```
////////////////////////////////////////////////////////////////////
// File - fpga_lib.c
//
// Library for accessing the FPGA card.
// Code was generated by DriverWizard v6.01 - http://www.jungo.com.
// The library accesses the hardware via WinDriver functions.
//
// Copyright (c) 2003 Jungo Ltd.  http://www.jungo.com
//
////////////////////////////////////////////////////////////////////

#include "fpga_lib.h"
#include "c:/windriver/include/windrvr_int_thread.h"
#include "c:/windriver/include/windrvr_events.h"
#include "c:/windriver/include/status_strings.h"
#ifndef __KERNEL__
       #include <stdio.h>
#endif

// If an error occurs, this string will be set to contain a relevant error message
CHAR FPGA_ErrorString[1024];

// internal data structures
typedef struct
{
    WD_INTERRUPT Int;
    HANDLE hThread;
    FPGA_INT_HANDLER funcIntHandler;
} FPGA_INT_INTERRUPT;

typedef struct
{
    DWORD index;
    BOOL  fIsMemory;
    BOOL  fActive;
} FPGA_ADDR_DESC;

typedef struct FPGA_STRUCT
{
    HANDLE hWD;
    HANDLE hEvent;
    FPGA_INT_INTERRUPT Int;
    WD_PCI_SLOT pciSlot;
    FPGA_ADDR_DESC addrDesc[FPGA_ITEMS];
    WD_CARD_REGISTER cardReg;
} FPGA_STRUCT;

// internal function used by FPGA_Open()
BOOL FPGA_DetectCardElements(FPGA_HANDLE hFPGA);

// Function: FPGA_CountCards()
//   Count the number of PCI cards that meets the given criteria.
// Parameters:
```

```
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   dwVendorID [in] indicates the vendor ID of the PCI card to search.
//     0 indicates all vendor IDs.
//   dwDeviceID [in] indicates the Device ID of the PCI card to search.
//     0 indicates all device IDs.
// Return Value:
//   the number of PCI cards found.
DWORD FPGA_CountCards (DWORD dwVendorID, DWORD dwDeviceID)
{
    WD_VERSION ver;
    WD_PCI_SCAN_CARDS pciScan;
    HANDLE hWD = INVALID_HANDLE_VALUE;
    DWORD dwStatus;

    FPGA_RegisterWinDriver();

    FPGA_ErrorString[0] = '¥0';
    hWD = WD_Open();

    // Verify that the handle is valid and that the version number is correct
    if (hWD==INVALID_HANDLE_VALUE)
    {
        sprintf(FPGA_ErrorString, "Failed opening WinDriver device¥n");
        return 0;
    }

    BZERO(ver);
    WD_Version(hWD,&ver);
    if (ver.dwVer<WD_VER)
    {
        sprintf(FPGA_ErrorString, "Incorrect WinDriver version. Expected %d.%02d,
got %d.%02d¥n",
             WD_VER/100, WD_VER, ver.dwVer/100, ver.dwVer);
        WD_Close(hWD);
        return 0;
    }

    BZERO(pciScan);
    pciScan.searchId.dwVendorId = dwVendorID;
    pciScan.searchId.dwDeviceId = dwDeviceID;
    dwStatus = WD_PciScanCards (hWD, &pciScan);
    WD_Close (hWD);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "WD_PciScanCards() failed with status 0x%x - %s
¥n",
            dwStatus, Stat2Str(dwStatus));
    }
    else if (pciScan.dwCards==0)
        sprintf(FPGA_ErrorString, "no cards found¥n");
    return pciScan.dwCards;
}
```

```
// Function: FPGA_Open()
//   Register a PCI card that meets the given criteria to enable working with it.
//   The handle returned from this function is used by most of the functions in
this file.
// Parameters:
//   phFPGA [out] returns the handle to the opened card.
//   dwVendorID [in] indicates the vendor ID of the PCI card to open.
