

# DYNAMIC ENGINEERING

435 Park Dr., Ben Lomond, Calif. 95005

831-336-8891 Fax 831-336-3840

<http://www.dyneng.com>

[sales@dyneng.com](mailto:sales@dyneng.com)

Est. 1988

# pb3\_mds1 & mds1\_chan

## Driver Documentation

Linux 2.6.16

Revision A

Corresponding Hardware: Revision C

10-2005-0203

Corresponding Firmware: Revision A

**pb3\_mds1 and mds1\_chan**  
Linux Device Drivers for the  
PMC-BiSerial-III Mds1  
Four-channel Manchester encoded  
PMC based Serial Interface

Dynamic Engineering  
435 Park Drive  
Ben Lomond, CA 95005  
831-336-8891  
831-336-3840 FAX

©2007 by Dynamic Engineering.  
Other trademarks and registered trademarks are owned by their  
respective manufactures.  
Manual Revision A. Revised February 22, 2007

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

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.

Connection of incompatible hardware is likely to cause serious damage.



---

---

# Table of Contents

---

---

Introduction .....	4
Note .....	4
Driver Installation .....	4
Driver Startup .....	5
IO Controls .....	6
IOCTL_PB3_MDS1_GET_INFO .....	6
IOCTL_PB3_MDS1_LOAD_PLL_DATA .....	6
IOCTL_PB3_MDS1_READ_PLL_DATA .....	6
IOCTL_MDS1_CHAN_GET_INFO .....	6
IOCTL_MDS1_CHAN_RESET_FIFOS .....	7
IOCTL_MDS1_CHAN_SET_CONFIG .....	7
IOCTL_MDS1_CHAN_GET_CONFIG .....	7
IOCTL_MDS1_CHAN_GET_STATUS .....	7
IOCTL_MDS1_CHAN_SET_FIFO_LEVELS .....	7
IOCTL_MDS1_CHAN_GET_FIFO_LEVELS .....	8
IOCTL_MDS1_CHAN_WRITE_FIFO .....	8
IOCTL_MDS1_CHAN_READ_FIFO .....	8
IOCTL_MDS1_CHAN_GET_FIFO_COUNTS .....	8
IOCTL_MDS1_CHAN_WAIT_ON_INTERRUPT .....	9
IOCTL_MDS1_CHAN_ENABLE_INTERRUPT .....	9
IOCTL_MDS1_CHAN_DISABLE_INTERRUPT .....	9
IOCTL_MDS1_CHAN_FORCE_INTERRUPT .....	9
IOCTL_MDS1_CHAN_GET_ISR_STATUS .....	10
IOCTL_MDS1_CHAN_GET_UNIT_ID .....	10
Write .....	11
Read .....	11
Warranty and Repair .....	11
Service Policy .....	12
Out of Warranty Repairs .....	12
For Service Contact: .....	12



## Introduction

The pb3\_mds1 and mds1\_chan drivers are Linux 2.6 device drivers for the PMC-BiSerial-III MDS1 from Dynamic Engineering. The PMC-BiSerial-III MDS1 has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for four serial channels using Manchester encoding. Each channel uses one RS-485 data input and one RS-485 data output. There is a programmable PLL with two clock outputs. One drives the 2x internal clock reference for the transmit state-machine and the other drives the 8x internal clock reference for the receive state-machine. There are two 1k x 32-bit data FIFOs for each channel, one for the transmit data and one for the receive data.

When the pb3\_mds1 module is installed, it interfaces with the PCI system configuration utility to acquire the memory and interrupt resources for each device installed. An mds1 bus is created for each device and four channel devices are allocated. The interrupt is assigned and the address space partitioned for the four channel devices. When the mds1\_chan driver is installed, it probes the mds1 bus and finds and initializes the four channel devices for each board. It allocates read and write list memory to hold the DMA page descriptors that are used by the hardware to perform scatter-gather DMA.

### 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-BiSerial-III MDS1 user manual (also referred to as the hardware manual).

### Driver Installation

The source files and makefiles for the drivers and test application are supplied in the driver package in a zip file. Copy the directory structure to the computer where the driver is to be installed. From the top-level directory type “make” to build the object files then type “make install” to copy the files to the target location (/lib/modules/\${VERSION}/kernel/drivers/misc/ for the driver and /usr/local/bin/ for the test app). If desired type “make clean” to remove interim files.

A load\_pb3mds1 script is provided that will load the base driver, parse the /proc/devices file for the device’s major number, count the number of entries in the /sys/bus/mds1/devices/ directory to determine the number of boards installed, create the required number of /dev/pb3\_mds1\_x (where x is the zero



based board number) device nodes, load the channel driver, find that major number and create the required number of /dev/mds1\_chan\_x device nodes as well.

