A Software Example using the Generic Option Board

M.H.Retzer
Professional & Commercial Radio
Worldwide Radio Solutions
Entreprise Mobility Solutions

***Abstract-* The Generic Option Board (GOB) and its companion XCMP/XNL interface protocol provide a rich and flexible toolbox for adding value to the MotoTrbo[TM] two-way radios. This richness and flexibility comes at a cost of complexity and detail, which often can be a steep learning curve to developers used to working with legacy analog radios with simpler hardware control lines. This Application Note provides an example using the on-board accelerometer, and works through all the steps needed to get any option board application accurately communicating with the radio.**

I. Introduction

First, we need to mention a few things this Application is not.. It is most likely not bug-free and perfect. It does, as far as we can tell, implement the XCMP/XNL protocols correctly, and avoids most of the common pit-falls encountered by someone new to these protocols. The C *Coding Style* used here may not be quite what you are used to. In many areas the author has broken the rules, sometimes for good reason, to clarify the direct manipulation of the hardware or economize in memory size or execution time. This particular application does not deal with transferring audio samples to and from the radio; audio is a big topic in itself, and may be covered in a future publication. This application passes ergonomic control information to and from the radio, and as such utilizes only the XNL Data Channel. Finally, although this note demonstrates the use of the accelerometer as a tilt sensor, it is not intended as a full-featured *Man-Down* emergency application.

The overall structure of the application is shown in Figure1. This is a simple controller loop without any operating system calls. Upon entry to *main()* some brief initialization is performed. Then the code drops into an infinite loop to do the actual work. While running, any process detecting a fatal error may set a *Panic* flag, forcing the application to re-initialize. Two interrupt service routines interface to the hardware. A peripheral DMA controller interrupt occurs at the 8KHz SSI radio frame rate, and is used to pull XNL fragments on and off the bus. A GPIO external interrupt is used to inform the control loop when a new *xyz* sample is ready from the accelerometer.

The complete listing of *main()* is provided in Appendix 1. The physical layer initialization of the hardware is covered in detail in Generic Option Board SDK Development Guide, and will only be briefly described here. A couple important points are highlighted.

The first thing to notice is the call to *Disable_global_interrupt()* right at the top of the code. For a normal power-up reset this call would not be necessary as interrupts default to disabled. The *Panic* error recovery used here requires explicit disabling of any interrupts which may be in progress.

The three lines which force the SSI Tx bus to tri-state are also needed for Panic recovery.

```
//Force SSC_TX_DATA_ENABLE Disabled.
  AVR32_GPIO.port[1].ovrs  =  0x00000001;
  AVR32_GPIO.port[1].oders =  0x00000001;
  AVR32_GPIO.port[1].gpers =  0x00000001;
```

Text Box 1 Forcing Tri-State

Another point to notice is a GPIO line is used to assist initial synchronization with the FSYNC signal coming from the radio. This serves two purposes. During normal operation this assures that the start up of our SSC receiver is properly phase aligned to the radio slots. Also, during debug, it allows the GOB to start up and wait in a known condition for the radio to wake up. After FSYNC is found and SSC initialized this GPIO is switched over to the Timer to maintain tri-state switching during the correct slots.

```
//Set up PB03 to watch FS.
//Waits for radio to start making FSYNC.
AVR32_GPIO.port[1].oderc = 0x00000002;
AVR32_GPIO.port[1].gpers = 0x00000002;
//Wait for FS High.
while ((AVR32_GPIO.port[1].pvr & 0x00000002) == 0);
//Wait for FS Low.
while ((AVR32_GPIO.port[1].pvr & 0x00000002) != 0);
```

Text Box 2 GPIO Synchronization

local_start_pll0 [Appendix 2] is application dependent. Here, the 12MHz crystal oscillator is enabled, and the phase locked loop is set up to provide a 48MHz clock to the processor and a 24MHz clock to peripheral bus A. One wait state is added to the FLASH controller to accommodate the faster processor speed.

The Interrupt Service drivers provided in the Atmel framework are highly flexible and re-configurable on the fly. Different service routines and priorities are dynamically swappable for each of the 38 possible interrupt events in the UC3B processor. This flexibility comes at a cost of a fairly big RAM table. Here we save RAM and some execution time by using a vector table in FLASH and writing custom INTC drivers.

local_start_SSC(), *local_start_PDC()*, and *local_start_timer()* [Appendix 3 to 5] are the same as discussed in *Generic Option Board SDK Development Guide*.

II. XNL PROCESSING

Once the hardware is initialized, the inner while loop in *main()* is continually executed, calling *process_XNL()* [Appendix 8] on every pass. This code processes the XNL state machine and retry scheduler diagrammed in Figure2. The purpose of this code is to establish and maintain the XNL connection with the radio. The machine is relatively simple; sequentially performing the tasks of:

- Receiving an XNL_MASTER_STATUS_BRDCST.
- Sending anXNL_DEVICE_AUTH_KEY_REQUEST.
- Receiving an XNL_DEVICE_AUTH_KEY_REPLY.
- Enciphering the returned 8 byte random number.
- Sending an XNL_ DEVICE_CONN_REQUEST.
- Receiving an XNL_DEVICE_CONN_REPLY.
- Verifying that the establishment was successful.
- Processing subsequent XNL_DATA_MSG's and XNL_DATA_MSG_ACK's.

The XNL processor passes messages back and forth to the radio using two dedicated buffer stores, Figure2. The *theRxCirBuffer* is organized as a wrap-around buffer containing fixed-size elements. The element size is 256 bytes in order to accommodate the largest physical layer fragment. The SSC interrupt handler receives non-Idle fragments and posts them sequentially into the circular buffer. The XNL processor handles received fragments in order. Flow control is managed in the buffer by the two indexes, *RxXNL_ProcessWaitingIndex* and *RxXNL_IsFillingMessageIndex*.

When no message is waiting to be processed, these indexes will be equal. *RxXNL_IsFillingMessageIndex* will always point to an empty element, ready for anything incoming to the interrupt service routine. After the interrupt routine fills an element, it will advance *RxXNL_IsFillingMessageIndex*. Non-equal indexes indicate to the XNL processor that a new fragment is waiting to be processed. The interrupt service routine will never advance *RxXNL_IsFillingMessageIndex* if it will collide with *RxXNL_ProcessWaitingIndex*; if such a collision would occur, the newly received fragment is discarded. The XNL processor will advance *RxXNL_ProcessWaitingIndex* after processing any waiting fragments.

```
typedef struct{
  U32            RxLinkCount;
  RxTemplate    *pRxTemplate;
  RxLink_State   theRxLink_State;
  S16          RxLink_Expected;  //Bytes Expected.
  U16            RxLink_CSUM;      //Scratch area
  S16            RxXNL_IsFillingMessageIndex;
  S16            RxXNL_IsFillingNextU16;
  S16            RxXNL_ProcessWaitingIndex;
} RxCirCtrlr;
```

Text Box 3 Structure of RxCirCtrlr

For transmitting XCMP/XNL messages the buffer storage arrangement is somewhat more complex; any message instance is comprised of one to six fragments. Any particular message has its own lifetime, and state behavior.

A message may be waiting in queue for a retry timeout, or may be associated through the *Transaction ID* with some response expected from the radio.

To accommodate this more complex behavior the transmit routines utilize a reusable pool of *TxInstance*'s, each maintaining its own behavior flags and retry variables, and up to six fragment indexes into a reusable *TxBufferPool* of physical blocks.

```
typedef struct {
      U32           behavior;
      U32           RetryTime;
      S32           BlockIndex[MAXPHYBLOCKS];
} TxInstance;
```

Text Box 4 Structure of TxInstance

To schedule a transmission, either immediately as SSC resources become available, or for a future retry, one must first call the following function in order to obtain a free TxInstance containing the requested number of physical blocks:

```
theInstance = reserveTxInstance(BLOCK_COUNT);
```

Text Box 5 reserveTxInstance [Appendix 12]

The caller must check the returned index, and execute appropriate error recovery should a free instance of the requested size not be available. A diagram of the transmit buffer storage is given in Figure 3. Book-keeping for determining availability and ownership of physical blocks uses the *TxBlockReservation* array. The caller populates the owned physical blocks, filling in the opcode, addresses, etc., and the fragment checksums.

```
sumTxInstance(theInstance);
```

Text Box 6 sumTxInstance [Appendix 13]

Communication between the background XCMP/XNL routines and the foreground SSC interrupt hardware controller is through the *TxXNL_schedule* structure and the *NextWaitingIndex* semaphore variable.

```
typedef struct {
      S32 AvailableInstanceCount;
      S32 AvailableBlockCount;
      U32 TxLinkState;
      S32 BytesRemaining;
      S32 CurrentInstanceIndex;
      S32 CurrentBlockIndex;
      S32 Next16TxIndex;
      S32 NextWaitingIndex;
} TxXNL_schedule;
```

Text Box 7 TxXNL_schedule structure

NextWaitingIndex is nulled to TXINSTANCESBOUND by the interrupt service routine indicating a new instance may be scheduled. The background XCMP/XNL routines may then write to *NextWaitingIndex*. [Except for initialization], the background may only write to *NextWaitingIndex* when it is equal to TXINSTANCESBOUND. While it equal to TXINSTANCESBOUND The foreground interrupt service will never write to *NextWaitingIndex*. The background may read the *CurrentInstanceIndex* at any time to determine if an actual transmission is in progress.

Depending on the application and messages coming from the radio, the option board background routines will write new messages into available *TxInstances*. These may be XNL Control messages, XNL Data messages containing XCMP broadcasts, requests, or replies, or XNL Data Acknowledgements, and in general all follow different behavior rules for retries and disposal. These different rules are handled, somewhat inelegantly, by various behavior flags in *TxInstance* structure. If *NextWaitingIndex* is free, the message may be signaled to the foreground interrupt service immediately, or it may be left in the message store for future transmission. Timing for scheduled transmissions is controlled by the 32 bit *RXLinkCount*, which is incremented by the interrupt service routine once every 125µS SSC frame.

Several helper routines aid in searching for and disposing of stored messages [Appendix 14]:

- findTxInstance_byOpCode()
- findTxInstance_byTransID()
- findTxInstance_byTimeout()
- releaseTxInstance(instanceIndex)
- garbageCollect()

III. XCMP

XCMP can be kind of tricky, a detailed walkthrough is in order. This implementation could be improved upon, but should serve to illustrate the XCMP steps.

Once we have completed all the necessary XNL connection steps, and received a *Success* code in the returned XNL_DEVICE_CONN_REPLY, there are a few remaining necessary XCMP steps. The XNL state is now XNL_CONNECTED, with *process_XNL()* operating in a continuous loop waiting to service XCMP transactions embedded within XNL data messages. In this example:

```
case XNL_CONNECTED:
if   (theXNL_Ctrlr.isIncomingMessage) {
     switch((theRxCirCtrlr.pRxTemplate)
               ->theXNL_Header.opcode){
     case XNL_DEVICE_SYSMAP_BRDCST:
          break;
     case XNL_DATA_MSG:
          processXNL_DATA_MSG();
          break;
     case XNL_DATA_MSG_ACK:
          processXNL_DATA_MSG_ACK();
          break;
     }//End of switch on XNL Header Opcode
     depleteAProcessedMessage();
 }
 break;
```

Text Box 7 Processing data messages

Upon receiving an XNL_DATA_MSG we *processXNL_DATA_MSG() [Appendix 15]*, deplete the received message, and break (which will continue processing XNL). To process an XNL data message, we first extract the message Destination Address, and determine if the message is for our Option Board. A message for the Option Board may be addressed specifically to the Option Board using the address previously supplied in the XNL_DEVICE_CONN_REPLY, or it may be a broadcast address.

```
DestinationAddress = (theRxCirCtrlr.pRxTemplate)
                    ->theXNL_Header.destination;
if ((0x000000 == DestinationAddress)
    || (theXNL_Ctrlr.XNL_DeviceAddress
     == DestinationAddress)){
     //The message is for me.
```

Text Box 8 Testing destination address

If any message is for us, we will try to schedule an XNL_DATA_MSG_ACK by calling *scheduleXNL_ACK() [Appendix 16]*. If memory resources are not available for sending this ACK, we will just ignore the incoming message, discontinue processing, and deplete it. The radio will eventually re-try. If memory resources are available, the scheduler may send the ACK immediately if the interrupt service *NextWaitingIndex* is available, or leave the message in queue with an immediate retry timeout.

The XNL_DATA_MSG processor then switches to a specific service routine based on the received XCMP opcode.

```
XCMPopcode = (theRxCirCtrlr.pRxTemplate)
->theXNL_Payload.ContentXNL_DATA_MSG.XCNPopcode;

switch (XCMPopcode){
case XCMP_DEVINITSTS:
  if ((theRxCirCtrlr.pRxTemplate)
     ->theXNL_Payload.ContentXNL_DATA_MSG.u8[4]
     == DIC){
     bunchofrandomstatusflags |= DIC;
     //Need do nothing else.
  }else{
     bunchofrandomstatusflags  &= 0xFFFFFFFC;
     //Device Init no longer Complete.
     sendDEVINITSTS();
  }
  break;

case XCMP_DEVMGMTBCST:
  temp  = (theRxCirCtrlr.pRxTemplate)
  ->theXNL_Payload.ContentXNL_DATA_MSG.u8[1] << 8;
  temp |= (theRxCirCtrlr.pRxTemplate)
  ->theXNL_Payload.ContentXNL_DATA_MSG.u8[2];
  if (temp ==
   theXNL_Ctrlr.XNL_DeviceLogicalAddress){
       if ((theRxCirCtrlr.pRxTemplate)
       ->theXNL_Payload.ContentXNL_DATA_MSG.u8[0]
       == 0x01){
            //Enable Option Board
            bunchofrandomstatusflags |= 0x00000002;
     }else{
            //Disable Option Board.
             bunchofrandomstatusflags &= 0xFFFFFFFD;
       }
  }
  break;

default:
  if ((XCMPopcode & XCMP_MTMask) == XCMP_requestMT){
     sendOpcode_Not_Supported(XCMPopcode);
  }
  break;
}
```

Text Box 9 Parsing XCMP Opcode

This demo application supports receiving only two XCMP opcodes: XCMP_DEVINITSTS, Device Initialization Status, and XCMP_DEVMGMTBCST, Device Management Broadcast. The Device Initialization Status message is used to announce that a device is present, as well as to provide information about the basic capabilities of the device. This message is also sent by the master to indicate that device initialization is complete (DIC). This is an indication that all devices may begin their normal XCMP messaging. After the master powers up, it broadcasts its Device Initialization Status message. Upon receiving the Device Initialization Status from the master, all devices broadcast their Device Initialization Status messages. There may be situations where a device powers up late or the master receives a device initialization after master sends out the Device Initialization Complete message. If the master detects that a new device powered on or it receives a Device Initialization Status message from a device after sending out Device Initialization Complete message then the Master sends out a Device Initialization Status message again to trigger all devices including the new device to resynchronize their status. In other words, if you are not DIC'ed, you must send an XCMP_DEVINITSTS message. This code does this by calling *sendDEVINITSTS() [Appendix 18]*.

The Device Management Broadcast message is used to manage the non-XNL master device from the XNL master or another non-XNL master device. The option board enabled or disabled selection in CPS is per channel. Therefore, at one channel the radio supports the option board operation and at another channel it may not. The radio will notify the option board if it is current in option board enabled personality or not by sending the XCMP Device Management Broadcast message. At power up or option board reset the radio will not send the notification if it is at an option board disabled personality. The option board shall default to disabled state unless it receives the enabled notification from the radio. The channel status is communicated to the main() loop using *bunchofrandomstatusflags*.

Unsupported XCMP messages must be ACK'ed, and requests must get a reply by calling *sendOpcode_Not_Supported() [Appendix 17]*.

XNL_DATA_MSG_ACK's scheduled by *scheduleXNL_ACK()* [Appendix 16] may begin transmission immediately if the foreground interrupt service is ready or the message may wait in queue until the foreground is ready.

```
if (txSchedule.NextWaitingIndex ==
TXINSTANCESBOUND){
    //Immediate transmission allowed.
    TxInstancePool[theInstance].RetryTime +=
                                STANDARDTIMEOUT;
    TxInstancePool[theInstance].behavior =
                              OWNEDXCMPACKPROTO;
    txSchedule.NextWaitingIndex = theInstance;
  }else{
    //Leave RetryTime at current time.
    TxInstancePool[theInstance].behavior =
                              TXXCMPACKPROTO;
  }
```

Text Box 10 Scheduling ACK

Here the STANDARDTIMEOUT is used only as an initializer, sometime in the far future. For the case where the ACK is queued the timeout is set to the current time; this will inform the retry scheduler to send the ACK as soon as foreground resources become available. In normal operation ACK's are not re-tried. Both behavior flags OWNEDXCMPACKPROTO and TXXCMPACKPROTO have CANDEPLETEAFTERSENT set. This allows the *garbageCollect()* routine to deplete this message as soon as the foreground has sent it out.

```
  void garbageCollect (void)
  {
    U8  i;

    for (i = 0; i < TXINSTANCESIMPLEMENTED;
         i++){ //Check each Instance.
         if (((TxInstancePool[i].behavior)
       & FGHANDSHAKEMASK)  == OKTOGARBAGECOLLECT)
         {  //Check for can delete.
             releaseTxInstance(i);
         }
    }
  }

// Note OKTOGARBAGECOLLECT  ==
              CANDEPLETEAFTERSENT | FGHASSENT
```

Text Box 11 GarbageCollection

The XCMP_DEVINITSTS message can also be started immediately or placed in queue. Since this is a message and not an ACK it should be re-tried if the transmission fails. The CANDEPLETEAFTERSENT flag is not set in these cases, and the STANDARDTIMEOUT is used.

```
if (txSchedule.NextWaitingIndex ==
TXINSTANCESBOUND){
   //Immediate transmission allowed.
   TxInstancePool[theInstance].RetryTime +=
                                 STANDARDTIMEOUT;
   TxInstancePool[theInstance].behavior =
                                OWNEDXCNPMSGPROTO;
   txSchedule.NextWaitingIndex = theInstance;
}else{
   //Leave RetryTime at current time.
   TxInstancePool[theInstance].behavior =
                                TXXCMPMSGPROTO;
}
```

Text Box 12 Scheduling Message

Following the transmission of the XCMP_DEVINITSTS message we expect to receive an XNL_DATA_MSG_ACK. All ACK's are caught by the *process_XNL()* loop, and passed to *processXNL_DATA_MSG_ACK() [Appendix 19]*. This routine determines if the ACK is for our device address and if the Transaction ID matches one of our scheduled transmissions. If so, it will release the Tx instance.

```
void processXNL_DATA_MSG_ACK(void)
{
  U16 DestinationAddress;
  U16 TransactionID;

  DestinationAddress = (theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Header.destination;
  if ((theXNL_Ctrlr.XNL_DeviceAddress) ==
                               DestinationAddress){
     //The ack is for me.
     TransactionID = (theRxCirCtrlr.pRxTemplate)
                      ->theXNL_Header.transactionID;

     releaseTxInstance(
          findTxInstance_byTransID(TransactionID));
  }
}
```

Text Box 13 Processing ACK

IV. Interrupt Service

The hardware interface to the radio SSC bus is diagramed in Figure 4. The hardware initialization sets up the SSC peripheral to synchronize with the frame sync coming from the radio and transfer the Slot #3 to #8 to and from the physical bus to the two double-buffers *RxBuffer[]* and *TxBuffer[]* using two DMA channels.

The Timer in Figure 4 is configured to automatically synchronize with FSYNC and hold our Transmit Data line in tri-state during the reserved Slot #1 and #2. Once set up, operation of this Timer and SSC peripheral are automatic, requiring no further software intervention. Every 125uS the *pdca_int_handler()* [Appendix 21] loads the alternate *TxBuffer[]* from the background random-access message store, and writes any received frames in the full *RxBuffer[]* into the background circular store. While these operations are occurring the DMA hardware is automatically filling the alternate *RxBuffer[]* and transmitting from the alternate *TxBuffer[]*.