//     0 indicates all vendor IDs.
//   dwDeviceID [in] indicates the device ID of the PCI card to open.
//     0 indicates all device IDs.
//   nCardNum [in] indicates the index number of the PCI card that meets
//     the criteria of the given vendor and device ID.
// Return Value:
//   TRUE if the card was opened successfully. FALSE if there were any errors.
BOOL FPGA_Open (FPGA_HANDLE *phFPGA, DWORD dwVendorID, DWORD dwDeviceID, DWORD
nCardNum)
{
    FPGA_HANDLE hFPGA = (FPGA_HANDLE)malloc(sizeof(FPGA_STRUCT));

    WD_VERSION ver;
    DWORD dwStatus;
    WD_PCI_SCAN_CARDS pciScan;
    WD_PCI_CARD_INFO pciCardInfo;

    *phFPGA = NULL;
    FPGA_ErrorString[0] = '¥0';
    if (!hFPGA)
    {
        sprintf(FPGA_ErrorString, "Failed allocating memory¥n");
        return FALSE;
    }
    BZERO(*hFPGA);

    FPGA_RegisterWinDriver();

    hFPGA->hWD = WD_Open();

    // Verify that the handle is valid and that the version number is correct
    if (hFPGA->hWD==INVALID_HANDLE_VALUE)
    {
        sprintf(FPGA_ErrorString, "Failed opening WinDriver device¥n");
        goto Exit;
    }

    BZERO(ver);
    WD_Version(hFPGA->hWD,&ver);
    if (ver.dwVer<WD_VER)
    {
        sprintf(FPGA_ErrorString, "Incorrect WinDriver version. Expected %d.%02d,
got %d.%02d¥n",
              WD_VER/100, WD_VER, ver.dwVer/100, ver.dwVer);
        goto Exit;
    }
```

```
    BZERO(pciScan);
    pciScan.searchId.dwVendorId = dwVendorID;
    pciScan.searchId.dwDeviceId = dwDeviceID;
    dwStatus = WD_PciScanCards(hFPGA->hWD, &pciScan);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "WD_PciScanCards() failed with status 0x%x - %s
¥n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    if (pciScan.dwCards==0)
    {
        sprintf(FPGA_ErrorString, "Could not find PCI card¥n");
        goto Exit;
    }
    if (pciScan.dwCards<=nCardNum)
    {
        sprintf(FPGA_ErrorString, "Card is out of range of available cards¥n");
        goto Exit;
    }
    // Found at least one card
    BZERO(pciCardInfo);
    pciCardInfo.pciSlot = pciScan.cardSlot[nCardNum];
    dwStatus = WD_PciGetCardInfo(hFPGA->hWD, &pciCardInfo);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "WD_PciGetCardInfo() failed with status 0x%x - %
s¥n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    hFPGA->pciSlot = pciCardInfo.pciSlot;
    hFPGA->cardReg.Card = pciCardInfo.Card;

    hFPGA->cardReg.fCheckLockOnly = FALSE;
    dwStatus = WD_CardRegister(hFPGA->hWD, &hFPGA->cardReg);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "Failed locking device. Status 0x%x - %s¥n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }

    if (!FPGA_DetectCardElements(hFPGA))
    {
        sprintf (FPGA_ErrorString, "Card does not have all items expected for FPGA
¥n");
        goto Exit;
    }

    // FPGA_Open() was successful
```

```
    *phFPGA = hFPGA;
    return TRUE;

Exit:
    // An error occured during the execution of FPGA_Open()
    if (hFPGA->cardReg.hCard)
        WD_CardUnregister(hFPGA->hWD, &hFPGA->cardReg);
    if (hFPGA->hWD!=INVALID_HANDLE_VALUE)
        WD_Close(hFPGA->hWD);
    free (hFPGA);
    return FALSE;
}

// Function: FPGA_Close()
//   Unregister an opened card.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
// Return Value:
//   None.
void FPGA_Close(FPGA_HANDLE hFPGA)
{
    // disable interrupts
    if (FPGA_IntIsEnabled(hFPGA))
        FPGA_IntDisable(hFPGA);

    // unregister card
    if (hFPGA->cardReg.hCard)
        WD_CardUnregister(hFPGA->hWD, &hFPGA->cardReg);

    // close WinDriver
    WD_Close(hFPGA->hWD);

    free (hFPGA);
}

// Function: FPGA_RegisterWinDriver()
//   Enter a license string into WinDriver module.