The pb3\_mds1\_api.h and mds1\_chan\_api.h files are C header files that define the Application Program Interface (API) to the drivers and contain the relevant bit defines for the control/status registers on the PMC-BiSerial-III MDS1.

The user\_app source code will provide examples of how to use the driver calls to control the hardware.

## Driver Startup

Install the hardware and boot the computer. After the drivers have been installed run the load\_pb3mds1 script to start the drivers and create the device interface nodes.

Handles can be opened to a specific board by using the open() function call and passing in the appropriate device names.

Below is example code for opening handles for device dev\_num.

```
    sprintf(Name, "/dev/pb3_mds1_%d", dev_num);
    hpb3_mds1 = open(Name , O_RDWR);
    if(hpb3_mds1 < 2)
    {
        printf("\n%s FAILED to open!\n", Name);
        return 1;
    }
    cdev_num = 4 * dev_num;
    for(i = 0; i < 4; i++)
    {
        sprintf(Name, "/dev/mds1_chan_%d", cdev_num + i);
        hmds1_chan[i] = open(Name , O_RDWR);
        if(hmds1_chan[i] < 2)
        {
            printf("\n%s FAILED to open!\n", Name);
            return 1;
        }
    }
}
```



## IO Controls

The driver uses ioctl() calls to configure the device and obtain status. The parameters passed to the ioctl() function include the handle obtained from the open() call, an integer command defined in the API header files and an optional parameter used to pass data in and/or out of the device. The ioctl commands defined for the PMC-BiSerial-III MDS1 are listed below.

### IOCTL\_PB3\_MDS1\_GET\_INFO

**Function:** Returns the Driver version, Xilinx revision, PLL device ID and Switch value.

**Input:** None

**Output:** PB3\_MDS1\_DRIVER\_DEVICE\_INFO structure

**Notes:** Switch value is the configuration of the on-board dip-switch that has been selected by the user (see the board silk screen for bit position and polarity). See pb3\_mds1\_api.h for the definition of PB3\_MDS1\_DRIVER\_DEVICE\_INFO.

### IOCTL\_PB3\_MDS1\_LOAD\_PLL\_DATA

**Function:** Loads the internal registers of the PLL.

**Input:** PB3\_MDS1\_PLL\_DATA structure

**Output:** None

**Notes:** The PB3\_MDS1\_PLL\_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write.

### IOCTL\_PB3\_MDS1\_READ\_PLL\_DATA

**Function:** Returns the contents of the PLL's internal registers.

**Input:** None

**Output:** PB3\_MDS1\_PLL\_DATA structure

**Notes:** The register data is output in the PB3\_MDS1\_PLL\_DATA structure in an array of 40 bytes.

### IOCTL\_MDS1\_CHAN\_GET\_INFO

**Function:** Returns the Driver version and Instance number.

**Input:** None

**Output:** MDS1\_CHAN\_DRIVER\_DEVICE\_INFO structure

**Notes:** See mds1\_chan\_api.h for the definition of MDS1\_CHAN\_DRIVER\_DEVICE\_INFO.



## IOCTL\_MDS1\_CHAN\_RESET\_FIFOS

*Function:* Resets the channel's FIFOs.

*Input:* None

*Output:* None

*Notes:* Resets the Tx and Rx FIFOs for the referenced channel.

## IOCTL\_MDS1\_CHAN\_SET\_CONFIG

*Function:* Writes to the channel's control register.

*Input:* Value of the control register (unsigned long integer)

*Output:* None

*Notes:* Only the bits in the CNTRL\_MASK are controlled by this command. See the bit definitions in mds1\_chan\_api.h for information on determining this value.

## IOCTL\_MDS1\_CHAN\_GET\_CONFIG

*Function:* Returns the configuration of the control register.

*Input:* None

*Output:* Value of control register (unsigned long integer)

*Notes:* The return value includes the bits in CNTRL\_MASK and CNTRL\_DMA\_WREN, CNTRL\_DMA\_RDEN and CNTRL\_MINTEN. This command is used mainly for testing.

## IOCTL\_MDS1\_CHAN\_GET\_STATUS

*Function:* Returns the channel's status value and clears the latched status bits.

*Input:* None

*Output:* Value of the channel's status register (unsigned long integer)

*Notes:* See mds1\_chan\_api.h for the status bit definitions. Only the bits in STATUS\_MASK will be returned. The bits in STATUS\_LATCH\_MASK will be cleared by this call only if they are set when the register is read. This prevents the possibility of missing an interrupt condition that occurs after the register read but before the register write that clears the bits.

## IOCTL\_MDS1\_CHAN\_SET\_FIFO\_LEVELS

*Function:* Sets the channel's receiver almost full and transmitter almost empty levels.

*Input:* MDS1\_CHAN\_FIFO\_LEVELS structure

*Output:* None

*Notes:* These FIFO levels are used to determine status when the FIFO data counts reach the specified levels. See mds1\_chan\_api.h for the definition of MDS1\_CHAN\_FIFO\_LEVELS.



