Freescale Semiconductor
Application Note

Document Number: AN4729
Rev. 0, , 5/2013

Time Stamp Synchronization Using RF4CE

Dennis Lui
Applications Engineering, Freescale Semiconductor

Contents

1 Introduction

This application note describes a method of timing synchronization using RF4CE technology as the transmission link. This feature can be applied in a group of remote controlled devices that require a common time base to ensure local task executions are in phase to others. For example, the sync signal transmission from TV to paired 3D-Glasses for lens switching, sequence control signal for couple of high power stages that are electrically isolated, or status information monitoring on different measurement devices in aligned time slots. Freescale's RF4CE starts with the 802.15.4 Standard, but incorporates improvements in interference avoidance by adding channel agility and low latency transmissions to address the specific needs of remote control applications.

2 System Overview

A typical remote control system is shown in Figure 1. The core time base signal is an intrinsic timing reference generated in the master with typical frequency equal to 50 Hz or 60 Hz. It is captured from the host processor and converted into a series of messages sending to other slave devices using RF4CE transmission with the broadcast option. The RF packets received in each slave device are decoded and re-formed into a sync signal that is identical to the core time base signal in frequency and phase. Freescale documents called BeeStack Consumer Application User's Guide (document number BSCONAUG) and ZigBee Remote Control (ZRC) Application Profile User's Guide (document number ZRCAPUG) provide detailed descriptions of the RF4CE application. This application note focuses on how to implement the synchronization with an existing RF4CE profile.

NOTE

In the RF4CE, controller and target are the names used for master and slave in this application note.

