DYNAMIC ENGINEERING
150 DuBois, Suite 3 Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
www.dyneng.com
sales@dyneng.com
Est. 1988

AltFm1

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision H
10-2002-0708
Corresponding Firmware: Altera FM1 VHDL Revision A

AltFm1
WDM Device Driver for the AltFm1 Altera design for the PCI-Altera-LVDS

Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revision A. Revised December 3, 2007.

Table of Contents

Introduction 4

Note 4

Driver Installation 4

Windows 2000 Installation 5

Windows XP Installation 6

Driver Startup 6

IO Controls 9
IOCTL_ALT_FM1_GET_INFO 9
IOCTL_ALT_FM1_SET_LEDS 9
IOCTL_ALT_FM1_GET_STATUS 9
IOCTL_ALT_FM1_SET_CHAN_CONFIG 9
IOCTL_ALT_FM1_GET_CHAN_CONFIG 9
IOCTL_ALT_FM1_SET_TX_CONFIG 10
IOCTL_ALT_FM1_GET_TX_CONFIG 10
IOCTL_ALT_FM1_SET_RX_CONFIG 10
IOCTL_ALT_FM1_GET_RX_CONFIG 10
IOCTL_ALT_FM1_START_TX 10
IOCTL_ALT_FM1_STOP_TX 10
IOCTL_ALT_FM1_START_RX 11
IOCTL_ALT_FM1_STOP_RX 11
IOCTL_ALT_FM1_GET_TX_DATA 11
IOCTL_ALT_FM1_PUT_RX_DATA 11
IOCTL_ALT_FM1_RESET_RX_FIFO 11
IOCTL_ALT_FM1_SET_RX_LEVEL 11
IOCTL_ALT_FM1_LOAD_PLL_DATA 12
IOCTL_ALT_FM1_READ_PLL_DATA 12
IOCTL_ALT_FM1_REGISTER_EVENT 12
IOCTL_ALT_FM1_ENABLE_INTERRUPT 12
IOCTL_ALT_FM1_DISABLE_INTERRUPT 12
IOCTL_ALT_FM1_FORCE_INTERRUPT 13
IOCTL_ALT_FM1_GET_ISR_STATUS 13

WARRANTY AND REPAIR 13

Service Policy 14
Out of Warranty Repairs 14

For Service Contact: 14

Introduction

The AltFm1 driver is a Win32 driver model (WDM) device driver for the AltFm1 Altera design from Dynamic Engineering. This design is for the Altera EP20K400EBC652 FPGA on the PCI-Altera board. The Altera is programmed from the PCI interface with a configuration file resident on the host hard drive. The Altera design ID field is then read and the appropriate driver is loaded. The ID number for this design is 0x81.

The Altera controls 40 LVDS transceivers. There is also a programmable PLL to supply a timing reference for the design. A transmit FIFO and a receive FIFO are connected between the Xilinx and the Altera to buffer data transfers for the I/O channel.

The Altera is treated as a hot-swappable child of the PciAlt parent. This means that different Altera configurations can be loaded at any time without powering down and a new Altera driver will be loaded automatically provided the design ID matches a known value. If the ID is not known, but the Altera loads successfully, a generic Altera driver will be loaded which allows the Rx FIFO reset and almost-full levels, LEDs, PLLs, and interrupts to be specifically controlled, but requires all other accesses to use a structure that contains two unsigned long integer fields: an address offset and a data value field.

A handle to the current Altera driver can be obtained using a CreateFile() call (see below). IO Control calls (IOCTLs) are used to configure the Altera and read status.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the device for each of these calls.

For more detailed information on the hardware implementation, refer to the PCI-Altera user manual (also referred to as the hardware manual).

Driver Installation

NOTE: The PciAlt driver must be installed before any Altera design can be recognized and the appropriate driver loaded!

There are several files provided in each driver package. These files include AltFm1.sys, AlteraDesigns.inf, DDAltFm1.h, AltFm1GUID.h, AFTest.exe, and AFTest source files.

DDAltFm1.h is a C header file that defines the Application Program Interface (API) to the driver. AltFm1GUID.h is a C header file that defines the device interface identifier for the AltFm1 driver. These files are required at compile time by any application that wishes to interface with the AltFm1 driver, but they are not needed for driver installation. AltFm1.sys is the actual driver executable that is loaded into kernel memory to provide

the software/hardware interface to control the Altera-FM1 Altera design. AlteraDesigns.inf is an information file to allow the system to identify the proper driver and where to load it. It also specifies registry information to be entered to facilitate plug-and-play functionality for the Altera-FM1.