With a 48MHz clock there are 6000 clock cycles between every pdca interrupt, setting an upper limit on any computing that can be accomplished inside the service routine. Any latency in starting the interrupt service, plus the computing time inside the interrupt service, cannot exceed this 6000 cycle upper bound or the signaling will be corrupted. The number of cycles used within the interrupt service routine may be instrumented using the system call *Get_system_register(AVR32_COUNT)*, obtaining a running 32-bit clock count since system reset. During development this instrumentation can be monitored to assure there is plenty of timing margin.

```
__attribute__((__interrupt__))
static void pdca_int_handler(void)
{
      intStartCount = Get_system_register(AVR32_COUNT);
      //TxBuffer and RxBuffer terminates the Option Card SSC Phlysical Layer.
      BufferIndex ^= 0x01; //Toggle Index.
      (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->marr =
                                                 (U32)(&TxBuffer[BufferIndex].theXNL_Channel.word);
      (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->tcrr = 3;  //Three words xfered each DMA.


      (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->marr =
                                                 (U32)(&RxBuffer[BufferIndex].theXNL_Channel.word);
      (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->tcrr = 3;  //Three words xfered each DMA.
      (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->isr;

      theRxCirCtrlr.RxLinkCount += 1;

      XNL_PhyRx(RxBuffer[BufferIndex].theXNL_Channel.word);
      TxBuffer[BufferIndex].theXNL_Channel.word = XNL_PhyTx();
      RxPhyMedia();


    //Dummy code to Idle Tx Media.
    TxBuffer[BufferIndex].thePayload_Channel.word[0] = PAYLOADIDLE0;
    TxBuffer[BufferIndex].thePayload_Channel.word[1] = PAYLOADIDLE1;
    intDuration = Get_system_register(AVR32_COUNT) - intStartCount;
}//End of pdca_int_handler.
```

Text Box 14 Interrupt Service

In the interrupt service code the first few lines perform the double-buffer swapping of *RxBuffer[]* and *TxBuffer[]*. The32-bit *RxLinkCount* is incremented providing a running count of received physical frames. This count is used in several places, notably in determining accurate XCMP/XNL retry timing.

The newly received 32-bit Slot #3 and #4 XNL Channel data, *RxBuffer[BufferIndex].theXNL_Channel.word*, is passed to the helper routine *XNL_PhyRx()* [Appendix 22] for storage into the circular buffer.

The *XNL_PhyTx()* [Appendix 24] helper routine returns a new 32-bit Slot #3 and #4 XNL Channel data word for filling the waiting *TxBuffer[]*.

The *RxPhyMedia()* helper routine, not discussed here, is called to handle any Slot #5 to #8 audio samples coming from the radio.

This application sends no audio samples to the radio. This interrupt service routine simply re-fills transmit Slots #5 to #8 with Idle Pattern. [This re-filling is strictly not necessary, as once filled nothing ever changes. But the operation is shown here for completeness.]

The real work of interfacing the foreground interrupt service to the background XCMP/XNL routines occurs in the *XNL_PhyRx()* and *XNL_PhyTx()* helper routines. Depending on the application, this interface may use different timing tradeoffs. The implementation presented next seems to work pretty well, but may not necessarily be the best strategy in all cases.

The *XNL_PhyRx()* [Appendix 22] helper routine state diagram is shown in Figure 5. This routine is used to load *theRxCirBuffer* from the received Slot #3 and #4 XNL physical channel. This routine will skip Idle Frames, and perform rudimentary formatting and checksum tests, and pass only good fragments to the circular buffer. It remains the responsibility of the background XNL processor to re-assemble any multiple fragments or account for missing fragments.

The routine spends most of its time simply waiting for a non-Idle message header. Once one is found, the state machine advances to process the subsequent checksum fields, message contents and terminator fields.

The *XNL_PhyTx()* [Appendix 24] helper state machine is diagrammed in Figure 6. This routine is used to fetch the next 32-bit Slot #3 and #4 word to be transmitted on the XNL physical channel. Most of the time this routine sends Idle pattern, waiting for the background to schedule a new message. When a message is ready, the routine will transmit all fragments comprising the message, back-to-back, inserting the proper headers and terminators. The checksum fields have been pre-computed in the background random store.

Once a message is started, the *NextWaitingIndex* semaphore is nulled (to TXINSTANCEBOUND). This enables the background to immediately schedule a new message for transmission.

Once a message is sent, the routine will manipulate the *TxInstance* behavior flags, signaling that the message is no longer needed by foreground, and may be deleted.

V. Accelerometer

The goals of this application are relatively modest:

- When first placed in a new position, determine the new reference gravity vector.
- Continue monitoring the accelerometer, filtering out any short-term motion variation.
- If the orientation of the gravity vector changes from the reference by more than a threshold amount, cause the radio to “beep,” and go back to the beginning to determine the new reference orientation.

Although simple in scope, this application illustrates all of the basic steps of initializing the hardware, collecting data, signal processing of the data, and communication of an event to the radio ergonomics.

An accelerometer is an electromechanical device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic - caused by moving or vibrating the accelerometer. By measuring the amount of static acceleration due to gravity, you can find out the angle the device is tilted at with respect to the earth. By sensing the amount of dynamic acceleration, you can analyze the way the device is moving. Note that the device is measuring acceleration forces, not actual acceleration, $dv/dt = d^2s/dt^2$. When the device is in free fall the x, y, and z outputs are zero. At rest, the vector V(x,y,z) points towards the center of the Earth.

The Generic Option Board includes the ST Microelectronics LIS302DL accelerometer, as a means to encourage development requiring motion sensing technology. These applications may encompass —emergency man-down applications, vibration or movement sensing, free-fall detection, motion telemetry, gesture interpretation, device abuse monitoring, or any other application requiring motion sensing or monitoring data. The LIS302DL may be configured to generate inertial interrupt signals when a new set of samples is ready or when a programmable acceleration threshold is crossed at least in one of the three axes. The thresholds and timing of interrupt generators are programmable by the G.O.B. application.

The LIS302DL’s SDA (Serial Data Input / Output) and SCL (Serial Clock) lines attach to the Atmel AVR32UC3B at TWI GPIO lines 9 & 10, respectively. The LIS302DL’s inertial interrupts (INT1 & INT2) interface to the AVR32UC3B at GPIO lines 5 and 6. These lines can be polled or configured by the AVR32UC3B as EXINT interrupts. Each pin on the AVR32UC3B has its own interrupt request, and can be individually masked.

The *accelerometer_init()* [Appendix 26] routine allocates processor I/O lines to the accelerometer:

```
//Allocate I/O's to TWI.
//AVR32_TWI_SDA_0_0_PIN port=0 pin=A mask=0x00000400
//AVR32_TWI_SCL_0_0_PIN port=0 pin=9 mask=0x00000200
//AVR32_TWI_SCL_0_0_FUNCTION
AVR32_TWI_SDA_0_0_FUNCTION
    AVR32_GPIO.port[0].pmr0c  = 0x000000600;
    AVR32_GPIO.port[0].pmr1c  = 0x000000600;
    AVR32_GPIO.port[0].gperc  = 0x000000600;
//  Accelerometer Outputs   GPIO Polling Inputs.
//INERTIAL INT1 port=0 pin=5 mask= 0x00000020
//INERTIAL INT2 port=0 pin=6 mask= 0x00000040
    AVR32_GPIO.port[0].oderc = 0x00000060;
    AVR32_GPIO.port[0].gpers = 0x00000060;
```

Text Box 15 AccelerometerI/O Lines

and initializes the Two Wire Interface (TWI) hardware:

```
//Init TWI.
AVR32_TWI.cr = AVR32_TWI_CR_SWRST_MASK;//resets TWI!
AVR32_TWI.sr;   // [shouldn't hurt]
AVR32_TWI.cwgr = 0x0000ECEC;//100KHz using 24Mhz PBA
```

Text Box16 TWI Interface

Communication with the accelerometer subregisters uses two driver routines, *my_writeabyte()* and *my_readabyte() [Appendix 27 and 28]*. Both routines return a status word which may be used to diagnose problems. Alternatively, the Atmel software framework provides perfectly good *twi_master_write()* and *twi_master_read()* drivers, though the ones used here seem somewhat easier to understand as they do not use interrupts. The low level interface to the accelerometer may be tested by reading the "Who Am I" register, with the result 0x3B stored to *data*.

```
//For Test, read Who_am_I
TWI_status = my_readabyte(LIS302DL_REG_WHO_AM_I,
&data);
```

Text Box 17 TWI reading [Appendix 28]

VI. Algorithm

Provided the radio is at rest, the output of the accelerometer is a vector (x,y,z) pointing towards the center of the Earth with the known magnitude of 1G. We can approximate the at rest condition by averaging (filtering) for a sufficiently long time, or by throwing out any tilt measurements when we detect the radio is moving. Since a general Man Down emergency application usually uses some combination of change in tilt and lack of motion to trigger the alarm these two criteria work well together; the motion detector can be used to discard tilt estimates. This demonstration application does not implement motion detection, but we will see that simple filtering alone does not perform too badly for determining change in tilt.

Suppose we have measured a reference vector (*a*,*b*,*c*), and we want to test a new vector measurement (*x*,*y*,*z*) to determine if the radio has tilted. The length, L, of each vector is the square root of the sum of the squares of the components; $\sqrt{a^2+b^2+c^2}$ and $\sqrt{x^2+y^2+z^2}$. Where here we happen to know that $L_{abc} \cong L_{xyz} \cong G$.

The dot product of two vectors is defined as the product of the lengths times the cosine of the angle between them.

$$(a,b,c)\bullet(x,y,z)=L_{abc}L_{xyz}\cos(\alpha)$$

The dot product may also be calculated by adding the products of like components.

$$(a,b,c)\bullet(x,y,z)=ax+by+cz$$

So:

$$\cos(\alpha)=\frac{ax+by+cz}{L_{abc}L_{xyz}}$$

The examples in the Atmel framework library for the accelerometer component use the math library to calculate this angle exactly. But since we are only interested in roughly knowing if the radio has changed tilt we can greatly simplify the calculation. First notice that if the tilt angle has not changed, *cos(0)* = 1, so:

$$\cos(0)=\frac{ax+by+cz}{L_{abc}L_{xyz}}=1$$

$$ax+by+cz=L_{abc}L_{xyz}$$

At some tilt difference, say 60°:

$$\cos(60^\circ)=\frac{ax+by+cz}{L_{abc}L_{xyz}}=0.5$$

so, to test if some new vector (a,b,c) is tilted more than 60° from our reference vector (x,y,z), all we need to do is calculate the dot product and compare the result with ½ G^2.

In Figure 8, as we turn the radio each component of the accelerometer output vector (x,y,z) can vary between ±G, with the magnitude, discounting the force needed to turn the radio, approximately equal to 1G. At a nominal sensitivity of 18mG/digit, 1G≈56. [Here we seem to be running somewhat hot, but still within spec-sheet tolerance.]

For the initialization parameters used, the accelerometer provides a new output set of samples every 10mS, equivalent to a Nyquist frequency of 50Hz. A sixteen tap FIR filter on each of the three channels provides some data smoothing and allows decimation by four to reduce computation time. The frequency response of this filter is shown in Figure 9. Integer arithmetic is used to implement this filter, with the coefficient taps scaled to avoid overflow in the dot-product calculation.

The accelerometer is polled within the infinite main() loop and if tilt is detected a flag is set in *bunchofrandomstatusflags*, and the radio may be alerted. XCMP/XNL only alerts the radio if the XNL Channel has successfully been established and the radio is on an Option Board enabled channel.

Upon entry, *processAccelerometer()* [Appendix 29] polls the GPIO line attached to the INERTIAL INT1 hardware line. This I/O is configured to go high when a set of three new (x,y,z) samples are ready. If samples are not ready, the routine just exits. Ready samples are read into three, sixteen element, circular buffer arrays according to the *accelerometerIndex*. This index is then incremented and wrapped. It is only necessary to calculate the filter output for four of the sixteen samples.

The remaining logic within *processAccelerometer()* is run every decimation time and is just timing and book-keeping. A full one Second of samples must be collected to establish the reference vector. Once a reference is established, every filtered output vector is compared against this reference by running the dot-product calculation in wearenottilted(). If any filtered accelerometer output within a one Second interval is not tilted, the whole one Second interval is determined to be not tilted. This helps to eliminate false detections caused by rapidly shaking the radio.

```
while(DontPanic){
  process_XNL();
  processAccelerometer();

  //Test if tilt is detected.
  if(0x00000040 == (bunchofrandomstatusflags & 0x00000040)){
        //Test if option board initialized and enabled.
        if (0x00000003 == (bunchofrandomstatusflags & 0x00000003)){
              if (sendTONECTRLREQ()){
                bunchofrandomstatusflags &= 0xFFFFFFBF;
              }
        }else{
                bunchofrandomstatusflags &= 0xFFFFFFBF;
        }
  }

  if (0x00000010 == (bunchofrandomstatusflags & 0x00000020)){
          processDoubleClick();
  }
}
```

TextBox 18 Main Loop using bunchofrandomstatusflags

```
void processAccelerometer(void)
{
  U32 TWI_status;

  if ((AVR32_GPIO.port[0].pvr & 0x00000020) != 0){//else, just leave.

    TWI_status = my_readabyte(LIS302DL_REG_OUT_X, &accsamples[accelerometerIndex][0]);
    TWI_status = my_readabyte(LIS302DL_REG_OUT_Y, &accsamples[accelerometerIndex][1]);
    TWI_status = my_readabyte(LIS302DL_REG_OUT_Z, &accsamples[accelerometerIndex][2]);
       //This should clear INERTIAL INT1.

       if (0x00000000 == (accelerometerIndex & 0x00000003)){ //Decimation
           filter(accelerometerIndex);

…

       }

    accelerometerIndex = (accelerometerIndex + 1) & 0x0000000F;
  }
}
```

Text Box 19 processAccelerometer

VII. Alerting the Radio

The accelerometer routines indicate to *main()* a tilt change by using *bunchofrandomstatusflags*.

```
while(DontPanic){
  process_XNL();
  processAccelerometer();
  //Test if tilt is detected.
  if(0x00000040 == (bunchofrandomstatusflags
     & 0x00000040)){
     //Test if option board initialized and enabled.
     if (0x00000003 == (bunchofrandomstatusflags
         & 0x00000003)){
        if (sendTONECTRLREQ()){
           bunchofrandomstatusflags &= 0xFFFFFFBF;
        }
     }else{
        bunchofrandomstatusflags &= 0xFFFFFFBF;
     }
  }
}
```

Text Box 20 Main Loop

If tilt is detected, and the Option Board is initialized and enabled on the channel, we attempt to send a tone request to the radio. Upon successful scheduling of the XNL_DATA_MSG, the tilt flag is released. This message will be retried until the XNL_DATA_MSG_ACK with corresponding Transaction ID is received. Should the Option Board be uninitialized of on a non-Option Board enabled channel, the tilt flag is just released.

VIII. Performance

The critical parameter in this application is the time spent within the PDCA interrupt service routine handling the SSI bus transactions. This must never exceed the number of clock cycles available. Measurements show this time between 386 to 739 clock cycles. With 6000 cycles available (48MHz), there is plenty of margin available, and better than 87% of the processor available to service application processing.

Figure 10 plots FLASH memory utilization for each of the code modules in the application. Total size is 11424 bytes. Of this, 1488 bytes are used for the actual Accelerometer application, with the rest needed to service the radio interface.

Michael Retzer is a Member of the Technical Staff in the Professional & Commercial Radio group, and has been a software gunslinger for Motorola since 1976. In this role he has had to pull the cat out of the hat on numerous occasions. He holds seventeen patents covering various wireless and signal processing techniques using small processors. Mr. Retzer has been developing embedded software and hardware since paper tape was high-tech: “The languages have changed; the parts have gotten faster; more people overlook the same bugs.” Recent projects have included the hardware and software design for the Generic Option Board, and a tutorial on two-way radio Trunking presented in Penang, Malaysia.

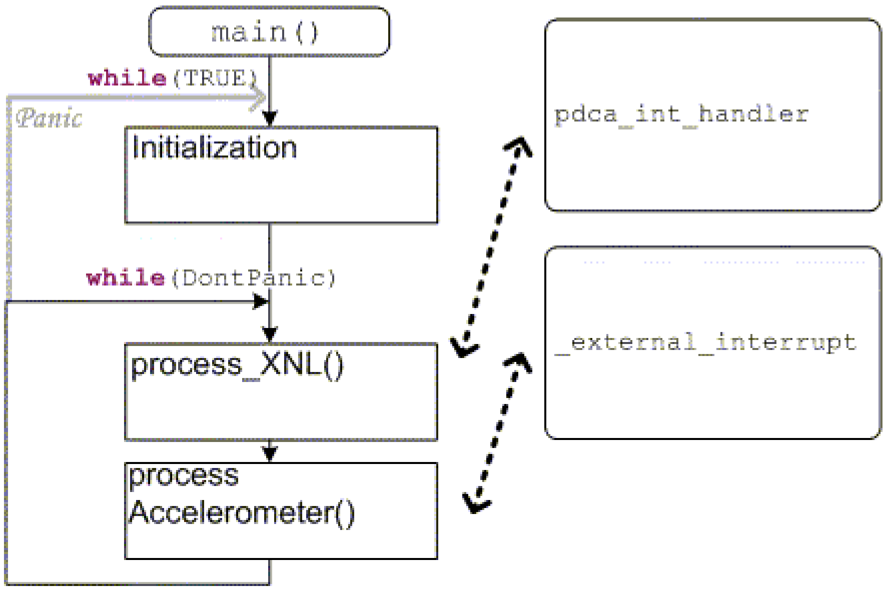


Figure 1 overall structure of the application

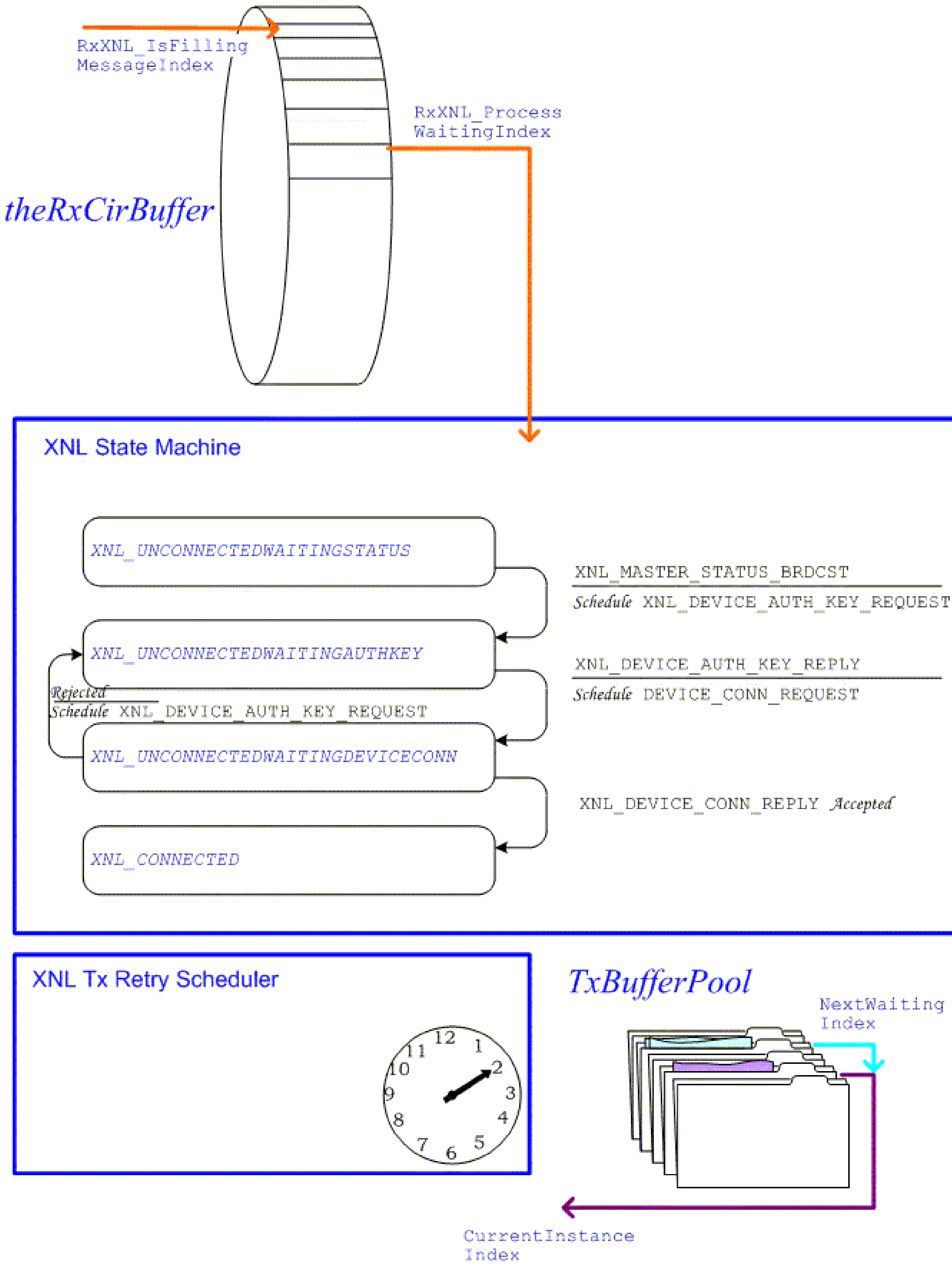


Figure 2 The XNL processor passes messages back and forth to the radio using two dedicated buffer stores.