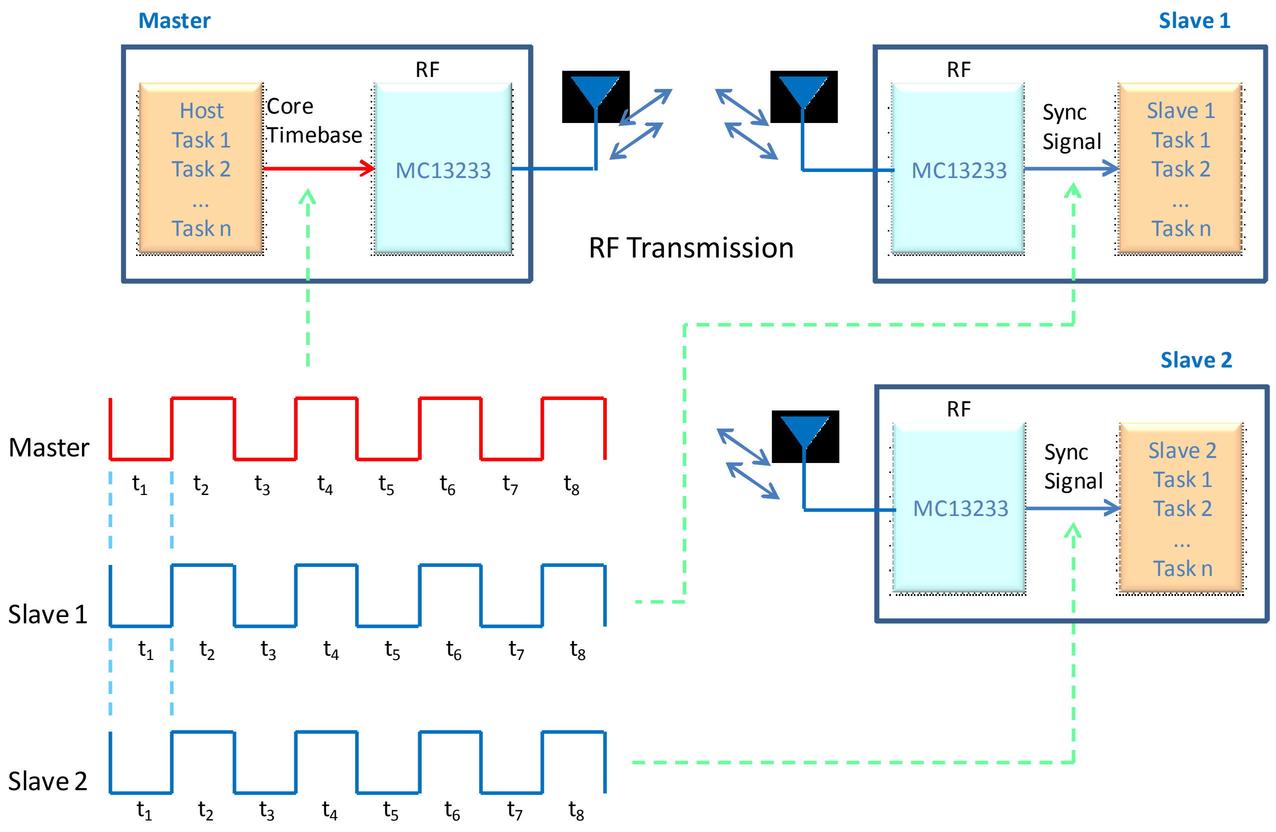

Figure 1. System Block Diagram

The simple method to achieve timing synchronization is continuous RF transmission from master to slave devices. The master sends out an RF packet every time the core timebase signal is toggled and the slave devices are operated in reverse order to toggle the sync signal when an RF packet is received. However, there are lots of disadvantages to using this kind of straightforward method without any control on data rate, power, error, and timing:

- Data traffic — Lot of unnecessary data packets are repeatedly transmitted over the RF link, which causes heavy traffic load over the air. The transmission effectiveness is quite low due to the overhead on data transmission. Part of the RF channel resources are always occupied and would not be released for other applications.
- Power consumption — The RF transmitted power in master side is almost proportional to the data transmission duty cycle, so short data packet length and low repeat rate are essential factors for energy saving. On the slave side, the RF receiver module is required to turn on all the time for data packet receiving from the master side and the average power dissipation is increased accordingly. The battery life drop is the key issue to address in power-sensitive portable applications.
- Packet loss — The slave device is just waiting for incoming data packets but does not have any information about the time for the next arriving packet, so it is difficult to maintain a stable sync signal reformation. One or more sync signal toggle edges would be missed if there was any data packet loss during the transmission period. The end result is a sharp change in frequency or time slot alignment, which disturbs the execution task sequence intermittently. The ability to handle a packet loss condition is highly desirable.
- Phase control — The threshold level setting for phase lock condition and the phase shift adjustment feature are not allowed in the system. There is no option to shut-down the system or disable a power function for system protection before the slave device starts running out of lock. Adding time delay configuration for different device types is a good feature to achieve a smooth transition in power switch application.

A time stamp synchronization method is introduced in following sections to resolve the above issues.

3 Synchronization Principle

The time stamp synchronization is organized in a master-slave hierarchy, a 50 Hz or 60 Hz square wave signal is generated in the master as the core time base signal to align all execution tasks of the whole system in a corresponding time slot. Figure 2 depicts the detail of timing relationship between the master and slave devices. Assume the core time base frequency is 50 Hz with equal duty of 10 ms, t_1 to t_{100} are aligned time slots with a repeat cycle equal to 1 second (100 x 10 ms). A sync message with pre-defined data structure is sent out periodically through RF transmission from master to slave devices with a long repeat cycle equal to 100 time slots and aligned at falling edge of t_1 (one sync message per second). The transmission duty and corresponding power consumption are much reduced by a factor of 100.

