Freescale Semiconductor
Application Note

Document Number: Order # AN3573
Rev. 0.0, 11/2007

Porting Freescale 802.15.4 MAC 1.06x Applications to MAC 2.0x (MAC for BeeKit)

1. Introduction

This note describes how to rewrite existing applications based on the Freescale 802.15.4 MAC version 1.06x so that they will work with the Freescale 802.15.4 MAC 2.0x (MAC for BeeKit).

This note describes the following:

- Improvements made to MAC for BeeKit.
- Editing existing 802.15.4 MAC 1.06x applications so they will work with MAC for BeeKit.

This note does not provide detailed information about writing 802.15.4 applications. For these details, see the following documents:

1. *802.15.4 MAC PHY Software Reference Manual*
2. *802.15.4 MAC MyWirelessApp User's Guide*
3. *802.15.4 MAC MyStarNetworkApp User's Guide*

Contents

2 Upgrading the Application Project

The MAC for BeeKit project file structure is different than the MAC 1.06x project file structure. The least complex method to create a file structure compatible with MAC for BeeKit is to use the BeeKit Wireless Connectivity Toolkit and generate an application starting from any existing MAC Codebase template. For more information on how to use BeeKit, see the *BeeKit Wireless Connectivity Toolkit User's Guide*.

2.1 Compiler Defines

The following compiler directives need to be added at the compiler line:

gTargetXXX_d=1	Specifies the type of the target board. At the time, the possible values for XXX are: gTargetMC13213SRB_d gTargetSARD_d gTargetMC13213NCB_d gTargetAxiomGB60_d For an updated list, refer to the `AppToPlatformConfig.h` file.
gZtcIncluded_d=0	Some platform components need this definition in order to compile.
gBeeStackIncluded_d=0	No BeeStack support is needed.
gMacStandAlone_d=1	Specifies that it is an 802.15.4 MAC application.
Type_XXX	Specifies the type of the MAC library that is used. Possible values for XXX are: Type_FFD Type_FFDNB Type_FFDNBNS Type_FFDNGTS Type_RFD Type_RFDNB Type_RFDNBNS For more information on the capabilities of each MAC library, see the *802.15.4 MAC/PHY Software Reference Manual.*

2.2 PHY Layer

For 1.06x MAC releases, the PHY layer is provided as a library which needs to be linked to the application. A CodeWarrior project, containing all the necessary source files of the PHY library, is included in the MACPHY release directory.

For MAC for BeeKit releases, the PHY layer source files are part of the application project. The following file structure needs to be added to the project:

```
Phy
|
|-Interface
| |-Phy.h
| |-PhyMacMsg.h
| |-MacPhyGlobalHdr.h
| |-MacPhyFLib.h
| |-MacPhy.h
| |-FunctionalityDefines.h
| |-MC1319xHandler.h
| |-MacPhyInit.h
|-Isr
| |-Eof.c
| |-MC1319xHandler.c
| |-StreamIsr.c
|-Primitives
| |-CcaEd.c
| |-Data.c
| |-GetSet.c
| |-SetRxTxState.c
|-Phy_Main.c
```

2.3 MAC Layer

The following MAC file structure must be added to the project, together with the appropriate MAC library:

```
Interface
|
|-AppAspInterface.h
|-FunctionLib.h
|-MsgSystem.h
|-NwkMacInterface.h
|-PublicConst.h
|-SecurityLib.h
```

For more information on the capabilities of each MAC library, see the *802.15.4 MAC/PHY Software Reference Manual*.

2.4 Platform and Software Support Modules

The PLM and SSM folders need to be added to the application project.

The PLM folder contains the platform components that help the application interact with the hardware.

The SSM folder contains the source code of the task scheduler.

3 Editing The Application

This section describes the revised Task Scheduler, sending data, active and passive scan, the ASP interface and platform components.