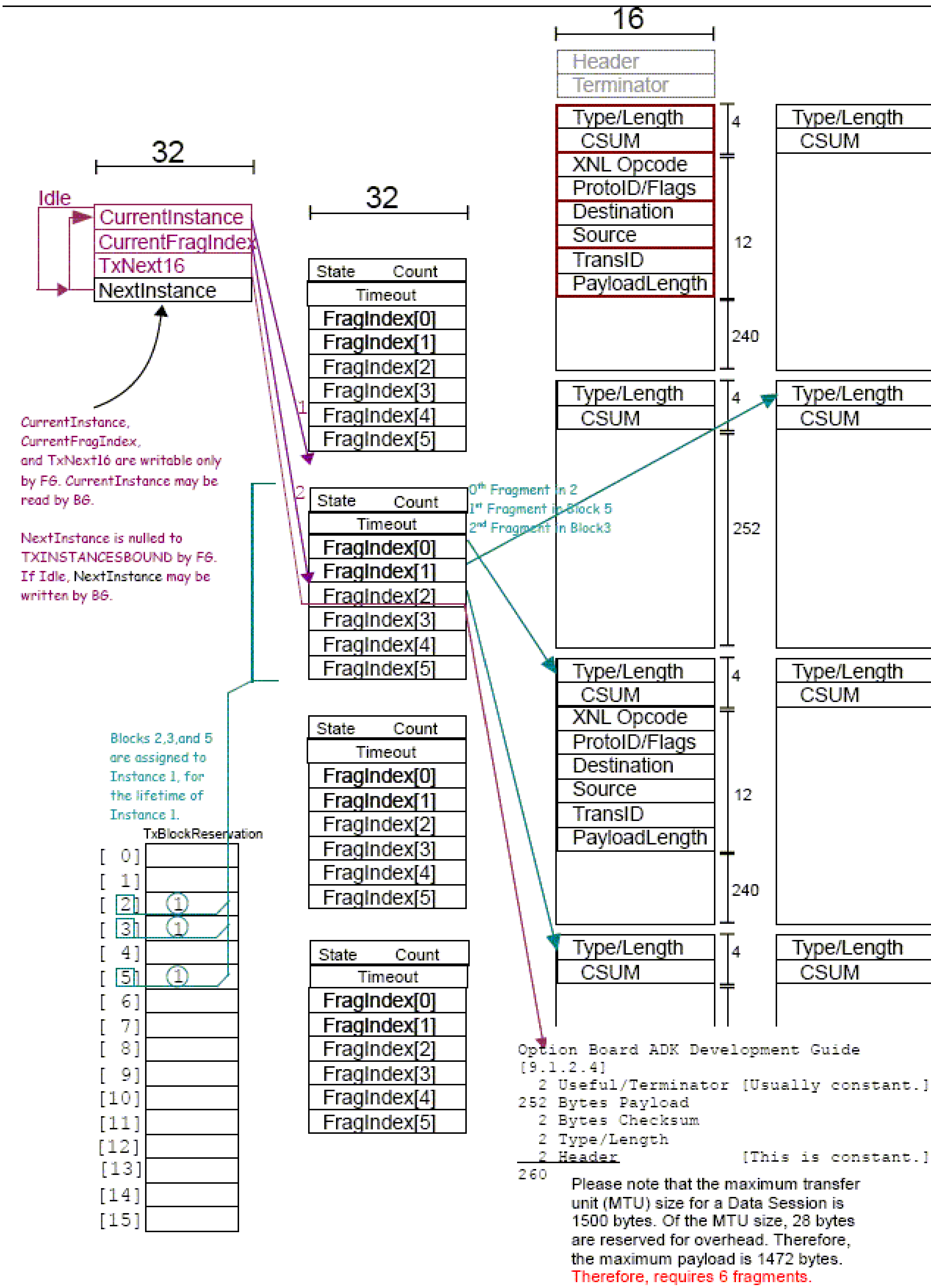

32
Idle
CurrentInstance
CurrentFragIndex
TxNext16
NextInstance
CurrentInstance, CurrentFragIndex, and TxNext16 are writable only by FG. CurrentInstance may be read by BG.
NextInstance is nulled to TXINSTANCESBOUND by FG. If Idle, NextInstance may be written by BG.
Blocks 2,3,and 5 are assigned to Instance 1, for the lifetime of Instance 1.
TxBlockReservation
[0]
[1]
[2] 1
[3] 1
[4]
[5] 1
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
32
State Count
Timeout
FragIndex[0]
FragIndex[1]
FragIndex[2]
FragIndex[3]
FragIndex[4]
FragIndex[5]
1
2
0th Fragment in 2
1st Fragment in Block 5
2nd Fragment in Block3
16
Header
Terminator
Type/Length
CSUM
XNL Opcode
ProtoID/Flags
Destination
Source
TransID
PayloadLength
4
12
240
252
Option Board ADK Development Guide [9.1.2.4]
2 Useful/Terminator [Usually constant.]
252 Bytes Payload
2 Bytes Checksum
2 Type/Length
2 Header [This is constant.]
260
Please note that the maximum transfer unit (MTU) size for a Data Session is 1500 bytes. Of the MTU size, 28 bytes are reserved for overhead. Therefore, the maximum payload is 1472 bytes.
Therefore, requires 6 fragments.

Figure 3 transmit buffer storage

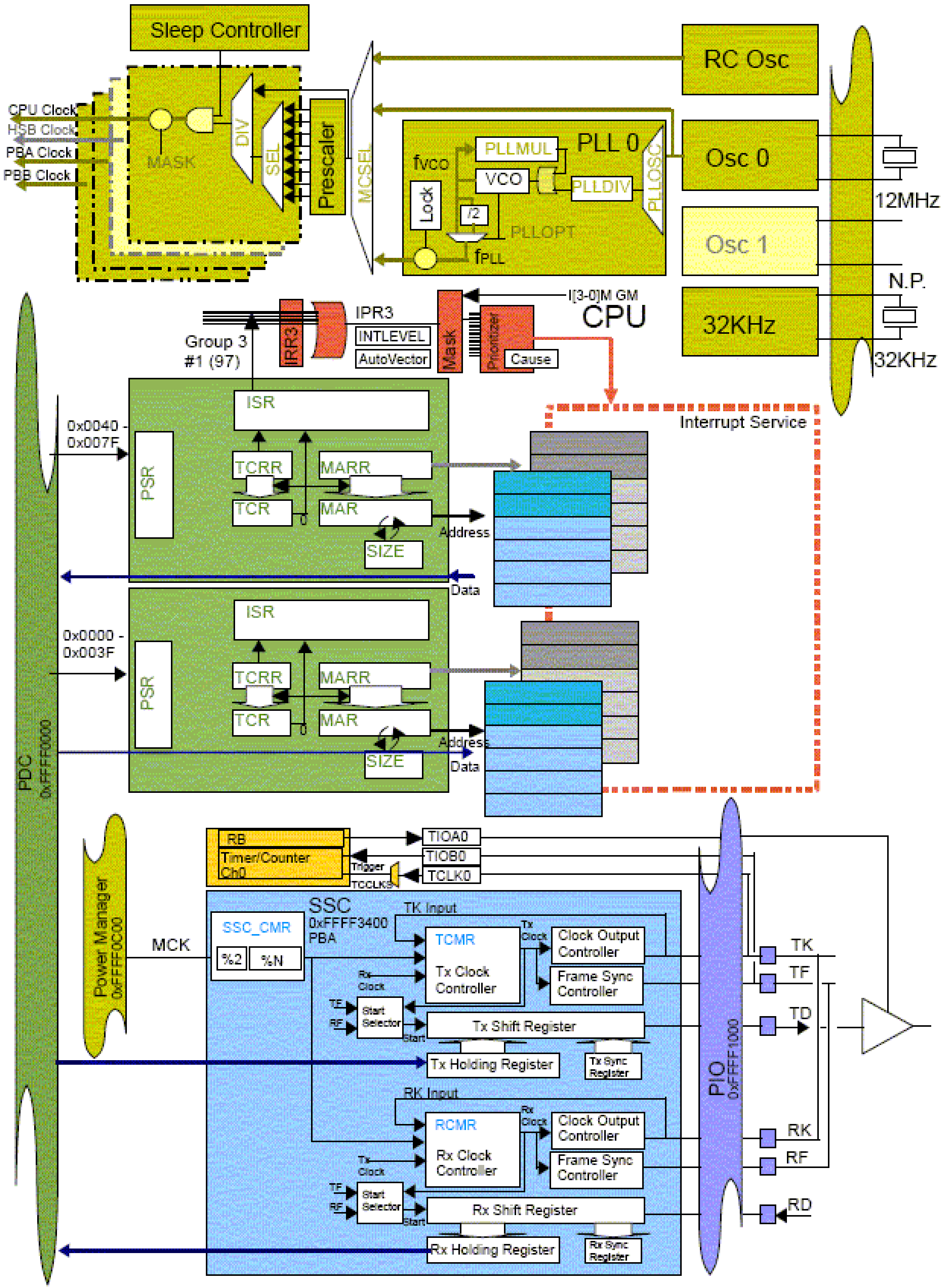
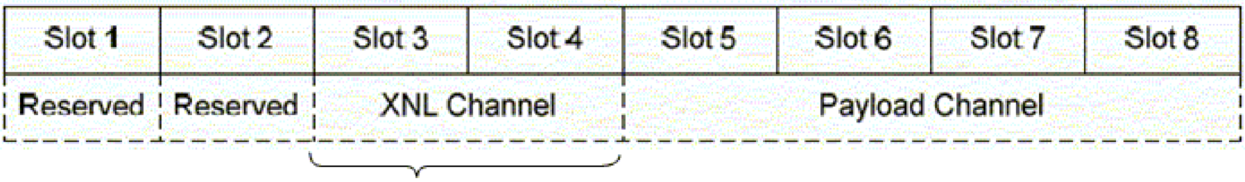


Figure 4 hardware interface to the radio SSC



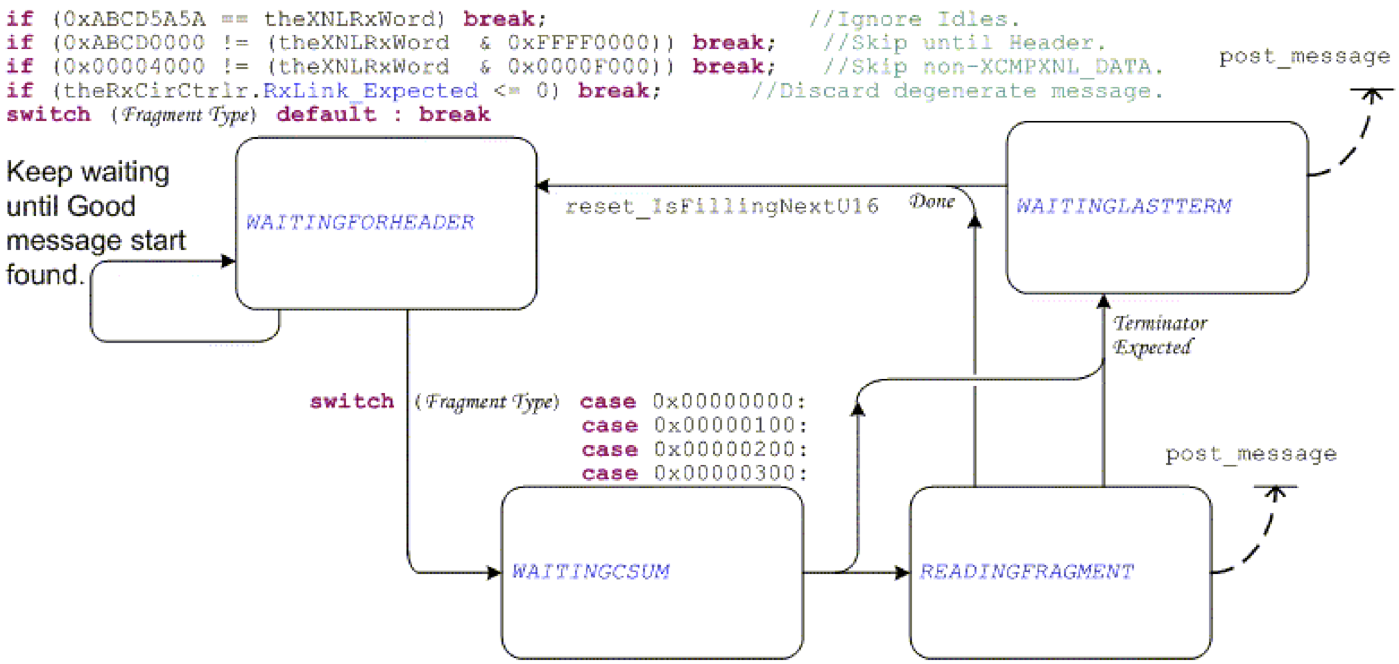


Figure 5 XNL_PhyRx() helper

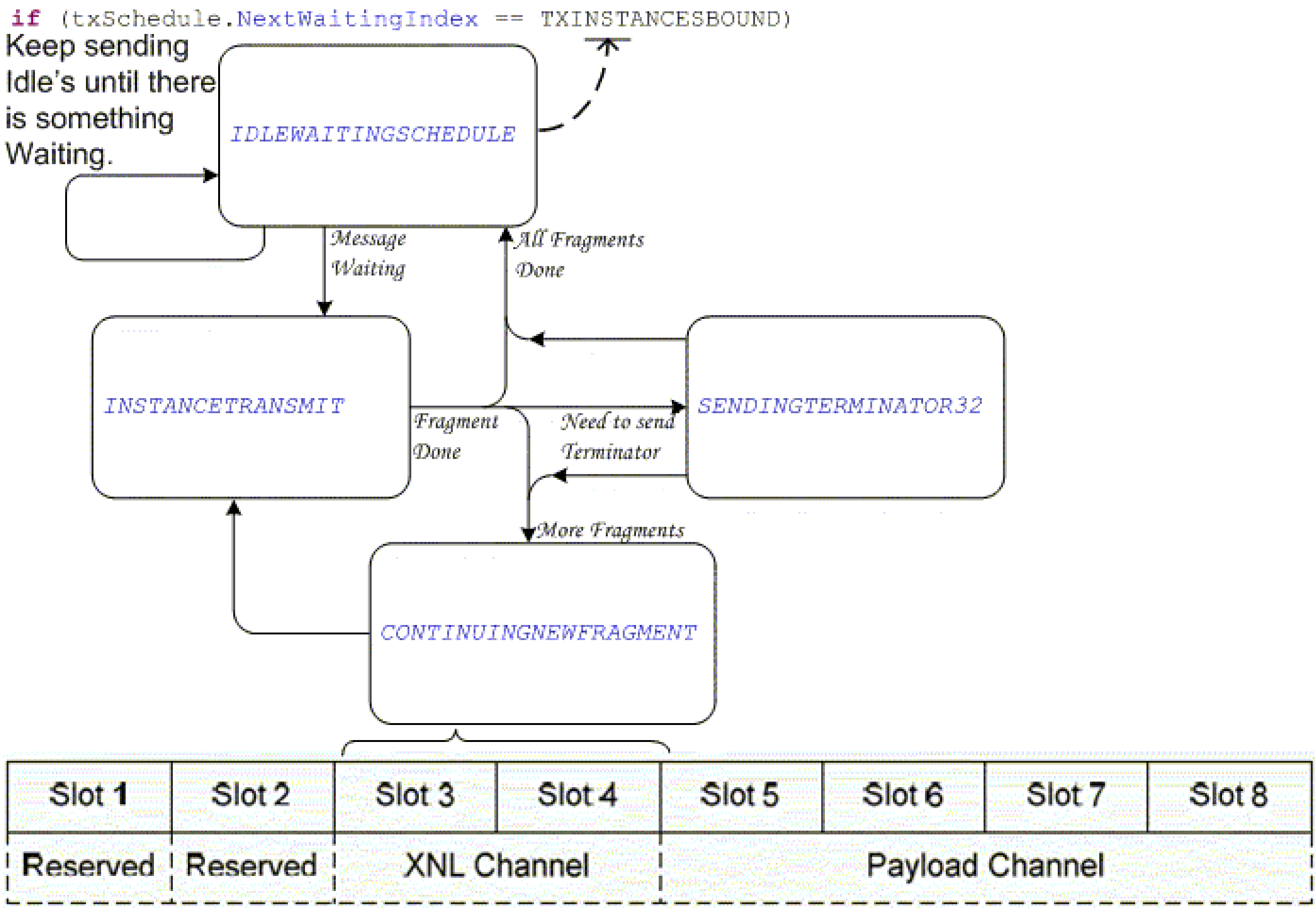


Figure 6 XNL_PhyTx() helper

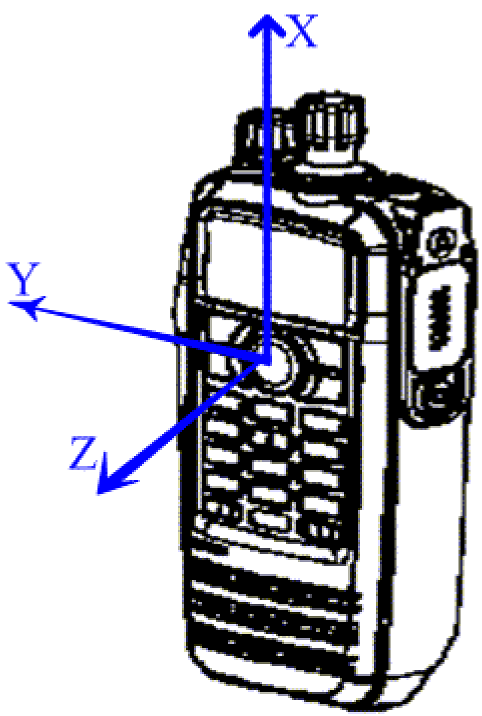


Figure 7 Accelerometer Orientation

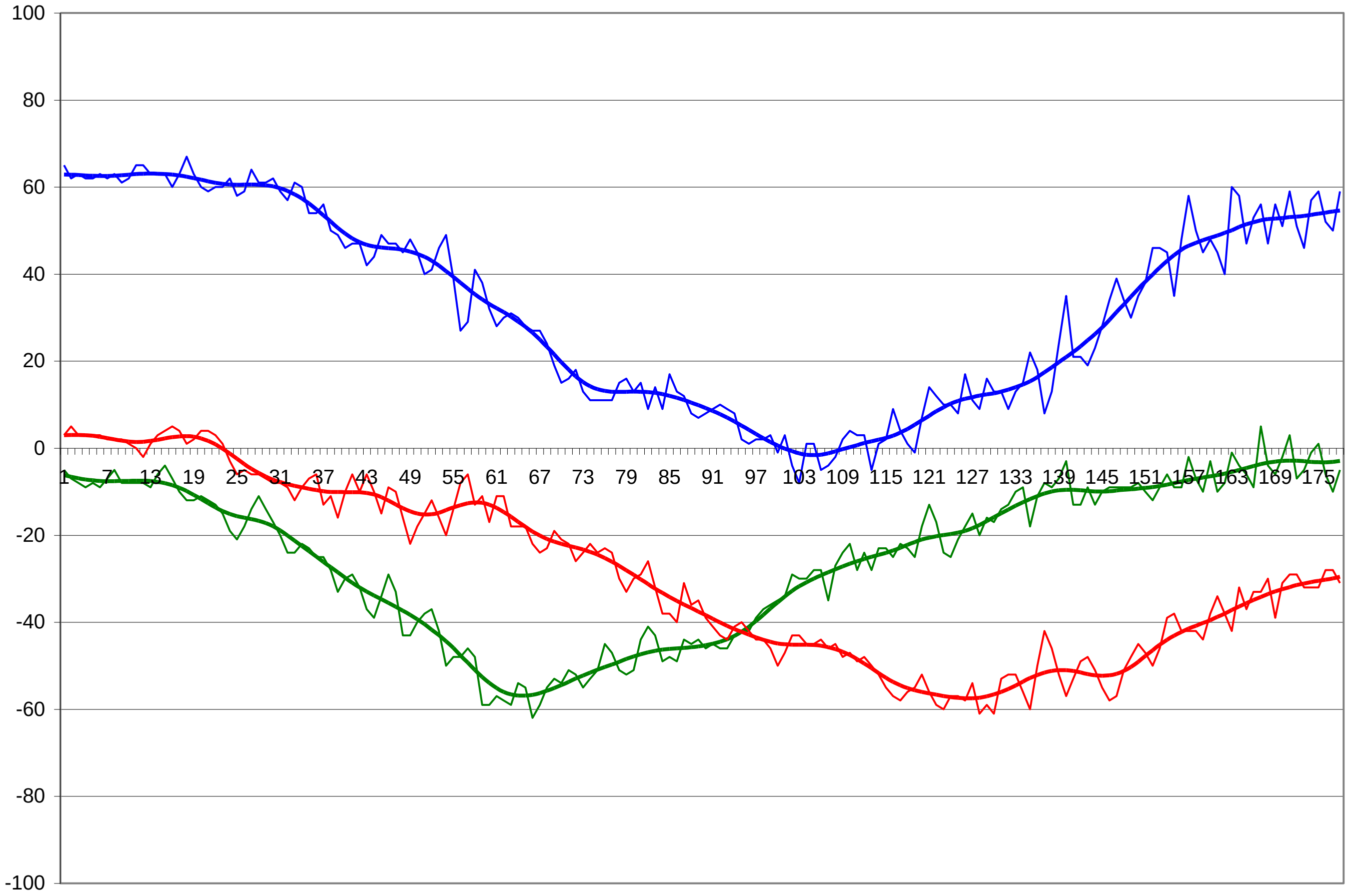


Figure 8 as we turn the radio each component of the accelerometer output vector (x,y,z) can vary between ±G

