TAN-200

Application Note – Exar APIs – OS Portability.

August 21, 2007

Revision 1.00

Exar APIs – OS Portability.

1 OVERVIEW

This document explains the procedure to port Exar APIs written for VxWorks, to other Operating Systems such as Linux.

VxWorks, as an operating system is quite different from Linux in the way it shares its memory space between the kernel and the application.

VxWorks has a flat memory model allowing the application to directly access a memory mapped device via pointer de-referencing. Linux, on the other hand, maintains a strict access separation between the application memory space and the kernel level space. While the device can be memory mapped and accessed in the kernel space, it cannot be directly accessed by the application.

Exar provides a sample device driver that can be modified for a given hardware and OS combination to present a gateway for using Exar APIs for VxWorks.

2 Basic S/W Operational Needs for an Exar Device

Any Exar device on a customer board will have a “base address”. The device registers are offset from this base address and if the device is memory mapped, these registers are accessible through pointer de-referencing, at least at the kernel level.

EXAR APIs write single byte data to any offset, read the single byte from any offset and have an interrupt handling routine that is invoked when the device generates an hardware interrupt.

Hence, there are three basic handlers that we need via the kernel.

1. Read a byte at any address offset.
2. Write a data byte at any address offset.
3. Get notification when an interrupt occurs.

2.1 VxWorks Programming Model

1. Read a byte at any address offset – handled via xrGetByte(ulAddr) call. This function implements direct pointer de-referencing.

2. Write a data byte at any address offset – handled via xrSetByte(ulAddr) call. This function, again, implements direct pointer de-referencing.

3. Get notification when an interrupt occurs – handled via intConnect() function in VxWorks that ties the interrupt vector to our interrupt handler directly in the application.

As we discussed above, this is not possible in Linux and similar operating systems.

2.2 Linux (or similar OSs) Programming Model

The user must install a "device driver" at the kernel level. In our case, it is like a "character device driver" with interrupt handling. This document provides a sample code for writing this driver, tested on our VxWorks platform. This is a working implementaion on our setup and will have to be modified to user specific hardware and OS combination. The user must write this kernel level character driver, which is a rather short piece of code. The Exar supplied code can be used as a guideline.

The first step is to install the driver and then install the device that uses this driver as part of the startup procedure.

This must be done on OS startup and prior to the application accessing the device. In the sample application provided with this document, see the first call to *myDeviceDriver()* in *myApp.c*. This function calls *myDriverInstall()* and then m*yDeviceInstall(). MyDevice* specific initialization (*myDeviceInit()*)is made part of *myDeviceInstall()* at the kernel level.

Once the device is installed via OS installation calls, the device appears as a "file" and shows up in the device list.

Thereafter, all access from the application must be made via standard OS device driver calls.

These are:

open()
close()
ioctl()

and,

select() – this is a special routine that lets us communicate up from device to application to handle asynchronous events – eg interrupts.

The other standard calls are *read()*, *write()*, *creat()* and *remove()* which are not required in our implementation. (*read*, *write* are for sequential reading and not convenient in our case.)

open() is the first call that opens the device and provides a file descriptor as a handler.

close(), likewise closes the device. User may not ever make a call to this routine, depending on application.

ioctl() is the most versatile and useful call. Based on the designed control value supplied, user can do a variety of things at the kernel level.

This brings us to the kernel side of things. Corresponding to each of the above calls in the application space, we write in the device driver, corresponding handlers, called:

myDrvOpen(), myDrvClose(), myDrvIoctl().

For each ioctlCode that we define for the user to use in the application, we write the handling code in a switch-case statement within *myDrvIoctl()*.

We define and use:

READ_ONE_BYTE
WRITE_ONE_BYTE
RE_ENABLE_INTERRUPTS

As may be evident, READ/WRITE ONE_BYTE call *getByte()* and *setByte()* which in the kernel space operate via pointer de-referencing.

3 Interrupt Handling

In the kernel space, we can tie an interrupt handler to the device interrupt line. That is quite straightforward. The issue is how to communicate using this handler upwards to the application, asynchronously.

This is where we employ *select()*.