// Parameters:
//   None.
// Return Value:
//   None.
void FPGA_RegisterWinDriver()
{
    HANDLE hWD;
    WD_LICENSE lic;

    hWD = WD_Open();
    if (hWD!=INVALID_HANDLE_VALUE)
    {
         // TO DO - When you get your registration string enter it in the
following line.
        //         To check the current license, run 'DriverWizard', and choose
'File / Register',
```

```
        //          leave the license string empty, and click 'Check License'.
        //          In the text box will appear the current valid licenses.
        strcpy(lic.cLicense, "12345abcde12345.abcde");
        WD_License(hWD, &lic);
        WD_Close(hWD);
    }
}

// Function: FPGA_WritePCIReg()
//   Write a DWORD to the PCI configuration space.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   dwReg [in] offset of the register in the PCI configuration space.
//     dwData [in] the value to write into the register.
// Return Value:
//   None.
void FPGA_WritePCIReg(FPGA_HANDLE hFPGA, DWORD dwReg, DWORD dwData)
{
    WD_PCI_CONFIG_DUMP pciCnf;

    BZERO(pciCnf);
    pciCnf.pciSlot = hFPGA->pciSlot;
    pciCnf.pBuffer = &dwData;
    pciCnf.dwOffset = dwReg;
    pciCnf.dwBytes = 4;
    pciCnf.fIsRead = FALSE;
    WD_PciConfigDump(hFPGA->hWD,&pciCnf);
}

// Function: FPGA_ReadPCIReg()
//   Read a DWORD from the PCI configuration space.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   dwReg [in] offset of the register in the PCI configuration space.
// Return Value:
//   the value of the register.
DWORD FPGA_ReadPCIReg(FPGA_HANDLE hFPGA, DWORD dwReg)
{
    WD_PCI_CONFIG_DUMP pciCnf;
    DWORD dwVal;

    BZERO(pciCnf);
    pciCnf.pciSlot = hFPGA->pciSlot;
    pciCnf.pBuffer = &dwVal;
    pciCnf.dwOffset = dwReg;
    pciCnf.dwBytes = 4;
    pciCnf.fIsRead = TRUE;
    WD_PciConfigDump(hFPGA->hWD,&pciCnf);
    return dwVal;
}

BOOL FPGA_DetectCardElements(FPGA_HANDLE hFPGA)
{
```

```
    DWORD i;
    DWORD ad_sp;

    BZERO(hFPGA->Int);
    BZERO(hFPGA->addrDesc);

    for (i=0; i<hFPGA->cardReg.Card.dwItems; i++)
    {
        WD_ITEMS *pItem = &hFPGA->cardReg.Card.Item[i];

        switch (pItem->item)
        {
        case ITEM_MEMORY:
            ad_sp = pItem->I.Mem.dwBar;
            hFPGA->addrDesc[ad_sp].fIsMemory = TRUE;
            hFPGA->addrDesc[ad_sp].fActive = TRUE;
            hFPGA->addrDesc[ad_sp].index = i;
            break;
        case ITEM_IO:
            ad_sp = pItem->I.IO.dwBar;
            hFPGA->addrDesc[ad_sp].fIsMemory = FALSE;
            hFPGA->addrDesc[ad_sp].fActive = TRUE;
            hFPGA->addrDesc[ad_sp].index = i;
            break;
        case ITEM_INTERRUPT:
            if (hFPGA->Int.Int.hInterrupt) return FALSE;
            hFPGA->Int.Int.hInterrupt = pItem->I.Int.hInterrupt;
            break;
        }
    }

    // Verify that all the necessary items were found
    // Verify that an interrupt was found
    if (!hFPGA->Int.Int.hInterrupt)
        return FALSE;

    // Verify that at least one memory/IO space was found
    for (i = 0; i<FPGA_ITEMS; i++)
        if (FPGA_IsAddrSpaceActive(hFPGA, (FPGA_ADDR) i)) break;
    if (i==FPGA_ITEMS) return FALSE;

    return TRUE;
}

// Function: FPGA_IsAddrSpaceActive()
//   Check if the given address space is configured and active.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space to check.