AFTest.exe is a sample Win32 console application that makes calls into the AltFm1 driver to test each driver call without actually writing any application code. It is also not required for driver installation.

To run AFTest.exe, open a command prompt console window and type ***AFTest -d0 -?*** to display a list of commands (the AFTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***AFTest -dn -im*** where ***n*** and ***m*** are the device instance number and driver ioctl number respectively. This application is only intended to test the proper functioning of the driver calls and should not be used for normal operation since it will result in diminished performance.

Windows 2000 Installation

Copy AlteraDesigns.inf and AltFm1.sys to a floppy disk, or CD if preferred.

With the PCI-Altera board installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

- Select ***Next***.
- Select ***Search for a suitable driver for my device.***
- Select ***Next***.
- Insert the disk prepared above in the desired drive.
- Select the appropriate drive e.g. ***Floppy disk drives***.
- Select ***Next***.
- The wizard should find and identify the AlteraDesigns.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

Windows XP Installation

Copy AlteraDesigns.inf and AltFm1.sys to a floppy disk, or CD if preferred.

With the PCI-Altera board installed, power-on the PCI host computer and wait for the for the ***Found New Hardware Wizard*** dialogue window to appear.

- Insert the disk prepared above in the desired drive.
- Select ***No when asked to connect to Windows Update***.
- Select ***Next***.
- Select ***Install the software automatically.***
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware.

A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system.

The interface to the device is identified using a globally unique identifier (GUID), which is defined in AltFm1GUID.h.

Below is example code for opening a handle for device devNum.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Device handle
HANDLE                          hAltFm1 = INVALID_HANDLE_VALUE;

// Return status from command
LONG                            status;

// Base configuration value
ULONG                           config;

// Handle to device information structure for the interface to devices
HDEVINFO                        hDeviceInfo;

// The actual symbolic link name to use in the createfile
CHAR                            deviceName[MAX_DEVICE_NAME];

// Size of buffer reguired to get the symbolic link name
DWORD                           requiredSize;

// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA        interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;
```

```
hDeviceInfo = SetupDiGetClassDevs(
                           (LPGUID)&GUID_DEVINTERFACE_ALT_FM1,
                                   NULL,
                                   NULL,
                                   DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                         (LPGUID)&GUID_DEVINTERFACE_ALT_FM1,
                                devNum,
                                &interfaceData))
{
   status = GetLastError();
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", devNum);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
   else
   {
      printf("**Error: couldn't enum device, (%d)\n", status);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Found our device, get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                    NULL,
                                    0,
                                    &requiredSize,
                                    NULL))
{
   if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
   {
      printf("**Error: couldn't get interface detail, (%d)\n",
             GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
```

```
if(pDeviceDetail == NULL)
{
   printf("**Error: couldn't allocate interface detail\n");
   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   exit(-1);
}

pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData, pDeviceDetail,
                                    requiredSize,
                                    NULL,
                                    NULL))
{
   printf("**Error: couldn't get interface detail(2), (%d)\n",
          GetLastError());

   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   free(pDeviceDetail);
   exit(-1);
}

// Save the name
lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

// Open driver
// Create the handle to the device
hAltFm1 = CreateFile(deviceName,
                     GENERIC_READ    | GENERIC_WRITE,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     NULL,
                     NULL);