3.1 Task Scheduler

The structure of the MAIN function for applications based on MAC 1.06x has the following common structure:

```
void main(void)
{
      /* Initialization code */
      ...

      /* Application state machine */
      while (state < stateTerminate)
      {
          switch (state)
          {
          case stateInit:
              /* code that runs while in this state */
              ...
              break;
          case stateListen:
              /* code that runs while in this state */
              ...
              break;
          }
          /* Call the MAC main function continuously */
          Mlme_Main();
      }
}
```

The frequency at which the Mlme_Main function is called depends on the execution time of the application machine state. As a result, the Mlme_Main function was not the most optimal.

The MAC for BeeKit based applications run under the control of a priority based, non-preemptive, event-driven task scheduler. For more information on the task scheduler, see the *802.15.4 MAC/PHY Software Reference Manual*.

The application consists of more tasks with different priorities. The more important tasks of the application are as follows:

Mlme_Main	This task executes the MAC state machine. (In MAC 1.06x this was executed by calling the Mlme_Main function as shown in the code snippet at the beginning of Section 3.1.)
AppTask	The application state machine runs within this task.
IdleTask	This task has the lowest priority and runs only if there are no events for the other tasks.

The MAC for BeeKit structure of the main function is as follows:

```
#define gTsFirstMacTaskPriorityLow_c        0x01
#define gTsUartTaskPriority_c               0x03
/* Reserved for application tasks. */
#define gTsFirstApplicationTaskPriority_c   0x40
/* Application main task. */
#define gTsAppTaskPriority_c                0x80
#define gTsMlmeTaskPriority_c               0xC6
#define gTsHighestTaskPriorityPlusOne_c     0xC7

tsTaskID_t gMacTaskID_c;
tsTaskID_t gAppTaskID_c;

void main(void)
{
      /* Init the 802.15.4 MAC state machine. */
      Init_802_15_4();

      /* Init the task scheduler. */
      TS_Init();

      /* Create the necessary tasks */
      gMacTaskID_c = TS_CreateTask(gTsMlmeTaskPriority_c, Mlme_Main);
      gAppTaskID_c = TS_CreateTask(gTsAppTaskPriority_c, AppTask);

      /* Initialize the needed platform components */
      Uart_ModuleInit();    /* Init the Uart */


      /* Start the task scheduler */
      TS_Scheduler();
}
```

First, the Init_802_15_4() needs to be called, in order to initialize the MAC state machine and the internal variables. The TS_Init function initializes the task scheduler module and creates the idle task. Then, the needed tasks are created by specifying the priority and the root function. The MAC task needs to have a higher priority than the application task.

NOTE

The gMacTaskID_c is used internally by the MAC library, so this variable must be declared.

Other platform components need to be initialized. In this example, the UART module is initialized. For more information on the available platform components, see the *Freescale Platform Reference Manual*.

Finally, the TS_Scheduler function starts up the scheduler.

NOTE

This function never returns, so any code after this function call will not be executed.