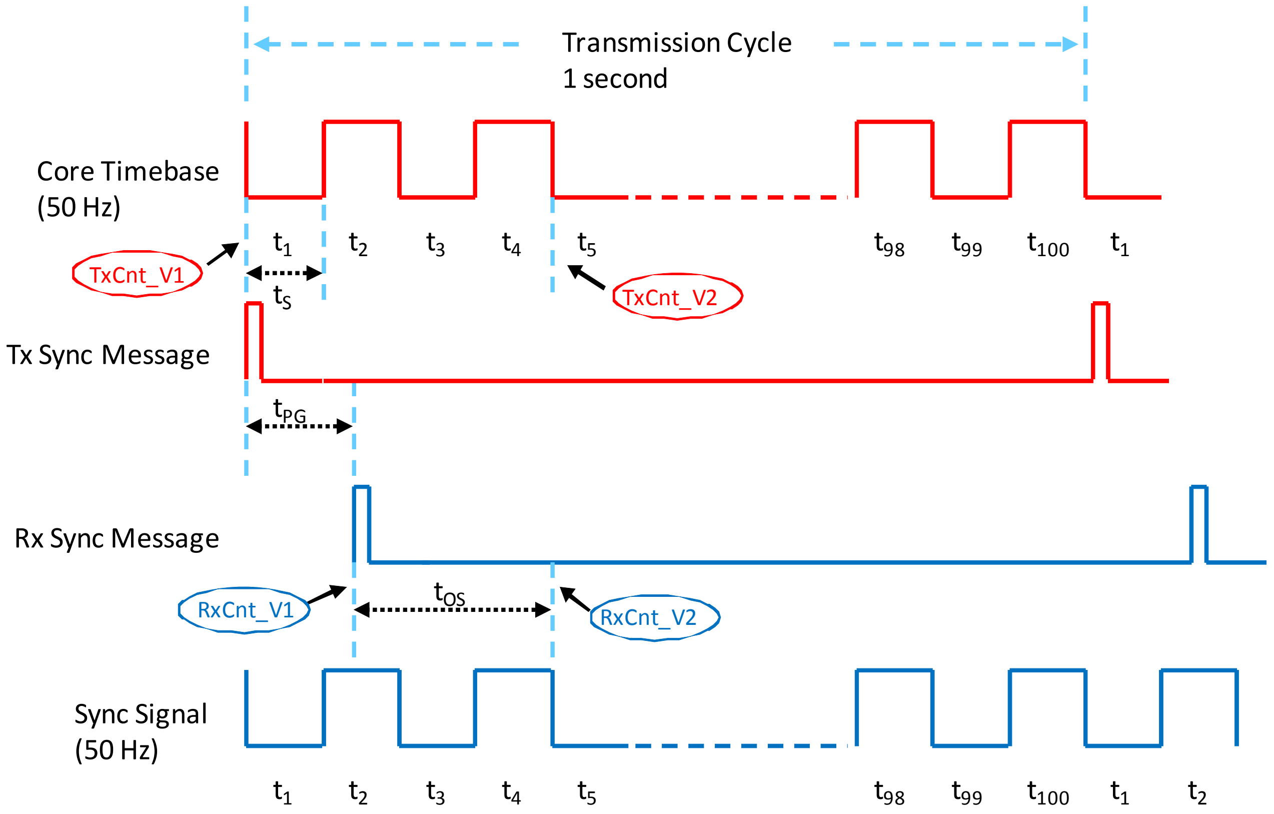

Figure 2. Synchronization Timing

The RF packet received within time slot t_2 in each slave device is decoded to reform a sync message that aligns all local time slots t_1 to t_{100} in phase to the master. The propagation delay (t_{PG}) from transmitter to receiver using RF4CE transmission is almost a constant for the same RF channel. However, the delay constant is different when other channels are selected by the frequency agility feature in high interference conditions. The channel number embedded in RF4CE protocol must be extracted in slave devices to determine the correct delay time, which allows the slave device to calculate the acceptable deviation level for phase lock condition.

4 Time Stamping

The local clock counter value in master device is captured at falling edge of time slot t_1 as shown in Figure 2. This is a reference point to calculate the counter value for coming time slot t_5 by adding four time slot intervals to counter value TxCnt_V1. The actual time stamp value used for sync message transmission is the projected counter value TxCnt_V2 in t_5 instead of the instant value TxCnt_V1 captured in t_1. The key advantages of using projected value are:

1. Provides simple calculation for fixed time slot interval.
2. Is independent of the transmission channel used.
3. Reserves enough timing for data processing on the receiver side.
4. Eliminates the effect of software delay by other routines in receiver side.

5. Avoids rollover of the counter module on the receiver side.

Similarly, the counter value RxCnt_V1 captured on the receiver side when receiving the sync message is stored immediately as a reference point. The counter value RxCnt_V2 in coming time slot t_5 can be estimated by adding an offset value (t_{OS}) into RxCnt_V1. The offset values are pre-set data stored in receiver side and equal to four time slot intervals minus the propagation delay (t_{PG}) for each transmission channels.

