorList[gScanResultsPerBlock_c];
    uint8_t                       panDescriptorCount;
    struct panDescriptorBlock_tag *pNext;
} panDescriptorBlock_t;
```

```
typedef PACKED_STRUCT panDescriptor_tag
{
    uint64_t            coordAddress;
    uint16_t            coordPanId;
    addrModeType_t      coordAddrMode;
    logicalChannelId_t  logicalChannel;
    //channelPageId_t     channelPage;
#ifdef gMAC2011_d
    resultType_t        securityStatus;
#else
    resultType_t        securityFailure;
#endif
    macSuperframeSpec_t superframeSpec;
    bool_t              gtsPermit;
    uint8_t             linkQuality;
    uint8_t             timeStamp[3];
    macSecurityLevel_t  securityLevel;
    keyIdModeType_t     keyIdMode;
    uint64_t            keySource;
    uint8_t             keyIndex;
#ifdef gMAC2011_d
    uint8_t*            pCodeList;
#endif
} panDescriptor_t;
```

The scan confirm can contain up to two (2) PAN descriptor blocks.

Using the list of `panDescriptor_t` structure, the device can decide which coordinator to associate to. The MyWirelessApp Demo (End Device) is a non beacon without security demo application, therefore does not use the following parameters: securityLevel, keyIdMode, keySource, keyIndex, securityFailure, gtsPermit and timeStamp.

The `coordAddrMode` denotes whether the coordinator is using its extended address or is using a short address. If set to `gAddrModeShort_c`, the coordinator uses a 16 bit short address and only the first two bytes in `coordAddress` are valid and contain (in little endian mode) the short address of the PAN coordinator. If `coordAddrMode` is set to `gAddrModeLong_c` all 8 bytes of the `coordAddress` parameter are valid and contain (in little endian mode) the full address of the PAN coordinator.

The `logicalChannel` denotes the channel where the PAN coordinator can be found. See section “Running Energy Detection Scan” for more details. The `coordPanId` contains the PAN ID of the PAN coordinator. See section “Starting a PAN.” The `linkQuality` contains a value in the range of

0x00 to 0xFF – the larger the value, the better the signal strength from the PAN coordinator. In most cases this means the closer the PAN coordinator is to the device.

2.4.10.1. Understanding the SuperFrame

The last parameter that the device must look at is the `superFrameSpec` parameter. This parameter contains the information as shown in the table below.

Table 3. SuperFrameSpec parameter contents

Bits: 0-3	4-7	8-11	12	13	14	15
Beacon order	Superframe order	Final CAP slot	Battery life extension	Reserved	PAN coordinator	Association permit

The bit definition is covered later in this chapter. However, Beacon order bits and Superframe order bits correspond to the beacon order and superframe order used when starting the PAN coordinator. See section “Starting a PAN”. The PAN coordinator bit must also be set as this bit is set as shown in section “Starting a PAN”. The last bit that is required to look at is the Association permit bit. This bit indicates if the PAN coordinator will process any incoming association requests. If set, it will process the requests, otherwise the association requests are ignored. This bit follows the macAssociatePermit PIB attribute of the coordinator as shown in section “Starting a PAN”.

The following code example from MyWirelessApp Demo Non Beacon (End Device) shows how an incoming scan confirmation on an active scan is handled.

```
static uint8_t App_HandleScanActiveConfirm(nwkMessage_t *pMsg)
{
  void    *pBlock;
  uint8_t panDescListSize = pMsg->msgData.scanCnf.resultListSize;
  uint8_t rc = errorNoScanResults;
  uint8_t j;
  uint8_t bestLinkQuality = 0;

  panDescriptorBlock_t *pDescBlock = pMsg->msgData.scanCnf.resList.pPanDescriptorBlockList;
  panDescriptor_t *pPanDesc;



  /* Check if the scan resulted in any coordinator responses. */
  if (panDescListSize > 0)
  {
    /* Check all PAN descriptors. */
    while (NULL != pDescBlock)
    {
      for (j = 0; j < pDescBlock->panDescriptorCount; j++)
      {
```

```
        pPanDesc = &pDescBlock->panDescriptorList[j];

        /* Only attempt to associate if the coordinator
           accepts associations and is non-beacon. */
        if( ( pPanDesc->superframeSpec.associationPermit ) &&
            ( pPanDesc->superframeSpec.beaconOrder == 0x0F) )
        {

          /* Find the nearest coordinator using the link quality measure. */
          if(pPanDesc->linkQuality > bestLinkQuality)
          {
            /* Save the information of the coordinator candidate. If we
               find a better candiate, the information will be replaced. */
            FLib_MemCpy(&mCoordInfo, pPanDesc, sizeof(panDescriptor_t));
            bestLinkQuality = pPanDesc->linkQuality;
            rc = errorNoError;
          }
        }
      }

      /* Free current block */
      pBlock = pDescBlock;
      pDescBlock = pDescBlock->pNext;
      MSG_Free(pBlock);
    }
  }

  if (pDescBlock)
      MSG_Free(pDescBlock);

  return rc;
}
```

As this code example shows, the device stores the PAN descriptor for the chosen coordinator in a global variable called `mCoordInfo`. The contents of this PAN descriptor will be used in the next section when the device requests to be associated with the coordinator.

Remember to free not only the scan confirm message (this is done in `AppTask()` in the example application) but also the data structures pointed to by `pMsg->msgData.scanCnf.resList.pPanDescriptorBlocks`.

2.4.11. Associating to a PAN

From the PAN descriptor that was returned by the PAN coordinator, users can see if the coordinator accepts the incoming association requests, its short address and PAN ID. Next, users can associate the device to the PAN coordinator. Use the association request message (see App.c from MyWirelessApp Demo (End Device)) to accomplish this as shown in the following code example.

```
static uint8_t App_SendAssociateRequest(void)
{
  mlmeMessage_t *pMsg;
  mlmeAssociateReq_t *pAssocReq;

  Serial_Print(interfaceId, "Sending the MLME-Associate Request message to the MAC...",
gAllowToBlock_d);

  /* Allocate a message for the MLME message. */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* This is a MLME-ASSOCIATE.req command. */
    pMsg->msgType = gMlmeAssociateReq_c;
    /* Create the Associate request message data. */
    pAssocReq = &pMsg->msgData.associateReq;

    /* Use the coordinator info we got from the Active Scan. */
    FLib_MemCpy(&pAssocReq->coordAddress, &mCoordInfo.coordAddress, 8);
    FLib_MemCpy(&pAssocReq->coordPanId,   &mCoordInfo.coordPanId, 2);
    pAssocReq->coordAddrMode      = mCoordInfo.coordAddrMode;
    pAssocReq->logicalChannel     = mCoordInfo.logicalChannel;
    pAssocReq->securityLevel      = gMacSecurityNone_c;
#ifdef gPHY_802_15_4g_d
    pAssocReq->channelPage = gChannelPageId9_c;
#else
    pAssocReq->channelPage = gDefaultChannelPageId_c;
#endif
    /* We want the coordinator to assign a short address to us. */
    pAssocReq->capabilityInfo     = gCapInfoAllocAddr_c;

    /* Send the Associate Request to the MLME. */
    if(NWK_MLME_SapHandler( pMsg, macInstance ) == gSuccess_c)
```

```
    {
      Serial_Print(interfaceId, "Done\n\r", gAllowToBlock_d);
      return errorNoError;
    }
    else
    {
      /* One or more parameters in the message were invalid. */
      Serial_Print(interfaceId, "Invalid parameter!\n\r", gAllowToBlock_d);
      return errorInvalidParameter;
    }
  }
  else
  {
    /* Allocation of a message buffer failed. */
    Serial_Print(interfaceId, "Message allocation failed!\n\r", gAllowToBlock_d);
    return errorAllocFailed;
  }
}
```

Most of the parameters in the association request message were described in the previous sections. One parameter that is new is `capabilityInfo`.

Table 4. capabilityInfo parameter

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4-5	Bit 6	Bit 7
Alternate PAN coordinator	Device type	Power source	Receiver on when idle	Reserved	Security capability	Allocate address

Table 5. capabilityInfo field definitions

Field Name	Setting
Alternate PAN coordinator	1 = If device is capable of being a PAN coordinator 0 = If device is not capable of being a PAN coordinator
Device Type	1 = If device is a Reduced Function Device (RFD) 0 = If device is not an Reduced Function Device (RFD)
Power Source	1 = If device is receiving power from AC power 0 = If device is not receiving power from AC power
Receiver on when idle	1= If device does not disable its receiver during idle periods 0 = If device does disable its receiver during idle periods
Reserved	
Security Capability	1 = If device is capable of sending and receiving MAC frames using the security suite. 0 = If device is not capable of sending and receiving MAC frames using the security suite. Field is one bit in length.
Allocate Address	1 = If device wants the coordinator to allocate a short address as a result of the association procedure. 0 = The special short address (0xFFFE) is allocated to the device and returned

	through the association response command. In this case, the device communicates on the PAN using only its 64 bit extended address. Field is one bit in length.

In this example, the device that users are trying to associate will only have the allocate address subfield set to 1, all others are set to 0. The capabilityInfo parameter is set to `gCapInfoAllocAddr_c` (0x80).

After sending the association request message to the MLME, an association request is sent from the device to the PAN coordinator. Assuming that the PAN coordinator receives, accepts, and responds positively to the association request by sending an association response as described in section "Adding a Device to the PAN", the device receives an association confirmation with status `gSuccess_c` and the `assocShortAddress` parameter set to the short address that the PAN coordinator has assigned to the device, as shown in the following example code.

```
static uint8_t App_HandleAssociateConfirm(nwkMessage_t *pMsg)
{
  /* If the coordinator assigns a short address of 0xfffe then,
     that means we must use our own extended address in all
     communications with the coordinator. Otherwise, we use
     the short address assigned to us. */
  if ( pMsg->msgData.associateCnf.status == gSuccess_c)
  {

        if( pMsg->msgData.associateCnf.assocShortAddress >= 0xFFFE)
        {
          mAddrMode = gAddrModeExtendedAddress_c;
          FLib_MemCpy(maMyAddress, (void*)&mExtendedAddress, 8);
        }
        else
        {
          mAddrMode = gAddrModeShortAddress_c;
          FLib_MemCpy(maMyAddress, &pMsg->msgData.associateCnf.assocShortAddress, 2);
        }

        return gSuccess_c;
  }

  else
  {
       return pMsg->msgData.associateCnf.status;
  }
}
```

As the example code shows, the address assigned by the coordinator is stored in `mAddress` and the addressing mode is stored in `mAddrMode`. If the coordinator assigns the short address 0xFFFE to the device, the device must always use its extended (8 byte) address.

In this particular scenario there is no checking on the `pMsg->msgData.associateCnf.status` parameter because the coordinator always accepts incoming association requests. However, in a more extensive application it would be necessary to check on this parameter.

2.4.12. Adding a device to the PAN

Until now examples have ignored what happens on the PAN coordinator when the association request from the device is received and passed up to the application as an association indication. The association indication contains the extended device address (in the `deviceAddress` parameter) of the associating device and a copy of the capability information that the device passed to the MLME in the association request message. The capability information is stored in the `capabilityInfo` parameter as described in section “Associating to a PAN”.

As the allocate address bit is set in the `capabilityInfo` parameter, the PAN coordinator must assign a short address to the device. Because the PAN coordinator is the only unit in the PAN that is allowed to assign short addresses to a device it can freely choose anything in the 0x0000-0xFFFD range. 0xFFFF is an invalid short address and 0xFFFE is the short address that must be assigned to devices that do not request a short address or that always must use a long address. In this example the incoming association indication is accepted and the short address 0x0001 is chosen for the device. As shown in the following code, this message must be sent to the MLME.

```
static uint8_t App_SendAssociateResponse(nwkMessage_t *pMsgIn, uint8_t appInstance)
{
  mlmeMessage_t *pMsg;
  mlmeAssociateRes_t *pAssocRes;

  Serial_Print(interfaceId,"Sending the MLME-Associate Response message to the MAC...",
gAllowToBlock_d);
  /* Allocate a message for the MLME */
  pMsg = MSG_AllocType(mlmeMessage_t);
  if(pMsg != NULL)
  {
    /* This is a MLME-ASSOCIATE.res command */
    pMsg->msgType = gMlmeAssociateRes_c;

    /* Create the Associate response message data. */
    pAssocRes = &pMsg->msgData.associateRes;

    /* Assign a short address to the device. In this example we simply
       choose 0x0001. Though, all devices and coordinators in a PAN must have
```

```
       different short addresses. However, if a device do not want to use
       short addresses at all in the PAN, a short address of 0xFFFE must
       be assigned to it. */
    if(pMsgIn->msgData.associateInd.capabilityInfo & gCapInfoAllocAddr_c)
    {
      /* Assign a unique short address less than 0xfffe if the device requests so. */
      pAssocRes->assocShortAddress = 0x0001;
    }
    else
    {
      /* A short address of 0xfffe means that the device is granted access to
         the PAN (Associate successful) but that long addressing is used.*/
      pAssocRes->assocShortAddress = 0xFFFE;
    }
    /* Get the 64 bit address of the device requesting association. */
    FLib_MemCpy(&pAssocRes->deviceAddress, &pMsgIn->msgData.associateInd.deviceAddress, 8);
    /* Association granted. May also be gPanAtCapacity_c or gPanAccessDenied_c. */
    pAssocRes->status = gSuccess_c;
    /* Do not use security */
    pAssocRes->securityLevel = gMacSecurityNone_c;

    /* Save device info. */
    FLib_MemCpy(&mDeviceShortAddress, &pAssocRes->assocShortAddress, 2);
    FLib_MemCpy(&mDeviceLongAddress,  &pAssocRes->deviceAddress,     8);

    /* Send the Associate Response to the MLME. */
    if( gSuccess_c == NWK_MLME_SapHandler( pMsg, macInstance ) )
    {
      Serial_Print( interfaceId,"Done\n\r", gAllowToBlock_d );
      return errorNoError;
    }
    else
    {
      /* One or more parameters in the message were invalid. */
      Serial_Print( interfaceId,"Invalid parameter!\n\r", gAllowToBlock_d );
      return errorInvalidParameter;
    }
  }
```

```
  else
  {
    /* Allocation of a message buffer failed. */
    Serial_Print(interfaceId,"Message allocation failed!\n\r", gAllowToBlock_d);
    return errorAllocFailed;
  }
}
```

The coordinator can also send associate response to the device with the `status` parameter set to `gPanAccessDenied_c` or `gPanAtCapacity_c`. If the PAN coordinator does not want to see any incoming association requests, it can set the PIB attribute macAssociatePermit to 0x00 (FALSE). See section “Starting a PAN” for details. As was the case for the device, the PAN coordinator must store the extended 8 byte address of the device contained in the `deviceAddress` parameter so that it can later disassociate from the device if it needs to.

As an indication of a successful association procedure the coordinator receives a MLME-COMM-STATUS.indication message which must be managed as shown in the following code example.

```
static uint8_t App_HandleMlmeInput(nwkMessage_t *pMsg, uint8_t appInstance)
{
  if(pMsg == NULL)
    return errorNoMessage;

  /* Handle the incoming message. The type determines the sort of processing.*/
  switch(pMsg->msgType) {
  case gMlmeAssociateInd_c:
    Serial_Print(interfaceId,"Received an MLME-Associate Indication from the MAC\n\r",
gAllowToBlock_d);
    /* A device sent us an Associate Request. We must send back a response.  */
    return App_SendAssociateResponse(pMsg, appInstance);

  case gMlmeCommStatusInd_c:
    /* Sent by the MLME after the Association Response has been transmitted. */
    Serial_Print(interfaceId,"Received an MLME-Comm-Status Indication from the MAC\n\r",
gAllowToBlock_d);
    break;
  }
  return errorNoError;
}
```

This concludes the association procedure for the PAN coordinator and completes the association procedure.

The example code does not expect that the association procedure fails and does not check the status code of the MLME-COMM-STATUS.indication. However, the MLME-COMM-STATUS.indication also contains other parameters such as the addresses of the coordinator and the device, their addressing modes, and the PAN ID used by the association procedure. See the *802.15.4 MAC/PHY Software Reference Manual* (document 802154MPSRM) for more information.

2.4.13. Data transfer

Now that the PAN coordinator and the device are associated, data can be transferred between these units.

In IEEE 802.15.4 Standard PANs most communications are driven by the devices in a network. The devices are typically battery powered and must be able to control the data flow in order to optimize battery life. This is accomplished by the device polling for data from the coordinator and transmitting the data directly to the coordinator. The coordinator only sends data to a device when it knows it is listening, for example, when a device has requested data.

2.4.13.1. Direct data

In many user scenarios, there will be one PAN coordinator with several devices associated to it and the PAN coordinator will be powered with AC power, where the devices will be powered by a battery. Also, the devices will decide when to transfer data. Therefore, the normal set up is to have the PAN coordinator listen for data all the time (except when it is transmitting) and have the devices initiate all data transfers.

This behavior is implemented in the PAN coordinator by setting the macRxOnWhenIdle PIB attribute to 0x01 (TRUE - the default value is 0x00 (FALSE) which corresponds to not listening). This PIB is automatically set to 0x01 when the coordinator starts the PAN. See section “Starting a PAN” for details.

Sending data using the MCPS interface is achieved similar to sending commands on the MLME interface. However, there is one major difference. The data may be required to be sent using high bandwidth. On the MLME interface, there can only be one outstanding command packet at a time. Only when a confirmation message has been received is the application allowed to send a new command. However, on the MCPS interface, there can be multiple outstanding data packets (as many as can be allocated by the application) at a time, keeping the throughput at a maximum.

In non-beacon mode, direct data, as opposed to indirect data, is sent immediately and is sent from the device to the coordinator. The device can send any time because the coordinator is always listening. The coordinator is AC powered and does not need to conserve battery power.

As shown in the following code example, App.c file from MyWirelessApp Demo (End Device) shows how to send data from the device to the coordinator and achieve maximum throughput while sending data.

Example 1. MCPS-DATA.Request example

```
static void App_TransmitUartData(void)

{

  static uint8_t keysBuffer[mMaxKeysToReceive_c];
```

```
  static uint8_t keysReceived = 0;
  uint16_t count = 0;

  /* get data from UART */
  if( keysReceived < mMaxKeysToReceive_c )
  {
    (void)Serial_GetByteFromRxBuffer(interfaceId, &keysBuffer[keysReceived], &count);
    if( count )
    {
      keysReceived++;
    }
  }
       /* Use multi buffering for increased TX performance. It does not really
     have any effect at a UART baud rate of 19200bps but serves as an
     example of how the throughput may be improved in a real-world
     application where the data rate is of concern. */
  if( (mcPendingPackets < mDefaultValueOfMaxPendingDataPackets_c) && (mpPacket == NULL) )
  {
    /* If the maximum number of pending data buffes is below maximum limit
       and we do not have a data buffer already then allocate one. */
    mpPacket = MSG_Alloc(sizeof(nwkToMcpsMessage_t) + gMaxPHYPacketSize_c);
  }
 if(mpPacket != NULL)
  {
    /* get data from UART */
      mpPacket->msgData.dataReq.pMsdu = &keysBuffer[0];
      /* Data was available in the UART receive buffer. Now create an
         MCPS-Data Request message containing the UART data. */
      mpPacket->msgType = gMcpsDataReq_c;
      /* Create the header using coordinator information gained during
         the scan procedure. Also use the short address we were assigned
         by the coordinator during association. */
      FLib_MemCpy(&mpPacket->msgData.dataReq.dstAddr, &mCoordInfo.coordAddress, 8);
      FLib_MemCpy(&mpPacket->msgData.dataReq.srcAddr, &maMyAddress, 8);
      FLib_MemCpy(&mpPacket->msgData.dataReq.dstPanId, &mCoordInfo.coordPanId, 2);
      FLib_MemCpy(&mpPacket->msgData.dataReq.srcPanId, &mCoordInfo.coordPanId, 2);
      mpPacket->msgData.dataReq.dstAddrMode = mCoordInfo.coordAddrMode;
      mpPacket->msgData.dataReq.srcAddrMode = mAddrMode;
```

```
      mpPacket->msgData.dataReq.msduLength = keysReceived;
      /* Request MAC level acknowledgement of the data packet */
      mpPacket->msgData.dataReq.txOptions = gMacTxOptionsAck_c;
      /* Give the data packet a handle. The handle is
         returned in the MCPS-Data Confirm message. */
      mpPacket->msgData.dataReq.msduHandle = mMsduHandle++;
      /* Don't use security */
      mpPacket->msgData.dataReq.securityLevel = gMacSecurityNone_c;

      /* Send the Data Request to the MCPS */
      (void)NWK_MCPS_SapHandler(mpPacket, macInstance);

      /* Prepare for another data buffer */
      mpPacket = NULL;
      mcPendingPackets++;
      /* Receive another pressed keys */
      keysReceived = 0;
  }
      /* If the keysBuffer[] wasn't send over the air because there are too many pending
packets, */
  /* try to send it later   */
  if (keysReceived)
  {
  OSA_EventSet(&mAppEvent, gAppEvtRxFromUart_c);
  }
```

As this code example shows, the function `App_TransmitUartData()` must be called every time a new character is received from the Serial Console in order to keep the throughput at a maximum. A good place to call this function is the application task. Also, the number of outstanding data packets in the example has been limited by a constant `mDefaultValueOfMaxPendingDataPackets_c`. This limitation is a matter of design and may influence the throughput if it is set too low depending on the data transfer scenario.

Most of the parameters in the data request message have already been explained in previous sections and basically just contain addresses and addressing modes. However, the `txOptions`, `msduHandle`, and `msduLength` require some explanation.

- The `msduHandle` parameter is a unique identifier for the data packet. This identifier can be used when receiving the MCPS-DATA.confirm message to identify which data packet the MCPS-DATA.message refers to. The `msduHandle` parameter must be chosen so that it can uniquely identify the outstanding data packets. The value is typically increased by one for each data packet that is sent as shown throughout this example.

- The `msduLength` parameter indicates how many valid data bytes the data buffer pPacket->msgData.dataReq.msdu contains.
- The pPacket->msgData.dataReq.msdu contains the data that must be transferred, up to 102 bytes.
- The txOptions parameter is a bit mask that tells the MCPS how to transfer the data. In this example, only the data packet was required to be acknowledged along with no GTS, no indirect transmission and no security options enabled.

The `txOptions` bit mask is encoded as shown below.

Table 6. txOptions parameter encoding

Bit 0	Bit 1	Bit 2	Bit 3-7
Acknowledge required	GTS transmission	Indirect transmission	0

The data sent by the device to the coordinator is sent as soon as possible because direct transmission is used. The coordinator is always listening and must acknowledge the reception of the data packet if it was received successfully. If the coordinator was not listening, the device will get a MCPS-DATA.confirm message indicating that no acknowledge was received. However, in the application it is assumed that the data was transferred successfully and a counter that keeps track of the number of outstanding data packets is decreased.

Example 2. Handle MCPS messages (end device)

```
static void App_HandleMcpsInput(mcpsToNwkMessage_t *pMsgIn)
{
  switch(pMsgIn->msgType)
  {
    /* The MCPS-Data confirm is sent by the MAC to the network
       or application layer when data has been sent. */
  case gMcpsDataCnf_c:
    if(mcPendingPackets)
      mcPendingPackets--;
    break;

  case gMcpsDataInd_c:
    /* Copy the received data to the UART. */
    Serial_SyncWrite(interfaceId, pMsgIn->msgData.dataInd.pMsdu, pMsgIn-
>msgData.dataInd.msduLength);
    /* Since we received data, the coordinator might have more to send. We
       reduce the polling interval to raise the throughput while data is
       available. */
    mPollInterval = mDefaultValueOfPollIntervalFast_c;
    /* Allow another MLME-Poll request. */
    mWaitPollConfirm = FALSE;
    break;
```

```
  }
}
```

On the coordinator side (see MApp.c file from MyWirelessApp Demo (Coordinator)) the data that the device sent is received much the same way, only it is receiving MCPS-DATA.indications instead of MCPS-DATA.confirms. The coordinator just passes the received data on to the UART.

Example 3. Handle MCPS messages (coordinator)

```
static void App_HandleMcpsInput(mcpsToNwkMessage_t *pMsgIn, uint8_t appInstance)
{
  switch(pMsgIn->msgType)
  {
    /* The MCPS-Data confirm is sent by the MAC to the network
       or application layer when data has been sent. */
  case gMcpsDataCnf_c:
    if(mcPendingPackets)
      mcPendingPackets--;
    break;

  case gMcpsDataInd_c:
    /* The MCPS-Data indication is sent by the MAC to the network
       or application layer when data has been received. We simply
       copy the received data to the UART. */
    Serial_SyncWrite( interfaceId,pMsgIn->msgData.dataInd.pMsdu, pMsgIn-
>msgData.dataInd.msduLength );
    break;
  }
}
```

For data transfer to actually take place, the SAP handler for the MCPS interface on both the coordinator and the device must now be implemented just like the MLME SAP handler was.

Example 4. MCPS_NWK SAP example

```
resultType_t MCPS_NWK_SapHandler (mcpsToNwkMessage_t* pMsg, instanceId_t instanceId)
{
  /* Put the incoming MCPS message in the applications input queue. */
  MSG_Queue(&mMcpsNwkInputQueue, pMsg);
  OSA_EventSet(&mAppEvent, gAppEvtMessageFromMCPS_c);
  return gSuccess_c;
}
```

The MCPS SAP handler uses its own queue and does not share queue with the MLME SAP handler because the message formats are different and cannot easily be distinguished by looking at the message.

Next, the *gAppEvtMessageFromMCPS_c* event is sent to the application task to notify it that a new message has arrived in the MCPS queue. The MCPS messages are implemented in the application task like in the code below:

Example 5. Handle MCPS confirms and transmit data from UART

```
if (ev & gAppEvtMessageFromMCPS_c)
{
    /* Get the message from MCPS */
    pMsgIn = MSG_DeQueue(&mMcpsNwkInputQueue);
    if (pMsgIn)
    {
        /* Process it */
        App_HandleMcpsInput(pMsgIn);
        /* Messages from the MCPS must always be freed. */
        MSG_Free(pMsgIn);
        pMsgIn = NULL;
    }
}
```

NOTE

The MCPS message *must* be freed by the application.

2.4.13.2. Indirect data

Sending data from the coordinator to the device is different than sending data from the device to the coordinator because the device is not necessarily listening. The device is usually battery powered and may shut down its transceiver at any time. This requires that the coordinator send its data indirectly. That is, the coordinator sends its data and the data is buffered until the device polls for it. The data is not sent immediately. The data message is created and set as was shown in section “Direct Data” except from the `txOptions` parameter (see App.c file from MyWirelessApp Demo (Coordinator)) as shown in the following example code.