The sample driver is written to support *select()*. In essence, when we get a file descriptor at the application level after the *open()* call, we register it with *select()* in a forked or a new spawned process. The *select()* call will not return till it is "woken up" by the kernel. We wake it up in the kernel space interrupt handler when it gets an interrupt from the device. At the application level, we loop forever on the *select()* call.

This is similar to giving a semaphore in VxWorks and having an application level interrupt handler pend on that semaphore forever.

In the kernel, when we get a device interrupt, we disable further interrupts from the device. At the application level, once we have unpended *select()* and have branched off to the application level interrupt processing, we would like to enable device level interrupts. It is here, we can again use our *ioctl()* function with the RE_ENABLE_INTERRUPTS code. This allows us to go back to the kernel level and enable interrupts on the device.

4 A word about Base Address

The sample code provided with this document, as part of the device initialization routine at the kernel level, discovers the interrupt vector and the base address associated with the a device. The device is in this example is via a PLX9030 PCI Bridge chip. It adds the base address to the offset received in the *ioctl()* calls for read and write. Consequently, the application must set the base address = 0 and just pass the offsets. In all our APIs, the base address is one of the arguments. This should be passed as zero if using the Linux like character driver since the base address will be automatically added by the device driver.

This is the basic architecture which allows us to migrate Exar APIs written for VxWorks to Linux and similar operating systems by replacing the xrSetByte() and xrGetByte() with the character device driver. It may be helpful to go through the sample code provided with this document to understand the detailed implementation.

5 APPENDIX A – REVISION CHANGE HISTORY

5.1 REVISION 1.00

Initial Release – 6/16/2004.

6 APPENDIX B – CODE EXAMPLE

6.1 myApp.h

```
/*

myApp.h
Piyush Gupta
6/1/04

Rev 1.0  - Original code.

My sample Device Driver header file for a character driver
in vxWorks.

*/

/* Application level Interrupt Service Routine */

void myDeviceDriver(void);
void myAppISR(void);


/* Primitive I/O functions, memory mapped device.*/

void xrSetByte(ULONG,UCHAR );

UCHAR xrGetByte(ULONG);

void xrReEnableIntrpt(void);
```