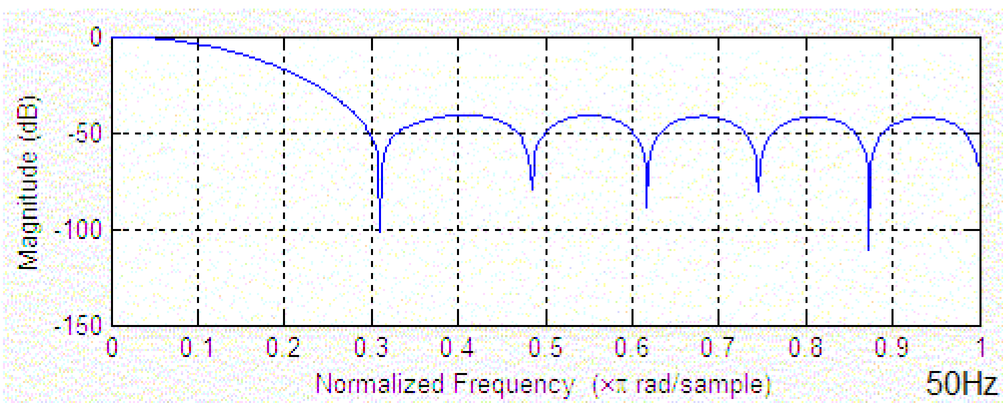


Figure 9 frequency response

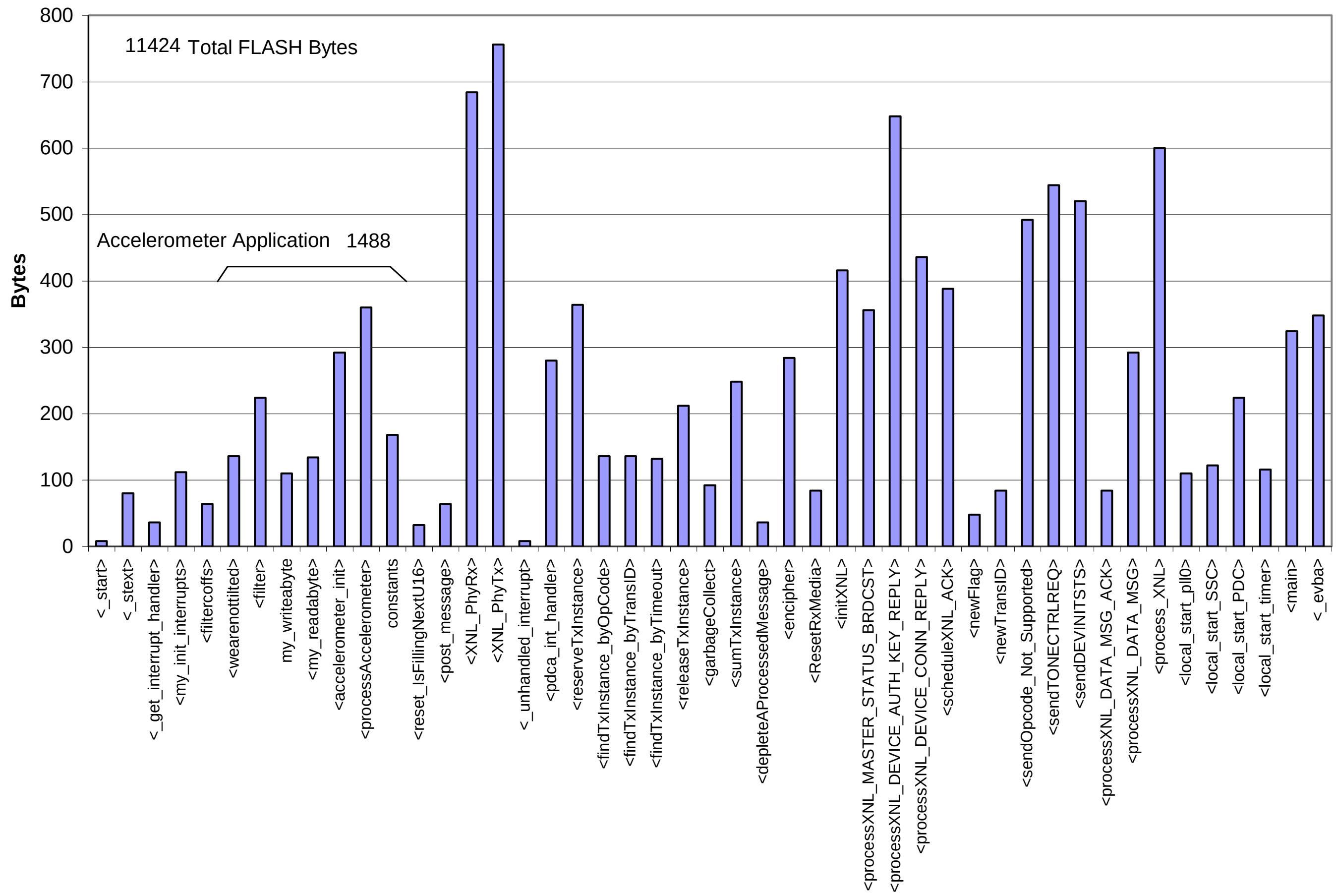


Figure10 FLASH Size

APPENDICES

Appendix 1	main()
Appendix 2	local_start_pll0()
Appendix 3	local_start_SSC()
Appendix 4	local_start_PDC()
Appendix 5	local_start_timer()
Appendix 6	Interrupt Initialization
Appendix 7	initXNL()
Appendix 8	process_XNL()
Appendix 9	processXNL_MASTER_STATUS_BRDCST()
Appendix 10	processXNL_DEVICE_AUTH_KEY_REPLY()
Appendix 11	processXNL_DEVICE_CONN_REPLY()
Appendix 12	reserveTXInstance()
Appendix 13	sumTxInstance()
Appendix 14	Helper Routines
Appendix 15	processXNL_DATA_MSG()
Appendix 16	scheduleXNL_ACK()
Appendix 17	sendOpcode_Not_Supported()
Appendix 18	sendDEVINITSTS()
Appendix 19	processXNL_DATA_MSG_ACK()
Appendix 20	newFlag() and newTransID()
Appendix 21	pdca_int_handler()
Appendix 22	XNL_PhyRx()
Appendix 23	reset_IsFillingNextU16() and post_message()
Appendix 24	XNL_PhyTx()
Appendix 25	Definitions & Structures Used By XCMP / XNL
Appendix 26	accelerometer_init()
Appendix 27	my_writeabyte()
Appendix 28	my_readabyte()
Appendix 29	processAccelerometer() and processDoubleClick()
Appendix 30	filter() and wearenottilted()
Appendix 31	sendTONECTRLREQ()

Appendix 1 main()

```
int main(void)
{
  while(TRUE){
    Disable_global_interrupt();
    DontPanic = TRUE;


    //Force SSC_TX_DATA_ENABLE Disabled as soon as possible.
      AVR32_GPIO.port[1].ovrs  =  0x00000001;  //Value will be high.
      AVR32_GPIO.port[1].oders =  0x00000001;  //Output Driver will be Enabled.
      AVR32_GPIO.port[1].gpers =  0x00000001;  //Enable as GPIO.

    bunchofrandomstatusflags = 0x00000000;

    local_start_pll0();
    my_init_interrupts();
    accelerometer_init();


    //Set up PB03 to watch FS.
    //Waits for radio to start making FSYNC.
      AVR32_GPIO.port[1].oderc = 0x00000002;
      AVR32_GPIO.port[1].gpers = 0x00000002;
      while ((AVR32_GPIO.port[1].pvr & 0x00000002) == 0); //Wait for FS High.
      while ((AVR32_GPIO.port[1].pvr & 0x00000002) != 0); //Wait for FS Low.

    local_start_SSC();
    local_start_PDC();
    initXNL();

    //Start the SSC Physical Layer.
    (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->cr = AVR32_PDCA_TEN_MASK;
    (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->cr = AVR32_PDCA_TEN_MASK;
    (&AVR32_SSC)->cr = AVR32_SSC_CR_RXEN_MASK | AVR32_SSC_CR_TXEN_MASK;
    (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->ier = AVR32_PDCA_RCZ_MASK;

    Enable_global_interrupt();

    while ((AVR32_GPIO.port[1].pvr & 0x00000002) == 0); //Wait for FS High.
    while ((AVR32_GPIO.port[1].pvr & 0x00000002) != 0); //Wait for FS Low.
    local_start_timer();


    while(DontPanic){
      process_XNL();
      processAccelerometer();
      //Test if tilt is detected.
      if(0x00000040 == (bunchofrandomstatusflags & 0x00000040)){
          //Test if option board initialized and enabled.
         if (0x00000003 == (bunchofrandomstatusflags & 0x00000003)){
             if (sendTONECTRLREQ()){
                 }
          bunchofrandomstatusflags &= 0xFFFFFFBF;
         }
      }
      if (0x00000010 == (bunchofrandomstatusflags & 0x00000020)){
          processDoubleClick();
      }
    }
    //Code will get here, and re-start, upon Panic.
  }
}
```

Appendix 2 local_start_pll0

```
void local_start_pll0 (void)
{
	//pm_switch_to_osc0(pm, 12000000, 3);
	//	pm_enable_osc0_crystal(pm, 12000000);
	//		pm_set_osc0_mode(pm,AVR32_PM_OSCCTRL0_MODE_CRYSTAL_G3);  0x00000007
	//	pm_enable_clk0(pm, 3);
	//		pm_enable_clk0_no_wait(pm, 3);
			(&AVR32_PM)->oscctrl0 = 0x00000307;
			(&AVR32_PM)->mcctrl   = 0x00000004;
	//		pm_wait_for_clk0_ready(pm);
			while (!((&AVR32_PM)->poscsr & AVR32_PM_POSCSR_OSC0RDY_MASK));
	//	pm_switch_to_clock(pm, AVR32_PM_MCSEL_OSC0);
			(&AVR32_PM)->mcctrl   = 0x00000005;

	//pm_pll_setup(pm,
	//		0,   // use PLL0
	//		7,   // MUL=7 in the formula
	//		1,   // DIV=1 in the formula
	//		0,   // Sel Osc0/PLL0 or Osc1/PLL1
	//		16); // lockcount in main clock for the PLL wait lock
	//pm_pll_set_option(pm, 0, //PLL number 0
	//			1, //freq Set to 1 for VCO frequency range 80-180MHz
	//			1, //div2 Divide the PLL output frequency by 2
	//			0);//0 to enable the Wide-Bandith Mode
	//pm_pll_enable(pm,0);
			(&AVR32_PM)->pll[0] = 0x1007010D;


	//pm_wait_for_pll0_locked(pm);
			while (!((&AVR32_PM)->poscsr & AVR32_PM_POSCSR_LOCK0_MASK));

	//pm_cksel(pm, 1,  //Bus A clock divisor enable = 1
	//		0,  //Bus A select = 0 (PBA clock = 48MHz/2 = 24MHz).
	//		0,  //B clock divisor enable = 0
	//		0,  //Bus B select = 0
	//		0,  //HS Bus clock divisor enable = 0
	//		0); //HS Bus select = 0
			(&AVR32_PM)->cksel = 0x00800000;

	//flashc_set_wait_state(1);
			AVR32_FLASHC.fcr = 0x00000040;

	//pm_switch_to_clock(pm, AVR32_PM_MCSEL_PLL0);
			(&AVR32_PM)->mcctrl   = 0x00000006;


			AVR32_HMATRIX.mcfg[AVR32_HMATRIX_MASTER_CPU_INSTRUCTION] = 0x1;
}
```

Appendix 3 **local_start_SSC**

```
void local_start_SSC(void)
{
//Before using the SSC receiver, the PIO controller must be configured to dedicate the SSC
//receiver I/O lines to the SSC peripheral mode. [23.6.1]
//Before using the SSC transmitter, the PIO controller must be configured to dedicate the SSC
//transmitter I/O lines to the SSC peripheral mode. [23.6.1]
// Assign GPIO to SSC.
        //gpio_enable_module
        //      gpio_enable_module_pin
            AVR32_GPIO.port[1].pmr0c = 0x00000DC0;
            AVR32_GPIO.port[1].pmr1c = 0x00000DC0;
            AVR32_GPIO.port[1].gperc = 0x00000DC0;

   //Software reset SSC
  (&AVR32_SSC)->cr = AVR32_SSC_CR_SWRST_MASK;

  (&AVR32_SSC)->cmr = AVR32_SSC_CMR_DIV_NOT_ACTIVE << AVR32_SSC_CMR_DIV_OFFSET;


  (&AVR32_SSC)->tcmr =
               AVR32_SSC_TCMR_CKS_RK_CLOCK              << AVR32_SSC_TCMR_CKS_OFFSET    |
                AVR32_SSC_TCMR_CKO_INPUT_ONLY            << AVR32_SSC_TCMR_CKO_OFFSET    |
                1                                        << AVR32_SSC_TCMR_CKI_OFFSET    |
                AVR32_SSC_TCMR_CKG_NONE                  << AVR32_SSC_TCMR_CKG_OFFSET    |
                4                                        << AVR32_SSC_TCMR_START_OFFSET  |
                32                                       << AVR32_SSC_TCMR_STTDLY_OFFSET |
                63                                       << AVR32_SSC_TCMR_PERIOD_OFFSET;

  (&AVR32_SSC)->tfmr =
                 31                                     << AVR32_SSC_TFMR_DATLEN_OFFSET |
                  0                                      << AVR32_SSC_TFMR_DATDEF_OFFSET |
                  1                                      << AVR32_SSC_TFMR_MSBF_OFFSET   |
                  2                                      << AVR32_SSC_TFMR_DATNB_OFFSET  |
                  0                                      << AVR32_SSC_TFMR_FSLEN_OFFSET  |
                  AVR32_SSC_TFMR_FSOS_INPUT_ONLY         << AVR32_SSC_TFMR_FSOS_OFFSET   |
                  0                                      << AVR32_SSC_TFMR_FSDEN_OFFSET  |
                  1                                      << AVR32_SSC_TFMR_FSEDGE_OFFSET;

  (&AVR32_SSC)->rcmr =
               AVR32_SSC_RCMR_CKS_RK_PIN                <<  AVR32_SSC_RCMR_CKS_OFFSET    |
                AVR32_SSC_RCMR_CKO_INPUT_ONLY            << AVR32_SSC_RCMR_CKO_OFFSET    |
                0                                        << AVR32_SSC_RCMR_CKI_OFFSET    |
                AVR32_SSC_RCMR_CKG_NONE                  << AVR32_SSC_RCMR_CKG_OFFSET    |
                AVR32_SSC_RCMR_START_DETECT_FALLING_RF   << AVR32_SSC_RCMR_START_OFFSET  |
                0                                        << AVR32_SSC_RCMR_STOP_OFFSET   |
                32                                       << AVR32_SSC_RCMR_STTDLY_OFFSET |
                63                                       << AVR32_SSC_RCMR_PERIOD_OFFSET;

  (&AVR32_SSC)->rfmr =
                 31                                     << AVR32_SSC_RFMR_DATLEN_OFFSET |
                  0                                      << AVR32_SSC_RFMR_LOOP_OFFSET   |
                  1                                      << AVR32_SSC_RFMR_MSBF_OFFSET   |
                  2                                      << AVR32_SSC_RFMR_DATNB_OFFSET  |
                  0                                      << AVR32_SSC_RFMR_FSLEN_OFFSET  |
                  AVR32_SSC_RFMR_FSOS_INPUT_ONLY         << AVR32_SSC_RFMR_FSOS_OFFSET   |
                  1                                      << AVR32_SSC_RFMR_FSEDGE_OFFSET;

}
```

Appendix 4 **local_start_PDC**

```
void local_start_PDC(void)
{
        BufferIndex = 1;

        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->idr = AVR32_PDCA_RCZ_MASK
                                                         | AVR32_PDCA_TRC_MASK
                                                         | AVR32_PDCA_TERR_MASK;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->isr;   //Dummy read?
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->mar = (U32)(&RxBuffer[0].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->tcr = 3;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->psr = AVR32_PDCA_PID_SSC_RX;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->marr = (U32)(&RxBuffer[1].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->tcrr = 3;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->mr = AVR32_PDCA_WORD;


        TxBuffer[0].theXNL_Channel.word = XNL_IDLE;
        TxBuffer[0].thePayload_Channel.word[0] = PAYLOADIDLE0;
        TxBuffer[0].thePayload_Channel.word[1] = PAYLOADIDLE1;
        TxBuffer[1].theXNL_Channel.word = XNL_IDLE;
        TxBuffer[1].thePayload_Channel.word[0] = PAYLOADIDLE0;
        TxBuffer[1].thePayload_Channel.word[1] = PAYLOADIDLE1;

        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->idr = AVR32_PDCA_RCZ_MASK
                                                          | AVR32_PDCA_TRC_MASK
                                                          | AVR32_PDCA_TERR_MASK; //Atomic!
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->isr; //Dummy read?
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->mar = (U32)(&TxBuffer[0].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->tcr = 3;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->psr = AVR32_PDCA_PID_SSC_TX;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->marr = (U32)(&TxBuffer[1].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->tcrr = 3;
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->mr = AVR32_PDCA_WORD;
}
```