The application task root function needs to be defined. It is responsible for executing the application state machine. An example of AppTask implementation is shown in the following example code:

```
#define gAppEvtDummyEvent_c            (1 << 0)
#define gAppEvtMessageFromMLME_c       (1 << 1)
#define gAppEvtMessageFromMCPS_c       (1 << 2)
#define gAppEvtMessageFromASP_c       (1 << 3)


void AppTask(event_t events)
{
  void *pMsgIn;
  uint8_t rc;
  pMsgIn = NULL;


  /* The application state machine */
  switch(gState)
  {
  case stateInit:
    /* Goto Active Scan state. */
    gState = stateScanActiveStart;
    TS_SendEvent(gAppTaskID_c, gAppEvtDummyEvent_c);
    break;

  case stateScanActiveStart:
    rc = App_StartScan(gScanModeActive_c);
    if(rc == errorNoError)
      gState = stateScanActiveWaitConfirm;
    break;

  case stateScanActiveWaitConfirm:
    if (events & gAppEvtMessageFromMLME_c)
    {
      if (pMsgIn)
      {
        rc = App_WaitMsg(pMsgIn, gNwkScanCnf_c);
        if(rc == errorNoError)
        {
          rc = App_HandleScanActiveConfirm(pMsgIn);
          if(rc == errorNoError)
          {
            gState = stateAssociate;
            TS_SendEvent(gAppTaskID_c, gAppEvtDummyEvent_c);
          }
        }
      }
    }
    break;

  case stateAssociate:
    /* Associate to the PAN coordinator */
    rc = App_SendAssociateRequest();
    if(rc == errorNoError)
      gState = stateAssociateWaitConfirm;
    break;

  case stateAssociateWaitConfirm:
    /* Stay in this state until the Associate confirm message
       arrives, and then goto the Listen state. */
    if (events & gAppEvtMessageFromMLME_c)
    {
      if (pMsgIn)
      {
        rc = App_WaitMsg(pMsgIn, gNwkAssociateCnf_c);
        if(rc == errorNoError)
        {
          rc = App_HandleAssociateConfirm(pMsgIn);
          if (rc == errorNoError)
          {
             gState = stateListen;
             TS_SendEvent(gAppTaskID_c, gAppEvtDummyEvent_c);
          }
        }
      }
    }
    break;

  case stateListen:

        if (pMsgIn)
        {
          /* Messages must always be freed. */
          MSG_Free(pMsgIn);
        }
    /* Handle MCPS confirms and transmit data from UART */
    if (events & 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);
      }
    }
  /* Check for pending messages in the Queue */
  if(MSG_Pending(&mMcpsNwkInputQueue))
    TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromMCPS_c);
  if(MSG_Pending(&mMlmeNwkInputQueue))
    TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromMLME_c);
  if(MSG_Pending(&mAspNwkInputQueue))
    TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromASP_c);
}
```

Like the other tasks (except the idle task), the application task does not execute if there are no events for it. In this example, the only events that are passed to this task are gAppEvtDummyEvent_c, gAppEvtMessageFromMLME_c and gAppEvtMessageFromMCPS_c.

The gAppEvtMessageFromMLME, gAppEvtMessageFromMCPS_c and gAppEvtMessageFromASP_c events are sent by the MLME, MCPS and ASP SAP handlers, to signal the application task that there are pending messages from the MAC. A possible implementation of these SAP handlers is shown in the following code snippet:

```
uint8_t MLME_NWK_SapHandler(nwkMessage_t * pMsg)
{
  /* Put the incoming MLME message in the applications input queue. */
  MSG_Queue(&mMlmeNwkInputQueue, pMsg);
  TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromMLME_c);
  return gSuccess_c;
}

uint8_t MCPS_NWK_SapHandler(mcpsToNwkMessage_t *pMsg)
{
  /* Put the incoming MCPS message in the applications input queue. */
  MSG_Queue(&mMcpsNwkInputQueue, pMsg);
  TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromMCPS_c);
  return gSuccess_c;
}

uint8_t ASP_APP_SapHandler(aspToAppMsg_t *pMsg)
{
  /* Put the incoming ASP message in the applications input queue. */
  MSG_Queue(&mAspNwkInputQueue, pMsg);
  TS_SendEvent(gAppTaskID_c, gAppEvtMessageFromASP_c);
  return gSuccess_c;
}
```

Notice the code executed in the stateInit. The state is set to stateScanActiveStart and then the application task sends itself a gAppEvtDummyEvent_c event. This event tells the scheduler to execute the application task the next time.

After a task execution, the events bitmap is cleared. If there are pending messages in the MLME or MCPS queues, these are extracted the next time the application task receives a gAppEvtMessageFromMLME, gAppEvtMessageFromMCPS_c or gAppEvtMessageFromASP_c event, respectively. In order to make sure that all of the messages are processed on time, the application task must check the queues and send itself the gAppEvtMessageFromMLME, gAppEvtMessageFromMCPS_c or gAppEvtMessageFromASP_c events if there are pending messages (See the AppTask implementation as already described).