6.2 myApp.c

```
/*

myApp.c
Piyush Gupta
6/1/04

Rev 1.0  - Original code.

My sample Test program for Character Device Driver
in vxWorks.  The idea is to show how to port EXAR APIs
to OSs that don't allow direct memory mapping and all
interface to the device must be via the OS through:

open()
close()
ioctl()  ... for read/write
&
select()  ... for device generated interrupts.

Not needed for Exar devices....

read()
write()

*/

#include "vxworks.h"
#include "myApp.h"
#include "myDeviceDriver.h"

#define LOOP_COUNT 5

/* Global variable. */
int fd;  /* Our one and only file descriptor for this demo.
           One file descriptor needed per device.
           Use array for multiple devices.*/


void myApp()
{

  UCHAR ucData, i;
  int iISRTaskID;

  myDeviceDriver();  /* Installs the device driver and calls it "/myDevice."
                 This must be executed ONLY ONCE at
                 system startup eg: in usrAppInit() */
```

```
fd = open("/myDevice",O_RDWR,0);

/*Since the driver automatically finds, the base address, we just need
  to pass the offset.

  To adapt our regular vxWorks APIs, to this architecture,
  set baseAddr=0 in the main API module.*/

/* See definitions of xrGetByte() and xrSetByte(). These can
   replace the same calls in our vxWorks API.*/

/* Example: Read from device. */

ucData=xrGetByte(0x104);
logMsg("Data at offset 0x104 is 0x%.2x \n",ucData,0,0,0,0,0);


/* Example: Write to device. */

ucData=xrGetByte(0x100);
logMsg("Data at offset 0x100 is 0x%.2x \n",ucData,0,0,0,0,0);

ucData = 0x3;
xrSetByte(0x100, ucData);

ucData=xrGetByte(0x100);
logMsg("Data at offset 0x100 is 0x%.2x \n",ucData,0,0,0,0,0);

/* Example: Interrrupt servicing task - pends on select(). */

/* Task Priority=100 */
iISRTaskID = taskSpawn("myAppISR",100,0,5000,(void*)myAppISR,0,0,0,0,0,0,0,0,0,0);/* */

/* Code below is specific to my demo hardware. It directly manipulates the PCI bridge
chip, to generate interrupts.*/

/* Simulate LOOP_COUNT number of PCI interrupts for test, one every second.*/

logMsg("Send 5 PCI interrupts.\n",0,0,0,0,0,0);

i=0;
while(i<LOOP_COUNT)
{
taskDelay(60);
setByte(0x8010004c,0xC0); /* Simulate interrupt for this test.
                            Command S/W interrupt generation to PCI Chip
                            directly (not going through driver).*/
```

```
 /* PCI Intrpt register at 0x8010004c, bits:
                          Disable Int Line 0 (bit 0), Active Low (Bit 1=0),Status on Bit 2.
                          Keep Disabled, Inl Line 1 (bit 3)
                          Keep Disabled, S/W Intrpt (bit 7).*/
 /* Clear S/W generated interrupt in device ISR automatically.*/
  logMsg("Sent PCI interrupt.\n",0,0,0,0,0,0);
  i++;
  }
  taskDelay(120);  /* Wait for Rx Interrupt demo to end.*/
  close(fd);
  /* End of demo app......... Real apps don't end!*/
}
/*****************************************************************************/

void myAppISR(void)
{
  extern int fd;
  struct fd_set readFds;
  int width=0;
  int i=0;


  while(i<LOOP_COUNT)  /* */
  {
    FD_ZERO(&readFds);
    FD_SET(fd, &readFds);/* Since we have only one file
                        descriptor in this example,
                        no need to loop through an array.*/
    width = fd + 1;

    if(select(width,&readFds, NULL, NULL, NULL)==ERROR) /* See OS specific select() definition.*/
    {
      logMsg("ERROR in select().\n",0,0,0,0,0,0);
      return(ERROR);
    }

    /* Task will pend here if there exists a device associated with fd. */

    /* In this character driver implementation, the select() call unpends
       when the device generates an interrupt. See the device driver code
       for details of implementation. */

    if(FD_ISSET(fd,&readFds))
    {

      logMsg("Received interrupt.\n",0,0,0,0,0,0);

      /* Call the EXAR supplied ISR in the API here.*/
    xrReEnableIntrpt();
   }
   i++;
  }/* END_FOREVER*/
}
/***************************************************************************/

/* Basic I/O functions. These will replace
   the memory mapped version in the EXAR API */

void xrSetByte(ULONG ulAddr,UCHAR ucData)
{

  extern int fd;
  MY_BYTE_IO myData;

  myData.iAddrOffset=ulAddr;
  myData.ucData = ucData;
  ioctl(fd, WRITE_ONE_BYTE,(int)&myData);

  return;
}
/***************************************************************************/

UCHAR xrGetByte(ULONG ulAddr)
{
  extern int fd;
  MY_BYTE_IO myData;

  myData.iAddrOffset=ulAddr;
  ioctl(fd, READ_ONE_BYTE,(int)&myData);

  return myData.ucData;

}


/***************************************************************************/

void xrReEnableIntrpt(void)
{
  extern int fd;
  MY_BYTE_IO myData;
  ioctl(fd, RE_ENABLE_INTRPT,(int)&myData);
  return;
}
/***************************************************************************/
void myDeviceDriver()  /* Call this ONLY ONCE at startup - eg. in usrAppInit() in vxWorks. */
{
  /* Intialize driver and device related variables. */

  int myDriverNumber = 0;
  BOOL isMyDeviceInstalled = FALSE;

  /*************************************************************************/
  /* Step 1. Call Function myDriverInstall() - calls iosDrvInstall() from ioslib
           to add driver to the driver table.

           Pass myDriverNumber (assumed to be 0) to the function by reference.
           Returns valid myDriverNumber in the variable. */

  myDriverInstall(&myDriverNumber);

  /* Step 2. Call Function myDeviceInstall() - calls iosDevAdd() from ioslib
           to add the device using the driver from the driver table.
           Returns with myDevice.isDeviceOpen=TRUE, if successful.*/

  myDeviceInstall("/myDevice",      /* Name of device to create. Name what you like.*/
                myDriverNumber,  /* Driver Number to associate with device.*/
                &isMyDeviceInstalled
               ) ;

  if(isMyDeviceInstalled)
   logMsg("Exar Device Driver installed successfully!\n",0,0,0,0,0,0);
  else
   logMsg("Unable to install Exar Device Driver.\n",0,0,0,0,0,0);

  /* Spawn an interrupt handler task in your application.  See myApp() above.*/


}
```