// Return Value:
//   TRUE if the address space is active. FALSE otherwise.
BOOL FPGA_IsAddrSpaceActive(FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace)
{
```

```
    return hFPGA->addrDesc[addrSpace].fActive;
}

// Function: FPGA_GetPciSlot()
//   Return the logical location (bus, slot and function) of the PCI card.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   pPciSlot [out] - pointer to a caller allocated space that will hold the
//     logical location of the card
// Return Value:
//   None.
void FPGA_GetPciSlot(FPGA_HANDLE hFPGA, WD_PCI_SLOT *pPciSlot)
{
    memcpy((PVOID)pPciSlot, &(hFPGA->pciSlot), sizeof(WD_PCI_SLOT));
}

// General read/write functions

// Function: FPGA_ReadWriteBlock()
//   Read/Write data from/to the card's memory/IO into/from a given buffer.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
//   fRead [in] direction of operation:
//     TRUE indicates read from the card's memory/IO into the given buffer.
//     FALSE indicates write from the given buffer to the card's memory/IO.
//   buf [in/out] a caller allocated buffer to read/write to/from.
//   dwBytes [in] the number of bytes to read/write. The allocated buffer should
be at least dwBytes long.
//   mode [in] perform the data transfer byte by byte / word by word / dword by
dword.
// Return Value:
//   None.
void FPGA_ReadWriteBlock(FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace,
    DWORD dwOffset, BOOL fRead, PVOID buf, DWORD dwBytes, FPGA_MODE mode)
{
    WD_TRANSFER trans;
    BOOL fMem = hFPGA->addrDesc[addrSpace].fIsMemory;
    // safety check: is the address range active
    if (!FPGA_IsAddrSpaceActive(hFPGA, addrSpace)) return;
    BZERO(trans);
    if (fRead)
    {
        if (mode==FPGA_MODE_BYTE) trans.cmdTrans = fMem ? RM_SBYTE : RP_SBYTE;
        else if (mode==FPGA_MODE_WORD) trans.cmdTrans = fMem ? RM_SWORD :
RP_SWORD;
        else if (mode==FPGA_MODE_DWORD) trans.cmdTrans = fMem ? RM_SDWORD :
RP_SDWORD;
    }
    else
    {
```

```
        if (mode==FPGA_MODE_BYTE) trans.cmdTrans = fMem ? WM_SBYTE : WP_SBYTE;
        else if (mode==FPGA_MODE_WORD) trans.cmdTrans = fMem ? WM_SWORD :
WP_SWORD;
        else if (mode==FPGA_MODE_DWORD) trans.cmdTrans = fMem ? WM_SDWORD :
WP_SDWORD;
    }
    if (fMem)
        trans.dwPort = hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwTransAddr;
    else trans.dwPort = hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.IO.dwAddr;
    trans.dwPort += dwOffset;

    trans.fAutoinc = TRUE;
    trans.dwBytes = dwBytes;
    trans.dwOptions = 0;
    trans.Data.pBuffer = buf;
    WD_Transfer (hFPGA->hWD, &trans);
}

// Function: FPGA_ReadByte()
//   Read a Byte from the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
// Return Value:
//   The value read from the card's memory/IO.