5 Phase Locking

In the ideal case, the calculated value of RxCnt_V2 is equal to the value of TxCnt_V2 retrieved from the sync message. This perfect match condition implies the sync signal in receiver side is locked in phase with the core time base signal on the transmitter side at the falling edge of t_4. An optional constant time delay at between TxCnt_V2 and RxCnt_V2 is achievable by using a non-zero value for the perfect match condition. The delta (t_D) at between the values of TxCnt_V2 and RxCnt_V2 must be checked periodically and keep as minimum by following steps:

1. Increase the counter value by delta Td if RxCnt_V2 is lagged behind TxCnt_V2.
2. Decrease the counter value by delta Td if RxCnt_V2 is leaded ahead TxCnt_V2.
3. No change on counter value if RxCnt_V2 is equal to TxCnt_V2.

The deviation of the delta t_D is an essential parameter to indicate the jitter performance of the locking process. The threshold level for the out-of-lock condition is adjusted by setting different limits for delta t_D in the lock range. The unlocked condition can be detected in advance and the software is able to make decision on system shut down if necessary. The receiver is able to generate the sync signal continuously even though one of the packet data is lost or corrupted in the transmission. The phase errors will be accumulated until a valid packet data is received for timing correction in next transmission. All execution tasks assigned for each time slot are still running smoothly without any abrupt change in time.

6 Conclusion

The concept of time stamping using RF4CE as a transmission link for a time aligned remote control system is illustrated in this application note. The system is able to address the issues of redundant data transmission, inefficient channel use, high power consumption, and is sensitive of packet data loss inherent in simple synchronization schemes.

7 References

The following reference documents are available on http://www.freescale.com

- AN4353: Flexible Active Shutter Control Interface Using the MC1323x.
- Document number BSCONAUG: BeeStack Consumer Application Users Guide.
- Document number ZRCAPUG: Zigbee Remote Control (ZRC) Application Profile Users Guide.