6.3 myDeviceDriver.h

```
/*

myDeviceDriver.h
Piyush Gupta
6/1/04

Rev 1.0  - Original code.

My sample Device Driver header file for a character driver
in vxWorks.

*/
/* Required vxWorks libraries.*/

#include "stdio.h"                              /* For printf */
#include "iv.h"                                 /* For INUM_TO_IVEC */
#include "intLib.h"                       /* For intConnect() */
#include "ioLib.h"                              /* For driver and device installation routines.*/
#include "iosLib.h"                             /* For driver and device installation routines.*/
#include "arch/ppc/vxPpcLib.h"    /* For PCI related routines. */
#include "drv/pci/pciConfigLib.h" /* For PCI related routines. */

#include "selectLib.h"       /* Support select() for signalling interrupts to application layer.*/

#include "errno.h"

/* Driver related definitions.*/

typedef struct  /* MY_DEVICE */
{
  DEV_HDR myDevHdr;  /* Required.*/
  UCHAR ucIntLine;
  SEL_WAKEUP_LIST selList;
  UINT32 iBaseAddr;
  BOOL isDeviceOpen;
  UINT32 iBridgeAddr;

} MY_DEVICE;

typedef struct  /* MY_BYTE_IO */
{
  UCHAR ucData;
  UINT32 iAddrOffset;
} MY_BYTE_IO;


STATUS myDriverInstall(int*);
```

```
/* Device related definitions.*/

STATUS myDeviceInstall(char* , int, BOOL*);

void myDeviceDelete(MY_DEVICE*);
STATUS myDeviceInit(MY_DEVICE*);
void myDeviceISR(MY_DEVICE*);

/* My character driver primitives. */

int myDrvOpen(DEV_HDR*, char*, int, int);       /* myDrvCreat() not defined. Use myDrvOpen() */
STATUS myDrvClose(int);
int myDrvRead(int,char*,int);   /* Not used here.*/
int myDrvWrite(int,char*,int);  /* Not used here.*/
int myDrvIoctl(int,int,int);    /* Use this function for all I/O*/
/*void myDrvDelete(void);  /*Optional, not defined. In RT apps, you typically would never delete a
driver.*/


/* void myErrorHandler(int);  /* TODO - myErrorHandler()*/

/* Define IOCTL codes. Must not conflict with any other pre-existing codes.*/

/* Note: 1-46 and 0x1003-0x100B are used by WindRiver */

#define READ_ONE_BYTE         0X8000
#define WRITE_ONE_BYTE        0X8001
#define RE_ENABLE_INTRPT      0X8002


/* Primitive I/O functions, memory mapped device at kernel level.*/

void setByte(ULONG,UCHAR );

UCHAR getByte(ULONG);
```