Appendix 5 **local_start_timer**

```
void local_start_timer(void)
{
    //Route CLK to Timer
      AVR32_GPIO.port[0].pmr0s = 0x00100000;
      AVR32_GPIO.port[0].pmr1c = 0x00100000;
      AVR32_GPIO.port[0].gperc = 0x00100000;
      //Route FS and Tri-State to Timer.
      AVR32_GPIO.port[1].pmr0c = 0x00000003;
      AVR32_GPIO.port[1].pmr1c = 0x00000003;
      AVR32_GPIO.port[1].gperc = 0x00000003;

      (&AVR32_TC)->bmr = 4;
      (&AVR32_TC)->channel[0].cmr =
                                  AVR32_TC_BSWTRG_NONE       << AVR32_TC_BSWTRG_OFFSET   |
                                  AVR32_TC_BEEVT_NONE        << AVR32_TC_BEEVT_OFFSET    |
                                  AVR32_TC_BCPC_NONE         << AVR32_TC_BCPC_OFFSET     |
                                  AVR32_TC_BCPB_NONE         << AVR32_TC_BCPB_OFFSET     |
                                  AVR32_TC_ASWTRG_SET        << AVR32_TC_ASWTRG_OFFSET   |
                                  AVR32_TC_AEEVT_SET         << AVR32_TC_AEEVT_OFFSET    |
                                  AVR32_TC_ACPC_NONE         << AVR32_TC_ACPC_OFFSET     |
                                  AVR32_TC_ACPA_CLEAR        << AVR32_TC_ACPA_OFFSET     |
                                  1                          << AVR32_TC_WAVE_OFFSET     |
                                  AVR32_TC_WAVSEL_UP_NO_AUTO << AVR32_TC_WAVSEL_OFFSET   |
                                  1                          << AVR32_TC_ENETRG_OFFSET   |
                                  AVR32_TC_EEVT_TIOB_INPUT   << AVR32_TC_EEVT_OFFSET     |
                                  AVR32_TC_EEVTEDG_POS_EDGE  << AVR32_TC_EEVTEDG_OFFSET  |
                                  0                          << AVR32_TC_CPCDIS_OFFSET   |
                                  0                          << AVR32_TC_CPCSTOP_OFFSET  |
                                  AVR32_TC_BURST_NOT_GATED   << AVR32_TC_BURST_OFFSET    |
                                  1                          << AVR32_TC_CLKI_OFFSET     |
                                  AVR32_TC_TCCLKS_XC0        << AVR32_TC_TCCLKS_OFFSET;


      (&AVR32_TC)->channel[0].ra = 32;
      (&AVR32_TC)->channel[0].ccr = AVR32_TC_SWTRG_MASK | AVR32_TC_CLKEN_MASK;
}
```

Appendix 6 Interrupt Initialization

```
//Values to store in the interrupt priority registers for the
//various interrupt priority levels.
extern const U32 ipr_val[4];  //These are the  glue routines provided by exception.S

const __int_handler interrupt_priority_handlers[4] =
{ &pdca_int_handler,
  &_external_interrupt,
  &_unhandled_interrupt,
  &_unhandled_interrupt
};

//A negative value will yield an unrecoverable exception.
const int priorityMapping[AVR32_INTC_NUM_INT_GRPS] =
{  -1,                      //Group  0 SYSBLOCK
    1,                      //Group  1 EIC, PM, RTC
   -1,                      //Group  2 GPIO
    0,                      //Group  3 DMA Controller
   -1,                      //Group  4 FLASH Controller
   -1,                      //Group  5 UART0
   -1,                      //Group  6 UART1
   -1,                      //Group  7 UART2
   -1,                      //Group  8 Unknown
   -1,                      //Group  9 SPI
   -1,                      //Group 10 Unknown
   -1,                      //Group 11 TWI
   -1,                      //Group 12 PWM
   -1,                      //Group 13 SSC
   -1,                      //Group 14 Timer/Counter
   -1,                      //Group 15 ADC
   -1,                      //Group 16 Unknown
   -1,                      //Group 17 USB
};



//Gets the interrupt handler of the current int_lev
__int_handler _get_interrupt_handler(unsigned int int_lev)
{
        return interrupt_priority_handlers[int_lev];
}

void my_init_interrupts(void)
{
  unsigned int int_grp;
  int requestedPriority;

  // For all interrupt groups,
  for (int_grp = 0; int_grp < AVR32_INTC_NUM_INT_GRPS; int_grp++)
  {
    requestedPriority = priorityMapping[int_grp];
    if (requestedPriority >= 0)
    {
       AVR32_INTC.ipr[int_grp] = ipr_val[requestedPriority];
    }
    else
    {
       AVR32_INTC.ipr[int_grp] = 0x00000000; //Zero Offset from _evba is unrecoverable.
    }
  }
}
```

Appendix 7 initXNL

```
void initXNL (void)
{
        int i;
        U8 masterQuery;

        ResetRxMedia();
        theRxCirCtrlr.theRxLink_State = WAITINGFORHEADER;
        theRxCirCtrlr.RxLinkCount =                   0;
        theRxCirCtrlr.RxXNL_IsFillingMessageIndex =  0;
        theRxCirCtrlr.RxXNL_IsFillingNextU16 =       0;
        theRxCirCtrlr.RxXNL_ProcessWaitingIndex =    0;
        theXNL_Ctrlr.XNL_State = XNL_UNCONNECTEDWAITINGSTATUS;
        theXNL_Ctrlr.isIncomingMessage =          FALSE;

          //TxBlockReservation has TXPOOLSIZE entries, [0,1,2...TXPOOLSIZE-1],
          //one for if each available fragment block.
          //Each entry is assigned to one of the available Instances (threads).
          //There are TXINSTANCESIMPLEMENTED implemented instances,
          //enabling TXINSTANCESIMPLEMENTED independent Tx mesage threads.
          //Initially each TxBlockReservation entry is set to
          //TXINSTANCESBOUND (== TXINSTANCESIMPLEMENTED), marking the block as available.
          for (i=0; i<TXPOOLSIZE; i++) TxBlockReservation[i] = TXINSTANCESBOUND;

          txSchedule.AvailableBlockCount = TXPOOLSIZE;

          //TxInstancePool has TXINSTANCESIMPLEMENTED entries, [0,1...TXINSTANCESIMPLEMENTED-1],
          //one for ack available Tx Instance (thread).
          //Each Instance State is initially set to NULLINSTANCESTATE,
          //flagging it as available.
          for (i=0; i<TXINSTANCESIMPLEMENTED; i++)
                  TxInstancePool[i].behavior = NULLINSTANCEBEHAVIOR;

          txSchedule.AvailableInstanceCount = TXINSTANCESIMPLEMENTED;

          //On initialization, the Tx scheduler is forced to TXINSTANCESBOUND,
          //indicating that none of the available threads are scheduled.
          //The FG will continually transmit the MAC Idle message.
          txSchedule.CurrentBlockIndex = TXINSTANCESBOUND;
          txSchedule.NextWaitingIndex  = TXINSTANCESBOUND;
          txSchedule.TxLinkState = IDLEWAITINGSCHEDULE;
          // All non-master devices must wait until a MASTER_STATUS_BRDCST is received. If no
          // MASTER_STATUS_BRDCST message is received within 500 ms, then the non-master
          // device should send a DEVICE_MASTER_QUERY message. If the master is present,
          // then it will respond with a MASTER_STATUS_BRDCST message. If a master is not
          // present, then the device shall re-send the DEVICE_MASTER_QUERY message. This
          // will handle the differences in power up times, as well as the case when a device is
          // connected after the rest of the system has already powered up. 5.2.1 XCMP/XNL
          // Development Guide.
          //
          // This code provides for this retry by creating an intance of the DEVICE_MASTER_QUERY
          // message, but not sending it. Should the MASTER_STATUS_BRDCST not be received
          // before the retry timout, the DEVICE_MASTER_QUERY will then be sent.
          masterQuery = reserveTxInstance(DEVICE_MASTER_QUERY_BLOCK_COUNT);
          if (masterQuery == TXINSTANCESBOUND){ //Good practice to test and recover.
                  DontPanic = FALSE;  //We just started, Panic!
                  return;
          }
          for (i=0; i<DEVICE_GENERIC_U16_COUNT; i++)
             TxBufferPool[TxInstancePool[masterQuery].BlockIndex[0]].u16[i]
             = DEVICE_GENERIC_PROTO[i];
          //Insert opcode.
          TxBufferPool[TxInstancePool[masterQuery].BlockIndex[0]].XNL.theXNL_Header.opcode
            = XNL_DEVICE_MASTER_QUERY;
          //fill in checksums.
          sumTxInstance(masterQuery);
          //Schedule in future.
          TxInstancePool[masterQuery].RetryTime += STANDARDTIMEOUT;
          TxInstancePool[masterQuery].behavior = TXXNLCTRLPROTO;
}
```

Appendix 8 process_XNL

```
void process_XNL (void)
{
  U32 temp;
  U8 SomeInstance;

  garbageCollect();

  theXNL_Ctrlr.isIncomingMessage = FALSE;
  if (theRxCirCtrlr.RxXNL_ProcessWaitingIndex != theRxCirCtrlr.RxXNL_IsFillingMessageIndex) {
     //Align XNL Template with message in circular buffer.
     theRxCirCtrlr.pRxTemplate=(RxTemplate*)(&(theRxCirBuffer.theRxFragment[theRxCirCtrlr.RxXNL_ProcessWaitingIndex]));
     if ( 0 == (((theRxCirCtrlr.pRxTemplate)->theMAC_Header.phy_control) & 0x0F00)){
         theXNL_Ctrlr.isIncomingMessage = TRUE;
     }else{ //This simple implementation throws away multiple fragments.
         depleteAProcessedMessage();
     }
  }

  switch (theXNL_Ctrlr.XNL_State) {
  case XNL_UNCONNECTEDWAITINGSTATUS:
    if   (theXNL_Ctrlr.isIncomingMessage) {
      if ( XNL_MASTER_STATUS_BRDCST ==
          (theRxCirCtrlr.pRxTemplate)->theXNL_Header.opcode ){
            processXNL_MASTER_STATUS_BRDCST();
      }
              depleteAProcessedMessage(); //This state depletes everything.
    }
    break;


  case XNL_UNCONNECTEDWAITINGAUTHKEY:
          if   (theXNL_Ctrlr.isIncomingMessage) {
                 if (XNL_DEVICE_AUTH_KEY_REPLY ==
                         (theRxCirCtrlr.pRxTemplate)->theXNL_Header.opcode ){
                         processXNL_DEVICE_AUTH_KEY_REPLY();
                 }
                 depleteAProcessedMessage(); //It is possible we could receive another XNL_MASTER_STATUS_BRDCST
                                             //here. It is possible the Master address could have changed.
                                             //Strict protocol would deplete any queued DEVICE_AUTH_KEY_REQUEST
                                             //and process the new XNL_MASTER_STATUS_BRDCST. If the Master address
                                             //has not changed, this is just a valid repeat of the same STATUS_BRDCST.
          }
          break;


  case XNL_UNCONNECTEDWAITINGDEVICECONN:
          if   (theXNL_Ctrlr.isIncomingMessage) {
                 if (XNL_DEVICE_CONN_REPLY ==
                         (theRxCirCtrlr.pRxTemplate)->theXNL_Header.opcode ){
                         processXNL_DEVICE_CONN_REPLY();
                 }
              depleteAProcessedMessage(); //It is possible we could receive another XNL_MASTER_STATUS_BRDCST
                                          //here.
                                          //It is possible we could receive another XNL_DEVICE_AUTH_KEY_REPLY
                                          //here, intended for us or someone else (no way to tell), having
                                          //a different Unencrypted Authentication Value. Strict protocol
                                          //should check for this, deplete any outstanding CONN_REQUESTs, and
                                          //process the new AUTH_KEY_REPLY. Since I suspect this will all
                                          //change in the future, I'll do nothing here.
          }
          break;
```

```
  case XNL_CONNECTED:
        if   (theXNL_Ctrlr.isIncomingMessage) {
               switch ((theRxCirCtrlr.pRxTemplate)->theXNL_Header.opcode){
               case XNL_DEVICE_SYSMAP_BRDCST:

                      break;
               case XNL_DATA_MSG:
                      processXNL_DATA_MSG();
                      break;
               case XNL_DATA_MSG_ACK:
                      processXNL_DATA_MSG_ACK();
                      break;
               }//End of switch on XNL Header Opcode
            depleteAProcessedMessage();
        }
        break;


  default:

    break;
  } //End of switch on XNL_State.

  //Need to search for active instances with timeouts.
  SomeInstance = findTxInstance_byTimeout();
  if (SomeInstance != TXINSTANCESBOUND){//Some Instance needs a retry.
        if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){//Scheduling allowed
            //Formally, shouldn't retry something that's stuck in transmitter.
               //Should in general never fail this test.
               if (((TxInstancePool[SomeInstance].behavior) & FGOWNSBEHAVIOR) == 0x00000000){
                      temp = ((TxInstancePool[SomeInstance].behavior) & TXINSTANCERETRYMASK) - 1;
                      if (temp == 0){ //All retries exhausted.
                             //In general case should maybe signal some error.
                             releaseTxInstance(SomeInstance);
                      }else{
                             TxInstancePool[SomeInstance].behavior &= 0x3FFF0000; //Clear owned, sent, old count.
                             TxInstancePool[SomeInstance].behavior |= temp;       //Update new count;
                             TxInstancePool[SomeInstance].behavior |= FGOWNSBEHAVIOR;
                             TxInstancePool[SomeInstance].RetryTime = theRxCirCtrlr.RxLinkCount
                                                                    + STANDARDTIMEOUT;
                             txSchedule.NextWaitingIndex = SomeInstance;
                      }
               }

        }
  }


}//End of process_XNL.
```

```
void depleteAProcessedMessage (void)
{
   theRxCirCtrlr.RxXNL_ProcessWaitingIndex =
              ((theRxCirCtrlr.RxXNL_ProcessWaitingIndex) + 1) & RXCIRBUFFERFRAGWRAP;
}
```

Appendix 9 processXNL_MASTER_STATUS_BRDCST

```
void processXNL_MASTER_STATUS_BRDCST (void)
{
        U8 i;
        U8 theInstance;

        //The XNL_MASTER_STATUS_BRDCST message is sent out by the master device to indicate that the master has been
        //determined and that non-master devices can now connect. The data payload for this
        //message will contain the XNL version as well as the logical device identifier for the
        //master device. The last field in the payload contains a flag that indicates whether or not
        //an XNL_DATA_MSG has been sent out. This will indicate to a connecting device that it
        //has missed messages. The XNL header will contain the master's XNL address. 5.4.1
        theXNL_Ctrlr.XNL_MasterAddress = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.source;
    //Could extract here Minor XNL Protocol Version Number.
    //Could extract here Major XNL Protocol Version Number.
    //Could extract here Master Logical Identifier.
    //Could extract here Message Sent Boolean.
        releaseTxInstance(findTxInstance_byOpCode(XNL_DEVICE_MASTER_QUERY));
        theXNL_Ctrlr.XNL_State = XNL_UNCONNECTEDWAITINGAUTHKEY;

        //This message is sent by all non-master devices in order to get the authentication key to
        //be used when establishing a connection. This message contains no payload data.
        //XCMP/XNL Development Guide 5.4.3
        theInstance = reserveTxInstance(DEVICE_AUTH_KEY_REQUEST_BLOCK_COUNT);
        if (theInstance == TXINSTANCESBOUND){ //Good practice to test and recover.
          DontPanic = FALSE;  //We just started, Panic!
              return;
        }

        for (i=0; i<DEVICE_GENERIC_U16_COUNT; i++)
                TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u16[i]
                = DEVICE_GENERIC_PROTO[i];

        //Insert opcode.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode
               = XNL_DEVICE_AUTH_KEY_REQUEST;

        //Use actual Master address.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination
        = theXNL_Ctrlr.XNL_MasterAddress;

        //fill in checksums.
        sumTxInstance(theInstance);

        //Attempt to schedule transmission.
        //Scheduling *should* be immediate here as we've just started up.
        //Failure to get immediate scheduling here may be concern for Panic.
        if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
               //Immediate transmission allowed
               TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
               TxInstancePool[theInstance].behavior = OWNEDXNLCTRLPRPTO;
               txSchedule.NextWaitingIndex = theInstance;
        }else{
               //Leave RetryTime at current time.
               TxInstancePool[theInstance].behavior = TXXNLCTRLPROTO;
        }

}
```

Appendix 10 **processXNL_DEVICE_AUTH_KEY_REPLY**

```
void processXNL_DEVICE_AUTH_KEY_REPLY(void)
{
    U32 v_vector[2], w_vector[2];
        U8 i, theInstance;

  //The payload for XNL_DEVICE_AUTH_KEY_REPLY message is a temporary XNL address to use during the connection
  //process and an unencrypted 8 byte random number generated by the master. This
  //number should be encrypted by the receiving device and will be used to authenticate
  //the connection request. 5.4.4
  //Temporarily use temporary device address
  theXNL_Ctrlr.XNL_DeviceAddress
    = (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.TemporaryXNLAddress;

  //Get Array of values to be encrypted into an aligned 2X32bits.
  v_vector[0] = ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[0])<<24
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[1])<<16
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[2])<<8
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[3]);
  v_vector[1] = ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[4])<<24
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[5])<<16
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[6])<<8
              | ((theRxCirCtrlr.pRxTemplate)
                        ->theXNL_Payload.ContentDEVICE_AUTH_KEY_REPLY.UnencryptedAuthenticationValue[7]);

  encipher(&v_vector[0], &w_vector[0], &authKey[0]);

  releaseTxInstance(findTxInstance_byOpCode(XNL_DEVICE_AUTH_KEY_REQUEST));
          theXNL_Ctrlr.XNL_State = XNL_UNCONNECTEDWAITINGDEVICECONN;

        //This message is sent by all non-master devices in order to establish a logical
        //connection with the master. If a particular XNL address is desired (for fixed address
        //systems), a preferred address field should be used. Otherwise a value of 0x0000 should
        //be used. For systems that contain both fixed address and dynamic address
        //assignments, the preferred address cannot be guaranteed. The payload for this
        //message also includes a device type value, authentication index, and the encrypted
        //authentication value. XCMP/XNL Development Guide 5.4.5
        theInstance = reserveTxInstance(DEVICE_CONN_REQUEST_BLOCK_COUNT);
                 if (theInstance == TXINSTANCESBOUND){ //Good practice to test and recover.
                   DontPanic = FALSE;  //We just started, Panic!
                       return;
                 }

                for (i=0; i<DEVICE_CONN_REQUEST_U16_COUNT; i++)
                        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u16[i]{
                        = DEVICE_CONN_REQUEST_PROTO[i];
                }
                //Use actual Master address.
                TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination
                     = theXNL_Ctrlr.XNL_MasterAddress;

                //Use Temporary address.
                //There is a possible flaw in the XNL protocol; several Option Cards may
                //power up at about the same time, XNL_DEVICE_AUTH_KEY_REPLY. Thus their Temporary Address
                //will also be the same. So, they will be sending the same message here,
                //and all will receive the same DEVICE_CONN_REPLY. Not real sure what's going
                //to happen with multiple conrol heads, etc. One suspects the Rocket Scientists will
                //eventually figure this out, and demand a transaction ID based on Device Type in
                //the XNL_DEVICE_AUTH_KEY_REQUEST.
                TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.source
                        = theXNL_Ctrlr.XNL_DeviceAddress;
```

```
            //We know encrypted array happens to be aligned to 32-bit boundary.
            TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u32[5]
                                                = w_vector[0];
            TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u32[6]
                                                = w_vector[1];

            //fill in checksums.
         sumTxInstance(theInstance);

         //Presently the protocol enables, but does not require the Device to Authenticate the Master.
         //The present exchange is vulnerable to playback attack anyway, and only one Master
         //can be talking on the physical bus, this is kind of silly. I've included it here as just
         //another opportunity to check that the sequence has sanity.
         encipher(&w_vector[0], &v_vector[0], &authKey[0]);
         //Squirrel away the result for comparison with value returned in DEVICE_CONN_REPLY.
         //This is not transmitted. Only using spare instance memory.
         TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u32[7]
                                                = v_vector[0];
         TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u32[8]
                                                = v_vector[1];

         //Attempt to schedule transmission.
            //Scheduling *should* be immediate here as we've just started up.
            //Failure to get immediate scheduling here may be concern for Panic.
            //In general, a schedule failure can just wait for retry.
            if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
                   //Immediate transmission allowed
                TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
                TxInstancePool[theInstance].behavior = OWNEDXNLCTRLPRPTO;
                   txSchedule.NextWaitingIndex = theInstance;
            }else{
                //Leave RetryTime at current time.
                TxInstancePool[theInstance].behavior = TXXNLCTRLPROTO;
            }

}


const U32 authKey[] = { 0x[redacted], 0x[redacted], 0x[redacted], 0x[redacted] };

void encipher (U32 *const v,
              U32 *const w,
       const U32 *const k)
{
        register U32 y=v[0], z=v[1], sum=0;
        register U32 delta= 0x[redacted];
        register U32 a=k[0], b=k[1], c=k[2], d=k[3];
        register U32 n=32;

        while(n-->0)
        {
        sum += delta;
        y += ((z << 4)+a) ^ (z+sum) ^ ((z >> 5)+b);
        z += ((y << 4)+c) ^ (y+sum) ^ ((y >> 5)+d);
        }

        w[0]=y; w[1]=z;
 }
```