BYTE FPGA_ReadByte (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset)
{
    BYTE data;
    if (hFPGA->addrDesc[addrSpace].fIsMemory)
    {
        PBYTE pData = (PBYTE) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
        data = *pData; // read directly from the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, TRUE, &data, sizeof
(BYTE), FPGA_MODE_BYTE);
    return data;
}

// Function: FPGA_ReadWord()
//   Read a Word from the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
// Return Value:
//   The value read from the card's memory/IO.
WORD FPGA_ReadWord (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset)
```

```
{
    WORD data;
    if (hFPGA->addrDesc[addrSpace].flsMemory)
    {
        PWORD pData = (PWORD) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
        data = *pData; // read directly from the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, TRUE, &data, sizeof
(WORD), FPGA_MODE_WORD);
    return data;
}

// Function: FPGA_ReadDword()
//   Read a Dword from the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
// Return Value:
//   The value read from the card's memory/IO.
DWORD FPGA_ReadDword (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset)
{
    DWORD data;
    if (hFPGA->addrDesc[addrSpace].flsMemory)
    {
        PDWORD pData = (PDWORD) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
        data = *pData; // read directly from the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, TRUE, &data, sizeof
(DWORD), FPGA_MODE_DWORD);
    return data;
}

// Function: FPGA_WriteByte()
//   Write a Byte to the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
//   data [in] the data to write to the card's memory/IO.
// Return Value:
//   None.
void FPGA_WriteByte (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset, BYTE
data)
{
    if (hFPGA->addrDesc[addrSpace].flsMemory)
    {
        PBYTE pData = (PBYTE) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
```

```
        *pData = data; // write directly to the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, FALSE, &data, sizeof
(BYTE), FPGA_MODE_BYTE);
}

// Function: FPGA_WriteWord()
//   Write a Word to the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
//   data [in] the data to write to the card's memory/IO.
// Return Value:
//   None.
void FPGA_WriteWord (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset, WORD
data)
{
    if (hFPGA->addrDesc[addrSpace].flsMemory)
    {
        PWORD pData = (PWORD) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
        *pData = data; // write directly to the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, FALSE, &data, sizeof
(WORD), FPGA_MODE_WORD);
}

// Function: FPGA_WriteDword()
//   Write a Dword to the card's memory/IO.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
//   addrSpace [in] the address space of the card to access.
//   dwOffset [in] offset relative to the beginning of the address space to
access.
//   data [in] the data to write to the card's memory/IO.
// Return Value:
//   None.
void FPGA_WriteDword (FPGA_HANDLE hFPGA, FPGA_ADDR addrSpace, DWORD dwOffset,
DWORD data)
{
    if (hFPGA->addrDesc[addrSpace].flsMemory)
    {
        PDWORD pData = (PDWORD) (hFPGA->cardReg.Card.Item[hFPGA->addrDesc
[addrSpace].index].I.Mem.dwUserDirectAddr + dwOffset);
        *pData = data; // write directly to the memory mapped range
    }
    else FPGA_ReadWriteBlock(hFPGA, addrSpace, dwOffset, FALSE, &data, sizeof
(DWORD), FPGA_MODE_DWORD);
}

// Function: FPGA_IntIsEnabled()
```

```
//   Check if the interrupt is enabled.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
// Return Value:
//   TRUE if the interrupt is enabled. FALSE otherwise.
BOOL FPGA_IntIsEnabled (FPGA_HANDLE hFPGA)
{
    if (!hFPGA->Int.hThread) return FALSE;
    return TRUE;
}

void FPGA_IntHandler (PVOID pData)
{
    FPGA_HANDLE hFPGA = (FPGA_HANDLE) pData;
    FPGA_INT_RESULT intResult;
    intResult.dwCounter = hFPGA->Int.Int.dwCounter;
    intResult.dwLost = hFPGA->Int.Int.dwLost;
    intResult.fStopped = hFPGA->Int.Int.fStopped;
    hFPGA->Int.funcIntHandler(hFPGA, &intResult);
}

// Function: FPGA_IntEnable()
//   Enable the interrupt.