6.4 myDeviceDriver.c

```
/*

myDeviceDriver.c
Piyush Gupta
6/1/04

Rev 1.0  - Original code.

My sample Device Driver source code file for a character driver
in vxWorks. For another OS, user must replace with equivalent
routines / code / function calls.

*/

#include "vxworks.h"
#include "myDeviceDriver.h"
extern int errno;

/*************************************************************************/
/* Function myDriverInstall() - calls iosDrvInstall() from ioslib
to add driver to the driver table.

Pass myDriverNumber (assumed to be 0) to the function by reference.
Returns valid myDriverNumber - save to a global variable in the application. */


STATUS myDriverInstall(int* pMyDriverNumber)
{

  /* If the driver is already installed, just return OK.*/
  if(*pMyDriverNumber>0)
    return(OK);

  /* Else, add the driver to the driver table. */

  if( (*pMyDriverNumber = iosDrvInstall(myDrvOpen, NULL, myDrvOpen, myDrvClose,
                                    myDrvRead, myDrvWrite, myDrvIoctl) )==ERROR)
  {
/*    myErrorHandler(errno);  /* TODO - myErrorHandler()*/
    return(ERROR);
  }

  return(OK);
}


/*************************************************************************/
```

```
/* Function myDeviceInstall() - calls iosDevAdd() from ioslib
to add the device using the driver from the driver table.
*/


STATUS myDeviceInstall(char* myDevName,     /* Name of device to create.*/
                    int myDriverNumber,   /* Driver Number to associate with device.*/
                    BOOL* pIsMyDeviceInstalled
                   )
{

  MY_DEVICE* pMyDevice ; /* pointer to myDevice struct*/


  /* Pre-installation checks. */

  /* Is the driver installed?*/

  if(myDriverNumber < 1)
  {
/*    myErrorHandler(S_ioLib_NO_DRIVER);  /* TODO - myErrorHandler()*/
    return(ERROR);
  }


  /* Is the device already created? */

  if(*pIsMyDeviceInstalled)
  {
/*    myErrorHandler(0xFF);  /* TODO - myErrorHandler()*/
    return(ERROR);
  }


  /* Allocate dynamic memory and zero it out. */

  if((pMyDevice = (MY_DEVICE*) malloc (sizeof (MY_DEVICE))) == NULL)
  {
/*    myErrorHandler(0xFF);  /* TODO - myErrorHandler()*/
    return(ERROR);
  }

  bzero(pMyDevice, sizeof (MY_DEVICE) );

  /* Initialize select wakeup list. */

  selWakeupListInit( &pMyDevice->selList );
  /* Initialize the device. ie link ISR to interrupt line etc.*/

  if(myDeviceInit(pMyDevice)==ERROR)
    return(ERROR);


  /* Add other code pertaining to MY_DEVICE struct. */



  /* Add the device to device list.*/

  if(iosDevAdd( (DEV_HDR*)pMyDevice, myDevName, myDriverNumber) == ERROR)
  {
/*    myErrorHandler(errno);  /* TODO - myErrorHandler()*/
     myDeviceDelete(pMyDevice);
     return(ERROR);
  }

  /* else, mark the device as opened.*/

  *pIsMyDeviceInstalled = TRUE;

  return(OK);

}

/**********************************************************************/
/* Function myDeviceInit() - initializes the device. */

STATUS myDeviceInit(MY_DEVICE* pMyDevice)
{

  /*  In our example, the exar device is at BAR3 of the PCI Bridge Chip.
      The device is in memory mapped mode. The interrupt is through the
      PCI interrupt line.


  /* Note: This routine is highly hardware design dependent.
           Modify per hardware impelementation. */


  UINT32 iPciBus, iPciDevice, iPciFunc, iMemBaseCsr;
  STATUS st;

  /* Step 1: Find the base address */


  if(pciFindDevice(0x10b5, 0x9030, 0, &iPciBus, &iPciDevice,&iPciFunc) == OK )
  {
    logMsg("XR PCI Bridge Chip found .... \n",0,0,0,0,0,0);

    /*Get the base address of DUT at BAR3, store in myDevice struct*/

    pciConfigInLong(iPciBus, iPciDevice, iPciFunc, PCI_CFG_BASE_ADDRESS_3, &iMemBaseCsr);