```
/* Request MAC level acknowledgement, and
indirect transmission of the data packet */
mpPacket->msgData.dataReq.txOptions |= gTxOptsIndirect_c;
```

The coordinator needs to set the `gTxOptsIndirect_c` bit in the `txOptions` parameter. If this bit is not set, the data is transmitted directly and immediately. This means that the data would probably be lost as the device is probably not listening. In non-beacon mode, the coordinator typically uses indirect data transmission.

By using the `gTxOptsAck_c` bit, the coordinator (and the device) will not only get a MCPS-DATA.confirm message stating that the packet was sent successfully, but they will also get a message if the data packet was not acknowledged and hence not received by the intended receiver. This allows for an opportunity to retransmit the data packet.

Because the coordinator sends its data indirectly, the device must poll the data out of the coordinator when the device decides that it is in a state where is it ready to receive and process a message from the coordinator. This is done using the MLME-POLL.request message (see MApp.c file from MyWirelessApp Demo (End Device)) as shown in the following example code.

Example 6. This is an MLME-POLL.req command.

```
    mlmeMessage_t *pMlmeMsg = MSG_AllocType(mlmeMessage_t);
    if(pMlmeMsg)
    {
      /* Create the Poll Request message data. */
      pMlmeMsg->msgType = gMlmePollReq_c;

      /* Use the coordinator information we got from the Active Scan. */
      FLib_MemCpy(&pMlmeMsg->msgData.pollReq.coordAddress, &mCoordInfo.coordAddress, 8);
      FLib_MemCpy(&pMlmeMsg->msgData.pollReq.coordPanId, &mCoordInfo.coordPanId, 2);
      pMlmeMsg->msgData.pollReq.coordAddrMode = mCoordInfo.coordAddrMode;
      pMlmeMsg->msgData.pollReq.securityLevel = gMacSecurityNone_c;

      /* Send the Poll Request to the MLME. */
      if( NWK_MLME_SapHandler( pMlmeMsg, macInstance ) == gSuccess_c )
      {
        /* Do not allow another Poll request before the confirm is received. */
        mWaitPollConfirm = TRUE;
}
```

Sending this request means that the device is asking the coordinator if there is any data available in the coordinator for the device. The coordinator sends back an answer in a MLME-POLL.response, resulting in a MLME-POLL.confirm on the device as shown in the following example code.

Example 7. Handle MLME messages

```
static uint8_t App_HandleMlmeInput(nwkMessage_t *pMsg)
{
  if(pMsg == NULL)
    return errorNoMessage;
  /* Handle the incoming message. The type determines the sort of processing.*/
  switch(pMsg->msgType) {
  case gMlmePollCnf_c:
    if(pMsg->msgData.pollCnf.status != gSuccess_c)
    {
      /* The Poll Confirm status was not successful. Usually this happens if
```

```
         no data was available at the coordinator. In this case we start
         polling at a lower rate to conserve power. */
      mPollInterval = mDefaultValueOfPollIntervalSlow_c;

      /* If we get to this point, then no data was available, and we
         allow a new poll request. Otherwise, we wait for the data
         indication before allowing the next poll request. */
      mWaitPollConfirm = FALSE;
    }
    break;
  }
  return errorNoError;
}
```

As this code example shows, a `status` parameter different from `gSuccess_c` indicates that there is no data available for the device on the coordinator. In this example, the device uses a timer for polling the coordinator at regular intervals. If the coordinator does not have any data, the polling rate is decreased until the coordinator starts transmitting data again and then the polling interval is again increased. Using this polling scheme conserves power in the device (the transceiver is on less often) but other applications may want to use another polling scheme and may choose to poll differently.

After the coordinator will send back a MLME-POLL.response that indicates that it has data queued for the device, the coordinator will, immediately after the MLME-POLL.response, send its data to the device, which in turn will receive the data packet on the MCPS interface. The application does not need to take any action on a MLME-POLL.confirm message with (`status==gSuccess_c`). This extends the `App_HandleMcpsInput()` function on the device that as shown earlier in this chapter.

As was the case for the coordinator, the device simply passes the data to the UART and completes the transparent UART example.

3. My Wireless Application

This guide provides information about creation and maintenance of a network based on the Freescale 802.15.4 Media Access Controller (MAC) implementation. The MyWirelessApp is a set of demo applications created on top of the Freescale 802.15.4 Media Access Controller (MAC).

The MyWirelessApp demo simply transfers all characters received on the End Device’s serial interface, over-the-air to the associated Coordinator, which will print the received data on its serial interface. Also, the Cordinaor will transfer indirectly all characters received over he serial interface to the associated EndDevice which will print the received data on its serial interface.

NOTE

All MyWirelessApp demo applications require a console application to monitor network activity.

The available MyWirelessApp demos are listed below:

- **MyWirelessApp Demo (coordinator):** Proper initialization of the MAC is demonstrated and

the MLME main function is called in the main loop. In this application, the coordinator sends indirect data to a device.

- **MyWirelessApp demo (Dual PAN):** In this application the coordinator starts two PANs and sends the same data to the devices associated on both PANs.
- **MyWirelessApp Demo (end device):** Proper initialization of the MAC is demonstrated and the MLME main function is called in the main loop. In this application, the device polls for data from the coordinator. The NVM and Low power features are also highlighted in this demo application.
- **MyWirelessApp Demo BE (coordinator):** This application builds upon the previous coordinator application**.** In this application, the coordinator starts a beaconed network.
- **MyWirelessApp Demo BE (end device)**: This application builds upon the previous device application. In this application, the device receives data using automatic polling.

- **MyWirelessApp Demo SE (coordinator)**: This application builds upon the previous coordinator application**.** In this application, the coordinator uses security.
- **MyWirelessApp Demo SE (end device)**: This application builds upon the previous device application. In this application, the device uses security.
- **MyWirelessApp demo (Coordinator FSCI_Host):** This application is built upon MAC FSCI App and behaves like MyWirelessApp Coordinator. In this application is demonstrated the use of 802.15.4 Software as a blackbox.
- **MyWirelessApp demo (End device FSCI_Host):** This application is built upon MAC FSCI App and behaves like MyWirelessApp end device. In this application is demonstrated the use of 802.15.4 Software as a blackbox.

3.1. MyWirelessApp

In this demo the Coordinator will run an Energy Detect Scan to choose the best channel, then it will start to listen for data from other devices (the *RxOnWhenIdle* PIB is set to TRUE at MLME-START.Request).

The End Device will run an Active Scan to find the nearest Coordinator (based on the LQI value) with which it will associate. After the End Device has joined the PAN (a short address will be assigned to it) it will periodically poll the PAN Coordinator for data (see the *mDefaultValueOfPollIntervalSlow_c* define in App.h). If data is available, the polling interval is reduced (see the *mDefaultValueOfPollIntervalFast_c* define from App.h) until all pending data is transferred.

3.2. MyWirelessApp dual PAN (coordinator)

To enable Dual PAN functionality, the *gMpmMaxPANs_c* define is set to 2 to enable the Multi PAN Manager (MPM), and the *gMacInstancesCnt_c* define is also set to 2 to duplicate the MAC's internal data.

After running an Energy Detect Scan to select the best channels, the application will start the coordinator functionality for both MAC instances (see the *App_StartCoordinator()* function).

The MPM module will share access to the PHY between the two MACs without the user's intervention.

After the two PAN Coordinators are started, End Devices can run Active Scan and associate with the desired Coordinator.

Data received from the End Devices is displayed into the Coordinator's console, specifying the PAN on which the data was received (`PAN0:` or `PAN1:`).

All characters typed into the Coordinator's console application will be sent on both PANs (both End Devices will receive the same data).

3.3. MyWirelessApp beacon enabled

In this demo, the PAN Coordinator is sending Beacon frames at a specified beacon interval (see the *mDefaultValueOfBeaconOrder_c* and *mDefaultValueOfSuperframeOrder_c* defines from App.h).

The End Device will synchronize with the Coordinator's beacons, and will automatically poll for data when his address is found inside the beacon payload.

3.3.1. Starting a beaconed network

To start a beaconed network, the application instructs the MAC to send out special packets at regular intervals. The packets (beacon frames) are not addressed to any particular device. That is, they are broadcast to every device that may be listening. A device that is synchronized to a coordinator's beacon, turns on its receiver at the time a beacon frame is expected.

On the coordinator side, the only change required is to change the values of the Beacon Order and Superframe Order. These values are parameters of the MLME-Start Request. By default they are both 0xF which means that a non-beaconed network is created. Setting the parameters to values between 0x0 and 0xE results in a beaconed network. The beacon interval is determined by the Beacon Order, and calculated as:

(960 · (2BO) · 16) / 1000000 seconds (approximately (2BO)/65.1 seconds)

The Superframe Order is the duration of the active period after the beacon frame. If the Superframe Order is the same as the Beacon Order, the active period stretches over the entire beacon interval. The beacon configuration is very application-specific. For example, a temperature sensor network might be configured to deliver temperature readings once every minute. The amount of data transferred is negligible compared to the Beacon Order of the network. A configuration could be BO=12 (63secs), and SO=1 (30.7msecs).

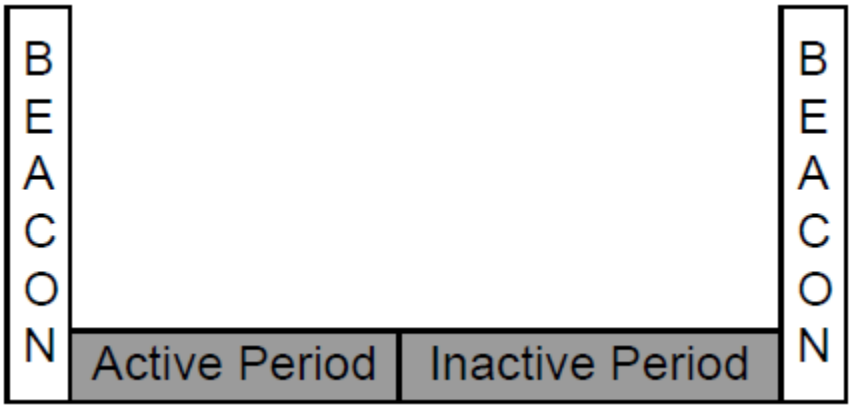


Figure 5. Data transfer

In the above figure, data was transferred between the device and coordinator. Assuming users do not want the application on the device to poll for data regularly, a beacon configuration suitable for the band-width requirements must be found. The data rate is 19200bps over the UART, and the application-specific data size limitation is 80 bytes. A transfer of 19200/10/80 = 24 packets each second, or one packet each 42 ms is required. Inverse math shows that the beacon order must be 1. The Superframe order must be the same. This configuration supports a bit rate of ~26000bps. In the example file `App.c` from the MyWirelessApp Demo BE (Coordinator) project, the coordinator is now set up to start a beaconed PAN instead of a non-beaconed PAN (`mDefaultValueOfBeaconOrder_c` and `mDefaultValueOfSuperframeOrder_c` are defined to 1):

Example 8. This is a MLME-START.req command

```
    pMsg->msgType = gMlmeStartReq_c;
    /* Create the Start request message data. */
    pStartReq = &pMsg->msgData.startReq;
    /* PAN ID - LSB, MSB. The example shows a PAN ID of 0xBEEF. */
    FLib_MemCpy(&pStartReq->panId, (void *)&mPanId, 2);
    /* Logical Channel - the default of 11 will be overridden */
    pStartReq->logicalChannel = mLogicalChannel;
#ifdef gPHY_802_15_4g_d
    pStartReq->channelPage = gChannelPageId9_c;
#endif
    /* Beacon Order - 0xF = turn off beacons */
    pStartReq->beaconOrder = mDefaultValueOfBeaconOrder_c;
    /* Superframe Order - 0xF = turn off beacons */
    pStartReq->superframeOrder = mDefaultValueOfSuperframeOrder_c;
    /* Be a PAN coordinator */
    pStartReq->panCoordinator = TRUE;
    /* Dont use battery life extension */
    pStartReq->batteryLifeExtension = FALSE;
```

```
    /* This is not a Realignment command */
    pStartReq->coordRealignment = FALSE;
    pStartReq->startTime = 0;

    /* Don't use security */
    pStartReq->coordRealignSecurityLevel = gMacSecurityNone_c;
    pStartReq->beaconSecurityLevel       = gMacSecurityNone_c;

    /* Send the Start request to the MLME. */
    if(NWK_MLME_SapHandler( pMsg, macInstance ) != gSuccess_c)
    {
      /* One or more parameters in the Start Request message were invalid. */
      Serial_Print(interfaceId,"Invalid parameter!\n\r", gAllowToBlock_d);
      return errorInvalidParameter;
}
```

This is all that is needed to apply and use beacons on a PAN, so from a coordinator point of view, adding beacons to the PAN does not require any extra code or even changing very much code. The beacon contains various information about the PAN and contains the addresses of the devices for which the coordinator has indirect data pending. This can be used by the device, as shown later in this section.

3.3.2. Understanding synchronization

The MAC on the device side can be instructed to follow the beacons from the coordinator. This is called synchronization, and is initiated by the MLME-Sync Request. A prerequisite for using the sync request is that the logical channel and the PAN ID of the coordinator users wish to communicate is already known. This information has been acquired during the scan procedure. Note that Passive Scan can replace Active Scan if the coordinator is beaconing. In that case, the Scan Duration parameter must be equal to or grater than the Beacon Order of the coordinator. This implies a priori knowledge of the network parameters. In `App.c` file from project MyWirelessApp Demo BE (End Device) the scan type has been changed from Active Scan to Passive scan by changing the `pScanReq->scanType` parameter in the scan request to`gScanModePassive_c`.

Active Scan is mostly used in non-beaconed networks. The MAC broadcasts a Beacon Request command frame on all channels specified in the channel list. Any coordinator listening on a particular channel responds to the request with a beacon frame. Because a coordinator in a beaconed network is already sending out beacons at some interval, the device does not have to send out the beacon request. This is called Passive Scan. For example, the device does not trigger the coordinator to send out a beacon frame before opening the receiver to listen for it and the beacon frame is sent out anyway.

When the device is part of a beaconed PAN, data reception can be significantly simplified because the MAC can be configured for automatic polling of data packets from the coordinator. In this application, all the polling code is removed from file `App.c` from project MyWirelessApp Demo BE (End Device) and replaced with the MLME-Sync Request, and a few MLME-Set Requests. These request messages

are only sent once right after the scan procedure has completed and before the association (see the `App.c` file from MyWirelessApp Demo BE (End Device) project):

Example 9. Synchronization with beacons

```
uint8_t value = TRUE;
uint8_t mBeaconOrder;
uint8_t mSuperFrameOrder;
/* Set MAC PIB auto request to TRUE. In this way the device will
   automatically poll for data if the pending address list of the
   beacon frame contains our address. */
pMsgOut->msgType = gMlmeSetReq_c;
pMsgOut->msgData.setReq.pibAttribute = gMPibAutoRequest_c;
pMsgOut->msgData.setReq.pibAttributeValue = &value;
/* Get/Set/Reset Request messages are NOT freed by the MLME. */
(void)NWK_MLME_SapHandler( pMsgOut, macInstance );

/* Since we are going to receive data from the coordinator
   using automatic polling we must synchronize to the beacon
   and keep tracking it. Before synchronizing it is required
   that the MAC PIB PAN ID, and the MAC PIB coordinator
   address is set. */
pMsgOut->msgType = gMlmeSetReq_c;
pMsgOut->msgData.setReq.pibAttribute = gMPibPanId_c;
pMsgOut->msgData.setReq.pibAttributeValue = &mCoordInfo.coordPanId;
/* Get/Set/Reset Request messages are NOT freed by the MLME. */
(void)NWK_MLME_SapHandler( pMsgOut, macInstance );

/* Set coordinator address PIB attribute according the the
   address mode of the coordinator (short or long address). */
pMsgOut->msgType = gMlmeSetReq_c;
pMsgOut->msgData.setReq.pibAttribute =
  mCoordInfo.coordAddrMode == gAddrModeShortAddress_c ? gMPibCoordShortAddress_c :
                                                        gMPibCoordExtendedAddress_c;
pMsgOut->msgData.setReq.pibAttributeValue = &mCoordInfo.coordAddress;
/* Get/Set/Reset Request messages are NOT freed by the MLME. */
(void)NWK_MLME_SapHandler( pMsgOut, macInstance );



/* Set macBeaconOrder PIB attribute according to the
```

```
        value found in beacon.*/
    pMsgOut->msgType = gMlmeSetReq_c;
    mBeaconOrder = mCoordInfo.superframeSpec.beaconOrder;
    pMsgOut->msgData.setReq.pibAttribute = gMPibBeaconOrder_c;
    pMsgOut->msgData.setReq.pibAttributeValue = &mBeaconOrder;
    /* Get/Set/Reset Request messages are NOT freed by the MLME. */
    (void)NWK_MLME_SapHandler( pMsgOut, macInstance );

    /* Set macSuperFrameOrder PIB attribute according to the
        value found in beacon.*/
    pMsgOut->msgType = gMlmeSetReq_c;
    mSuperFrameOrder = mCoordInfo.superframeSpec.superframeOrder;
    pMsgOut->msgData.setReq.pibAttribute = gMPibSuperframeOrder_c;
    pMsgOut->msgData.setReq.pibAttributeValue = &mSuperFrameOrder;
    /* Get/Set/Reset Request messages are NOT freed by the MLME. */
    (void)NWK_MLME_SapHandler( pMsgOut, macInstance );

    /* Now send the MLME-Sync Request. We choose to let the MAC track
        the beacons on the logical channel obtained by the passive scan.*/
    pMsgOut->msgType = gMlmeSyncReq_c;
    pMsgOut->msgData.syncReq.trackBeacon = TRUE;
    pMsgOut->msgData.syncReq.logicalChannel = mCoordInfo.logicalChannel;
    (void)NWK_MLME_SapHandler( pMsgOut, macInstance );
```

Notice that the PIB attribute macPanID must be set to the PAN ID of the coordinator that users want to synchronize to before actually synchronizing. Additionally, the PIB attributes macCoordShortAddress or macCoordExtendedAddress (depending on whether the coordinator uses a long or short address) must be set before synchronizing. The parameter `trackBeacon` from the Sync Request can be set to either TRUE or FALSE, depending on whether the device is requested to synchronize and track future beacons or just synchronize to the first beacon and ignore future beacons.

There is no MLME-SYNC.confirm primitive for verifying that the device was actually synchronized to the coordinator. It is assumed that the synchronization was successful. If the coordinator disappears or the device for some other reason loses track of the coordinator's beacons, the device receives a MLME-SYNC-LOSS.indication primitive. This is solved by re-issuing a MLME-SYNC.request (it is always allowed to re-synchronize to a coordinator).

Example 10. MLME-SYNC-LOSS.Indication

```
case gMlmeSyncLossInd_c:
    Serial_Print(interfaceId,"\n\rSynchronization lost!\n\r", gAllowToBlock_d);
    /* Resync with the coordinator */
```

```
if ( App_SendSyncRequest() != errorNoError)
{
  Serial_Print(interfaceId,"Error : Send Sync Request.\n\r", gAllowToBlock_d);
}
break;
```

Repeatedly getting MLME-SYNC-LOSS.indications typically means that the coordinator has disappeared, changed PAN ID, changed logical channel, or for some other reason is no longer available.

The coordinators beacon frame may, for example, carry information about indirect data pending in the coordinators transmit queue. If the device receiving the beacon frame finds its own address in the list of pending data packets, it can automatically send a Poll request to retrieve the data packet. Enable automatic polling by setting the macAutoRequest PIB attribute to TRUE (as shown in the previous code example). The benefit is that the device does not need to schedule Poll Requests at regular intervals, though, only one automatic Poll Request is sent for each beacon frame. If more data from the coordinator is required before the next beacon frame, the device still must poll for it.

3.4. MyWirelessApp security enabled

This demo is similar with the basic MyWirelessApp demo, the difference being that all data packets sent Over-the-Air are encrypted (see the *mDefaultValueOfSecurityLevel_c* define from App.h).

3.4.1. MAC 2006 security

MAC 2006 security provides a cryptographic mechanism to the upper layer with the following services:

- Data Confidentiality — Assurance that transmitted information is only disclosed to parties for which it is intended
- Data Authenticity — Assurance of the source of information
- Replay Protection — Assurance that duplicate information is detected

Cryptographic frame protection may use a key shared between two peer devices (link key) or a shared key among a group of devices (group key), thus allowing some flexibility and application-specific trade-offs between key storage and maintenance costs versus the cryptographic protection provided.

The information that determines how the security is provided is found in the security-related PIB.

The security-related MAC PIB attributes contain the following:

- Key Table — Holds the key descriptors that are required for security of outgoing and incoming frames
- Device Table — Holds the device descriptors that when combined with key-specific information from the key table, provide all the keying material needed to secure the outgoing and unsecure incoming frames
- Minimum Security Level Table — Holds information regarding the minimum security level the device expects to have been applied by the originator of a frame, depending on frame type and, if it concerns a MAC command frame, the command frame identifier

- Frame Counter — Provides replay protection. The frame is included in each secured frame and is one of the elements required for the un-securing operation at the recipient
- Automatic Request Attributes — Automatic request attributes hold the information needed to secure outgoing frames generated automatically and not as a result of higher layer primitive, as in the case with automatic data requests
- Default key source — This is the information commonly shared between originator and recipient(s) of a secured frame, which, when combined with additional information explicitly contained in the requesting primitive or the in the received frame, allows the originator or a recipient to determine the key required for securing or un-securing this frame, respectively
- PAN coordinator address — This is the information commonly shared between all devices in a PAN, which, when combined with additional information explicitly contained in the requesting primitive or in the received frame, allows an originator of a frame directed to the PAN coordinator or a recipient of a frame originating from the PAN coordinator to determine the key and security-related information required for securing or un-securing this frame

The security-related MACPIB attributes are structured as lists of attributes. The list sizes are given by constants defined in `MacGlobals.h` header. For example, the number of entries in the *macKeyTable* is given by: `#define gNumKeyTableEntries_c 2`.

MAC 2006 security allows for all frame types to be secure.

Table 7. Available MAC 2006 security levels

Security Level Identifier	Security Attributes	Data Confidentiality	Data Authenticity
0x00	None	OFF	NO (M = 0)
0x01	MIC-32	OFF	YES (M = 4)
0x02	MIC-64	OFF	YES (M = 8)
0x03	MIC-128	OFF	YES (M = 16)
0x04	ENC	ON	NO (M = 0)
0x05	ENC-MIC-32	ON	YES (M = 4)
0x06	ENC-MIC-64	ON	YES (M = 8)
0x07	ENC-MIC-128	ON	YES (M = 16)

3.4.1.1. Security material information

To secure a data frame on MAC 2006 it is required to complete the following security material stored in the security-related PIB attributes:

- The initial frame counter — *macFrameCounter*
- The key table information — *key, keyLookupData, keyLookupDataSize*
- The minimum security level information — *securityLevelFrameType, securityMinimum, deviceOverrideSecurityMinimum, uniqueDevice, blackListed*
- Device table information — *deviceDescriptorPanId, deviceDescriptorShortAddress, deviceDescriptorExtendedAddress, keyUsageFrameType, deviceDescriptorExempt*

The key value must be common for both the Coordinator and the End Device. The value can be set to one of the following: 0xCF, 0xCE, 0xCD, 0xCC, 0xCB, 0xCA 0xC9, 0xC8, 0xC7, 0xC6, 0xC5, 0xC4, 0xC3, 0xC2, 0xC1, 0xC0

The initial value of the frame counter is set to 0x00000000.

The *keyLookupData* contain information about the recipient device. If the data transmission is on extended address the value must be set at application generation stage in BeeKit interface: for Coordinator application must be set the End Device extended address and vice-versa.

3.4.1.2. Code example

Example 11. The code for setting up security on the device