3.2 Sending Data

The MAC 1.06x implements the MCPS-Data.request message using the following structure:

```
typedef struct mcpsDataReq_tag {
  uint8_t  dstAddr[8];  // First 0/2/8 bytes is the address as defined by dstAddrMode
  uint8_t  dstPanId[2]; // 16 bit word converted to little endian byte array
  uint8_t  dstAddrMode;
  uint8_t  srcAddr[8];  // First 0/2/8 bytes is the address as defined by srcAddrMode
  uint8_t  srcPanId[2]; // 16 bit word converted to little endian byte array
  uint8_t  srcAddrMode;
  uint8_t  msduLength;  // 0-102
  uint8_t  msduHandle;
  uint8_t  txOptions;
  uint8_t  msdu[1];     // Place holder. Data will start at the address of this byte
} mcpsDataReq_t;
```

The msdu field points to the first byte of the payload. The data request packet is created using the following code:

```
uint8_t *Message; /* global variable */

/* Allocate the message */
Message = MSG_Alloc(4);
FLib_MemCpy(Message, “Abc”, 4);

/* Create the data request message */
pPacket = MSG_Alloc(sizeof(nwkToMcpsMessage_t) - 1 + DEFAULT_DATA_LENGTH);
/* Fill the payload */
FLib_MemCpy(pPacket->msgData.dataReq.msdu, Message, 4);
/* Fill the other fields */
...
```

The MAC for BeeKit implementation of data request is similar, except that the msdu field is replaced with pMsdu, as shown in the following code example.

```
typedef struct mcpsDataReq_tag {
  ...
  uint8_t  *pMsdu;     //  Data will start at this address
} mcpsDataReq_t;
```

The MAC for BeeKit implementation allows the application to directly assign the pMsdu to an already allocated buffer, without the need to copy the entire contents of the buffer to the message. The MAC for BeeKit code looks like the following:

```
uint8_t *Message; /* global variable */

/* Allocate the message */
Message = MSG_Alloc(4);
Flib_MemCpy(Message, “Abc”, 4);

/* Create the data request message */
pPacket = MSG_Alloc(sizeof(nwkToMcpsMessage_t) - 1 + DEFAULT_DATA_LENGTH);
/* Fill the payload */
pPacket->msgData.dataReq.pMsdu = Message;
/* Fill the other fields */
```

...

In this code example, the deallocation of Message is the responsibility of the application because the MAC only deallocates the pPacket.

NOTE

It is still possible to use the 1.06x approach, by copying the Message to the pMsdu.

3.3 Active/Passive Scan

For scan, the difference between MAC for BeeKit and the 1.06x MAC versions lie in the implementation of the nwkScanCnf_t structure. The 1.06x versions contain a list of PAN descriptors as follows:

```
typedef struct nwkScanCnf_tag {
  uint8_t  status;
  uint8_t  scanType;
  uint8_t  resultListSize;
  uint8_t  unscannedChannels[4];
  union {
    uint8_t *pEnergyDetectList;          // pointer to 16 byte static buffer
    panDescriptor_t *pPanDescriptorList; // Array of pan descriptors [5] - this one must be
freed by MM_Free();
  } resList;
} nwkScanCnf_t;
```

The scan confirmation handler is implemented as follows:

```
uint8_t panDescListSize   = pMsg->msgData.scanCnf.resultListSize;
panDescriptor_t *pPanDesc = pMsg->msgData.scanCnf.resList.pPanDescriptorList;
uint8_t rc = errorNoScanResults;

  /* Check if the scan resulted in any coordinator responses. */
if(panDescListSize != 0)
{
  /* Initialize link quality to very poor. */
  uint8_t i, bestLinkQuality = 0;

  /* Check all PAN descriptors. */
  for(i=0; i<panDescListSize; i++, pPanDesc++)
  {
    /* Process each PAN descriptor */
    ...
  }
}
/* ALWAYS free the PAN descriptor list */
MSG_Free(pMsg->msgData.scanCnf.resList.pPanDescriptorList);
```