    /*Since the device is memory mapped, LSB address bit is 1. Make it zero.*/
    pMyDevice->iBaseAddr = iMemBaseCsr & 0xFFFFFFFE;

    logMsg("DUT Base address: 0x%x\n", pMyDevice->iBaseAddr,0,0,0,0,0);

    /*Get the bridge address for enabling / disabling interrupts.*/
    /*This is specific to Exar Eval Board / PPMC8260 hardware issue. */

    pciConfigInLong(iPciBus, iPciDevice, iPciFunc, PCI_CFG_BASE_ADDRESS_1, &iMemBaseCsr);

    /*Since the device is memory mapped, LSB address bit is 1. Make it zero.*/
    pMyDevice->iBridgeAddr = iMemBaseCsr & 0xFFFFFFFE;

    logMsg("DUT Bridge address: 0x%x\n", pMyDevice->iBridgeAddr,0,0,0,0,0);

    /* TODO - add base address of FPGA / CPLD etc in device struct
       and populate data.
       Not needed for this demo. */

  }
  else
  {
/*    myErrorHandler(errno);  /* TODO - myErrorHandler()*/
    return(ERROR);

  }

  /* Step 2: Find and Connect to the interrupt line. */

  pciConfigInByte(iPciBus, iPciDevice, iPciFunc,
                PCI_CFG_DEV_INT_LINE,&(pMyDevice->ucIntLine));

  logMsg("PCI Interrupt line found at: 0x%.2x\n",pMyDevice->ucIntLine,0,0,0,0,0);

  if(intConnect(INUM_TO_IVEC((int)(pMyDevice->ucIntLine)), myDeviceISR, (int)pMyDevice) ==
ERROR)
  {
/*    myErrorHandler(errno);  /* TODO - myErrorHandler()*/
    return(ERROR);

  }
  /* Step 3. Enable the interrupt line. */

  if(intEnable((int) pMyDevice->ucIntLine) == ERROR)
  {
/*    myErrorHandler(errno);  /* TODO - myErrorHandler()*/
    return(ERROR);

  }


 /*Enable Exar Device interrupts from the PCI Bridge at startup.*/

  setByte((pMyDevice->iBridgeAddr + 0x4C), 0x41); /* Enable PCI Intrpt, PLX 9030 device.
                         Enable Int Line 0 (bit 0 = 1), Active Low (Bit 1=0),Status on Bit 2.
                         Keep Disabled, Inl Line 1 (bit 3)
                         Enable PCI Interrupts (bit 6 = 1)
                         Keep Disabled, S/W Intrpt (bit 7).*/

  return(OK);
}

/**********************************************************************/
/* Function myDeviceDelete() - cleanup if failed to create the device. */

void myDeviceDelete(MY_DEVICE* pMyDevice)
{

  /* Delete any members that must be deleted or reset.*/

  pMyDevice->isDeviceOpen = FALSE;

  free(pMyDevice);  /*De-allocate the dynamic memory*/

  return;
}

void myDeviceISR(MY_DEVICE* pMyDevice)
{
 /*Disable Exar Device interrupts from the PCI Bridge.*/

  setByte((pMyDevice->iBridgeAddr + 0x4C), 0x40); /* PCI Intrpt control register.
                         Disable Int Line 0 (bit 0), Active Low (Bit 1=0),Status on Bit 2.
                         Keep Disabled, Inl Line 1 (bit 3)
                         Enable PCI Interrupts (bit 6 = 1)
                         Keep Disabled, S/W Intrpt (bit 7).*/

  *(volatile unsigned long*)(0x30000410)|= 0x00400000;
  /* Write 1 to clear PowerSpan isr0,
    interrupt status bit. Pulls interrupt
    line to 8260 back high. */

  *(volatile unsigned long*)(0xF0000000+0x010C08) |= 0x00004000;/*0xFFFFBFFF;
  /* Write 1 to Clear IRQ1, SIPNR_H, bit 17
    on MPC8260. This bit set High implies
    interrupt waiting to be servioed causing another ISR to be launched. */

/*  logMsg("In Device Level ISR\n",0,0,0,0,0,0);/**/

  selWakeupAll(&pMyDevice->selList, SELREAD);  /* This causes the select() in myApp.c to unpend.*/

/*  logMsg("Leaving Device Level ISR\n",0,0,0,0,0,0);/**/

  return;

}

/******************* CHARACTER DRIVER PRIMITIVES ******************/

/***************** OPEN, CLOSE, READ, WRITE, IOCTL ****************/


int myDrvOpen(DEV_HDR* pDevHdr, char* name, int flags, int mode)
{

  /* Instantiate a pointer to this device.*/

  MY_DEVICE* pMyDevice;