Appendix A Code Examples

The sample codes for time stamp capture and data transmission request on the transmitter side are shown at below:

```
/****************************************************************************/
/* Timer ISR. */
INTERRUPT_KEYWORD void CDTMR_VesaIn_Isr(void) {
  __asm {                      /* The HCS08 does not save H on interrupt. */
    PSHH
  }
  /* Clearing the interrupt flag requires reading it and then writing it. */
  if ( gCDTPMxVesaInCnSC_c & gCDTPMxCnSC_F_c ) {

    // Capture the counter value at each falling edge of t1
    CDTPMxReadVesaInCnVRegister(mVesaInCurrentTimeTf);

    if ( mVesaInStatus_d == TRUE ) {

      ++mVesaTransmitCount;

    }

  /* Reset the Counter */
    if(mVesaTransmitCount == mVesaTransmitPeriod){

    gCDTPMxVesaInCNTL_c = CDTPMxCNTLvalue(0x0000);
    gCDTPMxVesaInCNTH_c = CDTPMxCNTHvalue(0x0000);

    // Update the time stamp value per second
    mVesaInTimeStamp = mVesaInCurrentTimeTf;

    mVesaInCurrentTimeTf = 0;
    mVesaInPreviousTimeTf = 0;

    }

  /* or measure the time slot interval */
    else {

    mVesaInMeasureCurrentPeriod = (uint16_t)((int16_t)mVesaInCurrentTimeTf - 
(int16_t)mVesaInPreviousTimeTf);

    mVesaInMeasureAveragePeriod = ((mVesaInMeasureAveragePeriod >> 1) + 
(mVesaInMeasureCurrentPeriod >> 1));

    mVesaInPreviousTimeTf = mVesaInCurrentTimeTf;

    }

  /* Request to send the Sync Message */

    if ((mVesaTransmitCount == mVesaTransmitPeriod) && (mVesaInStatus_d == TRUE)) {

      if ((mVesaPairSuccess_d == TRUE) && (mVesaLinkEnable_d == TRUE)){

      // Enable the Sync message transmission
      mVesaSyncSend_d = TRUE;

      mVesaTransmitCount = 0;

      TS_SendEvent(gAppTaskID, gAppEvtStateStart_c);

      }

      else{
      mVesaTransmitCount = 0;
      }

    }


    ++mVesaInMeasureCount;
    mVesaInErrorCount = 0;

    CDTPMxVesaInStartTf();

    TS_SendEvent(gCDAppTaskID, gCDApp_Event_c);


  }

  __asm {
    PULH
  }

}
/*****************************************************************************/


/*****************************************************************************
*  App_DataState
*
*  Sends a data request specific with the current selected profileId.
*
*****************************************************************************/
static void App_DataState(event_t events, void* pMsgIn)
{
  uint8_t status;
  uint8_t startChannel = FaEnableStartChannel;
  static uint8_t baseChannel;
  static uint8_t originalBackoff;
  uint8_t beaconBackoff = 0;

  if((events & gAppEvtStateStart_c) && (appStateMachine.subState == gAppSubStateStart_c))
  {
    uint8_t appCmdPayload[]   = gAppCmdPayload_c;
    uint8_t CDappCmdPayload[] = gCDAppCmdPayload_c;
    uint8_t vendorId[] = gDefaultValueOfVendorId_c;

    /* Send Sync message using the nwk NLDE Data service */
    if(mVesaSyncSend_d == TRUE){

      CDTPMxReadVesaInCNTRegister(mVesaSyncTsDelay);
      CDappCmdPayload[0] = mVesaSendHeader;
      CDappCmdPayload[1] = mVesaInFreqState;
      CDappCmdPayload[2] = LoadHighValue(mVesaInDuty);
      CDappCmdPayload[3] = LoadLowValue(mVesaInDuty);
      CDappCmdPayload[4] = LoadHighValue(mVesaInTimeStamp);
      CDappCmdPayload[5] = LoadLowValue(mVesaInTimeStamp);
      CDappCmdPayload[6] = LoadHighValue(mVesaSyncTsDelay);
      CDappCmdPayload[7] = LoadLowValue(mVesaSyncTsDelay);
      CDappCmdPayload[8] = mShutterOnTimePercent;
