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

Sdlc

Driver Documentation

Developed with Windows Driver Foundation

Revision A
Corresponding Hardware: Revision E
10-2005-0205
Corresponding Firmware: Revision B

Sdlc
WDF Device Driver for the
PMC-BiSerial-III-SDLC
8-Channel PMC-Based SDLC Interface

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

Table of Contents

Introduction

The Sdlc driver is a Windows device driver for the PMC-BiSerial-III SDLC (Synchronous Data-Link Control) from Dynamic Engineering. This driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The PMC-BiSerial-III board has a Spartan3-1500/2000 Xilinx FPGA to implement the PCI interface, dual-port RAMs and protocol control and status for eight serial channels. Each channel has two 4kByte RAMs, one for transmit data and one for received data.

When the PMC-BiSerial-III SDLC is recognized by the PCI bus configuration utility it will start the Sdlc driver. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the I/O buffer memory.

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 PMC-BiSerial-III SDLC user manual (also referred to as the hardware manual).

Driver Installation

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

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

Windows 7 Installation

Copy Sdlc.inf, Sdlc.cat, Sdlc.sys and WdfCoInstaller01009.dll to a removable memory device or any other system accessible memory location as preferred.

With the PMC-BiSerial-III SDLC 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***.
- Browse to the location of the device prepared above.
- Select ***Next***.
- Select ***Close*** to close the update window.

***** 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 a globally unique identifier (GUID), which is defined in SdlcPublic.h. See main.c in the PB3SdlcUserApp project for an example of how to acquire a handle for the device.

Note: In order to build an application you must link with setupapi.lib.

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() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often custom structures are 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 Sdlc driver are described below:

IOCTL_SDLC_GET_INFO

Function: Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, and device instance number.
Input: None
Output: SDLC_DRIVER_DEVICE_INFO structure
Notes: The switch value is the configuration of the 8-bit onboard dipswitch that has been selected by the user (see the board silk screen for bit position and polarity). Instance number is the zero-based device number. See the definition of SDLC_DRIVER_DEVICE_INFO below.

```
//  Driver version and instance information
typedef struct _SDLC_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    SwitchValue;
   UCHAR    PllDeviceId;   // PLL device identifier (0x69 or 0x6A)
   USHORT   DesignId;      // Firmware design identifier
   USHORT   DesignRev;     // Firmware design revision
   ULONG    InstanceNumber;
} SDLC_DRIVER_DEVICE_INFO, *PSDLC_DRIVER_DEVICE_INFO;
```

IOCTL_SDLC_SET_WR_MEM_OFFSET

Function: Specifies the channel and the byte offset into the transmit data RAM to be used as the starting location for the next WriteFile call.
Input: SDLC_MEM_ACCESS structure
Output: None
Notes: This call must be made before any WriteFile call that targets a different channel or memory offset from the location that was previously accessed. Offset is a byte offset and must be between zero and SDLC_MEM_SIZE. Only the transmit RAM can be targeted. Use IOCTL_SDLC_PUT_DATA_WORD if you wish to write to the received data RAM. See the definition of SDLC_MEM_ACCESS below.

```
typedef struct _SDLC_MEM_ACCESS {
   UCHAR    Channel;
   USHORT   Offset;
} SDLC_MEM_ACCESS, *PSDLC_MEM_ACCESS;
```

IOCTL_SDLC_SET_RD_MEM_OFFSET

Function: Specifies the channel and the byte offset into the received data RAM to be used as the starting location for the next ReadFile call.
Input: SDLC_MEM_ACCESS structure
Output: None
Notes: This call must be made before any ReadFile call that targets a different channel or memory offset from the location that was previously accessed. Offset is a byte offset and must be between zero and SDLC_MEM_SIZE. Only the receive RAM can be targeted. Use IOCTL_SDLC_GET_DATA_WORD if you wish to read from the transmit data RAM. See the definition of SDLC_MEM_ACCESS above.

IOCTL_SDLC_PUT_DATA_WORD

Function: Writes a single long-word to any one of the dual-port RAMs.
Input: SDLC_WRITE_WORD structure
Output: None
Notes: This call is used to write a single 32-bit word to any of the 16 dual-port RAM blocks. Offsets between zero and SDLC_MEM_SIZE target the transmit data RAM for the referenced channel, whereas offsets between SDLC_MEM_SIZE and SDLC_CHAN_MEM_SIZE target the channel's received data RAM. See the definition of SDLC_WRITE_WORD below.

```
typedef struct _SDLC_WRITE_WORD {
   UCHAR    Channel;
   USHORT   Offset;
   ULONG    Data;
} SDLC_WRITE_WORD, *PSDLC_WRITE_WORD;
```

IOCTL_SDLC_GET_DATA_WORD

Function: Reads a single long-word from any one of the dual-port RAMs.
Input: SDLC_MEM_ACCESS structure
Output: Value of memory at specified address (unsigned long integer)
Notes: This call is used to read a single 32-bit word from any of the 16 dual-port RAM blocks. Offsets between zero and SDLC_MEM_SIZE target the transmit data RAM for the referenced channel, whereas offsets between SDLC_MEM_SIZE and SDLC_CHAN_MEM_SIZE target the channel's received data RAM. See the definition of SDLC_MEM_ACCESS above.