## IOCTL\_MDS1\_CHAN\_GET\_FIFO\_LEVELS

*Function:* Returns the channel's receiver almost full and transmitter almost empty levels.

*Input:* None

*Output:* MDS1\_CHAN\_FIFO\_LEVELS structure

*Notes:* See mds1\_chan\_api.h for the definition of MDS1\_CHAN\_FIFO\_LEVELS.

## IOCTL\_MDS1\_CHAN\_WRITE\_FIFO

*Function:* Writes a single data word to the channel's transmit FIFO.

*Input:* FIFO data word (unsigned long integer)

*Output:* None

*Notes:* Normally the write command is used to load data into the device. This call can be used for small amounts of data, but is much more inefficient for a transfer of any larger size.

## IOCTL\_MDS1\_CHAN\_READ\_FIFO

*Function:* Reads a data word from the channel's receive FIFO.

*Input:* None

*Output:* FIFO data word (unsigned long integer)

*Notes:* Normally the read command is used to read data from the device. This call can be used for small amounts of data, but is much more inefficient for a transfer of any larger size.

## IOCTL\_MDS1\_CHAN\_GET\_FIFO\_COUNTS

*Function:* Returns the number of data words in the transmit and receive FIFOs.

*Input:* None

*Output:* MDS1\_CHAN\_FIFO\_COUNTS structure

*Notes:* Returns the number of words in the referenced channels I/O data circuitry. For the transmitter this can be a maximum of one more than the FIFO size and for the receiver the maximum data-count can be as much as four words more than the FIFO size. The excess is due to data pipe-line latches in the I/O stream. See mds1\_chan\_api.h for the definition of MDS1\_CHAN\_FIFO\_COUNTS.



## IOCTL\_MDS1\_CHAN\_WAIT\_ON\_INTERRUPT

*Function:* Initiates a wait state for the current execution thread to allow the user code to respond to interrupt conditions.

*Input:* Time-out value in jiffies (jiffy = 10 milliseconds)

*Output:* none

*Notes:* This call is made in the user interrupt service routine to allow user-specified interrupt handlers for enabled interrupt conditions. The input parameter is a time-out value that causes the call to abort if the interrupt doesn't occur within the specified time. If the timeout is zero, the call will wait indefinitely for the interrupt to occur. The DMA interrupts do not use this mechanism; they are controlled automatically by the driver.

## IOCTL\_MDS1\_CHAN\_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 in order to respond to the next interrupt.

## IOCTL\_MDS1\_CHAN\_DISABLE\_INTERRUPT

*Function:* Disables the master interrupt.

*Input:* none

*Output:* none

*Notes:* This call is used when user interrupt processing is no longer desired.

## IOCTL\_MDS1\_CHAN\_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 call is used for development, to test interrupt processing.



## IOCTL\_MDS1\_CHAN\_GET\_ISR\_STATUS

**Function:** Returns the interrupt status that was read in the ISR from the last user interrupt.

**Input:** none

**Output:** Interrupt status value and time-out status (MDS1\_CHAN\_ISR\_STAT structure)

**Notes:** Returns the interrupt status that was read in the interrupt service routine for the last user interrupt serviced. If TimedOut is true, the time-out expired before the interrupt occurred.

## IOCTL\_MDS1\_CHAN\_GET\_UNIT\_ID

**Function:** Returns the unit ID read from the channel status register.

**Input:** none

**Output:** ID value (unsigned char)

**Notes:** When a data-block is received the last 16-bit word contains the unit id in bits 14 to 7. The receive state-machine will shift this value to bits 7 to 0 and store it in a latch. This call will return the unit id that was read in the last data block received. Although the unit id is read from the status port, this call is independent from the GET\_STATUS call and has no effect on latched status bits or other status values.



## Write

PMC-BiSerial-III MDS1 transmit data is written to the device using the write command. A handle to the device, a pointer to a pre-allocated buffer that contains the data to write and an unsigned long integer that represents the amount of data to write in bytes are passed to the write call. The driver will obtain physical addresses to the pages containing the data and will set-up a list of page descriptors in its list memory. The physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA to transfer the data.

## Read

PMC-BiSerial-III MDS1 received data is read from the device using the read command. A handle to the device, a pointer to a pre-allocated buffer that will contain the data read and an unsigned long integer that represents the amount of data to read in bytes are passed to the read call. The driver will obtain physical addresses to the buffer memory pages and will set-up a list of page descriptors in its list memory. The physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA to transfer the data.

## 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  
435 Park Dr.  
Ben Lomond, CA 95005  
831-336-8891  
831-336-3840 fax

[support@dyneng.com](mailto:support@dyneng.com)

All information provided is Copyright Dynamic Engineering.