Appendix 11 processXNL_DEVICE_CONN_REPLY

```
void processXNL_DEVICE_CONN_REPLY (void)
{
	U8 i, theInstance;

	//Bool MasterAuthenticate;
	//The reply to the Device Connection Request contains a result code (connection
	//successful or not) and the XNL address and device logical address to use for all future
	//communications. In addition, the reply will contain a unique 8-bit value that should be
	//used as the upper byte of the transaction ID and an 8-byte encrypted value that the
	//device can use to authenticate the master. XCMP/XNL Development Guide 5.4.6

	//Test result code
	if((((theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_CONN_REPLY.Result_Base)
	   & 0x0000FF00) != 0x00000100){
		//Rejected. The device must retry the authentication process at this point by sending out a
		//new AUTH_KEY_REQUEST message. XCMP/XNL Development Guide Section 5.2.3
		  releaseTxInstance(findTxInstance_byOpCode(XNL_DEVICE_CONN_REQUEST));
		  theXNL_Ctrlr.XNL_State = XNL_UNCONNECTEDWAITINGAUTHKEY;
		  theInstance = reserveTxInstance(DEVICE_AUTH_KEY_REQUEST_BLOCK_COUNT);
		  if (theInstance == TXINSTANCESBOUND){ //Good practice to test and recover.
			  DontPanic = FALSE;  //Panic!
			  return;
		  }

		  for (i=0; i<DEVICE_GENERIC_U16_COUNT; i++)
			  TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u16[i]
				 = DEVICE_GENERIC_PROTO[i];

		  //Insert opcode.
		  TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode
			   = XNL_DEVICE_AUTH_KEY_REQUEST;

		  //Use actual Master address.
		  TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination
			  = theXNL_Ctrlr.XNL_MasterAddress;

		  //fill in checksums.
		  sumTxInstance(theInstance);

		  if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
			  //Immediate transmission allowed.
			  TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
			  TxInstancePool[theInstance].behavior = OWNEDXNLCTRLPRPTO;
			  txSchedule.NextWaitingIndex = theInstance;
		  }else{
			  //Leave RetryTime at current time.
			  TxInstancePool[theInstance].behavior = TXXNLCTRLPROTO;
		  }

	}else{ //connection accepted
		//Record Transaction ID Base
	     theXNL_Ctrlr.XNL_TransactionIDBase
	      = (((theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_CONN_REPLY.Result_Base) & 0x000000FF) << 8;
	     //Record Device Logical Address
	     theXNL_Ctrlr.XNL_DeviceLogicalAddress
			       = (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_CONN_REPLY.LogicalAddress;
		//Record permanent device address
	    theXNL_Ctrlr.XNL_DeviceAddress
	     = (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_CONN_REPLY.XNLAddress;
	    //Initialize 3Bit  Rollover.
	    theXNL_Ctrlr.XNL_3BitRollover = 0x0000;

	    ////Optionally, authenticate Master.
	    //theInstance = findTxInstance_byOpCode(XNL_DEVICE_CONN_REQUEST));
	    //MasterAuthenticate = PASS;
	    //For (i=0; i<8; i++) {
	    //  if (
	    //((theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentDEVICE_CONN_REPLY.EncryptedAuthenticationValue[i])
	    //!= (TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].u8[i+20]))
```

```
            //  MasterAuthenticate = FAIL;
            //}

            releaseTxInstance(findTxInstance_byOpCode(XNL_DEVICE_CONN_REQUEST));
            theXNL_Ctrlr.XNL_State = XNL_CONNECTED;
        }
}
```

Appendix 12 reserveTxInstance

```
// This routine attrmpts to reserve a TxInstance
//with the number of requested fragment blocks.
//If successful, it will return an Index
//to the Instance (<TXINSTANCESBOUND). If unsucessful,
//it will return == TXINSTANCESBOUND. The requesting routine
//must check this!
U8 reserveTxInstance (int BlocksNeeded)
{
        U8  i;
        U8  blockIndex;
        U8  instanceIndex;

        //This is really un-necessary, and wastes two bytes of RAM. If the
        //values stored in AvailableInstanceCount or AvailableBlockCount
        //ever drift, there will be a problem. This *may* be useful for
        //automating performance metrics. [I *think* you can remove this test
        //and still work.]
        if ((txSchedule.AvailableInstanceCount < 1)
                | (txSchedule.AvailableBlockCount < BlocksNeeded))
                return TXINSTANCESBOUND;


        for (instanceIndex=0; instanceIndex<TXINSTANCESIMPLEMENTED; instanceIndex++){
                if (TxInstancePool[instanceIndex].behavior == NULLINSTANCEBEHAVIOR){
                        //Code gets here when the available instance is found.
                        TxInstancePool[instanceIndex].RetryTime = theRxCirCtrlr.RxLinkCount;
                        txSchedule.AvailableInstanceCount -= 1;
                        for (i=0; i<MAXPHYBLOCKS; i++){
                                if (i < BlocksNeeded){   //Still looking.
                                        for (blockIndex=0; blockIndex<TXPOOLSIZE; blockIndex++){
                                                if (TxBlockReservation[blockIndex] == TXINSTANCESBOUND){
                                                        TxBlockReservation[blockIndex] = instanceIndex;
                                                        TxInstancePool[instanceIndex].BlockIndex[i] = blockIndex;
                                                        txSchedule.AvailableBlockCount -= 1;
                                                        break;
                                                }
                                        }
                                }else{                  //Mark end of list.
                                        TxInstancePool[instanceIndex].BlockIndex[i] = TXBLOCKBOUND;
                                }
                        }
                        //Code gets here when all the BlocksNeeded are allocated.
                        return instanceIndex;
                }
        }
        //If using AvailableInstanceCount and AvailableBlockCount, code should never
        //get here unless something is very wrong. If these tests are not used,
        //code will get here if allocation fails. Then should call releaseTxInstance.
        // if (instanceIndex < TXINSTANCESBOUND) releaseTxInstance(instanceIndex);
        //Here, something is very wrong:
        DontPanic = FALSE;
        return TXINSTANCESBOUND;
}
```

Appendix 13 sumTxInstance

```
void sumTxInstance (U8 instanceIndex)
{
        U32  fragwithinInstance;
        U32  indextofrag;
        S32  hWordswithinFrag;
        U32  indextohWord;
        U16 sumScratch;

        if (instanceIndex >= TXINSTANCESBOUND) return;

        for (fragwithinInstance=0; fragwithinInstance<MAXPHYBLOCKS; fragwithinInstance++){
                indextofrag = TxInstancePool[instanceIndex].BlockIndex[fragwithinInstance];
                if (indextofrag != TXBLOCKBOUND){
                        sumScratch = 0;
                        hWordswithinFrag = ((TxBufferPool[indextofrag].MAC.theMAC_Header.phy_control) & 0x00FF) - 2;
                        hWordswithinFrag =  (hWordswithinFrag + (hWordswithinFrag & 0x0001)) >> 1; //Round up.
                        indextohWord = 2;
                        while (hWordswithinFrag>0){
                                sumScratch += TxBufferPool[indextofrag].u16[indextohWord];
                                indextohWord     += 1;
                                hWordswithinFrag -= 1;
                        }
                        TxBufferPool[indextofrag].XNL.theMAC_Header.checksumspacesaver = -sumScratch;
                }else{ //We've summed all blocks.
                        break;
                }
        }
}
```

Appendix 14 HelperRoutines

```
U8 findTxInstance_byOpCode(U16 opcode)
{
        U8 i;

        for (i = 0; i < TXINSTANCESIMPLEMENTED; i++){  //Check each Instance.
                if (TxInstancePool[i].behavior > 0) {  //For non-Null behavior,
                                                      //Check for opcode.
                   if (TxBufferPool[TxInstancePool[i].BlockIndex[0]].XNL.theXNL_Header.opcode == opcode) return i;
                }
        }
        return TXINSTANCESBOUND;  //Return none found.
}
```

```
U8 findTxInstance_byTransID(U16 TransID)
{
        U8 i;

        for (i = 0; i < TXINSTANCESIMPLEMENTED; i++){  //Check each Instance.
                if (TxInstancePool[i].behavior > 0) {  //For non-Null behavior,
                                                       //Check for TransID.
                 if (TxBufferPool[TxInstancePool[i].BlockIndex[0]].XNL.theXNL_Header.transactionID == TransID) return i;
                }
        }
        return TXINSTANCESBOUND;  //Return none found.
}
```

```
U8 findTxInstance_byTimeout(void)
{
        U8  i;

        for (i = 0; i < TXINSTANCESIMPLEMENTED; i++){  //Check each Instance.
               if (TxInstancePool[i].behavior > 0) {  //For non-Null behavior,
                       //FG writes to RxLinkCount *should* be atomic. If not, could be FG/BG contention.
                       if (theRxCirCtrlr.RxLinkCount - TxInstancePool[i].RetryTime < 0x7FFFFFFF) return i;
                }
        }
        return TXINSTANCESBOUND;  //Return none found.
}
```

```
void releaseTxInstance(U8 instanceIndex)
{
	U8  i;

	if (instanceIndex >= TXINSTANCESBOUND) return;

	//It is remotely possible we could be asked to release an instance
	//while it is still owned by the transmitter. An elegant solution
	//may be to mark this, and handle the deletion through garbage collection.
	//I'd need to set up the FG/BG semiphores differently for this.
	//Right now, I do not want to take the time to do this. Since this
	//really should never happen, and since time delays in BG really should not
	//cause any problems, I'm just using an ugly wait loop here.
	while (((TxInstancePool[instanceIndex].behavior) & FGOWNSBEHAVIOR) == FGOWNSBEHAVIOR);

	for (i=0; i<MAXPHYBLOCKS; i++){
		if (TxInstancePool[instanceIndex].BlockIndex[i] != TXBLOCKBOUND){
			TxBlockReservation[TxInstancePool[instanceIndex].BlockIndex[i]]
								= TXINSTANCESBOUND;
			TxInstancePool[instanceIndex].BlockIndex[i] = TXBLOCKBOUND;
			txSchedule.AvailableBlockCount += 1;
		}else{  //We've removed all allocations.
			break;
		}
	}
	TxInstancePool[instanceIndex].behavior = NULLINSTANCEBEHAVIOR;
	txSchedule.AvailableInstanceCount += 1;
}




void garbageCollect (void)
{
	U8  i;

	for (i = 0; i < TXINSTANCESIMPLEMENTED; i++){ //Check each Instance.
		if (((TxInstancePool[i].behavior) & FGHANDSHAKEMASK)  == OKTOGARBAGECOLLECT) {  //Check for can delete.
			releaseTxInstance(i);
		}
	}
}
```

Appendix 15 processXNL_DATA_MSG

```
void processXNL_DATA_MSG (void)
{
        U16 DestinationAddress;
        U16 XCMPopcode;
        U16 temp;

        DestinationAddress = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.destination;
        if ((0x000000 == DestinationAddress)
                || (theXNL_Ctrlr.XNL_DeviceAddress == DestinationAddress)){
         //The message is for me.
                if (scheduleXNL_ACK()){ //Try to schedule ACK.
                //If cannot schedule ACK, just leave without processing message;
                //XNL will retry again, and hopefully our Tx resources will then be free.
                //If ACK has been scheduled. It most likely is already      owned
                //by the transmitter, but possibly is waiting in Queue with immediate
            //timeout.
             XCMPopcode = (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentXNL_DATA_MSG.XCNPopcode;
            switch (XCMPopcode){
            case XCMP_DEVINITSTS:
               if ((theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentXNL_DATA_MSG.u8[4] == DIC){
                       bunchofrandomstatusflags |= DIC;  //Need do nothing else.
               }else{
                       bunchofrandomstatusflags  &= 0xFFFFFFFC; //Device Init no longer Complete.
                       sendDEVINITSTS();
               }
               break;
            case XCMP_DEVMGMTBCST:
               temp  = (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentXNL_DATA_MSG.u8[1] << 8;
               temp |= (theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentXNL_DATA_MSG.u8[2];
               if (temp == theXNL_Ctrlr.XNL_DeviceLogicalAddress){
                       if ((theRxCirCtrlr.pRxTemplate)->theXNL_Payload.ContentXNL_DATA_MSG.u8[0] == 0x01){
                               //Enable Option Board
                               bunchofrandomstatusflags |= 0x00000002;
                       }else{
                               //Disable Option Board.
                               bunchofrandomstatusflags &= 0xFFFFFFFD;
                       }
               }
               break;
            default:
               if ((XCMPopcode & XCMP_MTMask) == XCMP_requestMT){
                  sendOpcode_Not_Supported(XCMPopcode);
               }
               break;
            }

               }
        }
}
```

Appendix 16 scheduleXNL_ACK

```
Bool scheduleXNL_ACK (void)
{
        U8 theInstance;

        theInstance = reserveTxInstance(XNL_DATA_MSG_ACK_BLOCK_COUNT);
        if (theInstance == TXINSTANCESBOUND){ //Are we able to ACK?
                        return FALSE;
                }
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theMAC_Header.phy_control = 0x400E;

        //Turn around Flags.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.flags
                 = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.flags;

        //Insert opcode.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode
                         = XNL_DATA_MSG_ACK;

       //ACK Destination Address is Source of XNL_Message.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination
                  = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.source;

        //ACK Source Address is my address.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.source
              = theXNL_Ctrlr.XNL_DeviceAddress;

        //Turn around Transaction ID.
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.transactionID
              = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.transactionID;

        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.payloadLength
          = 0;

        //fill in checksums.
        sumTxInstance(theInstance);

        if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
                //Immediate transmission allowed.
                TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
               TxInstancePool[theInstance].behavior = OWNEDXCMPACKPROTO;
                txSchedule.NextWaitingIndex = theInstance;
        }else{
                //Leave RetryTime at current time.
                TxInstancePool[theInstance].behavior = TXXCMPACKPROTO;
        }


        return TRUE;
}
```

Appendix 17 **sendOpcode_Not_Supported**

```
void sendOpcode_Not_Supported(U16 XCMPopcode)
{
        U8 theInstance;

        theInstance = reserveTxInstance(DEVINITSTS_BLOCK_COUNT);
        if (theInstance == TXINSTANCESBOUND){ //Are we able to schedule?
                return;
        }

        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
                                                                .XNL
                                                                .theXNLpayload
                                                                .ContentXNL_DATA_MSG
                                                                .XCNPopcode
                                                             = XCMPopcode | 0x8000;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
                                                                .XNL
                                                                .theXNLpayload
                                                                .ContentXNL_DATA_MSG
                                                                .u8[0]
                                                             = XCMPRESULT_NOTSUPPORTED;

        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theMAC_Header.phy_control = 0x4011;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode = XNL_DATA_MSG;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.flags = 0x0100 | newFlag();
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination =
                                                (theRxCirCtrlr.pRxTemplate)->theXNL_Header.source;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.source =
                                                                theXNL_Ctrlr.XNL_DeviceAddress;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.transactionID =
                                         (theRxCirCtrlr.pRxTemplate)->theXNL_Header.transactionID;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.payloadLength = 0x0003;
        sumTxInstance(theInstance);

        if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
                        //Immediate transmission allowed.
                        TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
                        TxInstancePool[theInstance].behavior = OWNEDXCNPMSGPROTO;
                        txSchedule.NextWaitingIndex = theInstance;
        }else{
                        //Leave RetryTime at current time.
                        TxInstancePool[theInstance].behavior = TXXCMPMSGPROTO;
        }
}
```

Appendix 18 sendDEVINITSTS

```
void sendDEVINITSTS (void)
{
	U8 theInstance;
	U8 i;

	theInstance = reserveTxInstance(DEVINITSTS_BLOCK_COUNT);
	if (theInstance == TXINSTANCESBOUND){ //Are we able to schedule?
		return;
	}
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
						.XNL
						.theXNLpayload
						.ContentXNL_DATA_MSG
						.XCNPopcode = XCMP_DEVINITSTS;
	for (i=0; i<DEVINITSTS_BYTE_COUNT; i++){
		TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
						.XNL
						.theXNLpayload
						.ContentXNL_DATA_MSG
						.u8[i] = DEVINITSTSPROTO[i];
	}
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theMAC_Header.phy_control = 0x4019;
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode = XNL_DATA_MSG;
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.flags = 0x0100 | newFlag();
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination = 0x0000;
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.source =
							theXNL_Ctrlr.XNL_DeviceAddress;
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.transactionID = newTransID();
	TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.payloadLength = 0x000B;
	sumTxInstance(theInstance);

	if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
			//Immediate transmission allowed.
			TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
			TxInstancePool[theInstance].behavior = OWNEDXCNPMSGPROTO;
			txSchedule.NextWaitingIndex = theInstance;
	}else{
			//Leave RetryTime at current time.
			TxInstancePool[theInstance].behavior = TXXCMPMSGPROTO;
	}
}
```

Appendix 19 **processXNL_DATA_MSG_ACK**

```
void processXNL_DATA_MSG_ACK(void)
{
	U16 DestinationAddress;
	U16 TransactionID;

	DestinationAddress = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.destination;
	if ((theXNL_Ctrlr.XNL_DeviceAddress) == DestinationAddress){
		//The ack is for me.
		TransactionID = (theRxCirCtrlr.pRxTemplate)->theXNL_Header.transactionID;
		//if (instanceIndex >= TXINSTANCESBOUND) return;handled in releaseTxInstance.
		releaseTxInstance(findTxInstance_byTransID(TransactionID));
	}
}
```

Appendix 20 **newFlag and newTransID**

```
U16 newFlag(void)
{
	U16 flag;
	flag = (theXNL_Ctrlr.XNL_3BitRollover + 1) & 0x0007;
	theXNL_Ctrlr.XNL_3BitRollover = flag;
	return flag;
}

U16 newTransID(void)
{
	U16 TransID;
	TransID = ((theXNL_Ctrlr.XNL_TransactionIDBase & 0x00FF) + 1) & 0x00FF;
	TransID |= theXNL_Ctrlr.XNL_TransactionIDBase & 0xFF00;
	theXNL_Ctrlr.XNL_TransactionIDBase = TransID;
	return TransID;
}
```

Appendix 21 **pdca_int_handler**

```
__attribute__((__interrupt__))
static void pdca_int_handler(void)
{
        intStartCount = Get_system_register(AVR32_COUNT);
        //TxBuffer and RxBuffer terminates the Option Card SSC Phlysical Layer.
        BufferIndex ^= 0x01; //Toggle Index.
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->marr = (U32)(&TxBuffer[BufferIndex].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCTX_EXAMPLE])->tcrr = 3;  //Three words xfered each DMA.


        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->marr = (U32)(&RxBuffer[BufferIndex].theXNL_Channel.word);
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->tcrr = 3;  //Three words xfered each DMA.
        (&AVR32_PDCA.channel[PDCA_CHANNEL_SSCRX_EXAMPLE])->isr;

        theRxCirCtrlr.RxLinkCount += 1;

        XNL_PhyRx(RxBuffer[BufferIndex].theXNL_Channel.word);
        TxBuffer[BufferIndex].theXNL_Channel.word = XNL_PhyTx();
        RxPhyMedia();


    //Dummy code to Idle Tx Media.
    TxBuffer[BufferIndex].thePayload_Channel.word[0] = PAYLOADIDLE0;
    TxBuffer[BufferIndex].thePayload_Channel.word[1] = PAYLOADIDLE1;
    intDuration = Get_system_register(AVR32_COUNT) - intStartCount;

}//End of pdca_int_handler.
```

