DYNAMIC ENGINEERING

150 DuBois, Suite C

Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PMC Parallel IO

Software Manual

Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Revision A
10-1999-0110

PmcParIo
WDF Device Drivers for the
PMC-Parallel-IO Module

Dynamic Engineering

150 DuBois, Suite C

Santa Cruz, CA 95060

(831) 457-8891

FAX: (831) 457-4793

Manual Revision A. Revised July 31, 2017.

Table of Contents

Introduction

The PmcParIo driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The PmcParIo driver package has two parts. The driver is installed into the Windows® OS and the User Application "Userapp" exectutable.

The driver and test are delivered as installed or executable items to be used directly or indirectly by the user. The UserAp code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

UserAp is a stand-alone code set with a simple and powerful menu plus a series of "tests" that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering.

The test software can be ported to your application to provide a running start. The regtest's are simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system.

The PMC-Parallel-IO board features a PLX 9052 Altera FPGA and high current LVTH driver devices. When the PMC-Parallel-IO board is recognized by the PCI bus configuration utility it will start the PmcParIo driver which will create a device object for each board, initialize the hardware, create child devices for the two I/O channels and request loading of the PmcParIo driver. IO Control calls (IOCTLs) are used to configure the board and read status.

Note
This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PMC Parallel IO user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PmcParIoPublic.h, PmcParIo.inf, pmcpario.cat, PmcParIo.sys,and WdfCoInstaller01009.dll.

PmcParIoPublic.h is the C header file that defines the Application Program Interface (API) for the PmcParIo driver. This file is required at compile time by any application that wishes to interface with the drivers, but is not needed for driver installation.

Windows 7 Installation

Copy PmcParIo.inf, pmcpario.cat, PmcParIo.sys, and WdfCoInstaller01009.dll (Win7 version) to a floppy disk, CD or USB memory device as preferred.

With the PMC-Parallel-IO hardware installed, power-on the PCI host computer.
- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an ***Other PCI Bridge Device****.
- Right-click on the ***Other PCI Bridge Device*** and select ***Update Driver Software***.
- Insert the disk or memory device prepared above in the desired drive.
- Select ***Browse my computer for driver software***.
- Select ***Let me pick from a list of device drivers on my computer***.
- Select ***Next***.
- Select ***Have Disk*** and enter the path to the device prepared above.
- Select ***Next***.
- Select ***Close*** to close the update window.

The system should now display the PmcParIo PCI adapter in the Device Manager.

***** If the ***Other PCI Bridge Device*** is not displayed, click on the ***Scan for hardware changes*** icon on the tool-bar.

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 globally unique identifiers (GUID), which are defined in PmcParIoPublic.h. See main.c in the PmcParIoUserApp project for an example of how to acquire a handle to the device.

The main file provided is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without using user interaction. For multiple user systems it is suggested that the board number is associated with a switch setting so the calls can be associated with a particular board from a physical point of view.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE       hDevice,          // Handle opened with CreateFile()
  DWORD        dwIoControlCode,  // Control code defined in API header
file
  LPVOID       lpInBuffer,       // Pointer to input parameter
  DWORD        nInBufferSize,    // Size of input parameter
  LPVOID       lpOutBuffer,      // Pointer to output parameter
  DWORD        nOutBufferSize,   // Size of output parameter
  LPDWORD      lpBytesReturned,  // Pointer to return length parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to overlapped
structure
);                               //   used for asynchronous I/O
```

The IOCTLs defined for the PMC-Parallel-IO driver are described below:

IOCTL_PMC_PAR_IO_GET_INFO

Function: Returns the current driver version and instance number.
Input: none
Output: PMC_PAR_IO_DRIVER_DEVICE_INFO structure
Notes: This call does not access the hardware, only driver parameters. See the definition of UART_BASE_DRIVER_DEVICE_INFO below.

```
typedef struct _PMC_PAR_IO_DRIVER_DEVICE_INFO
{
   ULONG    DriverVersion;
   ULONG    InstanceNumber;
} PMC_PAR_IO_DRIVER_DEVICE_INFO, *PPMC_PAR_IO_DRIVER_DEVICE_INFO;
```

IOCTL_PMC_PAR_IO_SET_OUT_DATA

Function: Sets the value of the TTL outputs on the board.
Input: PMC_PAR_IO_DATA structure
Output: none
Notes: The input data structure has two unsigned long int fields, LoWord and HiWord. These correspond to the 64 TTL lines on the board. See the definition of PMC_PAR_IO_DATA below.