      CDappCmdPayload[9] = mTvProfileID;

      /* Set BaseChannel = StartChannel */
      NLME_GetRequest(gNwkNib_BaseChannel_c, 0 ,(uint8_t*)&baseChannel);
      NLME_SetRequest(gNwkNib_BaseChannel_c, 0, (uint8_t*)&startChannel);

      /* Set gNwkNib_MaxFirstAttemptCSMABackoffs_c = 0*/
      NLME_GetRequest(gNwkNib_MaxFirstAttemptCSMABackoffs_c, 0 ,(uint8_t*)&originalBackoff);
      NLME_SetRequest(gNwkNib_MaxFirstAttemptCSMABackoffs_c, 0, (uint8_t*)&beaconBackoff);

      /* For MC13233, Change power and CCA threshold */
      oldCCAThreshold = PhyPpGetCcaThreshold();
      oldPowerLevel   = Asp_GetPowerLevel();
      PhyPpSetCcaThreshold(0x00);
      Asp_SetPowerLevel(0xFF);

      /* Set backoffTime to 0 */
      Set_MacMinBE(0);

      status = NLDE_DataRequest(
                              appStateMachine.deviceId,
                              gSupportedProfile3_c,
                              vendorId,
                              gCDAppCmdPayloadLength_c,
                              CDappCmdPayload,
                              gCDTxOptions_c
                              );

    }

    else {

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

    /* Try to send data using the nwk NLDE Data service */
      status = NLDE_DataRequest(
                              appStateMachine.deviceId,
                              gSupportedProfile3_c,
                              vendorId,
                              gAppCmdPayloadLength_c,
                              appCmdPayload,
                              gTxOptions_c
                              );


    }
```

```
    /* Exit the state if this is not successful, otherwise wait confirm */
    if(gNWSuccess_c == status)
    {
      appStateMachine.subState = gAppSubStateWaitCnf_c;
    }
    else
    {
      appStateMachine.subState = gAppSubStateEnd_c;
    }
  }

  switch(appStateMachine.subState)
  {
    case gAppSubStateWaitCnf_c:
      if((events & gAppEvtMsgFromNlde_c) &&
         (pMsgIn != NULL))
      {
        nwkNldeToAppMsg_t* pNldeMsgIn = (nwkNldeToAppMsg_t*)pMsgIn;

    /* add for Sync message transmission */
        if(mVesaSyncSend_d == TRUE){

        /* Set BaseChannel to Original value */
        NLME_SetRequest(gNwkNib_BaseChannel_c, 0, (uint8_t*)&baseChannel);

        /* Set originalBackoff to gNwkNib_MaxFirstAttemptCSMABackoffs_c */
       NLME_SetRequest(gNwkNib_MaxFirstAttemptCSMABackoffs_c, 0, (uint8_t*)&originalBackoff);

        /* For MC13233, restore power level and CCA threshold */
          PhyPpSetCcaThreshold(oldCCAThreshold);
          Asp_SetPowerLevel(oldPowerLevel);

        /* Set backoffTime to 3(default) */
        Set_MacMinBE(3);
        }

        /* Data confirm received */
        if(pNldeMsgIn->msgType == gNwkNldeDataCnf_c)
        {
          status = pNldeMsgIn->msgData.nwkNldeDataCnf.status;
          appStateMachine.subState = gAppSubStateEnd_c;
        }
      }
      break;
    default:
      break;
  }

  if(appStateMachine.subState == gAppSubStateEnd_c)
  {
    /* Print the status */
    if(mVesaSyncSend_d == FALSE)   //no print for Vesa Sync Transmission

    App_PrintResult(status);

    /* Reset Vesa Sync Send Status */
    mVesaSyncSend_d      = FALSE;
    /* Send event to end the state */
    TS_SendEvent(gAppTaskID, gAppEvtStateEnd_c);
  }

}
/*****************************************************************************/
```

