DYNAMIC ENGINEERING

150 DuBois, Suite C

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

PMC-BiSerial-VI-UART

Linux Documentation

Developed/Tested on Linux Kernel
v. 5.4.0-74-generic

Revision 01p2 9/17/21
Corresponding Hardware: Revision 02+
PMC 10-2015-06XX
FLASH 0210

PMC-BiSerial-VI-UART
Linux Device Driver

Dynamic Engineering

150 DuBois, Suite C

Santa Cruz, CA 95060

(831) 457-8891

Table of Contents

Introduction

The PMC-BiSerial-VI is an eight channel, full duplex UART interface card supporting various modes of operation. All channels are supported with their own DMA engines (For a detailed description of the hardware including register definitions, see HW User Manual).

Driver Installation

Kernel drivers must be compiled to run on each specific kernel. As such, we distribute all the source code for the driver along with a make file (this will make the .ko file) and install script (this installs the driver and creates the device nodes for applications to access the ports “/dev/ deUart_<x>”, where <x> can be replaced by the port number 0-7, and finally an uninstall script (this uninstalls the driver and removes all device nodes).

Note: the driver does not install permanently with the current script. As such, the driver will need to be reinstalled if the computer is rebooted. If you would like the driver installed permanently, and you are having any difficulty with the process using a standard Linux distribution such as Ubuntu, CentOS, or RedHat, please contact us and we can assist you with this procedure.

The provided de_BiSerUart.h and de_common.h files are the C header files that define the Application Program Interface (API) for the BiSerUart driver. These files are required at compile time by any application that wishes to interface with the driver and for compiling the driver. The UserAp sample software package is written in C++ (with some legacy C embedded in it) to demonstrate how to use the C API within a C++ environment. The other example software is written in C.

Driver Software Description

The driver supports full duplex operation on all 8 channels.

A default configuration is applied when ports are opened for the first time. These default settings are defined in the driver header file, de_BiSerUart.h. The default I/O port config setting is named de_default_pt_config. The default config parameters can be customized for a particular application, and the driver recompiled. This may eliminate the need for invoking the config ioctl.

Applicable I/O configuration parameters include blocking timeout, baud-rate, mode, parity, flow control, inter-char timer (utilized for packet modes), and various UART options (data size, stop bits, and terminations). Blocking timeout provides a mechanism to timeout on blocking operations.

Default I/O configuration is as follows: Blocking timeout on reads = 5 sec. (if opened as blocking), 115200 baud-rate, packed mode of operation, even parity, flow control enabled (CTS/RTS), auto compute inter-char timer based upon baud-rate, 8-bit data, 1 stop bit, terminate CTS and Rx signals.

Modes of Operation

The HW and SW support 5 modes of operation on a port by port basis, all modes accept (writes) and return (reads) a packed byte stream. Please note I/O limitations between ports populating different platform types (little endian to/from big endian). If required for specific customer applications, these limitations can be addressed/resolved for an additional fee.

Unpacked

Prepends or strips 3 fill bytes for each data byte, max frame size = 255 bytes. Size does not have to be a multiple of 4 bytes. I/O between big/little endian platforms not supported.

Packed

Max frame size = 1020 bytes, size must be a multiple of 4 bytes

Packet

Packed data, max frame size = 1020 bytes, size does not have to be a multiple of 4 bytes, however for non-aligned receive packets least significant bytes are filled with zeros to force alignment. Non-aligned (not a multiple of 4 bytes) I/O between big/little endian platform not supported.

Alternate Packet

Prepends/strips control byte for every 3 bytes of data max frame size = 765 bytes. Does not have to be a multiple of 4 bytes, and received packet will contain no fill bytes. This mode is not supported on big endian platforms.

Test

Raw mode of operation supporting test.