if(hAltFm1 == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
   exit(-1);
}
```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object which controls a single board. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile(). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used. The IOCTLs defined in this driver are as follows:

IOCTL_ALT_FM1_GET_INFO

Function: Return the Driver Version and PLL device ID.
Input: None
Output: DRIVER_ALT_FM1_DEVICE_INFO structure
Notes: The PLL device ID is dynamically detected when the driver starts up. It can be one of two values: 0x69 or 0x6A.

IOCTL_ALT_FM1_SET_LEDS

Function: Controls the state of the four LEDs – A_Led0-3 on the upper right-hand corner of the board.
Input: Unsigned character
Output: None
Notes: A value of zero turns off all four LEDs. A one in a bit position 0..3 turns on the corresponding LED.

IOCTL_ALT_FM1_GET_STATUS

Function: Returns the FIFO, error and interrupt status.
Input: None
Output: Value of the status register (unsigned long integer)
Notes: See the bit definitions in DDAltFm1.h for information on interpreting this value.

IOCTL_ALT_FM1_SET_CHAN_CONFIG

Function: Sets the FIFO level interrupt enables, I/O clock polarity and the TX to RX FIFO data test enable.
Input: CHAN_CONFIG structure
Output: None
Notes: See DDAltFm1.h for the CHAN_CONFIG structure definition.

IOCTL_ALT_FM1_GET_CHAN_CONFIG

Function: Returns the configuration written in the previous call.
Input: None
Output: CHAN_CONFIG structure
Notes: See DDAltFm1.h for the CHAN_CONFIG structure definition.

IOCTL_ALT_FM1_SET_TX_CONFIG

Function: Sets the TX interrupt enable, start-bit clear enable and the transmit word count.
Input: TX_CONFIG structure
Output: None
Notes: See DDAltFm1.h for the TX_CONFIG structure definition.

IOCTL_ALT_FM1_GET_TX_CONFIG

Function: Returns the configuration written in the previous call.
Input: None
Output: TX_STATE structure
Notes: See DDAltFm1.h for the TX_STATE structure definition.

IOCTL_ALT_FM1_SET_RX_CONFIG

Function: Sets the RX interrupt enable, start-bit clear enable and termination enable.
Input: RX_CONFIG structure
Output: None
Notes: See DDAltFm1.h for the RX_CONFIG structure definition.

IOCTL_ALT_FM1_GET_RX_CONFIG

Function: Returns the configuration written in the previous call.
Input: None
Output: RX_STATE structure
Notes: See DDAltFm1.h for the RX_STATE structure definition.

IOCTL_ALT_FM1_START_TX

Function: Starts the transmitter.
Input: None
Output: None
Notes:

IOCTL_ALT_FM1_STOP_TX

Function: Stops the transmitter.
Input: None
Output: None
Notes:

IOCTL_ALT_FM1_START_RX

Function: Starts the receiver.
Input: None
Output: None
Notes:

IOCTL_ALT_FM1_STOP_RX

Function: Stops the receiver.
Input: None
Output: None
Notes:

IOCTL_ALT_FM1_GET_TX_DATA

Function: Read a Tx data byte.
Input: Unsigned character
Output: Unsigned character
Notes: The number of the transmit FIFO to read from is passed to this command and a byte of data read from the specified channel's FIFO is returned

IOCTL_ALT_FM1_PUT_RX_DATA

Function: Load an Rx data byte.
Input: RX_DATA_LOAD structure
Output: None
Notes: The RX_DATA_LOAD structure has two eight-bit fields: Channel – the number of the single receive FIFO to write to, and Data – the data byte to write.

IOCTL_ALT_FM1_RESET_RX_FIFO

Function: Resets Rx FIFO 0.
Input: None
Output: None
Notes: Resets receive FIFO zero which is the only receive FIFO used in this design.

IOCTL_ALT_FM1_SET_RX_LEVEL

Function: Sets the Rx FIFO 0 almost full level.
Input: Rx almost full value (unsigned short integer).
Output: None
Notes:

IOCTL_ALT_FM1_LOAD_PLL_DATA

Function: Load the internal registers of a PLL.
Input: PLL_LOAD structure
Output: None
Notes: The PLL_LOAD structure has two fields: Channel – the number of the PLL to write to, and Data – an array of 40 bytes containing the data to write. If channel is greater than seven, the first byte of data is written to the S2 bits for the eight PLLs.

IOCTL_ALT_FM1_READ_PLL_DATA

Function: Return the contents of a PLL's internal registers.
Input: Unsigned character
Output: PLL_READ structure
Notes: The channel number of the PLL to write to is passed in to this call and the register data is output in the PLL_READ struct in an array or 40 bytes. If channel is greater than seven, the first byte of the data array will return the value of the S2 bits from the eight PLLs.

IOCTL_ALT_FM1_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_ALT_FM1_ENABLE_INTERRUPT

Function: Enable the master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to interrupts. The master interrupt enable is disabled in the driver interrupt service routine. This command must then be run again to re-enable it.

IOCTL_ALT_FM1_DISABLE_INTERRUPT

Function: Disable the master interrupt.
Input: None
Output: None
Notes: This call is used when interrupt processing is no longer desired.

IOCTL_ALT_FM1_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_ALT_FM1_GET_ISR_STATUS

Function: Returns the FIFO, error and interrupt status.
Input: None
Output: Value of the status register (unsigned long integer)
Notes: See the bit definitions in DDAltFm1.h for information on interpreting this value.

Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering’s products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be “cockpit error” rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer’s making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer’s invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
(831) 457-8891 - Fax (831) 457-4793
support@dyneng.com