```
static void App_InitSecurity(void)
{
  mlmeMessage_t msg;
  uint8_t       pibVal;
  uint32_t      frameCounterInitVal = 0x00000000;
  uint8_t       key[16] = {0xCF, 0xCE, 0xCD, 0xCC, 0xCB, 0xCA, 0xC9, 0xC8, 0xC7, 0xC6, 0xC5,
0xC4, 0xC3, 0xC2, 0xC1, 0xC0};
  uint8_t       keyLookupData[9];
  uint8_t       lookupDataSize;
  uint64_t      tempExtAddress= 0xFFFFFFFFFFFFFFFF;

  //Set security PIB attributes
  pibVal = TRUE;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibSecurityEnabled_c;
  msg.msgData.setReq.pibAttributeValue = (uint8_t *)&pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibFrameCounter_c;
  msg.msgData.setReq.pibAttributeValue = (uint8_t *)&frameCounterInitVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  //RX security PIBS
  pibVal = 0;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibiSecurityLevelTableCrtEntry_c;
  msg.msgData.setReq.pibAttributeValue = &pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  pibVal = 1;
```

```
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevFrameType_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevSecurityMinimum_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );


pibVal = TRUE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevDeviceOverrideSecurityMinimum_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyTableCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyIdLookuplistCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKey_c;
msg.msgData.setReq.pibAttributeValue = key;
(void)NWK_MLME_SapHandler( &msg, macInstance );

//Create lookup data
if( mCoordInfo.coordAddrMode == 0x03 ) {
  //Copy device extended address
```

```
  FLib_MemCpy(keyLookupData, &mCoordInfo.coordAddress, 8);
  keyLookupData[8] = 0x00;
  lookupDataSize = 0x01;
}
else {
  //Copy PAN to lookupData
  FLib_MemCpy(keyLookupData, (void *)&mCoordInfo.coordPanId, 2);
  FLib_MemCpy(keyLookupData + 2, (void *)&mCoordInfo.coordAddress, 2);
  keyLookupData[4] = 0x00;
  lookupDataSize = 0x00;
}

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyIdLookupData_c;
msg.msgData.setReq.pibAttributeValue = keyLookupData;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyIdLookupDataSize_c;
msg.msgData.setReq.pibAttributeValue = &lookupDataSize;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = FALSE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibUniqueDevice_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = FALSE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibBlackListed_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyDeviceListCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
```

```
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorPanId_c;
msg.msgData.setReq.pibAttributeValue = &mCoordInfo.coordPanId;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorShortAddress_c;
msg.msgData.setReq.pibAttributeValue = &mCoordInfo.coordAddress;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeGetReq_c;
msg.msgData.getReq.pibAttribute =  gMPibCoordExtendedAddress_c;
msg.msgData.getReq.pibAttributeValue = &tempExtAddress;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorExtAddress_c;
msg.msgData.setReq.pibAttributeValue = &tempExtAddress;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiDeviceTableCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 1;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyUsageFrameType_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 1;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyUsageListCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
```

```
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  pibVal = FALSE;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibDeviceDescriptorExempt;
  msg.msgData.setReq.pibAttributeValue = &pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );
}
```

Example 12. The code for setting up security on the coordinator

```
static void App_InitSecurity(void)
{
  mlmeMessage_t msg;
  uint8_t       pibVal;
  uint32_t      frameCounterInitVal = 0x00000000;
  uint8_t       key[16] = {0xCF, 0xCE, 0xCD, 0xCC, 0xCB, 0xCA, 0xC9, 0xC8, 0xC7, 0xC6, 0xC5,
0xC4, 0xC3, 0xC2, 0xC1, 0xC0};
  uint8_t       keyLookupData[9];
  uint8_t       lookupDataSize;

  //Set security PIB attributes
  pibVal = TRUE;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibSecurityEnabled_c;
  msg.msgData.setReq.pibAttributeValue = (uint8_t *)&pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibFrameCounter_c;
  msg.msgData.setReq.pibAttributeValue = (uint8_t *)&frameCounterInitVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  pibVal = 0;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibiKeyTableCrtEntry_c;
  msg.msgData.setReq.pibAttributeValue = &pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );
```

```
pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyIdLookuplistCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKey_c;
msg.msgData.setReq.pibAttributeValue = key;
(void)NWK_MLME_SapHandler( &msg, macInstance );

//Create lookup data
if( mDeviceShortAddress == 0xFFFE ) {
  //Copy device extended address
  FLib_MemCpy(keyLookupData, &mDeviceLongAddress, 8);
  keyLookupData[8] = 0x00;
  lookupDataSize = 0x01;
}
else {
  //Copy PAN to lookupData
  FLib_MemCpy(keyLookupData, (void *)&mPanId, 2);
  FLib_MemCpy(keyLookupData + 2, &mDeviceShortAddress, 2);
  keyLookupData[4] = 0x00;
  lookupDataSize = 0x00;
}

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyIdLookupData_c;
msg.msgData.setReq.pibAttributeValue = keyLookupData;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyIdLookupDataSize_c;
msg.msgData.setReq.pibAttributeValue = &lookupDataSize;
(void)NWK_MLME_SapHandler( &msg, macInstance );

//RX security PIBs
pibVal = 0;
```

```
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiSecurityLevelTableCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 1;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevFrameType_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevSecurityMinimum_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = TRUE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibSecLevDeviceOverrideSecurityMinimum_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyIdLookupDataSize_c;
msg.msgData.setReq.pibAttributeValue = &lookupDataSize;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = FALSE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibUniqueDevice_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = FALSE;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibBlackListed_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
```

```
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyDeviceListCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorPanId_c;
msg.msgData.setReq.pibAttributeValue = (uint8_t *)&mPanId;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorShortAddress_c;
msg.msgData.setReq.pibAttributeValue = &mDeviceShortAddress;
(void)NWK_MLME_SapHandler( &msg, macInstance );

msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute =   gMPibDeviceDescriptorExtAddress_c;
msg.msgData.setReq.pibAttributeValue = &mDeviceLongAddress;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 0;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiDeviceTableCrtEntry_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 1;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibKeyUsageFrameType_c;
msg.msgData.setReq.pibAttributeValue = &pibVal;
(void)NWK_MLME_SapHandler( &msg, macInstance );

pibVal = 1;
msg.msgType = gMlmeSetReq_c;
msg.msgData.setReq.pibAttribute = gMPibiKeyUsageListCrtEntry_c;
```

```
  msg.msgData.setReq.pibAttributeValue = &pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );

  pibVal = FALSE;
  msg.msgType = gMlmeSetReq_c;
  msg.msgData.setReq.pibAttribute = gMPibDeviceDescriptorExempt;
  msg.msgData.setReq.pibAttributeValue = &pibVal;
  (void)NWK_MLME_SapHandler( &msg, macInstance );
}
```

3.4.1.3. Security summary

When applying security, extra processing power is required and the maximum bandwidth will be reduced from close to the theoretical maximum to about 50% of that value.

When applying MAC 2006 security, the frame overhead is represented by the auxiliary security header and the extra payload resulted from applying the cryptographic algorithm.

Users should now be able to understand the basic steps in creating and managing a simple beaconed, and non-beaconed 802.15.4 Standard PAN with and without security.

1. Starting and initializing the Freescale 802.15.4 MAC software.
2. Performing energy detection (ED) scan.
3. Starting a non-beaconed PAN.
4. Performing an active and passive scan.
5. Associating a device and a coordinator.
6. Performing a direct and indirect data transfer.
7. Starting a beaconed PAN.
8. Applying security.

When writing the application, users may choose to employ a less complex data transfer model. For example, implementing only one outstanding data request at a time or using a different polling scheme. The design depends on what the application is supposed to do and the performance criteria imposed on that application. Other items to consider are code size, power consumption, and throughput, among others. This example is intended to only introduce the basic concepts of an 802.15.4 PAN. Though there are many ways to achieve the same functionality, but the basic concepts remain the same.

3.5. MyWirelessApp FSCI host

This demo is similar with the basic MyWirelessApp demo, the difference being that the 802.15.4 MAC/PHY software is running on a black box platform (MAC FSCI App) and the application is running on another platform (FSCI Host). The two platforms are connected through the FSCI for IEEE 802.15.4 MAC/PHY protocol over one of the serial interfaces supported by both platforms (SPI, UART or I2C). For more information about FSCI for IEEE 802.15.4 MAC/PHY see the 802154FSCIRM.pdf.

The FSCI Host is responsible to register the serial interface through which the MAC is accessed while the MAC interface remains unchanged. In the example file `App.c` from the MyWirelessApp Fsci Host project the coordinator is set up to register the serial interface to the MAC.

```
static const gFsciSerialConfig_t mFsciSerials[] = {
/* Baudrate,                    interface type,              channel No,         virtual interface */
{gAppFSCIHostInterfaceBaud d, gAppFSCIHostInterfaceType d, gAppFSCIHostInterfaceId_d, 0},    {gUARTBaudRate115200_c,     APP_SERIAL_INTERFACE_TYPE, APP_SERIAL_INTERFACE_INSTANCE,       1}
};
void main_task(uint32_t param)
{
    static uint8_t initialized = FALSE;

    if( !initialized )
    {
        initialized = TRUE;
        hardware_init();

        MEM_Init();
        TMR_Init();
        LED_Init();
        SecLib_Init();
        SerialManager_Init();
#if gFSCI_IncludeLpmCommands_c
        PWRLib_Init();
        PWR_DisallowDeviceToSleep();
#endif
        FSCI_Init( (void*)mFsciSerials );
        RNG_Init(); /* RNG must be initialized after the PHY is Initialized */
        MAC_Init();

        /* Bind to MAC layer */
        macInstance = BindToMAC( (instanceId_t)0 );
        Mac_RegisterSapHandlers( MCPS_NWK_SapHandler, MLME_NWK_SapHandler, macInstance );
        fsciRegisterMacToHost( macInstance, 0 );

        /* Register ASP commands */
        fsciRegisterAspToHost(0, 0);
```

```
        App_init();
    }

    /* Call application task */
    AppThread( param );
}
```

The MAC SAPs are accessed in the same manner as in MyWirelessApp demo, the App to MAC messages being unchanged. To enable the FSCI Host feature the macro `gFsciHostMacSupport_c` should be defined.

After enabling the host support, the macro `gFsciHostInterfaceType_c` hast to be set to one of the supported values: `gFsciHostUseUart_c`, `gFsciHostUseSpi_c` or `gFsciHostUseI2c_c`.

If the macro `gFsciHostInterfaceType_c` is set to one of the supported value another set of macros have to be defined as presented in the code below:

```
/*
 * FSCI Host configuration
 */

#define gFsciHostUseUart_c      1
#define gFsciHostUseSpi_c       2
#define gFsciHostUseI2c_c       3

/* Select the serial type used for inter-processor communication */
#define gFsciHostInterfaceType_c  gFsciHostUseI2c_c

#if (gFsciHostInterfaceType_c == gFsciHostUseUart_c)
    #define gAppFSCIHostInterfaceBaud_d     gUARTBaudRate115200_c
    #define gAppFSCIHostInterfaceType_d     gSerialMgrUart_c
    #define gAppFSCIHostInterfaceId_d       3
    #define gSerialMgrUseUart_c             1
#elif (gFsciHostInterfaceType_c == gFsciHostUseSpi_c)
    #define gAppFSCIHostInterfaceBaud_d     gSPI_BaudRate_1000000_c
    #define gAppFSCIHostInterfaceType_d     gSerialMgrSPIMaster_c
    #define gAppFSCIHostInterfaceId_d       0
    #define gSerialMgrUseSPI_c              1
#elif (gFsciHostInterfaceType_c == gFsciHostUseI2c_c)
    #define gAppFSCIHostInterfaceBaud_d     gIIC_BaudRate_100000_c
```

```
    #define gAppFSCIHostInterfaceType_d     gSerialMgrIICMaster_c
    #define gAppFSCIHostInterfaceId_d       0
    #define gSerialMgrUseIIC_c              1
#endif
```

After the FSCI host was successfully configured, the FSCI black box has to be set to use the same serial interface as the FSCI host and the hardware connections (serial communication lines, jumpers etc.) must be checked in order to be properly configured. For more information, see 802154MPADG.pdf.

4. My Star Network Application

This guide provides information about creation and maintenance of non-beacon star networks based on the Freescale 802.15.4 Media Access Controller (MAC) implementation. The MyStarNetworkApp is a set of two (2) applications created on top of 802.15.4 Media Access Controller (MAC).

4.1. Running the MyStarNetwork demonstration

There are two possible use cases for employing the MyStarNetworkApp.

1. An autonomous network. The messages exchanged at the MAC level can be monitored using a network sniffer.
2. Users connect the PAN Coordinator and the End Devices to a computer through serial links. In this case, a terminal style program is used for watching the messages that the devices output to the UART interface.

If user interaction with the MyStarNetworkApp takes place through a terminal console connected to the PAN Coordinator and the End Device UART ports, messages typed in the terminal console connected to the PAN Coordinator are transmitted to all End Devices. The message is understood as a sequence of characters delimited by typing a carriage return, or a maximum of 20 characters. Due to the power-saving features enabled on the End Device MCU (STOP3 Mode) from the End Device to the PAN Coordinator, only a pre-defined message is sent when users press a button on the board.

Messages received by the Coordinator are displayed on the terminal console using the following format:

Device address [Device address] ([Link quality]): [received characters]

Messages received by the End Device are displayed on the terminal console using the following format:

PAN Coordinator ([Link quality]): [received characters]

MyStarNetworkApp users can monitor network activity even though no terminals are connected to the participants in the network by using a network sniffer on the network channel. Besides the devices regularly polling the Coordinator for data, the PAN Coordinator sends a timer value every mDefaultValueOfTimeInterval_c seconds to all End Devices, where mDefaultValueOfTimeInterval_c is the value of the Basic time interval property from the Star Network Demo (Coordinator) project in BeeKit. This value appears in binary format on the LEDs on the boards serving as the End Device. The LEDs on the board serving as the PAN Coordinator reflect the number of End Devices connected to the network.

If an End Device is turned off, the Coordinator detects that the End Device has left the network (even if it hasn't sent the MLME-DISASSOCIATION.request primitive), by looking for devices that haven't received their packets. In this case, the Coordinator removes the End Device from the queue of associated devices and it outputs the following message:

```
Disconnected device: 01
```

4.1.1. Running the end devices in autonomous mode

To run the end devices in autonomous mode, perform the following steps:

1. Plug in the PAN Coordinator and the End Devices. Use an external adapter DC voltage adapter or the power the devices through the USB port of the computer.
2. Turn on the PAN Coordinator and press any of the four switches on the board.
3. The PAN Coordinator creates and starts up the PAN. At this point, the LEDs on this board should be off to indicate that there are no devices connected.
4. Turn on the first End Device and press any of the four switches on the board.
5. The End Devices connect to the PAN and is ready to receive messages from its Coordinator. At this time, the LEDs on the PAN Coordinator display as follows:

6. This LED configuration indicates that the first available address has been allocated to the new connected End Device.
7. Repeat the Steps 1 through 5 for the second End Device. When it associates to the PAN, the LEDs on the PAN Coordinator display as follows:

Users can add up to four (4) End Devices to the PAN. When the PAN is full, all the LED's on the Coordinator device are on.

Every mDefaultValueOfTimeInterval_c seconds, the Coordinator sends the value of an internal counter to all of the End Devices currently associated to the PAN. The LEDs on each End Device show the current value (in binary) received from the Coordinator.

4.1.2. Starting the demonstration and creating the PAN

To start running the MyStarNetwork demonstration using the three evaluation boards, perform the following steps:

1. Install the batteries or plug a power supply into the Sensor Node and Low Power Node boards.
2. Connect the Network Node to the PC using a USB cable.
3. Power on the Network Node and follow the installation instructions as displayed in the Found New Hardware wizard on the PC. If the drivers do not automatically load, they can be found in the C:\Program Files\Freescale\Drivers directory.
4. Power on the Sensor Node and Low Power Node boards.

5. Launch the MyStarNetwork Demonstration GUI by clicking on Start -> Programs -> Freescale BeeKit -> MyStarNetwork Demonstration -> MyStarNetwork Demonstration.
6. The GUI starts and scans the serial port connections to auto detect the Network Node and the Application Log on the PC GUI displays which port the Coordinator was found as shown in the figure below. If the Coordinator port is not found, repeat the auto scan by pressing the "Autoconnect to MyStarNetwork Coordinator" button in the GUI toolbar. Ensure that no other application is keeping the Coordinator port open.

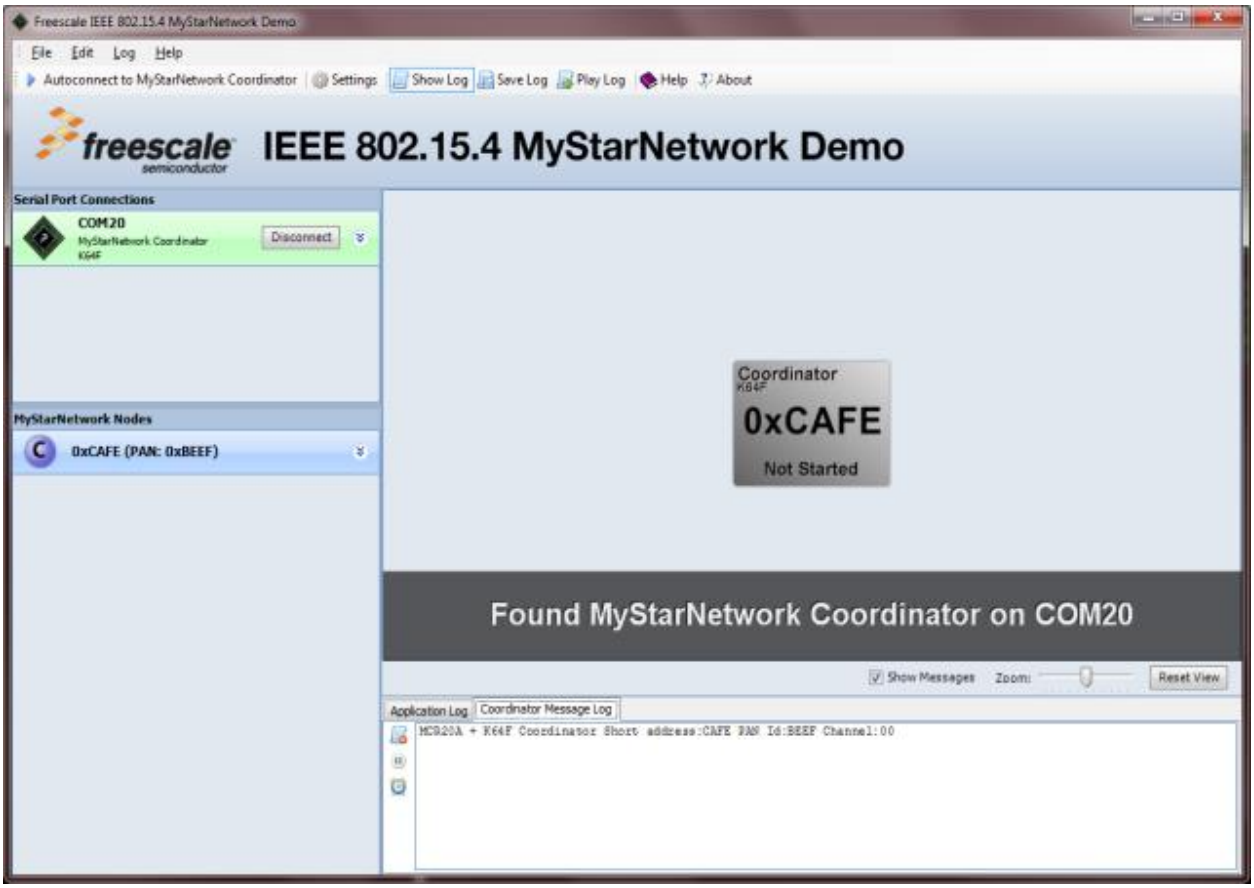


Figure 6. MyStarNetwork demonstration GUI showing detected coordinator port

7. The Coordinator icon in the GUI displays the node short address 0xCAFE and the "Not Started" message. The node has not yet initialized the IEEE 802.15.4 PAN. To initialize the PAN, press any of the switches (SW1 - SW4) on the Coordinator board or right-click on the Coordinator icon in the GUI and choose Start PAN Coordinator from the pop-up menu.
8. As the Coordinator forms the PAN, it performs an energy detection scan on all 802.15.4 RF channels and chooses a channel on which to start. The channel selected is the one that has the minimum energy level present at that point in time. The channel selected by the Coordinator is displayed in the GUI application log.
9. As the Coordinator initializes, the icon is no longer greyed out and the PAN ID is displayed on the icon as shown in the below figure.

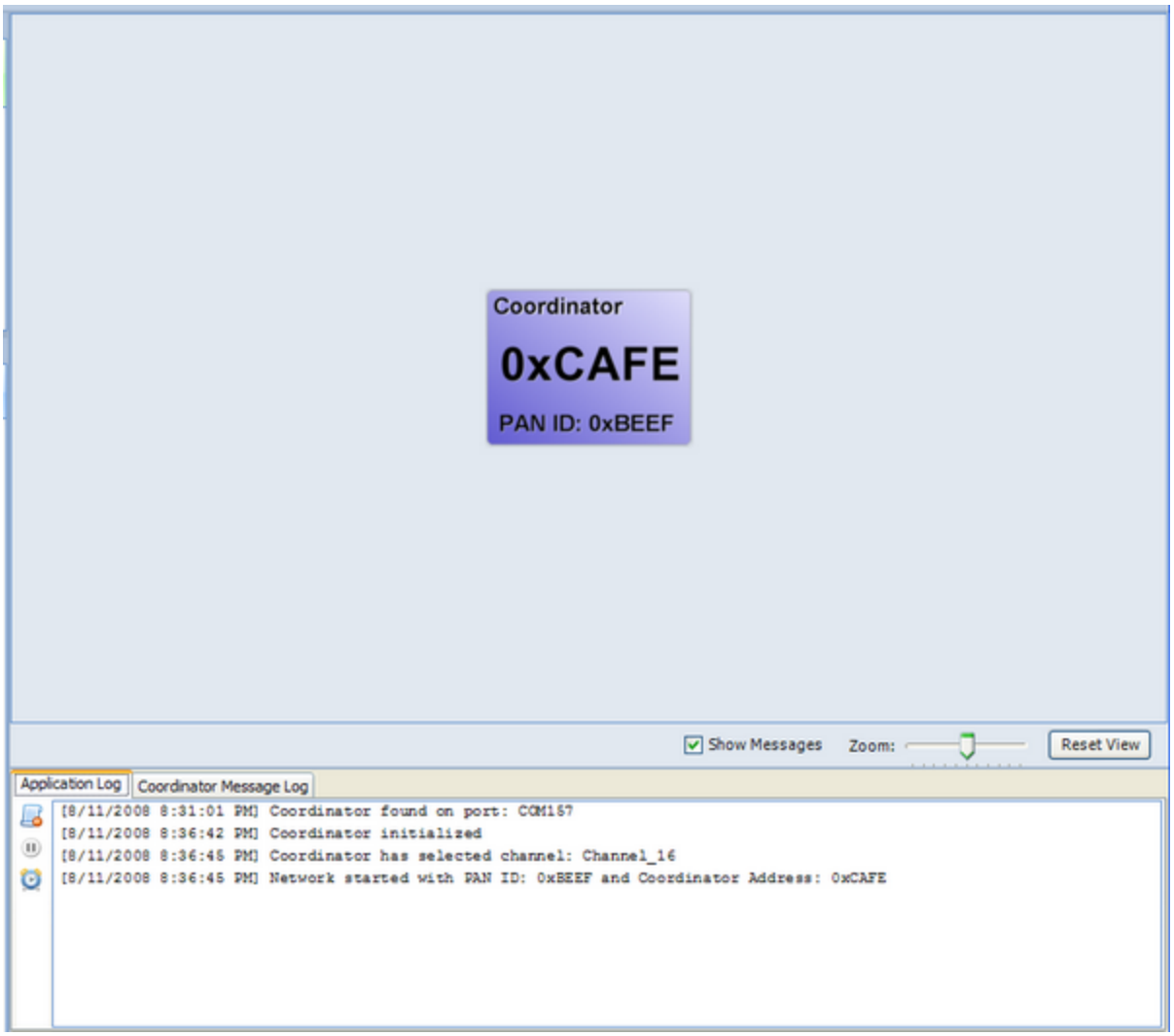


Figure 7. Coordinator has formed the PAN 0xBEEF on channel 16

10. Press any of the switches (SW1 - SW4) on the board to join the first IEEE 802.15.4 End Device to the network. When the End Device joins, the GUI displays a new icon for the End Device as shown in the figure from below. In a few seconds, the device begins sending sensor data messages to the Coordinator. This is depicted on the GUI by an animated sensor packet moving toward the Coordinator.

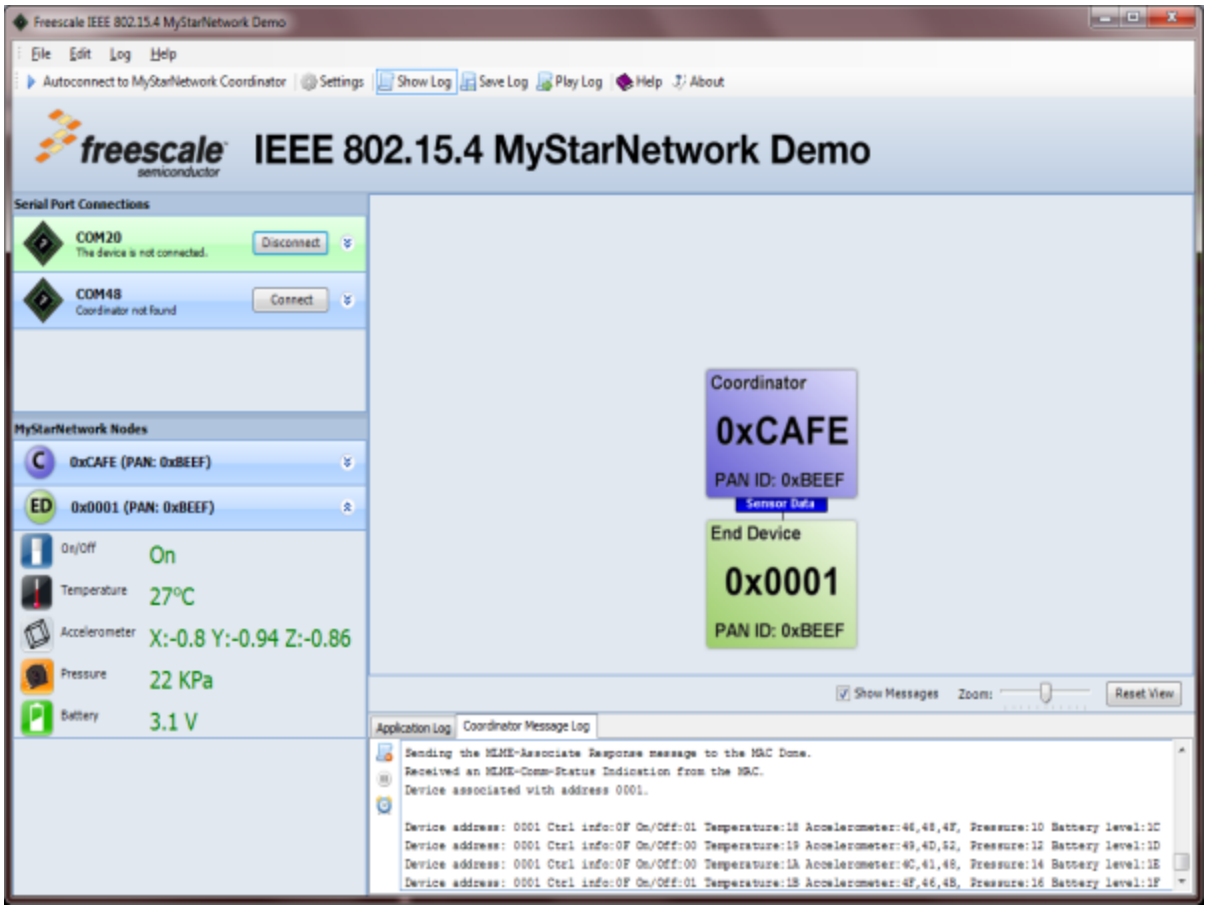


Figure 8. Sensor node end device associated and sending data

11. Press any key on the Low Power Node board to join the second IEEE 802.15.4 End Device to the network. When the device joins, another icon is displayed by the GUI which now shows the nodes in a star topology as shown in the figure below.

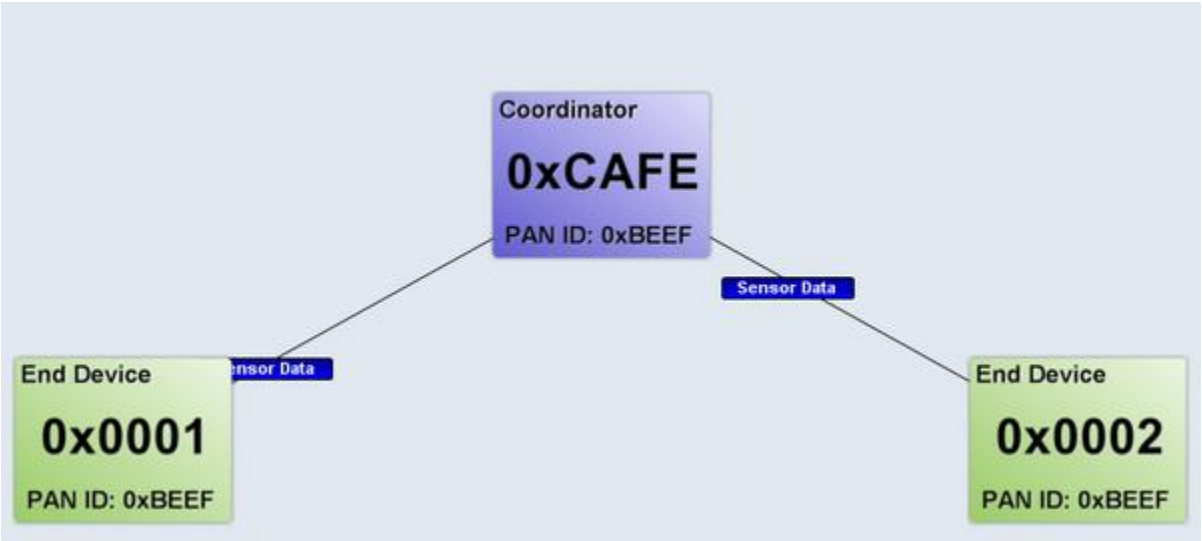


Figure 9. Coordinator with two end devices associated and sending data

4.1.3. Monitoring sensor data reports

The end devices are sending board sensor data to the Coordinator. Data sent from the boards is displayed on the MyStarNetwork Nodes panel on the bottom left side of the GUI as shown in the below figure.

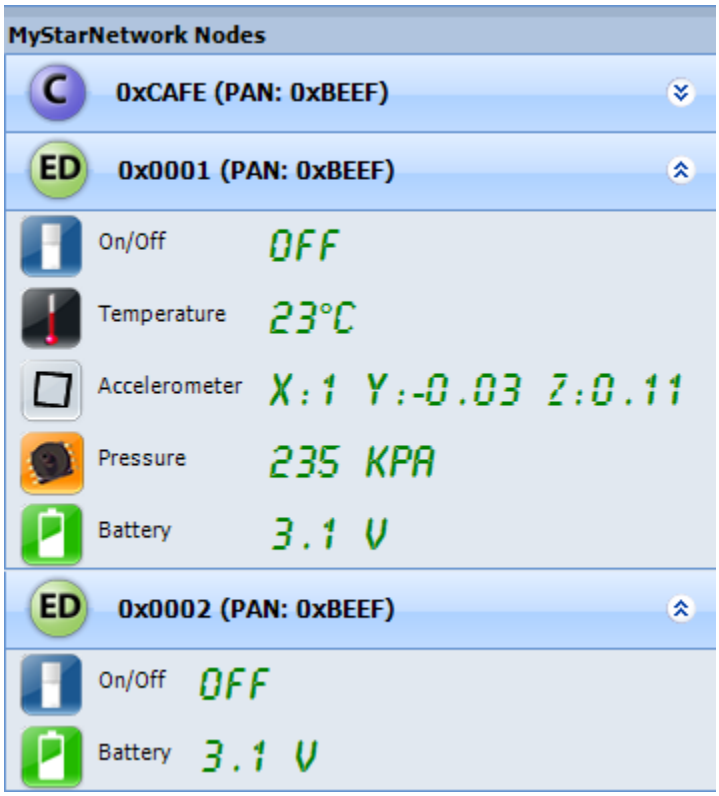


Figure 10. Sensor data received from the end devices

The Sensor Node reports five sensor items:

- On/off switch toggle state
- Temperature
- 3-axis acceleration
- Pressure
- Battery voltage

The Low Power Node reports two sensor items:

- On/off switch toggle state
- Battery voltage

To see how some of the data changes users can do the following:

1. Press SW1 on the boards to toggle the on/off switch state. The On/Off indication in the GUI also changes.

2. Physically move the Sensor Node board and monitor the 3-axis changes in the GUI. The three values displayed are the acceleration values on 3 axis, indicating tilt in the range from -1g to 1g. For example, when the sensor node is placed horizontally on a flat surface, the X axis displays a value close to 1g while the Y and Z axis will be close to 0. Turn the board upside down and notice how the X axis changes to a value close to -1g.
3. Insert the hose and its attached pump to the pressure sensor connector on the Sensor Node. Use the pump to make changes to the pressure and monitor the changes in the GUI.

All association and sensor data information that the GUI displays is obtained from the Network Node which sends the packets it receives over the air from the end devices via the USB connection. To view the raw data packets sent through the USB from the Network Node, users can change to the "Coordinator Message Log" tab as shown in the below figure.

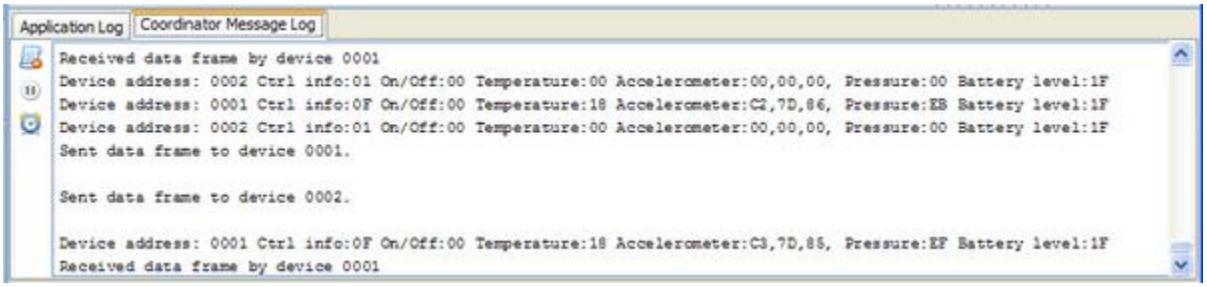


Figure 11. Coordinator message log showing UART communication with the coordinator

At the end of the demonstration, users can reset or power off the end devices. After a 15 second timeout the devices are grayed out by the GUI and the status is set to inactive as shown in Figure B-14. After 15 more seconds, the devices are removed from the display.

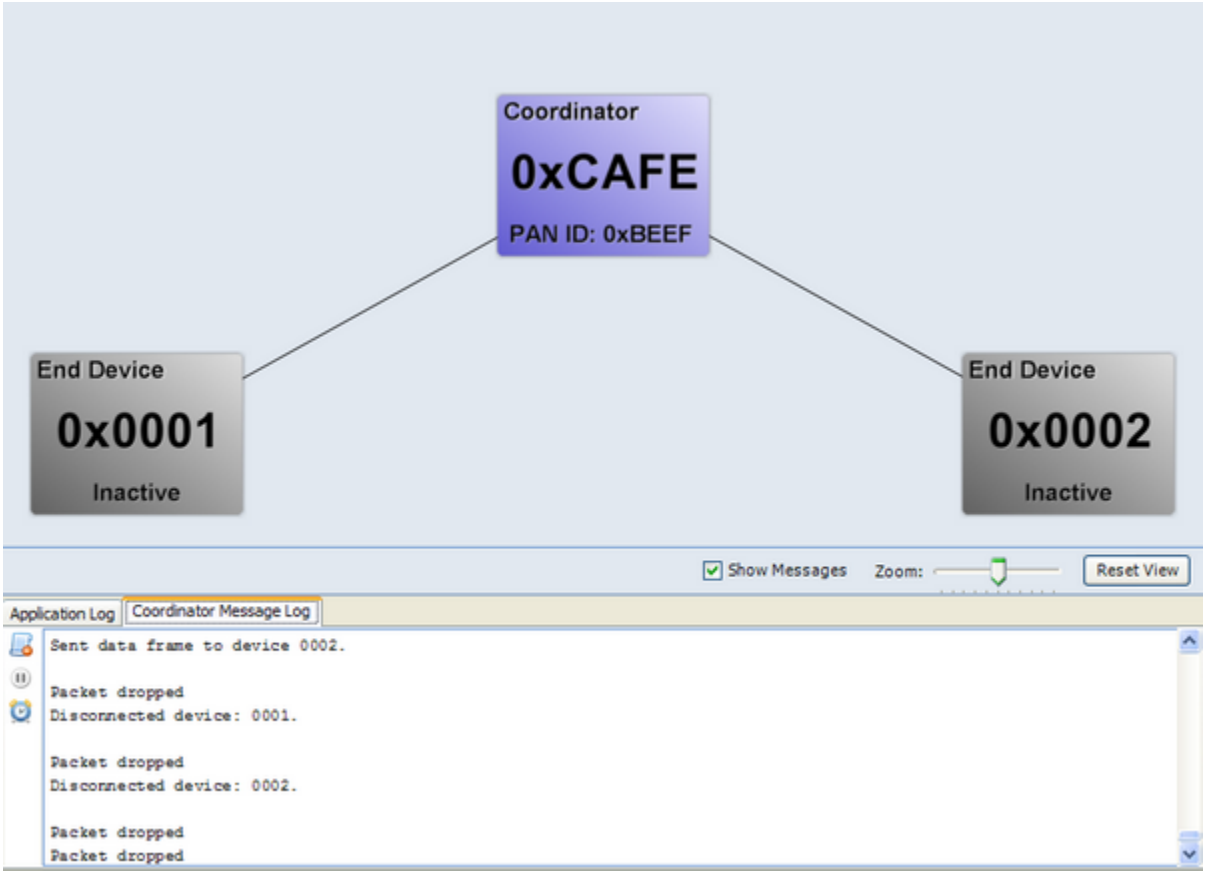


Figure 12. Inactive disconnected devices

4.1.4. MyStarNetwork demonstration GUI user interface overview

The MyStarNetwork Demonstration GUI main window consists of the Toolbar and Serial Port Connections Panel,

4.1.4.1. Application toolbar

Figure 13. MyStarNetwork demonstration toolbar

The application toolbar contains the following options which are also available from the drop-down menus:

Autoconnect to MyStarNetwork Coordinator

Use this option if the Coordinator's USB port is changed while the application is running. The new port will be detected by the GUI if it is not already opened.

Settings

Launches the MyStarNetwork Demonstration Settings window where GUI configuration settings can be changed.

Show Log

Toggle button which hides or shows the Log tabs at the bottom right of the application window.

Save Log

The current session's Coordinator messages are saved to a log file.

Play Log

A log file saved previously is played even if the boards are not connected. Do not use this option with a live Coordinator port connected.

Help

Displays the online help for the application.

About

Displays version and copyright information.

4.1.5. Serial port connections panel

Each time users connect Freescale evaluation boards to the PC using a USB connection, a virtual COM port is created. The ports will appear in the list once the boards are powered on and connected. If the background color of the board list item is blue, and the button text displays "Connect", the board communication port is available but it is not opened.

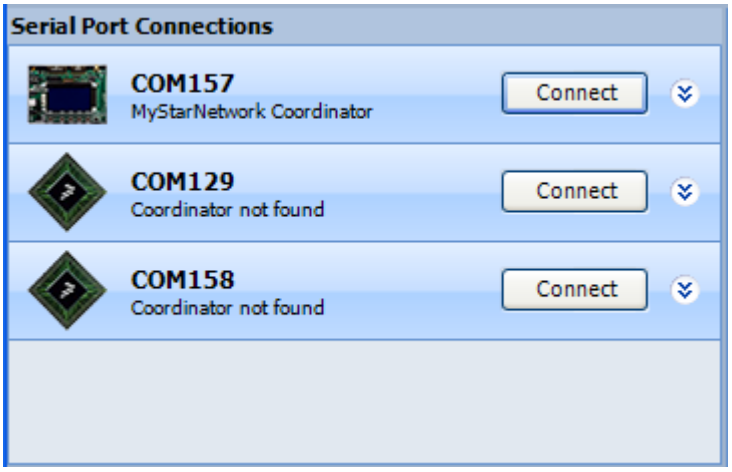


Figure 14. Serial port connections panel

The Coordinator port is usually opened automatically by the application when it scans to find the Coordinator. Users can use the Connect button to manually connect to the Coordinator. To manually connect to the Coordinator port, do the following:

1. Click on the expand button to the right of the “Connect” button. This expands the port setting control.

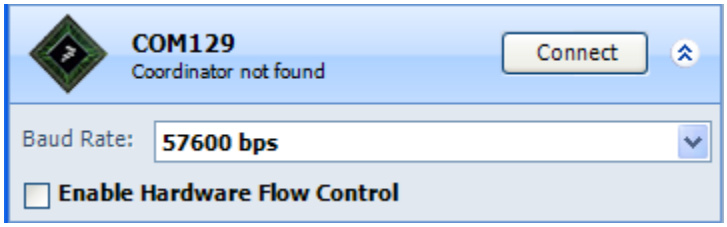


Figure 15. Expanded port setting controls

2. Choose a baud rate and flow control (enabled or disabled).
3. Click the Connect button.

If the connection is successful, the port list item is displayed in green and the button text changes to “Disconnect”. If the connection fails, the list entry is displayed in red and an error message is displayed below the port name.

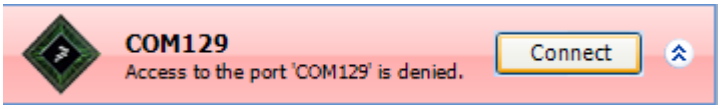


Figure 16. Expanded port setting controls

Once a port is connected, the messages it sends over the UART are displayed in the Coordinator Message Log. This information is useful for debugging purposes. To close the communication port, click “Disconnect” while a port is connected.

4.1.6. MyStarNetwork main panel

The MyStarNetwork main panel displays device icons in a star topology as they connect to the Coordinator. Click on a node icon so it can be displayed in close up and so that sensor data can be viewed on the End Device icons.

- Users can drag and drop in the panel to move the display’s viewpoint origin.
- Users can increase or decrease the size of the node icons by using the Zoom slider at the bottom of the panel or by using the mouse scroll wheel.

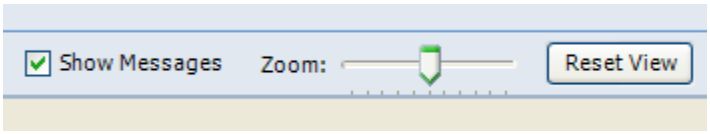


Figure 17. Main panel controls

Check or uncheck the “Show Messages” check box to show or hide “Sensor data” messages sent from the end devices to the Coordinator. Click Reset View to bring the node icon size and viewpoint origin to their initial positions.

4.1.7. MyStarNetwork nodes panel

The MyStarNetwork Nodes displays a list of the nodes with the Coordinator at the top followed by the end-devices connected to it. Sensor information sent from each end-device is also shown in this panel and updated each time a sensor data report is sent from the end devices to the Coordinator. Use the expand/contract arrow to the right of the device to show or hide the end device sensor information. Use the Settings option in the application toolbar to change the temperature and pressure sensor report measurement units.

4.1.8. MyStarNetwork log panel

The log panel contains two log displays, one for the application status messages and another which records the raw data sent from the Coordinator node over the UART. Use the buttons to the left of the panel to clear the log, pause message recording and show/hide message timestamps.

4.1.9. MyStarNetwork settings dialog

The settings dialog allows users to configure the way the MyStarNetwork Demonstration GUI works. There are three different configuration sections:

4.1.9.1. Port settings

The port settings window lets users configure how the virtual COM ports are managed by the application.

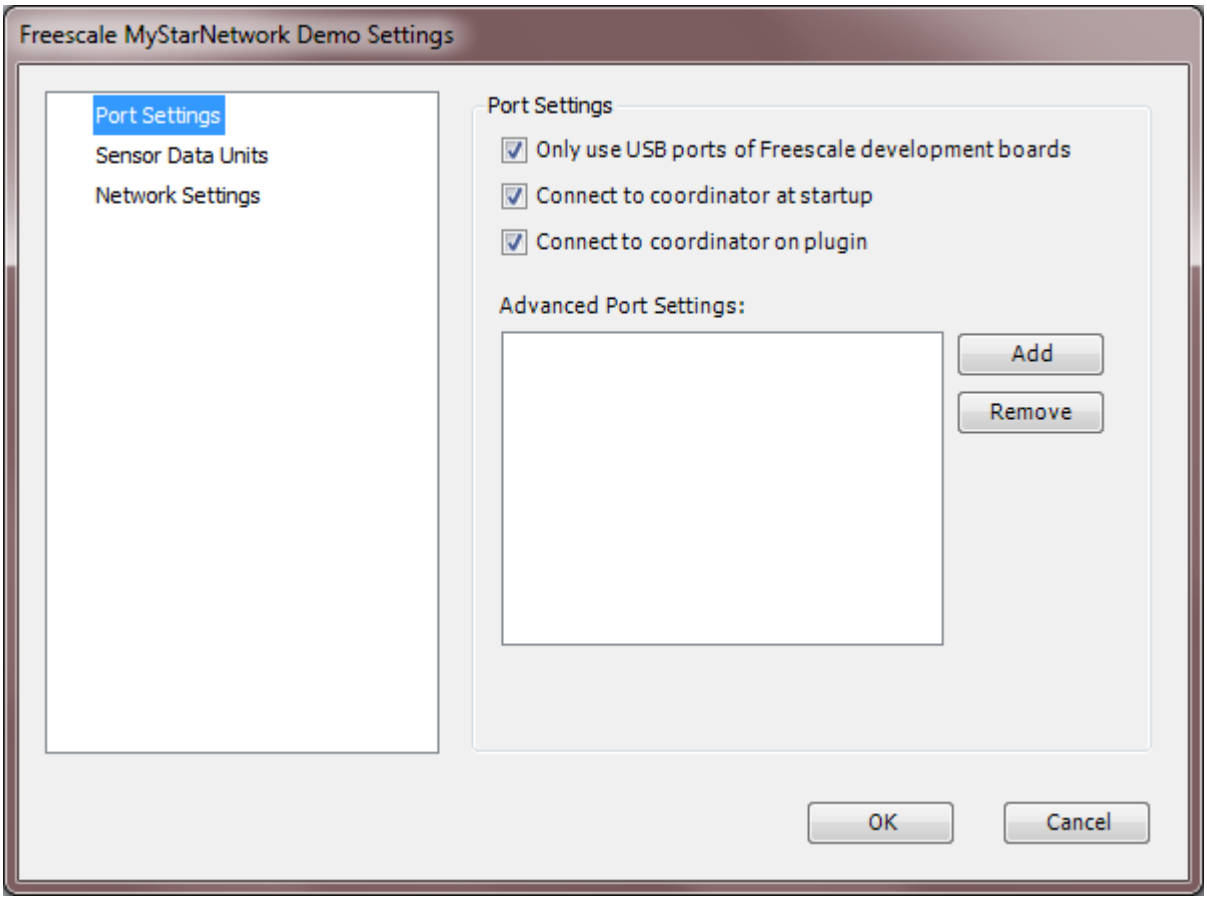


Figure 18. Port settings

The settings that can be performed are:

Use only USB ports of Freescale development boards

If checked, ports such as COM1 which are not detectable as development board USB ports are not used by the application even if they are active in the system.

Connect to coordinator at startup

If checked, the application will scan the active ports at startup, opening each one and sending an identification command to the port using default baud rate of 57600bps. If the response is received from the port that a coordinator is connected, the port is kept open, otherwise it is closed

Connect to coordinator on plug-in

If checked, the application will try and connect to the Coordinator as soon as a board is plugged in.

Advanced port settings

The list helps to do port pre-configuration; every time a port with the name appearing in the list is active in the system, the configuration specified in the list is used instead of the default one.

4.1.9.2. Sensor data units

The Sensor data units section allows user to set the temperature and pressure display units.

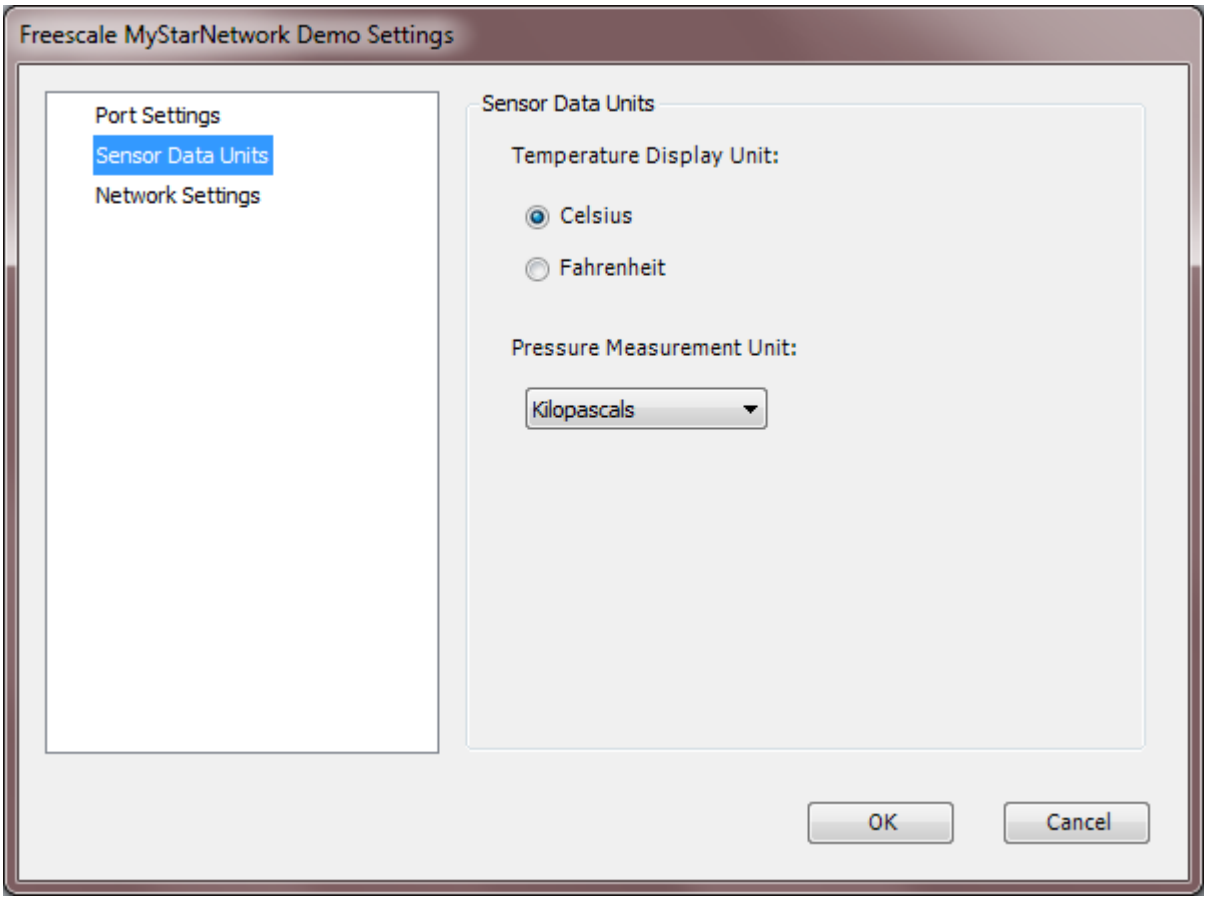


Figure 19. Sensor data units

Temperature

Degrees Celsius or degrees Fahrenheit.

Pressure

Kilo Pascals, PSI, Atmospheres, Torrs or Bars.

4.1.9.3. Network settings

The network settings section lets user configure time-outs for end devices.

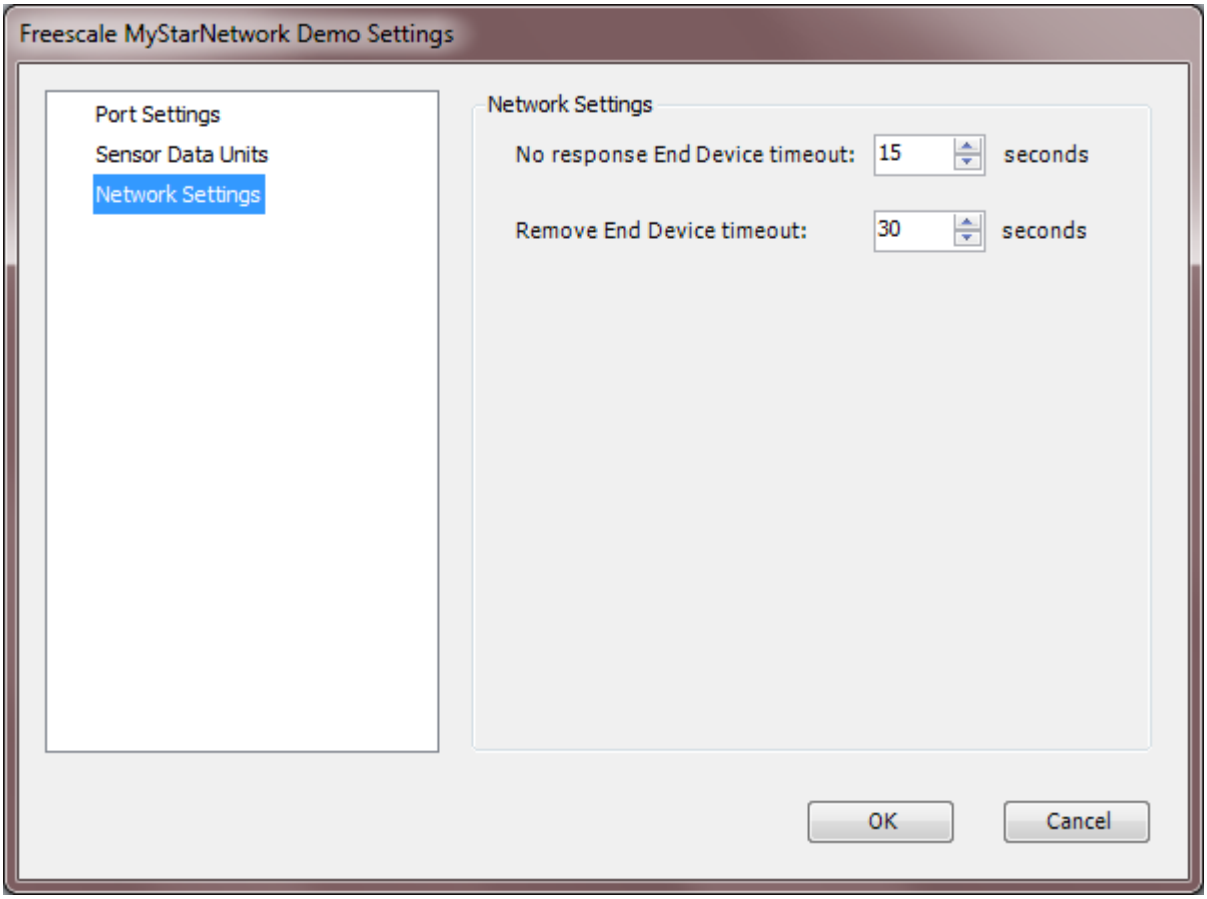


Figure 20. Network setting

No response end device timeout

If no sensor report has been received from an End Device for this period, the End Device is grayed out and put in an inactive state

Remove end device timeout

If no sensor report has been received from an End Device for this period, the End Device will be removed from the GUI.

4.2. Software implementation

The Purger is implemented as a separate module (*Purger.c*) and is used for tracking the data packets that are sent to the MCPS. The Purger allows the Coordinator to detect when End Devices disconnect from the PAN without invoking the DISASSOCIATE.request primitive.

4.2.1. Purger initialization

The Purger_Init function is called in the App_Init function before the first packet is sent. It initializes the array of packets, the packets life time and specifies the function that implements application-specific reaction to a purged packet. Its implementation is shown in the following code:

```
gPurgerExpireInterval = expireInterval;
ptAppProcessAddress = appFn;
for(i = 0; i < mPurgerNumPackets_c; i++)
{
  msgTrackArray[i].slotStatus = mPurgerUnusedSlot_c;
}
```

4.2.2. Purger tracking function

The Purger_Track function will add the packet handler to the list of tracked packets, along with the destination device's address and the expiration time.

```
uint8_t result = purgerNoSlot;
for(i = 0; i < mPurgerNumPackets_c; i++)
{
  if(mPurgerUnusedSlot_c == msgTrackArray[i].slotStatus)
  {
    msgTrackArray[i].msduHandle = msdu;
    msgTrackArray[i].destAddressHigh = destHigh;
    msgTrackArray[i].destAddressLow = destLow;
    msgTrackArray[i].expirationTime = time + gPurgerExpireInterval;
    msgTrackArray[i].slotStatus = mPurgerUsedSlot_c;
    result = purgerNoError;
    break;
  }
}
```

This function searches for an unused slot in the array of tracked packets and adds the new packet. It returns PurgerNoSlot error code if the array is full or PurgerNoError otherwise.

4.2.3. Purger check function

While in the listen state, the Coordinator periodically calls the Purger_Check function which sends purge requests to the MAC for the expired packets and calls the application-specific function handler. For example, the App_RemoveDevice defined in MyStarNetworkApp updates the mAddressesMap variable so it indicates that the address of the device that has left the PAN is available for another device as shown in the following code:

```
static void App_RemoveDevice(uint8_t shortAddrHigh, uint8_t shortAddrLow)
```

```
{
  /* Variable shortAddrHigh not used in this release*/
  uint8_t addHigh = shortAddrHigh;
  /* Remove the end device from the associated devices map - perhaps this should be a
call to an app provided fn.*/
  mAddressesMap &= ~(shortAddrLow);
  App_UpdateLEDs();
  UartUtil_Print("\n\rDisconnected device: ", gAllowToBlock_d);
  UartUtil_PrintHex(&shortAddrLow, 1, 0);
  UartUtil_Print(".\n\r", gAllowToBlock_d);
}
```

The Purger_check function iterates through the array of tracked packets. Each of the packets found in the array is checked. If it has expired, that is, the current time is greater than or equal to the packet's expiration time, a Purge.request is sent to the MCPS specifying the handle of the packet to be purged. Then the application-specific handler function pointed to by the ptAppProcessAddress is called. The number of purged packets is returned as shown in the following code:

```
for(i = 0; i < mPurgerNumPackets_c; i++)
{
  if((msgTrackArray[i].slotStatus == mPurgerUsedSlot_c) &&
     (time >= msgTrackArray[i].expirationTime) &&
     ((uint8_t)(msgTrackArray[i].expirationTime - time) > gPurgerExpireInterval))
  {
    purgeRequestPacket = MSG_Alloc(sizeof(primNwkToMcps_t) + sizeof(mcpsPurgeReq_t));
    if(purgeRequestPacket != NULL)
    {
      /* Create an MCPS-Purge Request message containing the msdu. */
      purgeRequestPacket->msgType = gMcpsPurgeReq_c;
      /* Specify the message to purge. */
      purgeRequestPacket->msgData.purgeReq.msduHandle = msgTrackArray[i].msduHandle;
      /* Send the Data Request to the MCPS */
      NR MSG_Send(NWK_MCPS, purgeRequestPacket);
#if mPurgerVerboseMode_c == 1
      UartUtil_Print("Sent purge request for ", gAllowToBlock_d);
      UartUtil_PrintHex(&purgeRequestPacket->msgData.purgeReq.msduHandle, 1,
gPrtHexNewLine_c);
#endif //mPurgerVerboseMode_c
      ptAppProcessAddress(msgTrackArray[i].destAddressHigh, msgTrackArray[i].destAddressLow);
      /* Remove it from the tracking list. */
      ret = Purger_Remove(msgTrackArray[i].msduHandle);
```

```
      result++;
    }
    else
    {
      /* If this happens, this function will get called until the memory manager
         has memory to allocate the packet. */
      UartUtil_Print("Can't allocate purge request packet.\n\r", gAllowToBlock_d);
    }
  }
}
```

4.2.4. Purger remove function

The packet is removed from the msgTrackArray list just after the purge message was sent. Another approach is to call the Purger_Remove() function when the MCPS-PURGE.confirm message was received. Given the fact that the purge request is treated locally in the MAC, it is not necessary to wait for the confirmation.

NOTE

There is a low probability that a packet that has its expirationTime field equal to 255 and the Purger_check function is not called when the time is 255, but called when time = 0. In this event, the packet is not purged according to the algorithm just described. It is dropped when the Purger_check function is called at time = 255.

The Purger_Remove function is called when the Coordinator receives a MCPS-DATA.confirmation. That is, when the packet has been received by the destination device.

```
uint8_t Purger_Remove(uint8_t msdu)
{
  uint8_t i;
  uint8_t result = purgerNoMessage;
  for(i = 0; i < mPurgerNumPackets_c; i++)
  {
    if(msgTrackArray[i].msduHandle == msdu)
    {
      msgTrackArray[i].slotStatus = mPurgerUnusedSlot_c;
#if mPurgerVerboseMode_c == 1
      UartUtil_Print("Untracked: ", gAllowToBlock_d);
      UartUtil_PrintHex(&msgTrackArray[i].msduHandle, 1, gPrtHexNewLine_c);
#endif //mPurgerVerboseMode_c
      result = purgerNoError;
      break;
```

```
    }
  }
  return result;
}
```

This function removes the specified packet identifier from the list of tracked packets. As shown in this code snippet, this is accomplished by marking the corresponding spot as unused. The PurgerNoMessage value is returned if there is no packet with the specified identifier, otherwise PurgerNoError is returned.

NOTE

The handler of a tracked packet is represented on one byte, so there can be at most 255 uniquely identified packets in the tracking list at a time. This can be achieved by choosing a *mMaxPendingDataPackets_c* value less than 255 and an appropriate expiration interval for the Purger. This ensures that the packets do not stay too long in the tracking list.

4.2.5. End device low-power mode overview

Low power consumption is one of the primary features of the 802.15.4 Standard. 802.15.4 Standard-compliant, battery powered End Devices can run for months or even years.

To implement the low power capability of the 802.15.4 stack, the Star Network Demonstration (End Device) uses the Power Management Library (PWRLIB) because it is fully configurable, simple to use and provides efficiency in conserving power.

The PWRLIB needs to be initialized before it can be used. This is performed in the init state of the End Device by calling the following function:

```
/* Initialize the Low Power Management module */
PWRLib_Init();
```

This function performs initialization of the PWRLIB and PWR functions. The End Device enters into Low Power Mode when no network activity is detected. Entering Low Power Mode is performed in the Idle task by calling the PWR_EnterLowPower() function as follows:

```
if(PWR_CheckIfDeviceCanGoToSleep())
{
  PWR_EnterLowPower();
}
```

Inside the `PWR_EnterLowPower` function, PWR_CheckForAndEnterNewPowerState (PWR_DeepSleep, cPWR_RTITicks) is called.

The PWR_DeepSleep parameter specifies that the transceiver is powered down and the MCU enters the Stop Mode.

The second parameter specifies the time interval in which the End Devices stay in Low Power Mode. The cPWR_RTITicks constant specifies the number of ticks from the Real Timer Interrupt. The total sleep interval is calculated by multiplying cPWR_RTITicks with cPWR_RTITickTime which specifies

the time interval between two consecutive ticks. These constants can be modified in the `LPMConfig.h` file to adjust the sleep interval.

The PWR_CheckForAndEnterNewPowerState function blocks the application until the power down time interval elapses or some event causes the MCU and the transceiver to resume. The return value specifies the cause for waking up the End Device. As shown in the previous code example, the End Device tests to see if a key was pressed and sets the appropriate flag.

5. Over-the-air Programming

This guide provides information about the 802.15.4 MAC Over-the-Air Programmer (MAC OTA Programmer) based on the Freescale 802.15.4 Media Access Controller (MAC) implementation.

The demo application presented in this document will load a new image (in form of a .srec file) using the Freescale TestTool 12 application into the OTA Server, then the image will be transfered Over-the-Air to an OTA Client, which will program and run the new image.

NOTE

Before starting the demo, the OTA Server and OTA Client devices must be associated.

5.1. OTA ProgrammingDemoApp (server)

This application is based on the MyWirelessApp Demo (Coordinator) with some portion of the FSCI module enabled. In this application, the server receives from a PC application a new image, which will be transferred Over-the-Air (OTA) to a Client device.

If `gOtapExternalMemorySupported_d` is set to TRUE, the server will first store the image into the External Memory.

5.2. OTA ProgrammingDemoApp (client)

This application is based on the MyWirelessApp Demo (EndDevice). In this application, the Client Device receives an image notify packet OTA when a new image is available or it can query the server any time. The image is transferred OTA, and the Client will store it into the External Memory. After the transfer is finished, the MCU resets giving control to the bootloader.

NOTE

The Bootloader represents a separate application and must be programmed separately into the MCU's Flash, before programming the OTAP Client demo application.

5.2.1. Project settings

To use the *OtaSupport* module and the OTAP Bootloader several configuration options must be set up in both the source files and the linker options of the toolchain.

First, the *OTASupport* and *Eeprom* module files must be included in the project. To configure the type of storage used the *gEepromType_d* preprocessor definition must be given a value. To use internal

storage put set the *gEepromType_d* value to *gEepromDevice_InternalFlash_c*. To use external storage set the *gEepromType_d* value to the appropriate type of EEPROM present on the board. The correct value for KW40Z demo boards is *gEepromDevice_AT45DB021E_c*.

Second, in the *Project->Options->Linker->Config*, the *gUseBootloaderLink_d* linker symbol must be defined with the value of 1. This will offset the code of the application, and will reserve the first 1/32 Flash for the bootloader code. The *gUseInternalStorageLink_d* should be set to 1 **only** if the value of *gEepromType_d* is set to *gEepromDevice_InternalFlash_c*. This symbol will reserve a Flash section between the application code and the NVM section (if NVM is used) to be used in the image upgrade process.

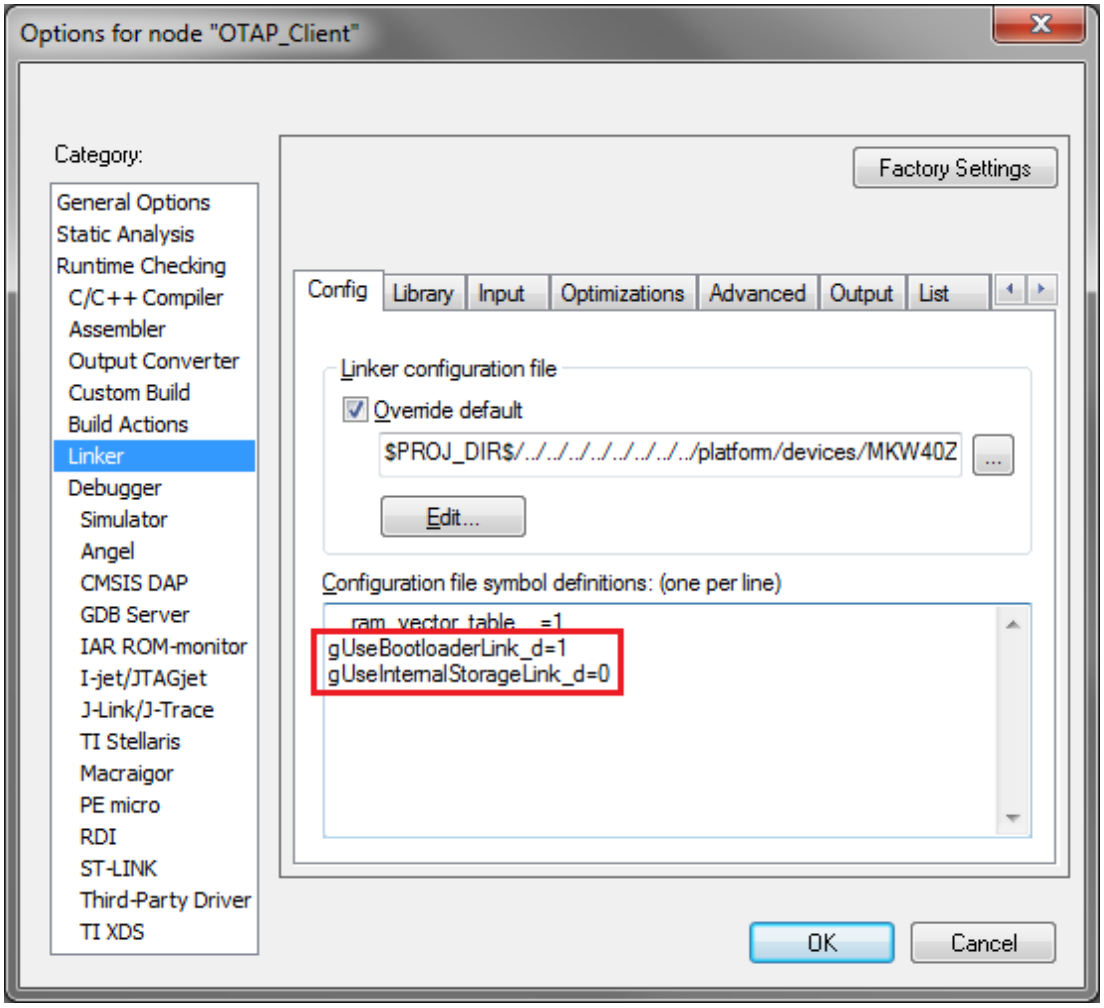


Figure 21. Linker Config - OTAP Client External Storage and Bootloader Configuration

5.3. Data structures and commands

The OTA file format is composed of a header followed by a number of sub-elements. The header describes general information about the file such as version, the manufacturer that created it, and the device it is intended for. Sub-elements in the file may contain upgrade data for the embedded device, certificates, configuration data, log messages, or other manufacturer-specific pieces.

Table 8. Sample OTA file

Octets	Variable	Variable	Variable	Variable
Data	OTA Header	Upgrade Image	Signer Certificate	Signature

NOTE

Signer Certificate and Signature sub-elements are not used by the described application.

The OTA header does not describe details of the particular sub-elements. Each sub-element is self-describing. With exception of a few sub-elements, the interpretation of the data contained is up to the manufacturer of the device.

5.3.1. Understanding OTA header format

Table 9. OTA header fields

Octets	Data Types	Field Names	Mandatory/Optional
4	Unsigned 32-bit integer	OTA upgrade file identifier	M
2	Unsigned 16-bit integer	OTA Header version	M
2	Unsigned 16-bit integer	OTA Header length	M
2	Unsigned 16-bit integer	OTA Header field control	M
2	Unsigned 16-bit integer	Manufacturer code	M
2	Unsigned 16-bit integer	Image type	M
4	Unsigned 32-bit integer	File version	M
2	Unsigned 16-bit integer	ZigBee Stack version	M
32	Character string	OTA Header string	M
4	Unsigned 32-bit integer	Total Image size (including header)	M
0/1	Unsigned 8-bit integer	Security credential version	O
0/8	IEEE Address	Upgrade file destination	O
0/2	Unsigned 16-bit integer	Minimum hardware version	O
0/2	Unsigned 16-bit integer	Maximum hardware version	O

5.3.1.1. OTA upgrade file identifier

The value is a unique 4-byte value that is included at the beginning of all ZigBee OTA upgrade image files to quickly identify and distinguish the file as being a ZigBee OTA cluster upgrade file, without having to examine the whole file content. This helps distinguishing the file from other file types on disk. The value is defined to be “0x0BEEF11E”.

5.3.1.2. OTA header version

The value enumerates the version of the header and provides compatibility information. The value is composed of a major and minor version number (one byte each). The high byte (or the most significant byte) represents the major version and the low byte (or the least significant byte) represents the minor version number. A change to the minor version means the OTA upgrade file format is still backward compatible, while a change to the major version suggests incompatibility.

The current OTA header version is 0x0100 with major version of “01” and minor version of “00”.

5.3.1.3. OTA header length

This value indicates full length of the OTA header in bytes, including the OTA upgrade file identifier, OTA header length itself to any optional fields. The value insulates existing software against new fields that may be added to the header. If new header fields added are not compatible with current running software, the implementations should process all fields they understand and then skip over any remaining bytes in the header to process the image or signing certificate. The value of the header length depends on the value of the OTA header field control, which dictates which optional OTA header fields are included. The length of the OTA header implemented by the demonstration apps is 60 bytes.

5.3.1.4. OTA header field control

The bit mask indicates whether additional information such as Image Signature or Signing Certificate are included as part of the OTA Upgrade Image.

Table 10. OTA header field control bitmask

Bits	Name
0	Security Credential Version Present
1	Device-Specific File
2	Device-Specific File
3-15	Reserved

Security credential version present bit indicates whether security credential version field is present or not in the OTA header.

Device-specific file bit in the field control indicates that this particular OTA upgrade file is specific to a single device.

Hardware version present bit indicates whether minimum and maximum hardware version fields are present in the OTA header or not.

NOTE

Only Hardware Versions are present in the header (Field Control = 0x04).

5.3.1.5. Manufacturer code

This is the ZigBee assigned identifier for each member company. When used during the OTA upgrade process, manufacturer code value of 0xffff has a special meaning of a wild card. The value has a ‘match all’ effect. OTA server may send a command with wild card value for manufacturer code to match all client devices from all manufacturers.

NOTE

The manufacturer code used by the demonstration apps is 0x1004.

5.3.1.6. Image type

The manufacturer should assign an appropriate and unique image type value to each of its devices to distinguish the products. This is a manufacturer-specific value. However, the OTA Upgrade cluster has reserved the last 64 values of image type value to indicate specific file types such as security credential,

log, and configuration. When a client wants to request one of these specific file types, it uses one of the reserved image type values instead of its own (manufacturer-specific) value when requesting the image via Query Next Image Request command.

Table 11. Image type values

File Type Values File Type Description	File Type Values File Type Description
0x0000 – 0xffbf Manufacturer-Specific	0x0000 – 0xffbf Manufacturer-Specific
0xffc0 Security credential	0xffc0 Security credential
0xffc1 Configuration	0xffc1 Configuration
0xffc2 Log	0xffc2 Log
0xffc3 – 0xfffe Reserved (unassigned)	0xffc3 – 0xfffe Reserved (unassigned)
0xffff Reserved: wild card	0xffff Reserved: wild card

Image type value of 0xffff has a special meaning of a wild card. The value has a ‘match all’ effect. For example, the OTA server may send Image Notify command with image type value of 0xffff to indicate to a group of client devices that it has all types of images for the clients. Additionally, the OTA server may send Upgrade End Response command with image type value of 0xffff to indicate a group of clients, with disregard to their image types, to upgrade.

NOTE

The image type used in this demonstration is 0x0000.

5.3.1.7. File version

For firmware image, the file version represents the release and build number of the image’s application and stack. The application release and build numbers are manufacturer-specific, however, each manufacturer should obtain stack release and build numbers from their stack vendor. OTA Upgrade cluster makes the recommendation regarding how the file version should be defined, in an attempt to make it easy for humans and upgrade servers to determine which versions are newer than others. The upgrade server should use this version value to compare with the one received from the client.

The server may implement more sophisticated policies to determine whether to upgrade the client based on the file version. A higher file version number indicates a newer file.

Table 12. Recommended file version definition

Application Release	Application Build	Stack Release	Stack Build
1 byte	1 byte	1 byte	1 byte
8-bit integer	8-bit integer	8-bit integer	8-bit integer

For example:

- • File version A: 0x10053519 represents application release 1.0 build 05 with stack release 3.5 b19.
- • File version B: 0x10103519 represents application release 1.0 build 10 with stack release 3.5 b19.
- • File version C: 0x10103701 represents application release 1.0 build 10 with stack release 3.7 b01.
- • File version B is newer than File version A because its application version is higher while File
- version C is newer than File version B because its stack version is higher.

For device-specific files, the file version value may be defined differently than that for firmware image to represent version scheme of different image types. For example, version scheme for security credential data may be different than that of log or configuration file. The specific implementation is manufacturer-specific.

NOTE

A binary-coded decimal convention (BCD) concept is used here for version number. This is to allow easy conversion to decimal digits for printing or display, and allows faster decimal calculations.

5.3.1.8. ZigBee stack version

This information indicates the ZigBee stack version that is used by the application. This provides the upgrade server the ability to coordinate the distribution of images to devices when the upgrades may cause a major jump that usually breaks the over-the-air compatibility, for example, from ZigBee Pro to ZigBee IP. The values as shown in the following table represent all currently available ZigBee stack versions.

ZigBee stack version values	Stack name
0x0000	ZigBee 2006
0x0001	ZigBee 2007
0x0002	ZigBee Pro
0x0003	ZigBee IP
0x0004 – 0xffff Reserved	Reserved

NOTE

The ZigBee Stack Version parameter is ignored by the applications.

5.3.1.9. OTA header string

This is a manufacturer-specific string that may be used to store other necessary information as seen fit by each manufacturer. The idea is to have a human readable string that can prove helpful during development cycle. The string is defined to occupy 32 bytes of space in the OTA header.

NOTE

The string used by this demonstration is “BeeStack Image File”.

5.3.1.10. Total image size

The value represents the total image size in bytes. This is the total of data in bytes that is transferred over-the-air from the server to the client. In most cases, the total image size of an OTA upgrade image file is the sum of the OTA header and the actual file data (along with its tag) lengths. If the image is a signed image and contains a certificate of the signer, then the Total image size also includes the signer certificate and the signature (along with their tags) in bytes.

This value is crucial in the OTA upgrade process. It allows the client to determine how many image request commands to send to the server to complete the upgrade process.

5.3.1.11. Minimum hardware version

The value represents the earliest hardware platform version this image should be used on. This field is defined as follows:

Table 13. Hardware version format

Version revision	Version revision
1 byte 1 byte	1 byte 1 byte
8-bit integer 8-bit integer	8-bit integer 8-bit integer

The high byte represents the version and the low byte represents the revision.

5.3.1.12. Maximum hardware version

The value represents the latest hardware platform this image should be used on. The field is defined the same as the Minimum Hardware Version (above).

The hardware version of the device should not be earlier than the minimum (hardware) version and should not be later than the maximum (hardware) version to run the OTA upgrade file.

5.3.1.13. Understanding subelement format

Sub-elements in the file are composed of an identifier followed by a length field, followed by the data. Theidentifier provides for forward and backward compatibility as new sub-elements are introduced. Existing devices that do not understand newer sub-elements may ignore the data.

Table 14. Subelement format

Octets	2-bytes	4-bytes	Variable
Data	Tag ID	Length Field	Data

Tag ID: The tag identifier denotes the type and format of the data contained within the sub-element. The identifier is one of the values from the following table.

Length Field: This value dictates the length of the rest of the data within the sub-element in bytes. It does not include the size of the Tag ID or the Length Fields.

Data: The length of the data in the sub-element must be equal to the value of the Length Field in bytes. The type and format of the data contained in the sub-element is specific to the Tag.

Tag Identifiers: Sub-elements are generally specific to the manufacturer and the implementation. However this specification has defined a number of common identifiers that may be used across multiple manufacturers.

Table 15. Tag identifiers

Tag identifiers description	Tag identifiers description
0x0000	Upgrade Image
0x0001	ECDSA Signature
0x0002	ECDSA Signing Certificate
0x0003 – 0xefff	Reserved
0xf000 – 0xffff	Manufacturer-Specific Use

Manufacturers may define tag identifiers for their own use and dictate the format and behavior of devices that receive images with that data.

NOTE

The TagId for the Sub Element containing the Sector Bitmap is 0xf000 and the TagId for the Sub Element containing the CRC is 0xf100.

5.4. OTA programmer commands

OTA Upgrade cluster commands, the frame control value is comprised of the following:

- Frame type is 0x01 — Commands are cluster-specific (not a global command).
- Manufacturer-specific is 0x00 — Commands are not manufacturer-specific.
- Direction — Is either 0x00 (client->server) or 0x01 (server->client) depending on the commands.

Command identifier values are shown in the following table.

Table 16. OTA upgrade cluster command frames

Command Identifier field value	Description	Direction	Disable default response	Mandatory / optional
0x00	Image Notify	Server -> Client(s) (0x01)	Set if sent as broadcast or multicast; Not Set if sent as unicast	O
0x01	Query Next Image Request	Client -> Server (0x00)	Not Set	M
0x02	Query Next Image Response	Server -> Client (0x01)	Set	M
0x03	Image Block Request	Client -> Server (0x00)	Not Set	M
0x04	Image Page Request	Client -> Server (0x00)	Not Set	O
0x05	Image Block Response	Server -> Client (0x01)	Set	M
0x06	Upgrade End Request	Client -> Server (0x00)	Not Set	M
0x07	Upgrade End Response	Server -> Client (0x01)	Set	M
0x08	Query-Specific File Request	Client -> Server (0x00)	Not Set	O
0x09	Query-Specific File Response	Server -> Client (0x01)	Set	O

5.4.1. OTA status code

OTA Upgrade cluster uses ZCL defined status codes during the upgrade process. These status codes are included as values in status field in payload of OTA Upgrade cluster’s response commands and in default response command.

Table 17. Status code defined and used by OTA upgrade cluster

Status code	Value	Description
SUCCESS	0x00	Success Operation
ABORT	0x95	Failed case when a client or a server decides to abort the upgrade process.
NOT_AUTHORIZED	0x7E	Server is not authorized to upgrade the client
INVALID_IMAGE	0x96	Invalid OTA upgrade image (ex. failed signature validation or signer information check or CRC check)
WAIT_FOR_DATA	0x97	Server does not have data block available yet
NO_IMAGE_AVAILABLE	0x98	No OTA upgrade image available for a particular client
MALFORMED_COMMAND	0x80	The command received is badly formatted. It usually means the command is missing certain fields or values included in the fields are invalid ex. invalid jitter value, invalid payload type value, invalid time value, invalid data size value, invalid image type value, invalid manufacturer code value and invalid file offset value
UNSUP_CLUSTER_COMMAND	0x81	Such command is not supported on the device
REQUIRE_MORE_IMAGE	0x99	The client still requires more OTA upgrade image files to successfully upgrade

5.4.2. Image Notify command

The purpose of sending Image Notify command is so the server has a way to notify client devices of when a new upgrade image is available for them. It eliminates the need for client devices having to check with the server periodically of when the new images are available.

However, all client devices still need to send in Query Next Image Request command to officially start the OTA upgrade process.

NOTE

This Demo Server sends an ImageNotify command only to the device with address 0x0001.

5.4.2.1. Payload format

Table 18. Format of Image Notify command payload

Field name	Payload type	Query jitter	Manufacturer code	Image type	(new) File version
Data type	8-bit Enumeration	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit
Length (bytes)	1	1	0/2	0/2	0/4

5.4.2.2. Payload field definitions

Image Notify command payload type

Payload type values	Description
0x00	Query jitter
0x01	Query jitter and manufacturer code
0x02	Query jitter, manufacturer code, and image type
0x03	Query jitter, manufacturer code, image type, and new file version
0x04-0xff	Reserved

Query jitter

To proceed, the device randomly chooses a number between 1 and 100 and compares it to the value of the QueryJitter value in the received message. If the generated value is less than or equal to the received value for QueryJitter, it queries the upgrade server. If not, then it discards the message and no further processing continues.

NOTE

This parameter is not used by the demonstration application (it's always 100).

Manufacturer code

Manufacturer code when included in the command should contain the specific value that indicates certain manufacturer. If the server intends for the command to be applied to all manufacturers then the value should be omitted.

Image type

Image type when included in the command should contain the specific value that indicates certain file type. If the server intends for the command to be applied to all image type values then the wild card value (0xffff) should be used.

(New) file version

The value is the OTA upgrade file version that the server tries to upgrade client devices in the network to. If the server intends for the command to be applied to all file version values then the wild card value (0xffffffff) should be used.

NOTE

The application uses 0xffffffff value. In this case downgrades can also be done.

5.4.3. Query Next Image Request command

5.4.3.1. Payload format

Table 19. Format of Query Next Image Request command payload

Field name	Field control	Manufacturer	Image type	(Current) File	Hardware

		code		version	version
Data type	8-bit Unsigned	16-bit Unsigned	Unsigned 16-bit	Unsigned 32-bit	Unsigned 16-bit
Length (bytes)	1	2	2	4	0/2

5.4.3.2. Payload field definitions

Query Next Image Request command field control

The field control indicates whether additional information such as device's current running hardware version is included as part of the Query Next Image Request command.

Table 20. Query next image Request field control bitmask

Bits	Name
0	Hardware Version Present
1-7	Reserved

Manufacturer code

The value is the device's assigned manufacturer code. Wild card value is not used in this case.

Image type

The value is between 0x0000–0xffbf (manufacturer specific value range).

(Current) file version

The file version included in the payload represents the device's current running image version. Wild card value is not used in this case.

(Optional) hardware version

The hardware version if included in the payload represents the device's current running hardware version. Wild card value is not used in this case.

5.4.4. Query Next Image Response command

5.4.4.1. Payload format

Field name	Status	Manufacturer code	Image type	File version	Image size
Data type	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit	Unsigned 32-bit
Length (bytes)	1	0/2	0/2	0/4	0/4

5.4.4.2. Payload field definitions

Query Next Image Response status

Only if the status is SUCCESS that other fields are included. For other (error) status values, only status field is present.

Manufacturer code

The value is the one received by the server in the Query Next Image Request command.

Image type

The value is the one received by the server in the Query Next Image Request command.

File version

The file version indicates the image version that the client is required to install. The version value may be lower than the current image version on the client if the server decides to perform a downgrade. The version value may be the same as the client's current version if the server decides to perform a reinstall.

However, in general, the version value should be higher than the current image version on the client to indicate an upgrade.

Image size

The value represents the total size of the image (in bytes) including header and all sub-elements.

5.4.5. Image Block Request command

5.4.5.1. Payload format

Table 21. Format of Image Block Request command payload

Field name	Field control	Manufacturer code	Image type	File version	File offset	Maximum data size	Request node address
Data type	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit	Unsigned 32-bit	Unsigned 8-bit	IEEE Address
Length (bytes)	1	2	2	4	4	1	0/8

5.4.5.2. Payload field definitions

Image Block Request command field control

Field control value is used to indicate additional optional fields that may be included in the payload of Image Block Request command. Currently, the device is required to support field control value of only 0x00; support for other field control value is optional.

Field control value 0x00 (bit 0 not set) indicates that the client is requesting a generic OTA upgrade file; hence, there is no need to include additional fields. The value of Image Type included in this case is manufacturer specific.

Field control value of 0x01 (bit 0 set) means that the client's IEEE address is included in the payload. This indicates that the client is requesting a device specific file such as security credential, log, or configuration; hence, the need to include the device's IEEE address in the image request command. The value of Image type included in this case is one of the reserved values that are assigned to each specific file type.

Table 22. Image Block Request field control bitmask

Bits name	Bits name
0	Request node's IEEE address Present
1-7	Reserved

Manufacturer code

The value is that of the client device assigned to each manufacturer by ZigBee.

Image type

The value is between 0x0000–0xffbf (manufacturer specific value range).

File version

The file version included in the payload represents the OTA upgrade image file version that is being requested.

File offset

The value indicates number of bytes of data offset from the beginning of the file. It essentially points to the location in the OTA upgrade image file that the client is requesting the data from. The value reflects the amount of (OTA upgrade image file) data (in bytes) that the client has received so far.

Maximum data size

The value indicates the largest possible length of data (in bytes) that the client can receive at once. The server respects the value and doesn't send data that is larger than the maximum data size. The server may send data that is smaller than the maximum data size value, for example, to account for source routing payload overhead if the client is multiple hops away. By having the client send both file offset and maximum data size in every command, it eliminates the burden on the server for having to remember the information for each client.

NOTE

The data size requested by the Client Device must be between

`gImageDataPacketMinSize_c` and `ImageDataPacketMaxSize_c`.

(Optional) request node address

This is the IEEE address of the client device sending the Image Block Request command.

NOTE

This parameter is not implemented in the demonstration applications.

5.4.6. Image Block Response command

5.4.6.1. Payload format

Field name	Field control	Manufacturer code	Image type	File version	File offset	Maximum data size	Request Node address
Data type	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit	Unsigned 32-bit	Unsigned 8-bit	IEEE Address
Length (bytes)	1	2	2	4	4	1	0/8

5.4.6.2. Payload field definitions

Image Block Request command field control

Field control value is used to indicate additional optional fields that may be included in the payload of Image Block Request command. Currently, the device is required to support field control value of only 0x00; support for other field control value is optional.

Field control value 0x00 (bit 0 not set) indicates that the client is requesting a generic OTA upgrade file; hence, there is no need to include additional fields. The value of Image Type included in this case is manufacturer specific.

Field control value of 0x01 (bit 0 set) means that the client's IEEE address is included in the payload. This indicates that the client is requesting a device specific file such as security credential, log, or configuration; hence, the need to include the device's IEEE address in the image request command. The value of Image type included in this case is one of the reserved values that are assigned to each specific file type.

Table 23. Image Block Request field control bitmask

Bits name	Bits name
0	Request node's IEEE address Present
1-7	Reserved

Manufacturer code

The value is that of the client device assigned to each manufacturer by ZigBee.

Image type

The value is between 0x0000–0xffbf (manufacturer specific value range).

File version

The file version included in the payload represents the OTA upgrade image file version that is being requested.

File offset

The value indicates number of bytes of data offset from the beginning of the file. It essentially points to the location in the OTA upgrade image file that the client is requesting the data from. The value reflects the amount of (OTA upgrade image file) data (in bytes) that the client has received so far.

Maximum data size

The value indicates the largest possible length of data (in bytes) that the client can receive at once. The server respects the value and doesn't send data that is larger than the maximum data size. The server may send data that is smaller than the maximum data size value, for example, to account for source routing payload overhead if the client is multiple hops away. By having the client send both file offset and maximum data size in every command, it eliminates the burden on the server for having to remember the information for each client.

NOTE

The data size requested by the Client Device must be between `gImageDataPacketMinSize_c` and `ImageDataPacketMaxSize_c`.

(Optional) request node address

This is the IEEE address of the client device sending the Image Block Request command.

NOTE

This parameter is not implemented in the demonstration applications.

5.4.7. Image Block Response command

5.4.7.1. Payload format

Field name	Status	Manufacturer code	Image type	File version	File offset	Data size	Image data
Data type	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit	Unsigned 32-bit	Unsigned 8-bit	Octet string
Length (bytes)	1	2	2	4	4	1	variable

5.4.7.2. Payload field definitions

Image Block Response status

The status in the Image Block Response command may be SUCCESS, ABORT, or WAIT_FOR_DATA.

Manufacturer code

The value is the same as the one included in Image Block/Page Request command.

Image type

The value is the same as the one included in Image Block/Page Request command.

File version

The file version indicates the image version that the client is required to install. The version value may be lower than the current image version on the client if the server decides to perform a downgrade. The version value may be the same as the client's current version if the server decides to perform a reinstall.

However, in general, the version value should be higher than the current image version on the client to indicate an upgrade.

File offset

The value represents the location of the data requested by the client. For most cases, the file offset value included in the (Image Block) response should be the same as the value requested by the client. For (unsolicited) Image Block responses generated as a result of Image Page Request, the file offset value increments to indicate the next data location.

Data size

The value indicates the length of the image data (in bytes) that is being included in the command. The value may be equal or smaller than the maximum data size value requested by the client.

Image data

The actual OTA upgrade image data with the length equals to data size value.

Current time and request time

If status is WAIT_FOR_DATA, the payload then includes the server's current time and the request time that the client retries the request command. The client waits at least the request time value before trying again. In case of a sleepy device, it may choose to wait longer than the specified time to not disrupt its sleeping cycle. If the current time value is zero that means the server does not support UTC time and the client treats the request time value as offset time. If neither time value is zero, and the client supports UTC time, it treats the request time value as UTC time. If the client does not support UTC time, it calculates the offset time from the difference between the two time values. The offset indicates the minimum amount of time to wait in seconds. The UTC time indicates the actual time moment that needs to pass before the client should try again.

NOTE

These fields are ignored by this Demo Application (are set to zero)

5.4.8. Upgrade End Request command

5.4.8.1. Payload format

Table 24. Format of Upgrade End Request command payload

Field name	Status	Manufacturer code	Image type	File version
Data type	Unsigned 8-bit	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit
Length (bytes)	1	2	2	4

5.4.8.2. Payload field definitions

Upgrade End Request command status

The status value of the Upgrade End Request command is SUCCESS, INVALID_IMAGE,

REQUIRE_MORE_IMAGE, or ABORT.

Manufacturer code

The value is that of the client device assigned to each manufacturer by ZigBee.

Image type

The value is between 0x0000–0xffbf (manufacturer specific value range).

File version

The file version included in the payload represents the newly downloaded OTA upgrade image file version.

5.4.9. Upgrade End Response command

5.4.9.1. Payload format

Field name	Manufacturer code	Image type	File version	Current time	Upgrade time
Data type	Unsigned 16-bit	Unsigned 16-bit	Unsigned 32-bit	Unsigned 32-bit	Unsigned 32-bit
Length (bytes)	2	2	4	4	4

5.4.9.2. Payload field definitions

The ability to send the command with wild card values for manufacturer code, image type, and file version is useful in this case because it eliminates the need for the server having to send the command multiple times for each manufacturer as well as having to keep track of all devices' manufacturers in the network.

Manufacturer code

Manufacturer code may be sent using wildcard value of 0xfffff to apply the command to all devices disregard of their manufacturers.

Image type

Image type may be sent using wildcard value of 0xfffff to apply the command to all devices disregard of their manufacturers.

File version

The file version included in the payload represents the newly downloaded OTA upgrade image file version. The value matches that included in the request. Alternatively, file version may be sent using wildcard value of 0xffffffff to apply the command to all devices regardless of their manufacturers.

Current time and upgrade time

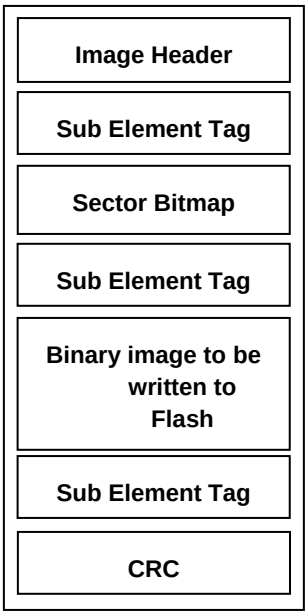
Current time and Upgrade time values are used by the client device to determine when to upgrade its running firmware image(s) with the newly downloaded one(s).

NOTE

These fields are ignored by this Demo Application (are set to zero).

5.5. OTA programmer image format

The following figures represent the OTA and EEPROM image formats.



PC Application and OTA Image Format

Client EEPROM Image Format

5.6. PC MAC OTA Programmer Application

The application used for uploading images (*.srec) to the OTAProgrammingDemoApp (Server) is integrated into the Freescale Test Tool, starting with Test Tool version 12.

Configure the Image Type, File Version, Min and Max HW version, and Sector Bitmap as shown below.

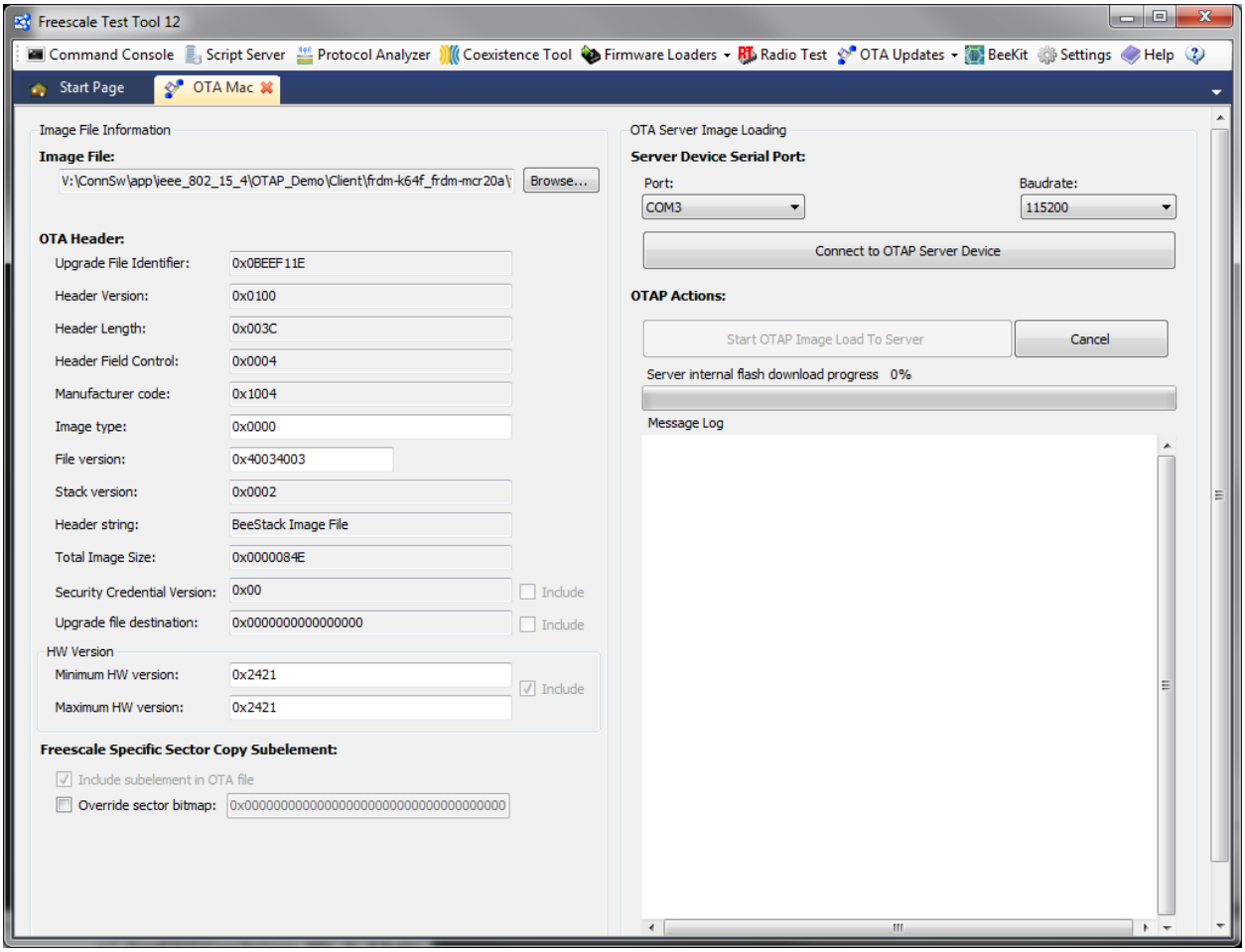


Figure 22. Parameter configuration in Freescale Test Tool 12

Every time an image file is selected, the following window appears:

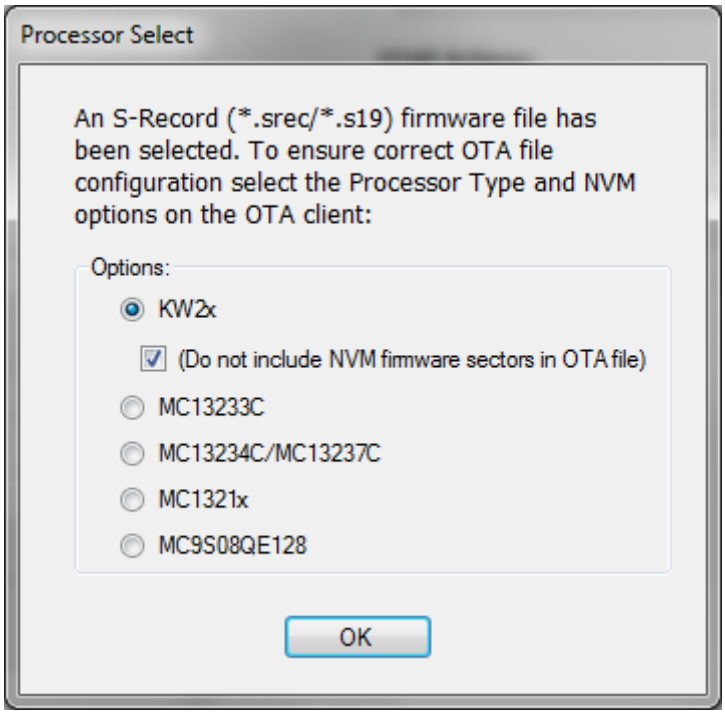


Figure 23. Processor selection

This window allow the selection of the processor for which the update is intended. In case of the Kinetis processor, there is an option to remove the NVM and ProductInfo sectors from the image.

To start downloading the new image to the Server, select the COM port on which the Coordinator board is connected, enter the baud rate (default bautrete is 115200 for Kinetis) and then press the *Open Serial Port to Server Gateway*. After this the transfer can be started by pressing Start Over the Air Programming button.

NOTE

No terminal sessions should be opened on the COM port on which the Server is connected.

5.7. Embedded applications

5.7.1. OTA Programmer Server (PAN coordinator)

To receive the image from the PC application, the FSCI module on the Server is enabled. The OtaSupport module is responsible for decoding the received packets. All data structures used by the OTA Programmer demo application are declared in *OtaMessages.h*.

The application is responsible for the initializarion of the FSCI and OtaSupport modules.

At startup, the function pointers used by the OtaSupport module to interact with the AppTask are initialized with the address of the functions that will process the specific event.