When operating in either of the packet modes, a read will return the next available packed irrespective of size. Thus, reads should be issued with a size of DE_MAX_FRAME. Please see HW manual for further discussion of advantages/disadvantages of each mode.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Node, which controls a single board or I/O channel. IOCTLs are called using the Linux function Ioctl(int fd, unsigned long request, …), and passing in the file descriptor to the device opened with Open(const char *pathname, int flags).

The IOCTLs defined for the BiSerUart driver are described below:

DE_GET_BD_INFO

Function: Returns a struct containing the, Xilinx flash revision (major/minor), type id, and the user switch value.
Input: None
Output: de_rev_t 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). Revision Major and Revision Minor represent the current Flash revision. The design is the design number for a particular version of the board based.

```
// Board information
typedef struct de_rev {
  uint8_t         major;
  uint8_t         minor;
  uint8_t         design;
  uint8_t         dips;
} de_rev_t;
```

DE_PLL

Function: Writes or Reads to the internal registers of the PLL.
Input: de_pll_cfg_t structure (if writing)
Output: de_pll_cfg_t structure (if reading)
Notes: The de_pll_cfg has two elements: op – which is an enum type with three possible values, DE_GET_OP, DE_SET_OP, and DE_RMW_OP. The first is used to read the PLL the second is to write. The third is not used, but could be used to do read/write/update (and is used in other ioctls). The second, dat, is an array of 40 bytes containing the PLL register data to write or that is read based on the op command.

```
// Structures for IOCTLs
typedef enum de_op {
   DE_GET_OP = 0,
   DE_SET_OP = 1,
   DE_RMW_OP = 2
} de_op_t;

typedef struct de_pll_cfg {
   de_opt_t          op;
   unsigned char     dat[PLL_MESSAGE_SIZE];
} de_pll_cfg_t;
```

DE_CONFIG_PT

Function: Reads/Writes the main configuration parameters for each port (depending one which device node was opened).
Input: de_port_cfg_t structure
Output: de_port_cfg_t structure
Notes: This ioctl is used to configure each ports primary settings. As with the PLL above, this requires the de_op_t to say if the configuration is being read or written.

```
// Port Configuration
typedef struct de_port_cfg {
  de_op_t                  op;
  long                     blocking_to;  //if in non-blocking user to pick timeout in milliseconds
  unsigned int             br_clk_src; // 0 = 32 Mhz osc., 1 = PLL
  unsigned int             baud_rate;  //
  unsigned char            mode; // (see de_mode_t below)
  unsigned char            parity; // (see de_parity_t below)
  unsigned char            flow_ctl; //(see flow_ctl_t below)
  unsigned int             ic_time; //
  unsigned                 options; //(see de_opts_t below and 'or' the values together to set configuration)
} de_port_cfg_t;

typedef enum de_mode {
  DE_UNPACKED          = 1,
  DE_PACKED            = 2,
  DE_PACKET            = 3,
  DE_ALT_PACKET        = 4,
  DE_TX_TEST           = 5,
} de_mode_t;

typedef enum de_parity {
  DE_NO_PARITY         = 0,
  DE_EVEN_PARITY       = 1,
  DE_ODD_PARITY        = 2,
  DE_STICK_PARITY      = 3,
} de_parity_t;

typedef enum de_flow {
  DE_NO_FLOW           = 0,
  DE_NORM_FLOW         = 1,
  DE_INVT_FLOW         = 2,
} de_flow_t;

typedef enum de_opts {
  DE_8_BIT             = 0x01,
  DE_2_STOP            = 0x02,
  DE__CTS_TERM         = 0x04,
  DE_RTS_TERM          = 0x08,
  DE_RX_TERM           = 0x10,
  DE_TX_TERM           = 0x20,
  DE_LOOPBACK         = 0x40,
} de_opts_t;
```