IOCTL_SDLC_SET_CHANNEL_CONTROL

Function: Sets the control parameters for the SDLC interface of the referenced channel.
Input: SDLC_CHAN_CNTL structure
Output: None
Notes: This call controls channel configuration items for sending and receiving SDLC data-frames. Among these parameters are the transmitter start and stop memory offset, the receiver start address, various interrupt enables, internal/external clock selection, transmit and receive enables and other control parameters. The addresses are all 16-bit word addresses. See the definition of SDLC_CHAN_CNTL below.

```
typedef struct _SDLC_CHAN_CNTL {
   UCHAR    Channel;
   BOOLEAN  TxEnable;
   BOOLEAN  RxEnable;
   BOOLEAN  TxExtClk;
   BOOLEAN  TxClearEnable;
   BOOLEAN  TxIntEnable;
   BOOLEAN  TxDnIntEnable;
   BOOLEAN  RxIntEnable;
   BOOLEAN  AbortIntEnable;
   BOOLEAN  TxIdleFrmEnd;
   BOOLEAN  TxFlgsShrZero;
   BOOLEAN  SendAbort;
   USHORT   RxStartAddress;
   BOOLEAN  LoadRxStartAddr;
   USHORT   TxStartAddress;
   BOOLEAN  LoadTxStartAddr;
   USHORT   TxEndAddress;
   BOOLEAN  LoadTxEndAddr;
} SDLC_CHAN_CNTL, *PSDLC_CHAN_CNTL;
```

IOCTL_SDLC_GET_CHANNEL_STATE

Function: Returns the control parameters set in the previous call, whether idle or abort states were detected and the receiver end address.
Input: None
Output: SDLC_CHAN_STATE structure
Notes: This call returns the state of various channel control parameters as well as some operational status values of the SDLC receiver. These include the 16-bit aligned end address of the last SDLC message-frame received and whether Idle or Abort conditions were detected. The Idle and Abort status bits are latched and, if set, they will be automatically cleared by this call. See the definition of SDLC_CHAN_STATE below.

```
typedef struct _SDLC_CHAN_STATE {
   BOOLEAN  TxEnable;
   BOOLEAN  RxEnable;
   BOOLEAN  TxExtClk;
   BOOLEAN  TxSndngFrm;
   BOOLEAN  TxFrmDone;
   BOOLEAN  TxClearEnable;
   BOOLEAN  TxIntEnable;
   BOOLEAN  TxDnIntEnable;
   BOOLEAN  RxIntEnable;
   BOOLEAN  AbortIntEnable;
   BOOLEAN  TxFlgsShrZero;
   BOOLEAN  TxIdleFrmEnd;
   USHORT   RxEndAddress;
   BOOLEAN  AbortReceived;
   BOOLEAN  IdleDetected;
} SDLC_CHAN_STATE, *PSDLC_CHAN_STATE;
```

IOCTL_SDLC_SET_DATA

Function: Sets the value of the 34 I/O data lines when they are used as a parallel port instead of for the SDLC interface.
Input: SDLC_IO
Output: None
Notes: IoData controls the lower 32 I/O lines (0-31) Bit32 and Bit33 control the two upper lines. See the definition of SDLC_IO below.

```
typedef struct _SDLC_IO {
   ULONG    IoData;
   BOOLEAN  Bit32;
   BOOLEAN  Bit33;
} SDLC_IO, *PSDLC_IO;
```

IOCTL_SDLC_GET_DATA

Function: Reads and returns the values set in the previous call.
Input: None
Output: SDLC_IO
Notes: See the definition of SDLC_IO above.

IOCTL_SDLC_SET_DIR

Function: Controls the direction of the 34 I/O data lines when they are used as a parallel port instead of for the SDLC interface.
Input: SDLC_IO
Output: None
Notes: When one of the control bits is a one, the corresponding I/O line is configured as an output from the board, when it's a zero, the I/O line is configured as an input. See the definition of SDLC_IO above.

IOCTL_SDLC_GET_DIR

Function: Reads and returns the values set in the previous call.
Input: None
Output: SDLC_IO
Notes: See the definition of SDLC_IO above.

IOCTL_SDLC_SET_TERM

Function: Controls the terminations on the 34 I/O data lines.
Input: SDLC_IO
Output: None
Notes: When one of the control bits is set to a one, the corresponding I/O line will be terminated with a nominal 100Ω shunt resistance, when it's a zero, the I/O line will not be terminated. See the definition of SDLC_IO above.

IOCTL_SDLC_GET_TERM

Function: Reads and returns the values set in the previous call.
Input: None
Output: SDLC_IO
Notes: See the definition of SDLC_IO above.