```
otaServer_AppCB_t mOTAcb = {
    NULL,
    SetOtapMode,
    QueryImageReqConfirm,
    ProcessImageChunk,
    NULL,
    NULL,
    NULL,
};
```

The initialization of the OtaSupport module is accomplished by calling *OTA_RegisterToFsci()*, with the FSCI interface used and a pointer to the OTA Server callbacks structure:

```
    OTA_RegisterToFsci(0, &mOTAcb);
```

The Message Types used by FSCI for OTA Programmer are:

```
    mFsciOtaSupportSetModeReq_c             = 0x28,
    mFsciOtaSupportStartImageReq_c          = 0x29,
    mFsciOtaSupportPushImageChunkReq_c      = 0x2A,
    mFsciOtaSupportCommitImageReq_c         = 0x2B,
    mFsciOtaSupportCancelImageReq_c         = 0x2C,
    mFsciOtaSupportSetFileVerPoliciesReq_c  = 0x2D,
    mFsciOtaSupportAbortOTAUpgradeReq_c     = 0x2E,
    mFsciOtaSupportImageChunkReq_c          = 0x2F,
    mFsciOtaSupportQueryImageReq_c          = 0xC2,
    mFsciOtaSupportQueryImageRsp_c          = 0xC3,
    mFsciOtaSupportImageNotifyReq_c         = 0xC4,
```

When one of the following message types are received over the serial interface, the *FSCI_OtaSupportHandlerFunc()* function is called, which calls the *OtaSupportCalback()* function.

The first commans received from the PC application (***mFsciOtaSupportStartImageReq_c*)** contains the image length without the length of the CRC Sub Element. The confirm to this command is used to notify the PC about the version of ZtcOtapSupport and the availability of an External Memory for the update process.

The second command received must be ***mFsciOtaSupportSetModeReq_c***. If an external Memory is available, this command will specify if it will be used or not in the upgrade process. The AppTask will be informed when this command is received thorough a callback (*otaServerSetModeCnf*).

If the upgrade mode (*gUpgradeMode* variable) is set to *gUseExternalMemoryForOtaUpdate_c* then the External Memory will be initialized.

The next step is to obtain a minimal set of information about the image by calling *OTA_QueryImageReq()* with the following parameters: device Id, manufacturer code, image type and file version. A value of 0xFFs has a special meaning of a wild card. The value has a ‘match all’ effect.

When the response is received, the AppTask will be informed thorough a callback (`otaServerQueryImageCnf`).

After this step, the Server will start requesting Image Chunks by calling *OTA_ImageChunkReq()*, and specifying the image offset, the chunk length and the device ID.

The PC application will send an ***mFsciOtaSupportPushImageChunkReq_c*** command, and the AppTask will be informed through a callback (*otaServerPushChunkCnf*).

The chunks are handled by the AppTask depeding on the availability of the external memory:

- If the external memory is used for OTA upgrade process, every time an image chunk is received, it will be stored in this memory. After the last chunk is received, the Server device will send OTA an *ImageNotify*. When a Client device request an image block, the block is read from the External Memory.
- If no external memory is used by the OTA Server for the OTA upgrade process, the server will send an *ImageNotify* when the *mFsciOtaSupportQueryImageRsp_c* command is received from the PC. The image chunks are requested from the PC only when an *ImageBlockRequest* command is received OTA vfrom a Client device.

If an error occurred during the image transfer (PC to Server device) the PC application sends ***mFsciOtaSupportCancelImageReq_c***, and the transfer is stopped. An error message will be displayed by the MAC OTA Programmer PC application.

After all chunks have been transferred, the PC app sends an ***mFsciOtaSupportCommitImageReq_c*** command that marks the end of the transfer.

NOTE

No processing will be done by the Server over the received image chunks.

5.7.2. OTA Programmer Client (end device)

Before sending/receiving any data, the device must join a PAN. This can be accomplished by pressing any switch on the Client's board.

After this operation the Client device can query the server for a new image (by pressing any switch on the board) or it can wait a notification from the server.

```
Press any switch to query for a new image.

->Sent Query Next Image req

No image available!
```

From the *ImageNotify* and *QueryNextImageResponse* commands the Client can determine whether it is necessary to download the image. The Client device must send an *ImageBlockRequest* with the desired file offset and payload size. The requested size must less or equal to the value of the *gImageDataPacketMaxSize_c* define (OtaMessages.h).

When the Client has received at least *sizeof(ImageHeader_t)* bytes, it will decode the Image Header into a RAM structure.

After the Header a variable number of Sub Elements can be received in a random order. The Client will interpret Sub Elements that have one of the following TagId:

```
gUpgradeImageTadId 0x0000
gSectorBitmapTagId 0xf000
gCRCTagId 0xf100
```

Other Sub Elements will be ignored but they will still be requested, to compute their CRC. The CRC is computed using the *OTA_CrcCompute()* function, over the whole image except the entire CRC Sub Element (CRC TagId, CRC TagLength and CRC field).

The Image Sub Element is saved into the External Memory using *OTA_StartImage()* to initiate the process and the *OTAP_PushImageChunk()* to store the received chunk.

A progress bar and a percentage are displayed on the Client's terminal console using the *PrintProgress()* function:

```
->Received an image Notify
->Sent Query Next Image req
OTA Transfer Progress:
100% [=========================]
```

If the computed CRC and the received CRC are the same the Client commits the image using the *OTAP_CommitImage()* function. This operation actually writes the real image size and the sector bitmap into the EEPROM memory.

Next an *UpgradeEndRequest* must be sent to the server with the status of the upgrade. If a response (*UpgradeEndResponse*) with the success status is received from the Server, a Flag will be written to FLASH (see *OTA_SetNewImageFlag()* function) to inform the Bootloader that a new image is available, and it will reset the MCU:

```
Transfer Successful!
Resetting MCU…
```

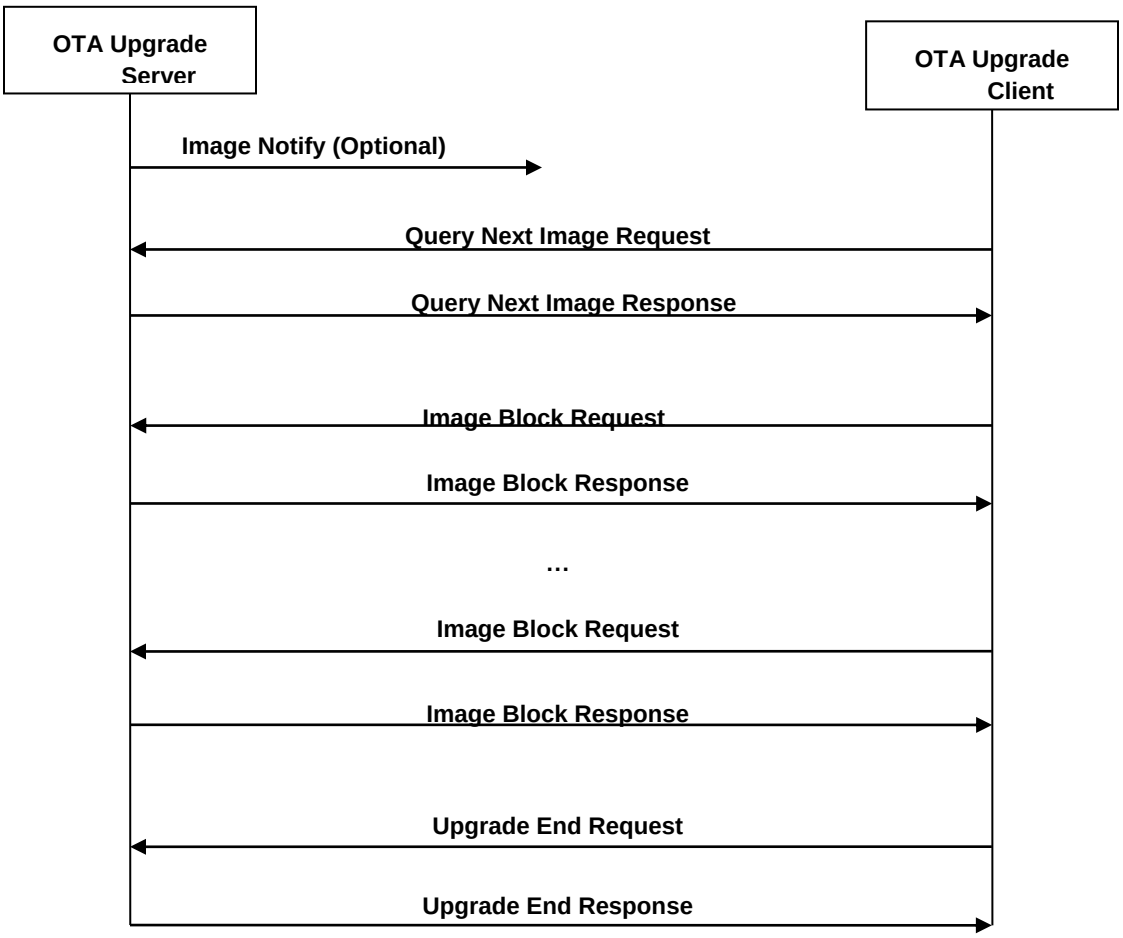
The Bootloader sees the flag and starts transferring the image from the storage memory into the MCU's Flash. After this step the MCU resets again and the new image will run.

During the OTA transfer if the Server doesn't respond in *gOtaTimeout_c* three times, the process is aborted.

NOTE

The Server and the Client can store only one image in the EEPROM memory.

5.7.3. OTA transfer diagram



6. Low Energy Demo

The 802.15.4e Media Access Control (MAC) layer provides two Low Energy mechanisms for application scenarios targeting an even more low power focus than previous 802.15.4 stadards have had. These features are Coordinated Sampled Listening (CSL) and Receiver Initiated Transmission (RIT), each being designed for certain use cases.

Usually the low power use case for an 802.15.4 network node is aimed for an End Device role, which could be battery powered, while the Coordinator implements additional MAC laye data exchange behaviorr in order to allow sleeping devices to receive data.

The Low Power application scenario is expected to achieve lower energy consumption providing a throughput and application data latency tradeoff. The latency in transmitting a packet is one of the main factors when choosing between CSL and RIT.

Coordinated Sampled Listening is recommended for an application data latency tolerance of few seconds, while the Receiver Initiated Transmission is designed for higher latency of up to tens of seconds. Also, RIT can be implemented in areas where the receiver on periods or the CSL continuous transmission time is too high can be limited by local regulation.

The Low Energy Applications do not require configuration of a serial interface. The application generated traffic is to be monitored using a sniffer and follow the behavior of the Coordinator and End Device when they periodically generate traffic.

The CSL and RIT demos cannot be run simultaneously, the type of demo scenario is configured at compile time on both Coordinator and End Device. The available demos are:

- **CSL** Demo PAN Coordinator
- **CSL** Demo End Device
- **RIT** Demo PAN Coordinator
- **RIT** Demo End Device

6.1. Coordinated sampled listening

6.1.1. Understanding coordinated sampled listening

Coordinated sampled listening is a method through which End Devices periodically monitor the channel in order to detect any incoming traffic. This is a channel sample which consists of an energy detection scan, lasting very short time, usually hundreds of microseconds. The Coordinator must generate additional MAC layer traffic before sending the actual higher layer data frame in order to make the receiving nodes aware of the incoming data frame and enable their receiver. Also, these 802.15.4e special MAC wakeup frames will announce the moment when the Application data frame will be sent.

CSL parameters that need to be configured on devices:

- CSL Max Period – on the Coordinator – this is the maximum listening period inside the network
- CSL Period – on the End Device – this is the distance between two channel samples

When the End Device performs the Energy Detection scan, the Coordinator generated traffic will trigger the energy level and the node enables its receiver. One of the wakeup frames is received and the node will enable its receiver for the higher layer data frame.

The acknowledgment of the frame includes a new behavior. The End Device will reply to the application data frame with an 802.15.4e Enhanced Acknowledgment. This frame will include the listening schedule of the device which is represented by the CSL period between channel samples and the absolute time in MAC symbols of the next channel sample.

Upon receiving the Enhanced Acknowledgment, the Coordinator will store this listening information in order to properly synchronize future communication and reduce transmission overhead.

The first communication to a new device is unsynchronized as the Coordinator does not know the listening schedule of the peer. What it does know is that the CSL Max Period will overlap any channel sample inside the network, so it starts a wakeup sequence having this maximum duration.

After the initial data exchange has been performed between peers, the Coordinator stores the listening schedule and following communication to the node will use a short wakeup sequence in order to just overlap the energy detection, announce the data transmission time through a wakeup frame and send the data frame.

6.1.2. CSL configuration on PAN coordinator

6.1.2.1. Macro definitions

The CSL demo is enabled through the following macro definition:

```
#define mDemoCsl_d 1
```

The Coordinator CSL Max period value is configured through the following macro definition. The value is calculated as a multiple of 10 MAC symbols. For instance, for a 20 microsecond MAC symbol duration, if the desired period is 1.5 seconds we get the following:

```
#define gMPibCslMaxPeriodValue_c 0x1D4C
```

NOTE

Ensure that the CSL Max period value is higher than any CSL period inside the network

The Coordinator higher layer periodically sends data to the MAC layer. The inter packet delay is configured through the following macro definition expressed in milliseconds:

```
#define gDataTxIntervalMs_c 10000
```

The maximum number of packets sent to the MAC layer for transmission is configured through the following macro definition:

```
#define mDefaultValueOfMaxPendingDataPackets_c 1
```

The other macro definitions are similar to the other applications provided.

6.1.2.2. CSL enablement

CSL is enabled by the PAN Coordinator before the periodically data transmission is performed.

Example 13. CSL Configuration on PAN coordinator

```
void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
```

```
    if( pMsg != NULL )
    {
        /* Initialize the MAC CSL Max Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibCslMaxPeriod_c;
        value = gMPibCslMaxPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.1.3. CSL configuration on end device

6.1.3.1. Macro definitions

The CSL demo is enabled through the following macro definition:

```
#define mDemoCsl_d 1
```

The CSL demo can be run in conjunction with a low power mode so that the node would sleep between channel samples, not just disable the receiver. This behavior is enabled through the following macro configuration:

```
#define mDemoLowPower_d 0
```

The CSL period represents the interval between the start of a channel sample expressed in a multiple of 10 MAC symbols. A common value is 1 second, which on a 20us symbol duration would result in the following value:

```
#define gMPibCslPeriodValue_c    0x1388
```

The other macro definitions are similar to the other applications.

6.1.3.2. CSL enablement

CSL is enabled on the End Device once the user has pressed a switch on the End Device board and the channel sample process begins.

The demo requires the setting of the addressing information on the End Device: the MAC short address and the PAN Id. Also, the channel the Coordinator has started the network on is set. These steps can be implemented similar to the other apps, but to obtain a simple approach to the Low Energy use case are presented as follows.

Example 14. CSL enablement on end device

```
void App_ConfigureMAC_LE_PIBs()
```

```
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC PAN ID */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macPanId;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC logical channel */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibLogicalChannel_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&macLogicalChannel;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC CSL Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibCslPeriod_c;
        value = gMPibCslPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
```

```
    }
}
```

6.1.3.3. CSL demo using low power

As presented before, the CSL demo can be run in conjunction with a sleep mode configure through the board corresponding low power mode. The configuration is performed through the following macro definition:

```
#define cPWR_DeepSleepMode 2
```

The available low power modes are:

- Deep Sleep mode 1
  - MCU is in LLS mode
  - Radio is in sleep mode
- Deep Sleep mode 2
  - MCU is in VLPS mode
  - Radio is in sleep mode

The only recommended setting the default value of deep sleep mode 2, as the time synchronization cannot be kept in the first variant. The synchronization is important when leaving the low power mode in order to adjust the running time to the absolute value spent sleeping. The low power clock may encounter some drifts and improperly adjust the running clock, resulting in an incorrect absolute time.

This behavior leads to incorrect channel sampling time which may eventually resort to packet loss. On the sniffer, the packet loss will oblige the Coordinator to revert to the maximum listening schedule wakeup sequence as the listening period has either changed or a clock drift has happened. The CSL mechanism has the recovery mechanism of using the long wakeup sequence and rediscover the listening schedule of the peeer device.

The initial state is of the demo is to disallow device to sleep:

Example 15. Low power initial state

```
#if mDemoLowPower_d
        PWR_CheckForAndEnterNewPowerState_Init();
        PWR_DisallowDeviceToSleep();
#endif
```

Once a switch is pressed the application allows the device to enter sleep mode entirely dependent on the MAC layer, so the control is fully leaved to the MAC events that will generate sleeping periods depending on the channel sampling period.

Example 16. Low power state after pressing a switch

```
#if mDemoLowPower_d
          /* Allow sleep by MAC LE events */
          PWR_AllowDeviceToSleep();
#endif
```

The low power enter mode is similar to the other MAC applications, on the Application idle task a check is made that all layers allow low power entry. The application allowed sleep mode enter after a switch has been pressed and then MAC events will drive this behavior:

Example 17. Application low power entry on idle task

```
void App_Idle_Task(uint32_t param)
{
    while(1)
    {
#if mDemoLowPower_d
        if( PWR_CheckIfDeviceCanGoToSleep() )
        {
            Led1Off();
            Led2Off();
            Led3Off();
            Led4Off();

            PWR_EnterLowPower();

            Led1On();
            Led2On();
            Led3On();
            Led4On();
        }
#endif
        if( !gUseRtos_c )
        {
            break;
        }
    }
}
```

The effect of the Low Power entry can be followed on the board leds. During sleep all the leds are off, while in run mode, they are on.

6.2. Receiver-initiated transmission

6.2.1. Understanding receiver-initiated transmission

Receiver Initiated Transmission is a mechanism describing how devices wanting to transmit a packet to a peer first start with a receiver on period that announces the availability of the peer to receive a packet.

When enabled, RIT requires nodes to enable their receiver only during dedicated intervals. These intervals are announce through a special 802.15.4e MAC command frame, the RIT Data Request. Immediately after sending the RIT Data Request, MAC layer will enable the receiver with a duration specified by the next higher layer and then will turn off the receiver until the next interval to send RIT Data Request again and the behavior will repeat.

The sender of a MAC data frame will start its receiver for a duration configured by the higher layer in waiting for its peer RIT Data Request. After receiving the frame, the sender will transmit the MAC data frame and consider the process ended.

The RIT configuration parameters are expressed in MAC symbols as follows:

- RIT Tx Wait duration – Rx on time spent by sender in searching for a RIT Data Request from its peer
- RIT Data Wait Duration – Rx on time spent by receiver after sending a RIT Data Request
- RIT Period – interval between RIT Data Requests
- RIT IE – listening schedule(optional)

The 802.15.4e standard includes two types of RIT functioning modes:

- Without listening schedule as described above
- With listening schedule included as a payload in the RIT Data Request

The RIT listening schedule is optional and can be configured in the following mode. The higher layer can set a schedule to turn the receiver on between two RIT Data Requests described by:

- T0 – time until first receiver on enablement
- T – time to keep the receiver on
- N – number of receiver on periods inside the RIT interval

Note that the MAC Role is not important when enabling RIT, both the Coordinator as sender and the End Device as receiver are enabling same parameters.

6.2.2. RIT configuration

6.2.2.1. Macro definitions

RIT demo can be enabled through the following macro definition:

```
#define mDemoRit_d 1
```

The RIT parameters are configured as follows in MAC symbols, taking into consideration that they were computed for a 20us MAC symbol duration.

RIT Tx wait duration or the time spent in searching for a RIT Data Request when wanting to transmit a MAC frame:

```
#define gMPibRitTxWaitDurationValue_c    0x4E2 // 24 sec
```

NOTE

Ensure that the RIT Tx Wait of the sender is greater than the RIT Data Wait of any of the peers it is to communicate with.

Rit Data wait duration or the time spent having the receiver on after announcing availability through a RIT Data Request:

```
#define gMPibRitDataWaitDurationValue_c  0x34  //  1 sec
```

RIT Interval or the time between sending the RIT Data Request command frames:

```
#define gMPibRitPeriodValue_c            0x410 // 20 sec
```

Optionally, the listening schedule can be set as follows for 3 receiver on enablements of 5 seconds each in a 20 seconds interval between RIT Data Requests(considering the above parameter values):

```
#define gMPibRitIeT0_c                   0x9C  //  3 sec
#define gMPibRitIeT_c                    0x104 //  5 sec
#define gMPibRitIeN_c                    0x03  //  3 Rx On repeats
```

The Coordinator higher layer periodically sends data to the MAC layer. The inter packet delay is configured through the following macro definition expressed in milliseconds:

```
#define gDataTxIntervalMs_c 10000
```

The maximum number of packets sent to the MAC layer for transmission is configured through the following macro definition:

```
#define mDefaultValueOfMaxPendingDataPackets_c 1
```

The other macro definitions are similar to the other applications provided.