The sample codes for data receiving and time stamp calculation on the receiver side are shown below:

```
/*****************************************************************************
*  App_HandleDataInd
*
*  Handles data received from paired devices
*
*****************************************************************************/
static void App_HandleDataInd(nwkNldeDataInd_t* pNwkNldeDataInd)
{
  // Sync message is received if Data[0] is 'c'
  if ((mVesaLinkEnable_d)&&(pNwkNldeDataInd->pData[0]==gVesaSendHeader_c)) {

  uint8_t status = 0;
  volatile uint8_t baseChannel;

  // Store the counter value when Sync message is received
  CDTPMxReadVesaOutCNTRegister(mVesaSyncRxCNTreadCur);

  status |= NLME_GetRequest(gNwkNib_BaseChannel_c, 0 ,(uint8_t*)&baseChannel);

  mVesaSyncRxChannelCur = baseChannel;

  mVesaSyncRxData[0] = pNwkNldeDataInd->pData[0];
  mVesaSyncRxData[1] = pNwkNldeDataInd->pData[1];
  mVesaSyncRxData[2] = pNwkNldeDataInd->pData[2];
  mVesaSyncRxData[3] = pNwkNldeDataInd->pData[3];
  mVesaSyncRxData[4] = pNwkNldeDataInd->pData[4];
  mVesaSyncRxData[5] = pNwkNldeDataInd->pData[5];
  mVesaSyncRxData[6] = pNwkNldeDataInd->pData[6];
  mVesaSyncRxData[7] = pNwkNldeDataInd->pData[7];
  mVesaSyncRxData[8] = pNwkNldeDataInd->pData[8];
  mVesaSyncRxData[9] = pNwkNldeDataInd->pData[9];

    ++mVesaSyncRxCount;
    mVesaSyncRxStatus_d = TRUE;
    TS_SendEvent(gCDAppTaskID, gCDApp_Event_c);

  }
  // else run normal code
  else {

  UartUtil_Print("\n\rData received from ", gAllowToBlock_d);
  UartUtil_Print(nodeData.pairTableEntry[pNwkNldeDataInd->deviceId].recipUserString,
gAllowToBlock_d);

  /* Print information about the received data indication */
  UartUtil_Print("\n\rReceived data parameters:", gAllowToBlock_d);
  /* Specify the payload of data indication, if present */
  UartUtil_Print("\n\r    1.Payload: ", gAllowToBlock_d);
  if(pNwkNldeDataInd->dataLength > 0)
    UartUtil_Tx(&pNwkNldeDataInd->pData[0], pNwkNldeDataInd->dataLength);
  else
    UartUtil_Print("None", gAllowToBlock_d);
  /* Specify the LQI of the received data */
  UartUtil_Print("\n\r    2.LQI: 0x", gAllowToBlock_d);
  UartUtil_PrintHex(&pNwkNldeDataInd->LQI, 1, gPrtHexSpaces_c);
  /* Specify the Rx options - Data was broadcasted or not */
  UartUtil_Print("\n\r    3.Broadcasted: ", gAllowToBlock_d);
  if(pNwkNldeDataInd->rxFlags & maskRxOptions_Broadcast_c)
    UartUtil_Print("Yes", gAllowToBlock_d);
  else
    UartUtil_Print("No", gAllowToBlock_d);
  /* Specify the Rx options - Data was secured or not */
  UartUtil_Print("\n\r    4.Secured: ", gAllowToBlock_d);
  if(pNwkNldeDataInd->rxFlags & maskRxOptions_UseSecurity_c)
    UartUtil_Print("Yes", gAllowToBlock_d);
  else
    UartUtil_Print("No", gAllowToBlock_d);
  /* Specify profile identifier */
  UartUtil_Print("\n\r    5.Profile ID: ", gAllowToBlock_d);
  UartUtil_PrintHex(&pNwkNldeDataInd->profileId, 1, 0);

  /* Specify the Rx options - Data is vendor specific or not */
  UartUtil_Print("\n\r    6.Vendor specific: ", gAllowToBlock_d);
  if(pNwkNldeDataInd->rxFlags & maskRxOptions_VendorSpecificData_c)
  {
    UartUtil_Print("Yes. Vendor ID: ", gAllowToBlock_d);
    UartUtil_PrintHex(&pNwkNldeDataInd->vendorId[0], 2, 0);
  }
  else
    UartUtil_Print("No", gAllowToBlock_d);

  UartUtil_Print("\n\r", gAllowToBlock_d);

  }

}
/*****************************************************************************/


/*****************************************************************************
*  3DApp_MainTask
*
*  3DApplication task. Responds to events for the application.
*****************************************************************************/
```

```
/* 3D Glasses event */
void CDApp_MainTask(event_t events)
{

    (void)events; /* remove compiler warning */

    if (mVesaLinkPowerSavingNwkReset_d == TRUE){
      mNwkDebugCode_d = NLME_ResetRequest(FALSE);
      mNwkDebugCode_d = NLME_StartRequest();
      mVesaLinkPowerSavingNwkReset_d = FALSE;
    }

    // Re-sync Vesa Out Phase base on data received from RF4CE Link
    if ((mVesaSyncRxStatus_d == TRUE) && (mVesaLinkStatus_d == TRUE)) {

      bDisableNwkDataSetUpdateInFlash = TRUE;

#ifdef VesaLinkPowerSavingFunction
      mNwkDebugCode_d = NLME_RxEnableRequest(0x00);
#endif
      bDisableNwkDataSetUpdateInFlash = FALSE;

      // Retrieve Sync message from RF4CE Link
      ReadVesaSendDataDuty(mVesaOutDuty);