  /* Cast pDevHdr. to it.*/

  pMyDevice = (MY_DEVICE*) pDevHdr;

  /*Open the device, if not already open. Check pDevHdr member openFlag.*/

  if(pMyDevice->isDeviceOpen == TRUE)
  {
/*    myErrorHandler(0xFE);  /* TODO - myErrorHandler()*/
    return(ERROR);

  }

  /* If they passed any kind of name tail, fail.*/

  if(*name != '\0')
  {
/*    myErrorHandler(0xFD);  /* TODO - myErrorHandler()*/
    return(ERROR);
  }

  /*Set the openFlag and return pointer to this device struct.*/

  pMyDevice->isDeviceOpen = TRUE ;

  return((int) pMyDevice);

}

/*-----------------------------------------*/

STATUS myDrvClose(int pDevice)
{

  /* Instantiate a pointer to this device.*/

  MY_DEVICE* pMyDevice;

  /* Cast pDevHdr. to it.*/

  pMyDevice = (MY_DEVICE*) pDevice;

  /*Set the openFlag to FALSE and return.*/

  pMyDevice->isDeviceOpen = FALSE ;

  return( OK);

}

/*-----------------------------------------*/

int myDrvRead(int pDevice, char* pBuf, int nBytes)
{
  /* This routine not required for our device.*/
  return (0);
}

/*-----------------------------------------*/

int myDrvWrite(int pDevice, char* pBuf, int nBytes)
{
  /* This routine not required for our device.*/
  return (0);

}
```

```
/*--------------- This is where all the action is! -------------------------*/

int myDrvIoctl(int pDevice, int ioctlCode, int arg)
{

  /* Typecast as required.*/

  MY_BYTE_IO* pIO = (MY_BYTE_IO*) arg;
  MY_DEVICE* pMyDevice = (MY_DEVICE*) pDevice;

  ULONG ulAddr;
  int status;

  switch(ioctlCode)
  {
  case READ_ONE_BYTE:
    ulAddr = pMyDevice->iBaseAddr + pIO->iAddrOffset;
    pIO->ucData = getByte(ulAddr);
    status=OK;
    break;

  case WRITE_ONE_BYTE:
    ulAddr = pMyDevice->iBaseAddr + pIO->iAddrOffset;
    setByte(ulAddr, pIO->ucData);
    status=OK;
    break;

  case RE_ENABLE_INTRPT:
    /*Enable Exar Device interrupts from the PCI Bridge.*/

    setByte((pMyDevice->iBridgeAddr + 0x4C), 0x41); /* PCI Intrpt control register.
                          Enable Int Line 0 (bit 0), Active Low (Bit 1=0),Status on Bit 2.
                          Keep Disabled, Inl Line 1 (bit 3)
                          Enable PCI Interrupts (bit 6 = 1)
                          Keep Disabled, S/W Intrpt (bit 7).*/
   /* logMsg("In ioctl - reenable interrupt.\n",0,0,0,0,0,0); /*  */
    status=OK;
    break;

    /* The FIOSELECT and FIOUNSELECT commands are supported
     * to provide support for select().
     * For FIOSELECT, add the task to the select wakeup list.*/

          case FIOSELECT:
                    selNodeAdd (&pMyDevice->selList, (SEL_WAKEUP_NODE *) arg);
    /*  logMsg("In ioctl - FIOSELECT\n",0,0,0,0,0,0); /* */
    status=OK;
                    break;
          case FIOUNSELECT:
```

```
                    selNodeDelete (&pMyDevice->selList, (SEL_WAKEUP_NODE *) arg);
   /*  logMsg("In ioctl - FIOUNSELECT\n",0,0,0,0,0,0); /* */
   status=OK;
                    break;

  default:
   status=ERROR;
   break;

  }  /* end-switch*/

  return(status);
}


/* Primitive Memory Mapped I/O - allowable at kernel
   level since our device is memory mapped in our hardware implementation.*/


void setByte(ULONG ulAddr, UCHAR ucByteData)
{
  typedef UCHAR  *puint;
  *(puint)ulAddr=ucByteData;
  return;
}


UCHAR getByte(ULONG ulAddr)
{
  UCHAR ucData;
  typedef UCHAR  *puint;
  ucData = *(puint)ulAddr;
  return ucData;
}


/***************************************************************************
*
* myErrorHandler - Error handling routine.  TODO.
*
* This routine logs an error message to the console whenever an error
* occurs.  It is included more for completeness than for actual utility.
*
* RETURNS: NONE
*/
```