The maximum number of PAN descriptors that pPanDescriptorList can contain is 5. To allow more descriptors, the pPanDescriptorList was replaced with a linked list of PAN descriptors blocks as follows:

```
typedef struct nwkScanCnf_tag {
  uint8_t  status;
  uint8_t  scanType;
  uint8_t  resultListSize;
  uint8_t  unscannedChannels[4];
  union {
    uint8_t *pEnergyDetectList;                   // pointer to 16 byte static buffer
    panDescriptorBlock_t *pPanDescriptorBlocks; // this one must be freed by the upper layer
  } resList;
} nwkScanCnf_t;
```

Each block contains a list of descriptors:

```
struct panDescriptorBlock_tag {
  panDescriptor_t descriptorList[aScanResultsPerBlock];
  uint8_t descriptorCount;
  struct panDescriptorBlock_tag *pNext;
};
```

The MAC for BeeKit scan confirmation handler is as follows:

```
static uint8_t App_HandleScanActiveConfirm(nwkMessage_t *pMsg)
{
  void    *pBlock;
  uint8_t panDescListSize = pMsg->msgData.scanCnf.resultListSize;
  uint8_t rc = errorNoScanResults;
  uint8_t j;
  panDescriptorBlock_t *pDescBlock = pMsg->msgData.scanCnf.resList.pPanDescriptorBlocks;
  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->descriptorCount; j++)
      {
        pPanDesc = &pDescBlock->descriptorList[j];;

        /* Process the PAN descriptor */
      }

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

3.4 ASP Interface

To reduce the size of the application image, the ASP calls have been replaced with direct function calls instead of message passing.

As an example, consider the following MAC 1.06x version code example that sends an event request to the ASP layer:

```
aspMsg_t aspEvent;
aspEventReq_t *pEventReq;
pEventReq = &aspEvent.appToAspMsg.msgData.aspEventReq;
aspEvent.msgType = gAppAspEventReq_c;
pEventReq->eventTime[0] = 0x00;
pEventReq->eventTime[1] = 0xff;
pEventReq->eventTime[2] = 0x00;
MSG_Send(APP_ASP, &aspEvent);
```

For the MAC for BeeKit, the above code is replaced with the following code:

```
uint8_t interval[3] = {0x00, 0xff, 0x00};
uint8_t res;
Asp_EventReq(interval);
```

For more information on the ASP interface, see the *802.15.4 MAC/PHY Software Reference Manual*.

3.5 Platform Components

For higher portability between different platforms, software drivers were written for each hardware component.

One of the most used platform components is the UART, which interacts with the serial port. For porting an existing application that calls UART functions, follow these steps:

1. Include the files `Uart_Interface.h`, `Uart.c`, `Uart.h`, `UartUtil.c`, `UartUtil.h` to the application project.
2. Set the appropriate UART interrupt handlers in the `isrvectors.c` file. Use the gUart1_TxIsr_c, gUart1_RxIsr_c, gUart1_ErrorIsr_c, gUart2_TxIsr_c, gUart2_RxIsr_c, gUart2_ErrorIsr_c defined in `UART_Interface.h`.
3. Add a Uart_ModuleInit() call inside the main() function (before TS_Scheduler()).
4. Replace all occurrences of Uart_Print with UartUtil_Print.
5. Replace all occurrences of Uart_PrintHex with UartUtil_PrintHex.
6. Replace all occurrences of Uart_Tx with UartUtil_Tx.
7. Replace all occurrences of Uart_Poll with UartX_GetByteFromRxBuffer.

For more information on the new UART functions and other available platform components, see the *Freescale Platform Reference Manual*.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSeminconductor@hibbertgroup.com

AN3573
11/2007