```
typedef struct _PMC_PAR_IO_DATA
{
   ULONG    LoWord;
   ULONG    HiWord;
} PMC_PAR_IO_DATA, *PPMC_PAR_IO_DATA;
```

IOCTL_PMC_PAR_IO_GET_OUT_DATA

Function: Returns the state of the TTL outputs in the output data register.
Input: none
Output: PMC_PAR_IO_DATA structure
Notes: This call returns the state of the output data registers on the board. The drivers are open collector with pull-up resistors, therefore if an IO line is being driven externally the actual value of the IO bus may not match this value. See the definition of PMC_PAR_IO_DATA above.

IOCTL_PMC_PAR_IO_READ_IN_DATA

Function: Reads the input/output data bus directly.
Input: none
Output: PMC_PAR_IO_DATA structure
Notes: This call reads the input data from the TTL input lines and returns a PMC_PAR_IO_DATA structure that reports the state of the 64 TTL IO bus lines. See the definition of PMC_PAR_IO_DATA above.

IOCTL_PMC_PAR_IO_SET_CLOCK_CONFIG

Function: Sets the clock configuration parameters.
Input: PMC_PAR_IO_CLOCK_CONFIG structure
Output: none
Notes: Controls the frequency of the internally generated clock, the state of the internal clock enable, and selects the internal or external source for the clock and clock enable. This clock and enable are used to clock the bus data value into the data read-back registers. See the definition of PMC_PAR_IO_CLOCK_CONFIG below.

```
typedef struct _PMC_PAR_IO_CLOCK_CONFIG
{
   UCHAR    FreqSelect;
   BOOLEAN  ExtClockSelect;
   BOOLEAN  ExtEnableSelect;
   BOOLEAN  IntEnableOn;
} PMC_PAR_IO_CLOCK_CONFIG, *PPMC_PAR_IO_CLOCK_CONFIG;
```

IOCTL_PMC_PAR_IO_GET_CLOCK_CONFIG

Function: Returns the configuration of the clock control register.
Input: none
Output: PMC_PAR_IO_CLOCK_CONFIG structure
Notes: Returns the values set in the previous call. See the definition of PMC_PAR_IO_CLOCK_CONFIG above.

IOCTL_PMC_PAR_IO_SET_INT_CONFIG

Function: Sets interrupt configuration parameters.
Input: PMC_PAR_IO_INT_CONFIG structure
Output: none
Notes: Enables and controls the behavior of the two interrupts connected to bit 0 and 1 of the IO bus data. These interrupts can be individually enabled and configured to respond to a high or low data value or a rising or falling edge on the corresponding data line. See the definition of PMC_PAR_IO_INT_CONFIG below.

```
typedef struct _PMC_PAR_IO_INT_CONFIG
{
   BOOLEAN  Int0Enabled;
   BOOLEAN  Int0Edge_Level;
   BOOLEAN  Int0PolHi_Lo;
   BOOLEAN  Int1Enabled;
   BOOLEAN  Int1Edge_Level;
   BOOLEAN  Int1PolHi_Lo;
} PMC_PAR_IO_INT_CONFIG, *PPMC_PAR_IO_INT_CONFIG;
```

IOCTL_PMC_PAR_IO_GET_INT_CONFIG

Function: Returns the configuration of the interrupt control register.
Input: none
Output: PMC_PAR_IO_INT_CONFIG structure
Notes: Returns the values set in the previous call. See the definition of PMC_PAR_IO_INT_CONFIG above.

IOCTL_PMC_PAR_IO_GET_INT_STATUS

Function: Returns the control/status bits in the Plx ICS register.
Input: none
Output: unsigned long int
Notes: The Plx-9052 interrupt control/status bits are read by this call. See the PLX 9052 manual for the bit definitions.

IOCTL_PMC_PAR_IO_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 an interrupt is serviced.

The user interrupt service routine waits on this event, allowing it to respond to the interrupt. When it is desired to un-register the event, set the event handle input parameter to NULL.

IOCTL_PMC_PAR_IO_ENABLE_INTERRUPT

Function: Enables the interrupts in the Plx-9052.
Input: none
Output: none
Notes: Sets the Plx interrupt enables. This IOCTL is used in the user interrupt processing function to begin interrupt processing or to re-enable the interrupts after they were disabled in the driver interrupt service routine.

IOCTL_PMC_PAR_IO_DISABLE_INTERRUPT

Function: Disables the Plx-9052 interrupts.
Input: none
Output: none
Notes: Clears the Plx interrupt enables. This IOCTL is used when interrupt processing is no longer desired.

IOCTL_PMC_PAR_IO_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: none
Output: none
Notes: Causes an interrupt to be asserted on the PCI bus provided the interrupts are enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_PMC_PAR_IO_GET_ISR_STATUS

Function: Returns the Plx-9052 interrupt status read in the last ISR.
Input: none
Output: ULONG
Notes: The status contains the status and control bits of the interrupt register read in the last ISR execution.

Warranty and Repair

Please refer to the warranty page on our website for the current warranty offered and options.

http://www.dyneng.com/warranty.html

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.

Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

For Service Contact:

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