Appendix 22 **XNL_PhyRx**

```
void XNL_PhyRx(U32 theXNLRxWord)
{
   //This is the code for parsing the incoming physical message.
   switch (theRxCirCtrlr.theRxLink_State){

   // Note that all segments must align with a 32-bit boundary and beginning of
   // each XCMP/XNL payload frame must start on slot 3 Thus, segments of odd length
   // must append a 0x0000 at the end (slot 4) to ensure alignment. [9.1.3]
   case WAITINGFORHEADER:                     //Waiting for something. Most frequent visit.
      if (0xABCD5A5A == theXNLRxWord) break;                   //Ignore Idles.
      if (0xABCD0000 != (theXNLRxWord  & 0xFFFF0000)) break;    //Skip until Header.
      if (0x00004000 != (theXNLRxWord  & 0x0000F000)) break;    //Skip non-XCMPXNL_DATA.
      theRxCirCtrlr.RxLink_Expected = (theXNLRxWord & 0x000000FF) - 2; //Length excluding CSUM.
      if (theRxCirCtrlr.RxLink_Expected <= 0) break;           //Discard degenerate message.
      //Do not need to check for buffer wrap here. Should point to clean buffer.
      theRxCirBuffer.CirBufferElement16[(theRxCirCtrlr.RxXNL_IsFillingNextU16)++] = theXNLRxWord;
      theRxCirBuffer.CirBufferElement16[(theRxCirCtrlr.RxXNL_IsFillingNextU16)++]
                     = (theRxCirCtrlr.RxLinkCount) & 0x0000FFFF; //Time stamp.

      //This switch tests the fragment type, and adjusts receiver state accordingly.
      switch (theXNLRxWord & 0x00000F00) {  //Check frag type.
      case 0x00000000:                      //Only Fragment.
      case 0x00000100:                      //First of Multifragment.
      case 0x00000200:                      //Continuing Multifragment.
      case 0x00000300:                      //Last Multifragment.
           theRxCirCtrlr.theRxLink_State = WAITINGCSUM;
           break;
      default:                              //No other values allowed.
           reset_IsFillingNextU16();
           break;
      }
      break;


  case READINGFRAGMENT:
      theRxCirBuffer.CirBufferElement16[(theRxCirCtrlr.RxXNL_IsFillingNextU16)++] = (theXNLRxWord & 0xFFFF0000) >> 16;
      theRxCirCtrlr.RxLink_CSUM += (theXNLRxWord & 0xFFFF0000) >> 16;
      theRxCirCtrlr.RxLink_Expected -= 2;
      if (theRxCirCtrlr.RxLink_Expected <= 0) {
          //All read in.
          //Terminator should be in 2nd hWord.
          //Shaoqun says useful bits not used. The packet will always end with $00BA. [9.1.2.8]
          if ( (0x000000BA == (theXNLRxWord  & 0x0000FFFF))
               && (theRxCirCtrlr.RxLink_CSUM == 0)  )  {
                   post_message();
          }
          reset_IsFillingNextU16();
          theRxCirCtrlr.theRxLink_State = WAITINGFORHEADER;
          break;
      }

      //Have not broken. 2nd hWord contains payload.
      theRxCirBuffer.CirBufferElement16[(theRxCirCtrlr.RxXNL_IsFillingNextU16)++] = (theXNLRxWord & 0x0000FFFF);
      theRxCirCtrlr.RxLink_CSUM  += (theXNLRxWord & 0x0000FFFF);
      theRxCirCtrlr.RxLink_Expected -= 2;
      if (theRxCirCtrlr.RxLink_Expected <= 0) {
          //All read in. Next Word should be 0x00BA0000.
          theRxCirCtrlr.theRxLink_State = WAITINGLASTTERM;
      } //else, next Word contains more payload.
      break;
```

```
 case WAITINGCSUM:   //Gets here on CSUM. Expect at least one hWord payload. Gets here once on every fragment.
     theRxCirCtrlr.RxLink_CSUM  = (theXNLRxWord & 0xFFFF0000) >> 16;  //Stores CSUM
     theRxCirCtrlr.RxLink_CSUM += (theXNLRxWord & 0x0000FFFF);        //sums in first hWord
     theRxCirBuffer.CirBufferElement16[(theRxCirCtrlr.RxXNL_IsFillingNextU16)++] = (theXNLRxWord & 0x0000FFFF);
     theRxCirCtrlr.RxLink_Expected -= 2;
     if (theRxCirCtrlr.RxLink_Expected > 0) { //Normal case for greater than one byte payloads.
         theRxCirCtrlr.theRxLink_State = READINGFRAGMENT;
     }else{ //Sort of strange, one byte payload. Should not happen, but follow protocol.
            //Expect next word 0x00BA0000.
            //Note that all segments must align with a 32-bit boundary and beginning
            //of each XCMP/XNL payload frame must start on slot 3 Thus, segments
            //of odd length must append a 0x0000 at the end (slot 4) to ensure alignment. [9.1.3]
            theRxCirCtrlr.theRxLink_State = WAITINGLASTTERM;
     }
     break;



case WAITINGLASTTERM:     //Expecting last terminator 0x00BA0000.
     if (  (0x00BA0000 == (theXNLRxWord  & 0x00FF0000)) //Expected found.
         &&  (theRxCirCtrlr.RxLink_CSUM == 0)  ) {       //Good checksum.
              post_message();
     }
     theRxCirCtrlr.theRxLink_State = WAITINGFORHEADER;
     reset_IsFillingNextU16();
     break;


  }//End of theRxLink_State switch.
}
```

Appendix 23 **reset_IsFillingNextU16 and post_message**

```
void reset_IsFillingNextU16(void)
{
        theRxCirCtrlr.RxXNL_IsFillingNextU16 =
                theRxCirCtrlr.RxXNL_IsFillingMessageIndex << 7;
}

void post_message(void)
{
        S16 nextpossiblebufferindex;

        nextpossiblebufferindex = ((theRxCirCtrlr.RxXNL_IsFillingMessageIndex) + 1) & RXCIRBUFFERFRAGWRAP;
        if ( nextpossiblebufferindex
              != theRxCirCtrlr.RxXNL_ProcessWaitingIndex ){ //Test for collision, discard.
                theRxCirCtrlr.RxXNL_IsFillingMessageIndex =
                        nextpossiblebufferindex;
        }
}
```

Appendix 24 **XNL_PhyTx**

```
  U32 XNL_PhyTx(void)
  {
        U32 theReturn;
    //This is the code for handling any outgoing XNL Phy message.
    switch (txSchedule.TxLinkState){
    case IDLEWAITINGSCHEDULE:
       //Test to see if there is anything to transmit.
       if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){ //Nothing new to transmit. Send Idle.
              theReturn = XNL_IDLE;// We're done here.
       }else{ //A new instance has has been scheduled.
              txSchedule.CurrentInstanceIndex = txSchedule.NextWaitingIndex; //Begin handling waiting instance.
              txSchedule.NextWaitingIndex = TXINSTANCESBOUND;                //Allow BG to schedule new instance.
              txSchedule.CurrentBlockIndex = 0;                              //Handle to first fragment. Assume index
                                                                             //to a valid fragment block.
              //txSchedule.Next16TxIndex = 0;    //Points to first hWord in fragment block. Init below.
              theReturn =
TxBufferPool[TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[0]].XNL.theMAC_Header.phy_control;
              txSchedule.BytesRemaining = theReturn & 0x000000FF;
              theReturn |= PHYHEADER32; //Transmit 0xABCD0000 | Type/Length.
              txSchedule.Next16TxIndex = 1;
              txSchedule.TxLinkState = INSTANCETRANSMIT;
              //We're done here. The new transmission has started.
       }
       break;

    case INSTANCETRANSMIT:
       theReturn =
TxBufferPool[TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[txSchedule.CurrentBlockIndex]].u16[txSchedule.Ne
xt16TxIndex] << 16;
       txSchedule.Next16TxIndex += 1;
       txSchedule.BytesRemaining -= 2;
       if (txSchedule.BytesRemaining <= 0){ //Have written all the bytes (including 16-bit pad).
              //Must immediately send 0x00BA in Slot 4.
              theReturn |= PHYTERMRIGHT;
              txSchedule.CurrentBlockIndex += 1;  //Advance to next fragment in this instance.
              if (TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[txSchedule.CurrentBlockIndex]
                            == TXBLOCKBOUND){ //All fragments of this instance have been transmitted.
                     TxInstancePool[txSchedule.CurrentInstanceIndex].behavior &= 0x7FFFFFFF; //Release ownership.
                     TxInstancePool[txSchedule.CurrentInstanceIndex].behavior |= FGHASSENT;  //Flag sent.
                     txSchedule.CurrentInstanceIndex = TXINSTANCESBOUND; //Signal for anyone watching.
                     txSchedule.TxLinkState = IDLEWAITINGSCHEDULE;       //Go back to waiting.
              }else{ //Fragments remain in this instance.
                     //Next interrupt will send Header for next fragment.
                     //txSchedule.CurrentInstanceIndex is unchanged.
                     //txSchedule.CurrentBlockIndex already points to next valid fragment block.
                     //txSchedule.BytesRemaining needs to be initialized.
                     //txSchedule.Next16TxIndex needs to be initialized.
                     txSchedule.TxLinkState = CONTINUINGNEWFRAGMENT;
              }
              break; //This fragment finished.
       }

       //Have not broken. Transmit 2nd hWord.
       theReturn |=
TxBufferPool[TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[txSchedule.CurrentBlockIndex]].u16[txSchedule.Ne
xt16TxIndex];
       txSchedule.Next16TxIndex += 1;
       txSchedule.BytesRemaining -= 2;
       if (txSchedule.BytesRemaining <= 0){ //Have written all the bytes (including 16-bit pad).
              //Must send 0x00BA0000 next interrupt in Slot 3&4.
              //txSchedule.CurrentInstanceIndex is unchanged.
                     //txSchedule.CurrentBlockIndex needs advancing and checking.
                     //txSchedule.BytesRemaining needs to be initialized.
                     //txSchedule.Next16TxIndex needs to be initialized.
              txSchedule.TxLinkState = SENDINGTERMINATOR32;
       }
       break;
```

```
    case SENDINGTERMINATOR32:
        theReturn = PHYTERMLEFT;
        txSchedule.CurrentBlockIndex += 1;  //Advance to next fragment in this instance.
        if (TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[txSchedule.CurrentBlockIndex]
                            == TXBLOCKBOUND){ //All fragments of this instance have been transmitted.
                TxInstancePool[txSchedule.CurrentInstanceIndex].behavior &= 0x7FFFFFFF; //Release ownership.
                TxInstancePool[txSchedule.CurrentInstanceIndex].behavior |= FGHASSENT;  //Flag sent.
            txSchedule.CurrentInstanceIndex = TXINSTANCESBOUND; //Signal for anyone watching.
            txSchedule.TxLinkState = IDLEWAITINGSCHEDULE;   //Go back to waiting.
        }else{ //Fragments remain in this instance.
                    //Next interrupt will send Header for next fragment.
                    //txSchedule.CurrentInstanceIndex is unchanged.
                    //txSchedule.CurrentBlockIndex already points to next valid fragment block.
                    //txSchedule.BytesRemaining needs to be initialized.
                    //txSchedule.Next16TxIndex needs to be initialized.
                    txSchedule.TxLinkState = CONTINUINGNEWFRAGMENT;
        }
        break; //This fragment finished.

    case CONTINUINGNEWFRAGMENT:
        theReturn =
TxBufferPool[TxInstancePool[txSchedule.CurrentInstanceIndex].BlockIndex[txSchedule.CurrentBlockIndex]].MAC.theMAC_Header
.phy_control;
        txSchedule.BytesRemaining = theReturn & 0x00FF;
        theReturn |= PHYHEADER32;     //Transmit 0xABCD0000  | Type/Length.
        txSchedule.Next16TxIndex = 1;
        txSchedule.TxLinkState = INSTANCETRANSMIT;
        break; //We're done here. The continuing fragment has started.
    }//End of TxLinkState Switch
    return theReturn;
  }
```