6.2.2.2. RIT enablement without listening schedule

On the Coordinator, RIT is enabled without a listening schedule. The demo starts once the application starts and the higher layer generates data packets periodically.

Example 18. RIT without listening schedule

```
void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
```

```
    if( pMsg != NULL )
    {
        /* Initialize the MAC RIT Tx Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitTxWaitDuration_c;
        value = gMPibRitTxWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Data Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitDataWaitDuration_c;
        value = gMPibRitDataWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitPeriod_c;
        value = gMPibRitPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.2.2.3. RIT enablement with listening schedule

On the End Device, the RIT demo is configured with a listening schedule. RIT is enabled at the press of a switch on the board.

For simplicity purposes, the End Device will set the addressing information consisting of the MAC short address and the PAN ID the Coordinator is using. Also, the MAC channel is set to the value the Coordinator has started the network on. These values can be obtained similar to the other application scenarios, but they are configured as such for the sole purpose of focusing on RIT configuration.

Example 19. RIT with optional listening schedule

```
void App_ConfigureMAC_LE_PIBs()
{
    uint32_t value;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg;

    /* Allocate a message for the MLME (We should check for NULL). */
    pMsg = MSG_AllocType(mlmeMessage_t);
    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC PAN ID */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibPanId_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macPanId;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* MAC logical channel */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibLogicalChannel_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&macLogicalChannel;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Tx Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitTxWaitDuration_c;
        value = gMPibRitTxWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
```

```
        /* Initialize the MAC RIT Data Wait Duration */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitDataWaitDuration_c;
        value = gMPibRitDataWaitDurationValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint8_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT IE Listening schedule */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitIe_c;
        ((macRitIe_t*)&value)->T0 = gMPibRitIeT0_c;
        ((macRitIe_t*)&value)->N = gMPibRitIeN_c;
        ((macRitIe_t*)&value)->T = gMPibRitIeT_c;
        pMsg->msgData.setReq.pibAttributeValue = (macRitIe_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Initialize the MAC RIT Period */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibRitPeriod_c;
        value = gMPibRitPeriodValue_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint32_t*)&value;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Free message, all requests were sync */
        MSG_Free(pMsg);
    }
}
```

6.2.2.4. RIT demo using low power

As presented before, the RIT demo can be run in conjunction with a sleep mode configure through the board corresponding low power mode. The configuration is performed through the following macro definition:

```
#define cPWR_DeepSleepMode 2
```

The available low power modes are:

- Deep Sleep mode 1
  - MCU is in LLS mode
  - Radio is in sleep mode
- Deep Sleep mode 2

- MCU is in VLPS mode
- Radio is in sleep mode

The only recommended setting the default value of deep sleep mode 2, as the time synchronization cannot be kept in the first variant. The RIT demo is not as much affected by the clock drift as the CSL demo, should the user require usage of the first deep sleep mode. The effect of the clock drift can be seen by a retransmission of the MAC data frame with Ack required, but the second attempt is successfully acknowledged.

The synchronization is important when leaving the low power mode in order to adjust the running time to the absolute value spent sleeping. The low power clock may encounter some drifts and improperly adjust the running clock, resulting in an incorrect absolute time.

The initial state is of the demo is to disallow device to sleep:

Example 20. Low power initial state

```
#if mDemoLowPower_d
        PWR_CheckForAndEnterNewPowerState_Init();
        PWR_DisallowDeviceToSleep();
#endif
```

Once a switch is pressed the application allows the device to enter sleep mode entirely dependent on the MAC layer, so the control is fully leaved to the MAC events that will generate sleeping periods depending on the channel sampling period.

Example 21. Low power state after switch press

```
#if mDemoLowPower_d
          /* Allow sleep by MAC LE events */
          PWR_AllowDeviceToSleep();
#endif
```

The low power enter mode is similar to the other MAC applications, on the Application idle task a check is made that all layers allow low power entry. The application allowed sleep mode enter after a switch has been pressed and then MAC events will drive this behavior:

Example 22. Application low power entry on idle task

```
void App_Idle_Task(uint32_t param)
{
    while(1)
    {
#if mDemoLowPower_d
        if( PWR_CheckIfDeviceCanGoToSleep() )
        {
            Led1Off();
            Led2Off();
```

```
            Led3Off();
            Led4Off();

            PWR_EnterLowPower();

            Led1On();
            Led2On();
            Led3On();
            Led4On();
        }
#endif
        if( !gUseRtos_c )
        {
            break;
        }
    }
}
```

6.3. Running the demos

To run CSL demo follow these steps:

1. Setup sniffer on the channel configured in the applications
2. Compile MAC_LE_DEMO_Coordinator with CSL enabled
3. Run Coordinator application and you can see the wakeup frames and data frames on the sniffer
4. Compile MAC_LE_DEMO_EndDevice with CSL enabled
5. Start End Deivce application
6. Press any switch on the End Device board to start channel sampling
7. Monitor sniffer to see the packets exchanged between devices

To run the RIT demo follow these steps:

1. Setup sniffer on the channel configured in the applications
2. Compile MAC_LE_DEMO_Coordinator with RIT enabled
3. Start Coordinator application and you can see Coordinator RIT Data Requests on the sniffer
4. Compile MAC_LE_DEMO_EndDevice with RIT enabled
5. Run End Deivce application

6. Press any switch on the End Device board to enable RIT and you can now see RIT Data Requests from End Device also
7. Monitor sniffer to see the packets exchanged between devices

7. Time-slotted Channel-hopping Demo

The 802.15.4e standard introduces a Media Access Control layer (MAC) feature that allows network nodes to hop on a set of channels based on a time division manner, the mechanism being Time Slotted Channel Hopping (TSCH). TSCH allows networks nodes to exchange data frames on various frequencies being recommended for regions where local regulations limit the continuous transmission time on a frequency of a node to a certain duration.

The TSCH mechanism allows numerous topologies from star to mesh, introduces multiple patterns of communication inside the network, adds dedicated transmission links between nodes, and removes chances of packet collisions.

The available demo applications are:

- TSCH Demo PAN Coordinator
- TSCH Demo End Device

There is no need to configure a serial interface to run the demo, just run the applications and monitor traffic on one or multiple sniffers set on the channels the two devices hop on.

TSCH communications occur on units of time called timeslots. In a timeslot, a single frame can be exchanged with an optional acknowledgment. So, a node is in transmission and its peer must be configured to be in reception for the same timeslot and channel.

7.1. Understanding TSCH

7.1.1. Slotframes

A slotframe is a collection of timeslots reapeating in time. A slotframe is represented by a slotframe handle and its size. The size of a slotframe specifies when each timeslot of the frame is repeated again. Multiple slotframe with various sizes can be set on a device. This allows for various communication patterns to be configured. Lower slotframe handles have priority over higher slotframe handles, while any transmission has higher priority over reception regardless of the slotframe handle.

The coordinator that starts the network and enables TSCH will set the absolute slot number (ASN) inside the network which is used in synchronization to the network by other joining devices. The absolute slot number represents the number of timeslots that has elapsed since the start of the network. It is also used as a part of the CCM* nonce when security is enabled and TSCH is enabled, instead of the classic frame counter. This introduces a time dependent component in the security process, as the ASN increments with every timeslot that passes, rather than incrementation only when frames are sent.

7.1.2. Links

A link is an oriented communication between two nodes taking place on a certain timeslot with a certain channel offset. Links are a pair wise communication between nodes, one of the peers being assigned to transmit and the other to be in reception.

Links are assigned to slotframes, every link handle must be provided with a slotframe handle. Other parameters of a link are a timeslot and a channel offset. The timeslot assigned is the timeslot inside the slotframe when the link is available for use and the channel offset is used in the computation of the channel on which the communication is to take place.

Links can be of multiple types:

- Tx Links – used to perform Transmission
- Rx Links – used to perform Reception
- Normal Links – used to send MAC data frames
- Advertising Links – used to send MAC Enhanced Beacons and/or MAC data frames
- Shared Links(Tx) – allocated to more than one device for Tx
- Timekeeping Links(Rx) – used for maintaining clock source from peer node

Certain combinations have no meaning for instance Shared Rx Links or Timekeeping Tx Links.

7.1.3. Channel hopping

Channels can be configured by setting two PIBs:

- gMPibHoppingSequenceLength_c – length of the channel hopping sequence
- gMPibHoppingSequenceList_c – list of channels in the hopping sequence

The channel hopping can include same channel multiple times.

The current channel in a timeslot is calculated depending on the slotframe and link set for that timeslot, following this formula:

Channel = HoppingSequenceList [(ASN + link channel offset) % HoppingSequenceLength]

Note that a channel is set if and only if a link from any slotframe is selected for that timeslot to perform either a transmission or a reception.

The channel offset is present to allow for each timeslot a number of up to HoppingSequenceLength synchronious communications on different channels, so inside a slotframe the node communication matrix is slotframe size in timeslots multiplied by the hopping sequence length channels.

7.1.4. Network joining

Timeslot communication makes the MAC Active Scan procedure to have almost to zero chances to success, so when TSCH is enabled the recommended way to join devices to a network is to perform a Passive Scan.

This requires the higher layer of the coordinator to set an advertising procedure. This represents the set procedure of a Tx link, with the advertising type set and usually a broadcast address as a node address.

When the higher layer of a Coordinator then decides to advertise the parameters required for a node to join the network, it must issue the MLME-BEACON.Request primitive, with an Enhanced Beacon type and a channel from the channel hopping sequence to send the beacon on. Note that the address from the MLME-BEACON.Request primitive has to match the node address from the advertising Tx link set and short addressing has to be used due to the node address short addressing mode.

The End Device is performing a passive scan on a channel mask that must include the advertising channel used by the PAN Coordinator and also be part of the channel hopping sequence of the network. The higher layer is presented the IEs and the beacon payload and is responsible for configuring the slotframe and link parameters to be used inside the network.

7.1.5. Clock synchronization

It is strongly recommended to use at least one timekeeping Rx link for every new device that joins the network. A clock source for each node is required in order to correct any clock drifts that may appear between devices and properly detect the timeslot start from the PAN Coordinator perspective.

Timeslot start is very important for the precision of the actions performed by both devices exchanging a frame in the timeslot. Time perspective is exchanged through MAC data frames or Enhanced Acknowledgments through the ACK/NACK Information element. The expected time when receiving a MAC frame or an ACK frame is compared with the timestamp of the frame and adjusted locally or communicated through an EACK frame to the other node.

A node can have more than one peer as a time source. When multiple nodes are configured as such, the node adjusts its timeslot start with a fraction of 1 divided by the number of clock sources for each adjustment made after a frame exchange.

Should no frames be exchange by the higher layer of two nodes, one being a clock source for the other, the node with the timekeeping Rx link will send MAC Keepalive frame with an ACK required flag set, requiring clock source to specify its time perspective. The keep alive period after which this exchange is performed is configured by the higher layer through MLME-KEEP-ALIVE.request primitive and it is expressed in timeslots. Note that this primitive should be issued after an Rx Link with Keepalive option set has been set on the node.

7.1.6. TSCH enablement

Prior to enabling TSCH, the TSCH Role must be set on the device. The TSCH PAN is started (ASN = 0) by a TSCH PAN Coordinator, usually the PAN Coordinator. The other nodes join a TSCH network that has already been started, and they must be configured as TSCH devices through the TSCH role. Then, TSCH can be enabled through TSCH-MODE.Request primitive with TSCH mode parameter set to On and disabled through Off.

7.2. TSCH configuration on PAN coordinator

7.2.1. Macro definitions

The Hopping sequence is configured through the following macro definitions:

```
#define mMacHoppingSequenceId_c         0
#define mMacHoppingSequenceLen_c        3
#define mMacHoppingSequenceList_c       { 0, 1, 2 }
```

NOTE

Ensure that the Hopping Sequence is the same as the one used by the End Device.

The PAN Coordinator advertising interval represents the time in milliseconds between sending Beacon Requests and is configured through the following macro definition:

```
#define gBeaconAdvertiseIntervalMs_c     3000
```

The End Device higher layer periodically sends data to the MAC layer. The inter packet delay is configured through the following macro definition expressed in milliseconds:

```
#define gDataTxIntervalMs_c 10000
```

The maximum number of packets sent to the MAC layer for transmission is configured through the following macro definition:

```
#define mDefaultValueOfMaxPendingDataPackets_c 1
```

The other macro definitions are similar to the other applications provided.

7.2.2. Slotframes

The slotframes are configured through the following constant array:

```
const macSlotframeIe_t slotframeArray[] =
{
  {
    .macSlotframeHandle = 0,
    .macSlotframeSize = 10,
  },
};
```

NOTE

Ensure that the slotframe is the same as the one used by the End Device.

7.2.3. Links

An advertising link is required in order to promote the network TSCH parameters. Also, Tx and Rx links need to be set in order to send to and receive data from the End Device. The links are configure through the following constant array:

```
const macLink_t linkArray[] =
{
    {
        .macLinkHandle = 0,
```

```
        .macNodeAddress = 0xFFFF,
        .macLinkType = gMacLinkTypeAdvertising_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 0,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                    },
    },
    {
        .macLinkHandle = 1,
        .macNodeAddress = mDefaultValueOfEDShortAddr_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 1,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                    },
  },
  {
        .macLinkHandle = 2,
        .macNodeAddress = mDefaultValueOfEDShortAddr_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 2,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .rx = 1,
                        },
```

```
                    },
  },
};
```

7.2.4. TSCH configuration on the PAN coordinator

The TSCH enablement on the PAN Coordinator is done as follows:

Example 23. TSCH configuration on PAN coordinator

```
static uint8_t App_Configure_MAC_TSCH_Params(void)
{
    uint32_t i;
    uint16_t hoppingSequenceLength = mMacHoppingSequenceLen_c;
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;
    macTschRole_t tschRole = gMacTschRolePANCoordinator_c;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);

    if( pMsg != NULL )
    {
        /* Set MAC Hopping Sequence Length */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceLength_c;
        pMsg->msgData.setReq.pibAttributeValue = &hoppingSequenceLength;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence List */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceList_c;
        pMsg->msgData.setReq.pibAttributeValue = mHoppingSequenceList;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set slotframes */
        for( i=0; i<sizeof(slotframeArray)/sizeof(macSlotframeIe_t); i++ )
        {
            pMsg->msgType = gMlmeSetSlotframeReq_c;
            pMsg->msgData.setSlotframeReq.operation = gMacSetSlotframeOpAdd_c;
            pMsg->msgData.setSlotframeReq.slotframeHandle =
slotframeArray[i].macSlotframeHandle;
```

```
            pMsg->msgData.setSlotframeReq.size = slotframeArray[i].macSlotframeSize;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

        /* Set links */
        for( i=0; i<sizeof(linkArray)/sizeof(macLink_t); i++ )
        {
            pMsg->msgType = gMlmeSetLinkReq_c;
            pMsg->msgData.setLinkReq.operation = gMacSetLinkOpAdd_c;
            pMsg->msgData.setLinkReq.slotframeHandle = linkArray[i].slotframeHandle;
            pMsg->msgData.setLinkReq.linkHandle = linkArray[i].macLinkHandle;
            pMsg->msgData.setLinkReq.linkType = linkArray[i].macLinkType;
            pMsg->msgData.setLinkReq.nodeAddr = linkArray[i].macNodeAddress;
            pMsg->msgData.setLinkReq.timeslot = linkArray[i].macLinkIe.timeslot;
            pMsg->msgData.setLinkReq.channelOffset = linkArray[i].macLinkIe.channelOffset;
            pMsg->msgData.setLinkReq.linkOptions = linkArray[i].macLinkIe.macLinkOptions;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

      /* Set TSCH Role as PAN Coordinator */
      pMsg->msgType = gMlmeSetReq_c;
      pMsg->msgData.setReq.pibAttribute = gMPibTschRole_c;
      pMsg->msgData.setReq.pibAttributeValue = &tschRole;
      (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

       /* Enable TSCH */
       pMsg->msgType = gMlmeTschModeReq_c;
       pMsg->msgData.tschModeReq.tschMode = gMacTschModeOn_c;
       (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

       /* Free message, all requests were sync */
       MSG_Free(pMsg);
    }
    else
    {
        /* Allocation of a message buffer failed. */
        return errorAllocFailed;
    }
```

```
    return errorNoError;
}
```

7.2.5. TSCH advertising

Periodicall, the Coordinator sends Enhanced Beacons to allow new devices to join the network.

Example 24. TSCH advertising by the PAN coordinator

```
    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_Alloc(sizeof(mlmeMessage_t) + gMaxPHYPacketSize_c);
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;
    if(pMsg != NULL)
    {
        /* Create MLME-BEACON Request message. */
        pMsg->msgType = gMlmeBeaconReq_c;
        pMsg->msgData.beaconReq.beaconType = gMacEnhancedBeacon_c;
        pMsg->msgData.beaconReq.channel = mHoppingSequenceList[0];
        pMsg->msgData.beaconReq.channelPage = gChannelPageId9_c;
        pMsg->msgData.beaconReq.superframeOrder = 0x0F;
        pMsg->msgData.beaconReq.beaconSecurityLevel = gMacSecurityNone_c;
        pMsg->msgData.beaconReq.dstAddrMode = gAddrModeShortAddress_c;
        pMsg->msgData.beaconReq.dstAddr = 0xFFFF;
        pMsg->msgData.beaconReq.bsnSuppression = FALSE;

        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
 }
```

7.3. TSCH configuration on end device

7.3.1. Macro definitions

The Hopping sequence is configured through the following macro definitions:

```
#define mMacHoppingSequenceId_c         0
#define mMacHoppingSequenceLen_c        3
#define mMacHoppingSequenceList_c       { 0, 1, 2 }
```

NOTE

Ensure that the Hopping Sequence is the same as the one used by the PAN Coordinator.

The Keepalive period represents the amount of time expressed in timeslots when no frames are exchanged with the node clock source after which the keep alive MAC frames are sent. It can be configured through the following macro definition:

```
#define mMacKeepAlivePeriod_c           1000    /* timeslots */
```

The End Device higher layer periodically sends data to the MAC layer. The inter packet delay is configured through the following macro definition expressed in milliseconds:

```
#define gDataTxIntervalMs_c 10000
```

The maximum number of packets sent to the MAC layer for transmission is configured through the following macro definition:

```
#define mDefaultValueOfMaxPendingDataPackets_c 1
```

The other macro definitions are similar to the other applications provided.

7.3.2. Slotframes

The slotframes are configured through the following constant array:

```
const macSlotframeIe_t slotframeArray[] =
{
  {
    .macSlotframeHandle = 0,
    .macSlotframeSize = 10,
  },
};
```

NOTE

Ensure that the slotframe is the same as the one used by the PAN Coordinator.

7.3.3. Links

A Timekeeping Rx link is recommended to be configured with the timekeeping option. Also, the demo requires data transfer to the PAN Coordinator so a Tx Link needs to be set also. The links are configured through the following constant array:

```
const macLink_t linkArray[] =
{
    {
        .macLinkHandle = 0,
        .macNodeAddress = mDefaultValueOfCoordAddress_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
```

```
                        .timeslot = 1,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .rx = 1,
                          .timekeeping = 1,
                        },
                      },
  },
  {
        .macLinkHandle = 1,
        .macNodeAddress = mDefaultValueOfCoordAddress_c,
        .macLinkType = gMacLinkTypeNormal_c,
        .slotframeHandle = 0,
        .macLinkIe = {
                        .timeslot = 2,
                        .channelOffset = 0,
                        .macLinkOptions =
                        {
                          .tx = 1,
                        },
                      },
  },
};
```

NOTE

Ensure the the Rx link of the End device matches the timeslot and channel offset of the PAN Coordinator Tx link and viceversa.

7.3.4. TSCH configuration on the end device

The association is not available when TSCH is enabled, so the MAC short address needs to be set on the End Device. The rest of the configuration is done as follows:

Example 25. TSCH configuration on the end device

```
static uint8_t App_Configure_MAC_TSCH_Params(void)
{
    uint32_t i;
    uint16_t hoppingSequenceLength = mMacHoppingSequenceLen_c;
    uint8_t mHoppingSequenceList[] = mMacHoppingSequenceList_c;
```

```
    macTschRole_t tschRole = gMacTschRoleDevice_c;

    /* Message for the MLME will be allocated and attached to this pointer */
    mlmeMessage_t *pMsg = MSG_AllocType(mlmeMessage_t);

    if( pMsg != NULL )
    {
        /* Configure addressing PIBs */
        /* MAC short address */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibShortAddress_c;
        pMsg->msgData.setReq.pibAttributeValue = (uint16_t*)&macShortAddress;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence Length */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceLength_c;
        pMsg->msgData.setReq.pibAttributeValue = &hoppingSequenceLength;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set MAC Hopping Sequence List */
        pMsg->msgType = gMlmeSetReq_c;
        pMsg->msgData.setReq.pibAttribute = gMPibHoppingSequenceList_c;
        pMsg->msgData.setReq.pibAttributeValue = mHoppingSequenceList;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

        /* Set slotframes */
        for( i=0; i<sizeof(slotframeArray)/sizeof(macSlotframeIe_t); i++ )
        {
            pMsg->msgType = gMlmeSetSlotframeReq_c;
            pMsg->msgData.setSlotframeReq.operation = gMacSetSlotframeOpAdd_c;
            pMsg->msgData.setSlotframeReq.slotframeHandle =
slotframeArray[i].macSlotframeHandle;
            pMsg->msgData.setSlotframeReq.size = slotframeArray[i].macSlotframeSize;
            (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
        }

        /* Set links */
```

```
    for( i=0; i<sizeof(linkArray)/sizeof(macLink_t); i++ )
    {
        pMsg->msgType = gMlmeSetLinkReq_c;
        pMsg->msgData.setLinkReq.operation = gMacSetLinkOpAdd_c;
        pMsg->msgData.setLinkReq.slotframeHandle = linkArray[i].slotframeHandle;
        pMsg->msgData.setLinkReq.linkHandle = linkArray[i].macLinkHandle;
        pMsg->msgData.setLinkReq.linkType = linkArray[i].macLinkType;
        pMsg->msgData.setLinkReq.nodeAddr = linkArray[i].macNodeAddress;
        pMsg->msgData.setLinkReq.timeslot = linkArray[i].macLinkIe.timeslot;
        pMsg->msgData.setLinkReq.channelOffset = linkArray[i].macLinkIe.channelOffset;
        pMsg->msgData.setLinkReq.linkOptions = linkArray[i].macLinkIe.macLinkOptions;
        (void)NWK_MLME_SapHandler( pMsg, mMacInstance );
    }

    /* Configure PANC as time source neighbor */
    pMsg->msgType = gMlmeKeepAliveReq_c;
    pMsg->msgData.keepAliveReq.dstAddr = mDefaultValueOfCoordAddress_c;
    pMsg->msgData.keepAliveReq.keepAlivePeriod = mMacKeepAlivePeriod_c;
    (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

  /* Set TSCH Role as normal node */
  pMsg->msgType = gMlmeSetReq_c;
  pMsg->msgData.setReq.pibAttribute = gMPibTschRole_c;
  pMsg->msgData.setReq.pibAttributeValue = &tschRole;
  (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

    /* Enable TSCH */
    pMsg->msgType = gMlmeTschModeReq_c;
    pMsg->msgData.tschModeReq.tschMode = gMacTschModeOn_c;
    (void)NWK_MLME_SapHandler( pMsg, mMacInstance );

    /* Free message, all requests were sync */
    MSG_Free(pMsg);
}
else
{
    /* Allocation of a message buffer failed. */
    return errorAllocFailed;
```

```
    }

    return errorNoError;
}
```

7.4. Running the demo

To run the TSCH demo follow these steps:

1. Setup sniffers on the channels configured in the applications
2. Run MAC_TSCH_DEMO_Coordinator
3. You can see on the sniffers Coordinator periodically advertising Enhanced Beacons and sending MAC data frames
4. Run MAC_LE_TSCH_EndDevice
5. Press any switch on the End Device board to start passive scan, enable TSCH and then periodically generate MAC data frames
6. Monitor sniffers to see the packets exchanged between devices

8. Revision History

Table 25. Revision history

Rev. number	Date	Substantive changes
0	03/2015	Initial release.
1	10/2015	Changes related to IEEE 802.15.4g/e specific new features
2	01/2016	Removed OTAP Bootloader section. Added Section 3.5, “MyWirelessApp FSCI host”.
3	04/2016	Added support for KW4x.
4	09/2016	Public information

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: 802154MPDAUG
Rev. 4
09/2016