      ReadVesaSendDataStamp(mVesaOutTimeStamp);
      ReadVesaSendDataTsDelay(mVesaSyncTsDelay);

      mVesaOutFreqState = mVesaSyncRxData[gVesaIndexFreq_c];

      if(mVesaSyncRxChannelCur != mVesaSyncRxChannelPre) {
        if(mVesaOutFreqState == gVesaOutFreq100Hz_c){
        mVesaOutDuty = gVesaOutDuty100Hz_c;
        }else if(mVesaOutFreqState == gVesaOutFreq120Hz_c){
        mVesaOutDuty = gVesaOutDuty120Hz_c;
        }
        mVesaSyncRxChannelPre = mVesaSyncRxChannelCur;
      }

      if(mVesaOutFreqState == gVesaOutFreq100Hz_c){

        mVesaSyncRxPeriod = gVesaSyncRxPeriod100Hz_c;

        if(mVesaSyncRxChannelCur == 15){
          mVesaSyncRxChOffset = gVesaSyncRx100HzOffsetCh15_c;
        }else if(mVesaSyncRxChannelCur == 20){
          mVesaSyncRxChOffset = gVesaSyncRx100HzOffsetCh20_c;
        }else if(mVesaSyncRxChannelCur == 25){
          mVesaSyncRxChOffset = gVesaSyncRx100HzOffsetCh25_c;
        }
      }

      if(mVesaOutFreqState == gVesaOutFreq120Hz_c){

        mVesaSyncRxPeriod = gVesaSyncRxPeriod120Hz_c;

        if(mVesaSyncRxChannelCur == 15){
          mVesaSyncRxChOffset = gVesaSyncRx120HzOffsetCh15_c;
        }else if(mVesaSyncRxChannelCur == 20){
          mVesaSyncRxChOffset = gVesaSyncRx120HzOffsetCh20_c;
        }else if(mVesaSyncRxChannelCur == 25){
          mVesaSyncRxChOffset = gVesaSyncRx120HzOffsetCh25_c;
        }
      }

      // Calculate the projected counter value for phase lock checking
      mVesaOutPredictInterruptTime = (mVesaSyncRxCNTreadCur) + (3*mVesaOutDuty);

      mVesaOutPredictInterruptTime = (uint16_t)((int16_t)(mVesaOutPredictInterruptTime) -
(int16_t)(mVesaSyncRxChOffset));

      mVesaOutPredictInterruptTime = (uint16_t)((int16_t)(mVesaOutPredictInterruptTime) -
(int16_t)(mVesaSyncTsDelay));

      mVesaOutPredictTimeStamp = mVesaOutPredictInterruptTime + mVesaOutDuty;

      mVesaOutEdgeJitter = (int16_t)((int16_t)(mVesaOutTimeStamp) -
(int16_t)(mVesaOutPredictTimeStamp));

      // Correct the jitter errors
      if(mVesaOutEdgeJitter >= 0)                      // timer clock = 250kHz
      {
          mVesaOutEdgeJitterAdjPolarity_d = TRUE;

          if(mVesaOutEdgeJitter >= 20){
          mVesaOutEdgeJitterAdjCounter = 20;
          }else if(mVesaOutEdgeJitter >= 10){
          mVesaOutEdgeJitterAdjCounter = 10;
          }else if(mVesaOutEdgeJitter >= 5){
          mVesaOutEdgeJitterAdjCounter = 5;
          }else if(mVesaOutEdgeJitter >= 3){
          mVesaOutEdgeJitterAdjCounter = 3;
          }else{
          mVesaOutEdgeJitterAdjCounter = (mVesaOutEdgeJitter + 0);
          }


      }else {
          mVesaOutEdgeJitterAdjPolarity_d = FALSE;

          if(-mVesaOutEdgeJitter >= 20){
          mVesaOutEdgeJitterAdjCounter = -20;
          }else if(-mVesaOutEdgeJitter >= 10){
          mVesaOutEdgeJitterAdjCounter = -10;
          }else if(-mVesaOutEdgeJitter >= 5){
          mVesaOutEdgeJitterAdjCounter = -5;
          }else if(-mVesaOutEdgeJitter >= 3){
          mVesaOutEdgeJitterAdjCounter = -3;
          }else{
          mVesaOutEdgeJitterAdjCounter = (mVesaOutEdgeJitter + 0);
          }

      }
      mVesaOutPredictInterruptTime = (uint16_t)((int16_t)(mVesaOutPredictInterruptTime) +
(int16_t)(mVesaOutEdgeJitterAdjCounter));

      mVesaLinkErrorCount = 0;
      mVesaSyncRxStatus_d = FALSE;
      mVesaOutAlignEdge_d = TRUE;

    }


    if(((mVesaOutEdgeJitter < 0)&&(-mVesaOutEdgeJitter < 200))||((mVesaOutEdgeJitter >=
0)&&(mVesaOutEdgeJitter < 200))){

#ifdef VesaLinkPowerSavingFunction
     mVesaLinkPowerSavingEnable_d = TRUE;
#else
     mVesaLinkPowerSavingEnable_d = FALSE;
#endif

    }

    else{

     mVesaLinkPowerSavingEnable_d = FALSE;

      // set the bus clock to 16Mhz in case we have to access flash or enter low power mode
      SOMC1 = 0x0C;
         __asm nop
         __asm nop
         __asm nop
         __asm nop

      gCDTPMxVesaOutSC_c = 0x4E;
      gCDTPMxSinSC_c = 0x4E;
      gCDTPMxRoutSC_c = 0x4E;

      mVesaLinkPowerSavingLowBusClk_d = FALSE;

     bDisableNwkDataSetUpdateInFlash = TRUE;
     mNwkDebugCode_d = NLME_RxEnableRequest(0xFFFFFF);
     bDisableNwkDataSetUpdateInFlash = FALSE;

     mVesaLinkPowerSavingNvAllow_d = TRUE;

    }

    SinStatusCheck();

    VesaOutStatusCheck();

}
/****************************************************************************/
```

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4729
Rev. 0,
5/2013