Appendix 25 Definitions and Structures used by XCMP/XNL

```
//_______________________________________________________________________
//XNL State Machine.

typedef enum {
  XNL_UNCONNECTEDWAITINGSTATUS,
  XNL_UNCONNECTEDWAITINGAUTHKEY,
  XNL_UNCONNECTEDWAITINGDEVICECONN,
  XNL_CONNECTED
} XNL_States;

typedef struct {
       U16        XNL_MasterAddress;
       U16        XNL_DeviceAddress;
       U16        XNL_DeviceLogicalAddress;
       U16        XNL_TransactionIDBase;
       U16        XNL_3BitRollover;
       XNL_States XNL_State;
       Bool       isIncomingMessage;
}XNL_Ctrlr;



//_________________________________________________________________
//XNL/XCMP Definitions.


#define XNL_MASTER_STATUS_BRDCST     0x0002
#define XNL_DEVICE_MASTER_QUERY      0x0003
#define XNL_DEVICE_AUTH_KEY_REQUEST  0x0004
#define XNL_DEVICE_AUTH_KEY_REPLY    0x0005
#define XNL_DEVICE_CONN_REQUEST      0x0006
#define XNL_DEVICE_CONN_REPLY        0x0007
#define XNL_DEVICE_SYSMAP_REQUEST    0x0008
#define XNL_DEVICE_SYSMAP_BRDCST     0x0009
#define XNL_DATA_MSG                 0x000B
#define XNL_DATA_MSG_ACK             0x000C


#define XNL_PROTO_XNL_CTRL           0x0000
#define XNL_PROTO_XCMP               0x0100



#define DEVICE_GENERIC_BLOCK_COUNT 1
#define DEVICE_GENERIC_U16_COUNT 8
#define DEVICE_GENERIC_MACBYTE_COUNT 0xE
static const U16 DEVICE_GENERIC_PROTO[DEVICE_GENERIC_U16_COUNT] =
{
              0x400E,   //XCMPXNL_DATA | 0x0E Bytecount.
              0x0000,   //Cannot precalculate checksum.
              0x0000,   //Must be filled in.
              0x0000,   //XNL control message | Unused XNL Flags.
              0x0000,   //Must substitute Master address.
              0x0000,   //Unasigned device address.
              0x0000,   //No transaction ID required for this message.
              0x0000    //This message contains no payload.
};

#define DEVICE_MASTER_STATUS_BRDCST_MACBYTE_COUNT 21 //extended length
typedef struct {                 //XCMP/XNL Development Guide Section 5.4.1
U16    MinorXNLVersionNumber;
U16    MajorXNLVersionNumber;
U16    MasterLogicalIdentifier;    //Upper byte is device type. Lower byte is device number.
U8     DataMessageSent;
U8     u8;                         //Pad.
U16    u16[3];
} contentMASTER_STATUS_BRDCST;
```

```
//XCMP/XNL Development Guide  5.4.2.
#define DEVICE_MASTER_QUERY_BLOCK_COUNT 1
#define DEVICE_MASTER_QUERY_U16_COUNT 8
typedef struct {                  //XCMP/XNL Development Guide Section 5.4.2
U16   u16[7];                     //This message contains no payload.
} contentDEVICE_MASTER_QUERY;




//XCMP/XNL Development Guide  5.4.3.
#define DEVICE_AUTH_KEY_REQUEST_BLOCK_COUNT 1
#define DEVICE_AUTH_KEY_REQUEST_U16_COUNT 8
typedef struct {                  //XCMP/XNL Development Guide Section 5.4.3
U16   u16[7];                     //This message contains no payload.
} contentDEVICE_AUTH_KEY_REQUEST;




#define DEVICE_AUTH_KEY_REPLY_MACBYTE_COUNT 24 //extended length
typedef struct {                  //XCMP/XNL Development Guide Section 5.4.4
U16   TemporaryXNLAddress;
U8    UnencryptedAuthenticationValue[8];
U16   u16[2];
} contentDEVICE_AUTH_KEY_REPLY;



//XCMP/XNL Development Guide 5.4.5
#define DEVICE_CONN_REQUEST_BLOCK_COUNT 1
#define DEVICE_CONN_REQUEST_U16_COUNT 14
typedef struct {                  //XCMP/XNL Development Guide Section 5.4.5
U16   PreferredXNLAddress;
U8    DeviceType;
U8    AuthenticationIndex;
U8    EncryptedAuthenticationValue[8];
U16   u16;
} contentDEVICE_CONN_REQUEST;

static const U16 DEVICE_CONN_REQUEST_PROTO[DEVICE_CONN_REQUEST_U16_COUNT] =
{
               0x401A,   //XCMPXNL_DATA | 0x1A Bytecount.
               0x0000,   //Cannot precalculate checksum.

               XNL_DEVICE_CONN_REQUEST,   //Opcode.
               0x0000,   //XNL control message | Unused XNL Flags.

               0x0000,   //Must substitute Master address.
               0x0000,   //Must substitute Temporary XNL address.

               0x0000,   //No transaction ID required for this message.
               0x000C,   //This message contains 12 payload bytes.

               0x0000,   //No Preferred XNL Address.
               0x0702,   //XCMP/XNL Development Specification Section 4.5.3.2.1.
                         //Same as in MOTOTRBO™ XCMP/XNL Development Specification?
                         //Array aligned at u32[5]
               0x0000,   //Must substitute value encrypted.
               0x0000,   //Must substitute value encrypted.
               0x0000,   //Must substitute value encrypted.
               0x0000    //Must substitute value encrypted.
};
```

```
#define DEVICE_CONN_REPLY_MACBYTE_COUNT 28 //Extended count
typedef struct {                //XCMP/XNL Development Guide Section 5.4.6
U16   Result_Base;
U16   XNLAddress;
U16   LogicalAddress;          //Upper byte is device type. Lower byte is device number.
U8    EncryptedAuthenticationValue[8];
} contentDEVICE_CONN_REPLY;


typedef struct {                //XCMP/XNL Development Guide Section 5.4.7
U16   u16[7];                   //This message contains no payload.
} contentDEVICE_SYSMAP_REQUEST;


typedef struct {                //XCMP/XNL Development Guide Section 5.4.8
U16   SizeofSysMaparray;
U8    u8[10];                   //Packed bytes.
U16   u16;
} contentDEVICE_SYSMAP_BRDCST;

#define XCMP_MTMask      0xF000
#define XCMP_requestMT   0x0000
#define XCMP_DEVINITSTS  0xB400   //Device Initialization Status
#define DIC              0x01
#define XCMP_DEVMGMTBCST 0xB428   //Device Management Broadcast
#define XCMP_TONECTRLREQ 0x0409   //Tone Control Request
#define XCMP_TONECTRLREP 0x8409   //Tone Control Reply
#define XCMP_TONECTRLBRDCST 0xB409 //Tone Control Broadcast


typedef struct {                //XCMP/XNL Development Guide Section 5.4.9
U16  XCNPopcode;
U8   u8[12];                    //Packed Bytes.
} contentXNL_DATA_MSG;


//XCMP/XNL Development Guide 5.4.10
#define XNL_DATA_MSG_ACK_BLOCK_COUNT 1
#define XNL_DATA_MSG_ACK_U16_COUNT 8
typedef struct {                //XCMP/XNL Development Guide Section 5.4.10
U16   u16[7];                   //This message contains no payload.
} contentXNL_DATA_MSG_ACK;


typedef union {
  contentMASTER_STATUS_BRDCST     ContentMASTER_STATUS_BRDCST;
  contentDEVICE_MASTER_QUERY      ContentDEVICE_MASTER_QUERY;
  contentDEVICE_AUTH_KEY_REQUEST  ContentDEVICE_AUTH_KEY_REQUEST;
  contentDEVICE_AUTH_KEY_REPLY    ContentDEVICE_AUTH_KEY_REPLY;
  contentDEVICE_CONN_REQUEST      ContentDEVICE_CONN_REQUEST;
  contentDEVICE_CONN_REPLY        ContentDEVICE_CONN_REPLY;
  contentDEVICE_SYSMAP_REQUEST    ContentDEVICE_SYSMAP_REQUEST;
  contentDEVICE_SYSMAP_BRDCST     ContentDEVICE_SYSMAP_BRDCST;
  contentXNL_DATA_MSG             ContentXNL_DATA_MSG;
  contentXNL_DATA_MSG_ACK         ContentXNL_DATA_MSG_ACK;
} XNLpayload;

#define TONECTRLREQ_BLOCK_COUNT 1
#define TONECTRLREQ_BYTE_COUNT 8
static const U8 TONECTRLREQPROTO[TONECTRLREQ_BYTE_COUNT] =
{
               //This test uses 0x000C [PRIORITY_BEEP]
               0x01, 0x00, 0x0C, 0x00, 0x00, 0x00, 0x00, 0x00
};

#define DEVINITSTS_BLOCK_COUNT 1
#define DEVINITSTS_BYTE_COUNT 9
static const U8 DEVINITSTSPROTO[DEVINITSTS_BYTE_COUNT] =
{
               0x00, 0x01, 0x00, 0x05, 0x00, 0x07, 0x00, 0x00, 0x00
};
```

```
#define XCMPRESULT_SUCCESS 0x00
#define XCMPRESULT_FAILURE 0x01
#define XCMPRESULT_INCORRECTMODE 0x02
#define XCMPRESULT_NOTSUPPORTED 0x03
#define XCMPRESULT_INVALPARAM 0x04
#define XCMPRESULT_TOOBIG 0x05
#define XCMPRESULT_SECLOCK 0x06




//________________________________________________
//Structures defining the SSC physical frame.
typedef union {
  U32     word;
  U16 hword[2];
  U8   byte[4];
} Reserved_Channel;

typedef union {
  U32     word;
  U16 hword[2];
  U8   byte[4];
} XNL_Channel;

typedef union {
  U32  word[2];
  U16 hword[4];
  U8   byte[8];
} Payload_Channel;

typedef struct {
  Reserved_Channel theReserved_Channel;
  XNL_Channel           theXNL_Channel;
  Payload_Channel   thePayload_Channel;
 } SSC_Frame;

#define PHYHEADER32      (U32)0xABCD0000
#define PHYTERMRIGHT     (U32)0x000000BA
#define PHYTERMLEFT      (U32)0x00BA0000
#define XNL_IDLE         (U32)0xABCD5A5A
#define PAYLOADIDLE0     (U32)0xABCD5A5A
#define PAYLOADIDLE1     (U32)0x00000000


//_____________________________________________________________________________
//XNL/SCC Rx Media Controller.

#define RXMEDIABUFFERSIZE 5120
  typedef enum {
    WAITINGABAB,
    READINGARRAYDISCRPT,
    READINGMEDIA,
    BGFORCERESET
  } RxMediaStates;  //enums are 32 bits.
typedef struct{
       RxMediaStates RxMediaState;
       S32           RxMedia_IsFillingNext16;
       S32           RxBytesWaiting;
       S32           ArrayDiscLength;
}RxMediaCtrlr;



//_____________________________________________________________________________
```

```
//XNL/SSC Rx Circular Buffer Link Controller.

#define RXCIRBUFFERMAXFRAGS      16
#define RXCIRBUFFERMAX16         2048
#define RXCIRBUFFERFRAGWRAP      0x000F

 typedef enum {
   WAITINGFORHEADER,
   READINGFRAGMENT,
   WAITINGCSUM,
   WAITINGLASTTERM
 } RxLink_State;



typedef struct {
  U16            phy_control;
  U16            checksumspacesaver;
} MAC_Header;

//Option Board ADK Development Guide [9.1.2.4].
//The 0xABCD Header is a constant, so need not be stored.
typedef struct {   //XCMP/XNL Development Guide Section 5.1
  U16            opcode;
  U16            flags;
  U16            destination;
  U16            source;
  U16            transactionID;
  S16            payloadLength;
}XNL_Header;

//Fixed size 256 byte Rx fragment.
typedef struct {
  MAC_Header     theMAC_Header;  // 2  hWords
  XNL_Header     theXNL_Header;  // 6  hWords
  XNLpayload     theXNL_Payload; // 7  hWords (most common messages)
  U16            theRest[113];
} RxTemplate;

typedef union {
  RxTemplate     theRxTemplate;
  U16            RxFragmentElement16[128];
} RxFragment;

typedef union {
  RxFragment     theRxFragment[RXCIRBUFFERMAXFRAGS];
  U16            CirBufferElement16[RXCIRBUFFERMAX16];
} RxCirBuffer;

// "RxCirCtrlr" is the state machine controller for the Rx circular buffer.
typedef struct{
  U32            RxLinkCount;      //Frames received since powerup.
  RxTemplate    *pRxTemplate;
  RxLink_State   theRxLink_State;
  S16            RxLink_Expected;  //Bytes Expected.
  U16            RxLink_CSUM;      //Scratch area frag CSUM.
  S16            RxXNL_IsFillingMessageIndex;
  S16            RxXNL_IsFillingNextU16;
  S16            RxXNL_ProcessWaitingIndex;
} RxCirCtrlr;
```

```
//_________________________________________________________________________
//XNL/SSC Tx buffers.

#define TXPOOLSIZE        16     //16 phy_block's of 256 Bytes = 4096 Bytes
#define TXBLOCKBOUND      TXPOOLSIZE


typedef struct {
         MAC_Header     theMAC_Header;                         //  2 hWords
         XNL_Header     theXNL_Header;                         //  6
         XNLpayload     theXNLpayload;                         //  7
         U16            overflow[113];                         //113
}MACXNL_Template;

typedef struct {
         MAC_Header     theMAC_Header;                         //  2 hWords
         U16            overflow[126];                         //126
}MACRAW_Template;

typedef union {
       MACXNL_Template   XNL;
       MACRAW_Template   MAC;
       U32           u32[64];
       U16          u16[128];
       U8            u8[256];
}phy_block;





//_________________________________________________________________________
//XNL Tx Controller.

#define STANDARDTIMEOUT              4000;     //500mS*8KHz.
#define TXINSTANCESIMPLEMENTED       4
#define TXINSTANCESBOUND             TXINSTANCESIMPLEMENTED
#define NULLINSTANCEBEHAVIOR         0x00000000   //Zero Behavior | Zero Retries.
#define ALLOCATEDINSTANCEBEHAVIOR    0x00010005   //Allocates     | 5 Retries.

//"behavior" word can only be written by BG if FGOWNSBEHAVIOR flag is clear.
//BG must verify that this flag is clear before writing anything to this word.
//In particular, BG cannot deplete this instance if FGOWNSBEHAVIOR is set.
//To schedule an instance for transmission, BG must verify that a transmission
//may be scheduled, NextInstance == TXINSTANCEBOUNT. Then set FGOWNSBEHAVIOR to 1,
//then put instance index into NextInstance.
#define FGHANDSHAKEMASK          0xE0000000
#define FGOWNSBEHAVIOR           0x80000000
#define FGHASSENT                0x40000000
#define CANDEPLETEAFTERSENT      0x20000000
#define OKTOGARBAGECOLLECT       0x60000000 //Un-owned, sent, can deplete.
#define TXINSTANCESTATEMASK      0x0FFF0000
#define TXINSTANCERETRYMASK      0x0000FFFF
#define TXINSTANCESTATE          0x00010000 //Others may be needed

                                            //The retry test decrements count,
                                            //and if equal to Zero, stops retrying.
#define TXXNLCTRLPROTO           0x00010006
#define OWNEDXNLCTRLPRPTO        0x80010005
#define TXXCMPACKPROTO           0x20010002
#define OWNEDXCMPACKPROTO        0xA0010001
#define TXXCMPMSGPROTO           0x00010006
#define OWNEDXCNPMSGPROTO        0x80010005
```

```
//Please note that the maximum transfer unit (MTU) size for a Data Session
//is 1500 bytes. Of the MTU size, 28 bytes are reserved for overhead.
// Therefore, the maximum payload is 1472 bytes. Therefore, requires 6 fragments.
#define MAXPHYBLOCKS      6   //Maximum number of fragments possible in a single Instance.
typedef struct {
       U32           behavior;
       U32           RetryTime;
       S32           BlockIndex[MAXPHYBLOCKS];
} TxInstance;


typedef enum {
  IDLEWAITINGSCHEDULE,
  INSTANCETRANSMIT,
  CONTINUINGNEWFRAGMENT,
  SENDINGTERMINATOR32
} TXLINKSTATES;  //enums are 32 bits.

typedef struct {
       S32 AvailableInstanceCount;      //Used to speed up BG.
       S32 AvailableBlockCount;         //Used to speed up BG.
       U32 TxLinkState;                 //Tx Phy machine State.
       S32 BytesRemaining;              //Phy Bytes remaining in this fragment.
       S32 CurrentInstanceIndex;        //CurrentInstanceIndex, CurrentBlockIndex,
       S32 CurrentBlockIndex;           //and Next16TxIndex are writable only by FG.
       S32 Next16TxIndex;               //CurrentInstanceIndex may be read by BG.
       S32 NextWaitingIndex;            //Upon NextWaitingIndex Tx start by FG,
} TxXNL_schedule;                    //NextWaitingIndex is nulled to TXINSTANCESBOUND.
                                     //This indicates a new instance may be scheduled.
                                     //The BG may then write to NextWaitingIndex.
                                     //[Except for initialization], the BG may only
                                     //Write to NextWaitingIndex when it is == TXINSTANCESBOUND.
                                     //The FG must never write to NextWaitingIndex
                                     //while it == TXINSTANCESBOUND. This is the Tx FG/BG
                                     //semiphore.
```

Appendix 26 **accelerometer_init**

```
void accelerometer_init(void)
{
  U32 TWI_status;
  U32 temp;
  S8 data;

  accstatus.theAccStatus = 0x00;
  accstatus.seconds = 0;
  accstatus.mS_40 = ACC_DECPERSEC;
  accelerometerIndex = 0;
  for (TWI_status = 0; TWI_status < 16; TWI_status++){
        for (temp = 0; temp < 4; temp++){
              accsamples[TWI_status][temp] = 0;
        }
  }

  //Allocate I/O's to TWI.
    //AVR32_TWI_SDA_0_0_PIN  port=0 pin=A mask=0x00000400
    //AVR32_TWI_SCL_0_0_PIN  port=0 pin=9 mask=0x00000200
    //AVR32_TWI_SCL_0_0_FUNCTION AVR32_TWI_SDA_0_0_FUNCTION functions = 0
    AVR32_GPIO.port[0].pmr0c  = 0x000000600;
    AVR32_GPIO.port[0].pmr1c  = 0x000000600;
    AVR32_GPIO.port[0].gperc  = 0x000000600;
  //  Accelerometer Outputs   GPIO Polling Inputs.
    //INERTIAL INT1           port=0 pin=5 mask= 0x00000020
    //INERTIAL INT2           port=0 pin=6 mask= 0x00000040
    AVR32_GPIO.port[0].oderc = 0x00000060;
    AVR32_GPIO.port[0].gpers = 0x00000060;



  //Init TWI.
       AVR32_TWI.cr    =  AVR32_TWI_CR_SWRST_MASK;  //resets TWI!
       AVR32_TWI.sr;                                // [shouldn't hurt]
       AVR32_TWI.cwgr =  0x0000ECEC;                //100KHz using 24Mhz PBA Clock.

  //For Test, read Who_am_I
  //TWI_status = my_readabyte(LIS302DL_REG_WHO_AM_I, &data);

  //Data Rate 100Hz, Power Active, Full Scale 2.3G, All Axis Enabled.
  TWI_status = my_writeabyte(LIS302DL_REG_CTRL1, ACCREADY);
  //Lines Active High, INT2 (temporarily) disabled, INT1 on Data Ready.
  TWI_status = my_writeabyte(LIS302DL_REG_CTRL3, LIS302DL_CTRL3_DATAONLY);


      //Discard dummy sample.
         TWI_status = my_readabyte(LIS302DL_REG_OUT_X, &data);
         TWI_status = my_readabyte(LIS302DL_REG_OUT_Y, &data);
         TWI_status = my_readabyte(LIS302DL_REG_OUT_Z, &data);
}
```

Appendix 27 **my_writeabyte**

```
U32 my_writeabyte(U32 subaddress, U32 datatosend)
{
       U32 TWI_Status = 0;

       AVR32_TWI.cr =   AVR32_TWI_CR_MSEN_MASK | AVR32_TWI_CR_SVDIS_MASK;
       AVR32_TWI.mmr =  ACC_ADDRESS        << AVR32_TWI_MMR_DADR_OFFSET   |
                        ADDR_LGT           << AVR32_TWI_MMR_IADRSZ_OFFSET |
                        0                  << AVR32_TWI_MMR_MREAD_OFFSET;
       AVR32_TWI.iadr =  subaddress;

       AVR32_TWI.thr = datatosend;  //Higher order bits discarded.

       do
       {
         TWI_Status =  AVR32_TWI.sr & 0x00000104;
       }
       while (TWI_Status == 0);
       while ((AVR32_TWI.sr & 0x00000001) == 0x00000000); //Wait for complete.
       return (TWI_Status);
}
```

Appendix 28 my_readabyte

```
U32 my_readabyte (U32 subaddress, S8 *datareceived)
{
       U32 TWI_Status = 0;

       AVR32_TWI.cr   =  AVR32_TWI_CR_MSEN_MASK | AVR32_TWI_CR_SVDIS_MASK;
       AVR32_TWI.mmr =  ACC_ADDRESS        << AVR32_TWI_MMR_DADR_OFFSET   |
                        ADDR_LGT           << AVR32_TWI_MMR_IADRSZ_OFFSET |
                        1                  << AVR32_TWI_MMR_MREAD_OFFSET;
       AVR32_TWI.iadr =  subaddress;

       AVR32_TWI.cr   =  AVR32_TWI_START_MASK | AVR32_TWI_STOP_MASK;

       do
       {
         TWI_Status =  AVR32_TWI.sr & 0x00000102;
       }
       while (TWI_Status == 0);

       if (!(TWI_Status & 0x00000100))
       {
         *datareceived = AVR32_TWI.rhr;
       }
       while ((AVR32_TWI.sr & 0x00000001) == 0x00000000); //Wait for complete.
       return (TWI_Status);
}
```

Appendix 29 **processAccelerometer and processDoubleClick**

```
        //The code reads the accelerometer status register (to clear the
        //hardware interrupt line), and then reads the three x, y, z samples into
        //a circular array. [The array is of dimension four for alignment.]
        //The filter is lowpass FIR for smoothing. The filter should remove "most"
        //of the fast motion variation, so the acceleration vector should be "mostly"
        //due to gravity. With an accelerometer gain of 0.018G/digit, 1G acceleration
        //should read a maximum of about 56 levels for an aligned axis. The filter
        //coefficients are scaled so that the filter output for an aligned axis
        //is a maximum of about 1G = 939523648. The dot product of two 1G aligned
        //vectors is then 205492225. A threshold of 102746112 is about a 60 degree
        //tilt.
void processAccelerometer(void)
{
        U32 TWI_status;

        if ((AVR32_GPIO.port[0].pvr & 0x00000020) != 0){//else, just leave.


        TWI_status = my_readabyte(LIS302DL_REG_OUT_X, &accsamples[accelerometerIndex][0]);
        TWI_status = my_readabyte(LIS302DL_REG_OUT_Y, &accsamples[accelerometerIndex][1]);
        TWI_status = my_readabyte(LIS302DL_REG_OUT_Z, &accsamples[accelerometerIndex][2]);
       //This should clear INERTIAL INT1.


        if (0x00000000 == (accelerometerIndex & 0x00000003)){ //Decimation
                filter(accelerometerIndex);

                accstatus.mS_40 -= 1;
                if (ACC_ISINITIALIZED == (accstatus.theAccStatus & ACC_ISINITIALIZED)){
                        if (wearenottilted()){
                                //Any good angle during interval is a good interval.
                                accstatus.theAccStatus |= ACC_GOODANGLEHIT;
                        }

                        if (0 == accstatus.mS_40){

                                if (ACC_GOODANGLEHIT != (accstatus.theAccStatus & ACC_GOODANGLEHIT)){
                                        //Angle change detected. Reset and let settle detector.
                                         accstatus.theAccStatus = 0x00;
                                        //report angle change.
                                        bunchofrandomstatusflags |= 0x00000040;
                                }else{
                                        accstatus.theAccStatus &= ACC_GOODANGLEMASK;
                                }
                                accstatus.mS_40 = ACC_DECPERSEC;
                                accstatus.seconds += 1;
                        }

                }else{
                        if (0 == accstatus.mS_40){//Then initialization interval complete.
                                accstatus.mS_40 = ACC_DECPERSEC;
                                accstatus.seconds = 0;
                                accstatus.theAccStatus = ACC_INITIALVALUE;
                                oldresults[0] = filterresults[0];
                                oldresults[1] = filterresults[1];
                                oldresults[2] = filterresults[2];
                        }
                }
        }

        accelerometerIndex = (accelerometerIndex + 1) & 0x0000000F;
        }
}

void processDoubleClick(void)
{
        bunchofrandomstatusflags &= 0xFFFFFFDF;
}
```

Appendix 30 **filter and wearenottilted**

```
const S32 filtercoffs[16] =
{
                133756,
                213239,
                435727,
                779334,
                1191587,
                1599781,
                1926767,
                2108413,
                2108413,
                1926767,
                1599781,
                1191587,
                779334,
                435727,
                213239,
                133756
};

S32 filterresults[3];
S32 oldresults[3];
S32 accelerometerIndex;
S8 accsamples[16][4];


void filter(U32 index)
{
        S32 j, k;
        filterresults[0] = 0;
        filterresults[1] = 0;
        filterresults[2] = 0;

        for (j=0;j<16;j++)
        {
                k = (index - j) & 0x0000000F;
                filterresults[0] += (accsamples[k][0])*filtercoffs[j];
                filterresults[1] += (accsamples[k][1])*filtercoffs[j];
                filterresults[2] += (accsamples[k][2])*filtercoffs[j];
        }
}



Bool wearenottilted(void)
{
        S32 dotproduct;

        dotproduct  = (oldresults[0]>>16)*(filterresults[0]>>16);
        dotproduct += (oldresults[1]>>16)*(filterresults[1]>>16);
        dotproduct += (oldresults[2]>>16)*(filterresults[2]>>16);
        if (ACC_TILTTHRESHOLD < dotproduct){
                return TRUE;    //We are not tilted.
        }else{
                return FALSE;   //We are tilted.
        }
}
```

Appendix 31 **sendTONECTRLREQ**

```
Bool sendTONECTRLREQ(void)
{
        U8 theInstance;
        U8 i;

        theInstance = reserveTxInstance(TONECTRLREQ_BLOCK_COUNT);
        if (theInstance == TXINSTANCESBOUND){ //Are we able to schedule?
                        return FALSE;
        }

        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
                                                            .XNL
                                                            .theXNLpayload
                                                            .ContentXNL_DATA_MSG
                                                            .XCNPopcode = XCMP_TONECTRLREQ;
        for (i=0; i<DEVINITSTS_BYTE_COUNT; i++){
             TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]]
                                                               .XNL
                                                               .theXNLpayload
                                                               .ContentXNL_DATA_MSG
                                                               .u8[i] = TONECTRLREQPROTO[i];
        }

        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theMAC_Header.phy_control = 0x4018;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.opcode = XNL_DATA_MSG;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.flags = 0x0100 | newFlag();
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.destination                           =
theXNL_Ctrlr.XNL_MasterAddress;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.source                                =
theXNL_Ctrlr.XNL_DeviceAddress;
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.transactionID = newTransID();
        TxBufferPool[TxInstancePool[theInstance].BlockIndex[0]].XNL.theXNL_Header.payloadLength = 0x000A;
        sumTxInstance(theInstance);

        if (txSchedule.NextWaitingIndex == TXINSTANCESBOUND){
                        //Immediate transmission allowed.
                        TxInstancePool[theInstance].RetryTime += STANDARDTIMEOUT;
                    TxInstancePool[theInstance].behavior = OWNEDXCNPMSGPROTO;
                        txSchedule.NextWaitingIndex = theInstance;
        }else{
                        //Leave RetryTime at current time.
                        TxInstancePool[theInstance].behavior = TXXCMPMSGPROTO;
        }
        return TRUE;
}
```