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open.
//   funcIntHandler [in] The call back function to be called upon interrupt.
// Return Value:
//   TRUE if the interrupt was successfully enabled. FALSE otherwise.
BOOL FPGA_IntEnable (FPGA_HANDLE hFPGA, FPGA_INT_HANDLER funcIntHandler)
{
    DWORD dwStatus;

    // Check if interrupt is already enabled
    if (hFPGA->Int.hThread)
        return FALSE;

    // Calls WD_IntEnable() and creates an interrupt handler thread
    hFPGA->Int.funcIntHandler = funcIntHandler;
    dwStatus = InterruptEnable(&hFPGA->Int.hThread, hFPGA->hWD, &hFPGA->Int.Int,
FPGA_IntHandler, (PVOID) hFPGA);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "InterruptEnable() failed with status 0x%x - %s
¥n",
            dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }

    return TRUE;
}

// Function: FPGA_IntDisable()
//   Disable the interrupt.
```

```
// Parameters:
//   hFPGA [in] handle to the card as received from FPGA_Open().
// Return Value:
//   None.
void FPGA_IntDisable (FPGA_HANDLE hFPGA)
{
    if (!hFPGA->Int.hThread) return;

    // this calls WD_IntDisable()
    InterruptDisable(hFPGA->Int.hThread);

    hFPGA->Int.hThread = NULL;
}

// Function: FPGA_RegisterEvent()
//   Register to receive Plug-and-Play and power notification events according to
the given criteria.
// Parameters:
//   hFPGA [in] handle to the PCI card as received from FPGA_Open().
//   dwAction [in] bit-field data that indicated on what type of actions to be
notified.
//   dwVendorID [in] indicates the vendor ID of the PCI card to register events
upon.
//     0 indicates all vendor IDs.
//   dwDeviceID [in] indicates the device ID of the PCI card to register events
upon.
//     0 indicates all device IDs.
//   pciSlot [in] indicates the logical location (bus, lot and function) of the
PCI card
//     to register events upon. 0 indicates all
//   funcHandler [in] the call back function to be called when event is received.
// Return Value:
//   TRUE is the registration was successful. FALSE otherwise.
BOOL FPGA_RegisterEvent(FPGA_HANDLE hFPGA, DWORD dwAction, DWORD dwVendorID, DWORD
dwDeviceID,
    WD_PCI_SLOT pciSlot, EVENT_HANDLER funcHandler)
{
    HANDLE hEvent;
    WD_EVENT Events;    DWORD dwStatus;

    BZERO(Events);
    Events.dwCardType = WD_BUS_PCI;
    Events.u.Pci.pciSlot = pciSlot;
    Events.u.Pci.cardId.dwVendorId = dwVendorID;
    Events.u.Pci.cardId.dwDeviceId = dwDeviceID;
    Events.dwAction = dwAction;
    dwStatus = EventRegister(&hEvent, hFPGA->hWD, &Events, funcHandler, NULL);
    if (dwStatus)
    {
        sprintf(FPGA_ErrorString, "EventRegister() failed with status 0x%x - %s
¥n",
            dwStatus, Stat2Str(dwStatus));
        return FALSE;
```

```
    }
    hFPGA->hEvent = hEvent;
    return TRUE;
}

// Function: FPGA_UnregisterEvent()
//   Unregister events notification.
// Parameters:
//   hFPGA [in] handle to the PCI card as received from FPGA_Open().
// Return Value:
//   None.
void FPGA_UnregisterEvent(FPGA_HANDLE hFPGA)
{
    if (hFPGA->hEvent)
        EventUnregister(hFPGA->hEvent);
}
```