IOCTL_SDLC_SET_MUX

Function: Determines whether the data and direction of the referenced I/O line is controlled by the SDLC state machine or the data and direction registers described above.
Input: SDLC_IO
Output: None
Notes: When one of the control bits is a one, the corresponding I/O line will be controlled by the respective SDLC state-machine, when it's a zero, the I/O line is controlled by the data and direction I/O registers. See the definition of SDLC_IO above.

IOCTL_SDLC_GET_MUX

Function: Reads and returns the values set in the previous call.
Input: None
Output: SDLC_IO
Notes: See the definition of SDLC_IO above.

IOCTL_SDLC_READ_DATA

Function: Reads and returns the values seen on the external I/O bus.
Input: None
Output: SDLC_IO
Notes: Whatever value is present on the respective I/O lines whether an input or an output will be returned by this call. See the definition of SDLC_IO above.

IOCTL_SDLC_GET_INT_STATUS

Function: Reads and returns the status of all the channel interrupt conditions.
Input: None
Output: Value of interrupt conditions (unsigned long integer)
Notes: Each of the eight SDLC channels has four possible interrupt conditions transmission done, transmitter end-of-frame, receiver end-of-frame and received message aborted. These interrupt conditions are represented in that order with channel 0 in the least significant hex digit, channel 1 in the next digit and so on. After reading the interrupt status, the status latches will be cleared.

IOCTL_SDLC_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the 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_SDLC_ENABLE_INTERRUPT

Function: Enables the master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to user interrupts. The master interrupt enable is disabled in the driver interrupt service routine. Therefore this command must be run after each user interrupt occurs to re-enable the interrupts.

IOCTL_SDLC_DISABLE_INTERRUPT

Function: Disables all user interrupts.
Input: None
Output: None
Notes: This call is used when user interrupt processing is no longer desired.

IOCTL_SDLC_FORCE_INTERRUPT

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

IOCTL_SDLC_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (SDLC_CHAN_INT_STAT)
Notes: Returns the status that was read in the interrupt service routine for the last user interrupt serviced. The interrupt status that was read in the ISR is returned, but the status latches will have been automatically cleared in the interrupt DPC. See the definition of SDLC_CHAN_INT_STAT below.

```
typedef struct _SDLC_CHAN_INT_STAT {
  ULONG    Status;  // Value of status register read in ISR
  BOOLEAN  New;     // True if the status has changed since the last call
} SDLC_CHAN_INT_STAT;
```

IOCTL_SDLC_WRITE_I2O_ADDRESS

Function: Writes the physical address used for I2O accesses to the SDLC device.
Input: I2O memory buffer physical address (unsigned long integer)
Output: None
Notes: A buffer is allocated from the kernel non-paged memory pool. The physical address of this buffer is stored in the SDLC device and interrupt status is automatically written to that address if an enabled interrupt occurs and the I2O interface is enabled.

IOCTL_SDLC_SET_I2O_CONTROL

Function: Sets the control configuration for the I2O interface.
Input: SDLC_I2O_CONTROL structure
Output: None
Notes: This call allows the driver to enable the I2O interface and clear the stored interrupt status. See the definition of SDLC_I2O_CNTL below.

```
typedef struct _SDLC_I2O_CNTL {
   BOOLEAN  Enable;
   BOOLEAN  Clear;
} SDLC_I2O_CNTL, *PSDLC_I2O_CNTL;
```

IOCTL_SDLC_I2O_TEST_READ

Function: Reads from the physical address allocated for testing the I2O interface.
Input: None
Output: Interrupt status word (unsigned long integer)
Notes: The physical address of the allocated buffer is obtained and stored in the device extension. When the I2O interface is enabled and an enabled interrupt condition occurs, the interrupt status is written to this address by the SDLC hardware. This call reads and returns this status word.

IOCTL_SDLC_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL device.
Input: SDLC_PLL_DATA structure
Output: None
Notes: The SDLC_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write to the PLL device. See below for the definition of SDLC_PLL_DATA.

```
#define PLL_MESSAGE1_SIZE  16
#define PLL_MESSAGE2_SIZE  24
#define PLL_MESSAGE_SIZE  (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _SDLC_PLL_DATA {
   UCHAR    Data[PLL_MESSAGE_SIZE];
} SDLC_PLL_DATA, *PSDLC_PLL_DATA;
```

IOCTL_SDLC_READ_PLL_DATA

Function: Returns the contents of the internal registers of the PLL device.
Input: None
Output: SDLC_PLL_DATA structure
Notes: The register data is read from the PLL device and loaded into the SDLC_PLL_DATA structure in an array of 40 bytes. See definition of SDLC_PLL_DATA above.

Write

PMC-BiSerial-III SDLC data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the target device, a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous I/O. The data will be written to the transmit data RAM starting at the byte address specified by the IOCTL_SDLC_SET_WR_MEM_OFFSET call.

Read

PMC-BiSerial-III SDLC data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the target device, a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous I/O. The data will be read from the received data RAM starting at the byte address specified by the IOCTL_SDLC_SET_RD_MEM_OFFSET call.

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 merchantability 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 C Santa Cruz, CA 95060
(831) 457-8891 Fax (831) 457-4793
support@dyneng.com