DE_GET_STATS

Function: This ioctl fetches and possibly clears stats
Input: de_get_stats_t structure
Output: de_get_stats_t structure

```
// Board information
typedef struct de_get_stats {
  int              clear;
  de_pt_stats_t    stats; // (see de_pt_stats_t below)
} de_get_stats_t;

typedef struct de_pt_stats {
  unsigned int     frame_err_cnt;
  unsigned int     re_ovfl_cnt;
  unsigned int     parity_err_cnt;
  unsigned int     break_cnt;
  unsigned int     last_rx_err;
  unsigned int     rx_cnt;
  unsigned int     tx_cnt;
} de_pt_stats_t;
```

DE_REG

Function: Reads/Writes any register value.
Input: de_reg_cmd_t structure
Output: de_reg_cmd_t structure
Notes: The struct uses the same op code above to determine if reading or writing. The de_reg_cmd_t has five components, the first is the op code, the second is the base address used to determine of you are accessing the board registers or the ports registers, the third is the value read or written, the fourth element is the offset for the specific register you are trying to read/write. The final element can be used to do a RMW mask defined in de_opt_t.

```
typedef struct de_reg_cmd {
    de_op_t             op;
    de_reg_off_t        base; // determines if accessing port or board level registers for this device node
    unsigned int        val; // Value to be written or value read back
    unsigned int        reg; // #define offsets from header file use here to say which register
    unsigned int        mask; //can be used with DE_RMW_OP
} de_reg_cmd_t;
```

```
typedef enum de_reg_off {
  DE_REG_BASE         = 0,
  DE_REG_PT           = 1,
  DE_REG_INV          = 2,
} de_reg_off_t;
```

DE_SEND_BREAK

Function: Sends break
Input: de_break_cmd_t
Output: None
Notes: RETURNS 0 upon success, -EINVAL on failure.

```
typedef struct de_break_cmd_t {
    unsigned int           period;
} de_reg_cmd_t;
```

DE_FIFO_READ

Function: This reads data from the FIFO 32-bits at a time
Input: None
Output: uint32_t
Notes: None

DE_FIFO_WRITE

Function: Writes data to FIFO 32-bits at a time
Input: uint_32
Output: None
Notes: None

DE_FORCE_INT

Function: This will cause the device to trigger an interrupt**.**
Input: None
Output: None
Notes: This is primarily used for testing the boards interrupts

Open

All ioctls, read, write and close, use the file descriptor (fd) returned from an open call that is passed the device node as a parameter (i.e. “/dev/deUart_n”).
The only configuration used in the open call that is supported is the O_NONBLOCK. if O_NONBLOCK and the timeout can be configured with DE_CONFIG_PT ioctl by setting the blocking_to parameter of the struct.

Close

This is the standard Linux system call close() that takes as a parameter the file descriptor returned from open().

Read and Write

Data is written/read to/from the device (and out the port) using the Linux write() and read() system calls. The size of the buffer allowed is constrained by the DE_MAX_FRAME value set as a macro in the header file as well as mode of operation selected during configuration (See Modes of Operation above for reference). If you are experiencing failures on read or write, it is suggested to look at the kernel messages using dmesg to determine the source of failure as there are several debugging messages available for mode/buffer size related errors.

User Software Description

We have provided a **UserAp**, which serves as 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 testing at Dynamic Engineering. The test software can be ported to your application to provide a running start. The tests 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.

In addition to the UserAp, there are a few smaller sample applications included to demonstrate some of the basic means of using the device (open, config, read, write, statistics).

The three sample applications are de_IoApp.c, de_IoAppS.c, de_IoctlApp.c, and demonstrate configuration, ioctl invocation, and I/O in the supported modes, respectively. Various modes of operation and options maybe validated/demonstrated by changing port configuration parameters in the application and recompiling.

Specifically, de_IoApp.c is a board to board test. It requires two boards to be installed in the platform and connected via a board-to-board test fixture. A minimum of two instances must be invoked, first the reader, then the writer within 5 seconds. The applications run asynchronously to one another. Port 0 is connected to port 8, port 1 to port 9, and so on via test fixture.

de_IoAppS.c is a single board test. Ports are looped back to themselves externally via single board test fixture. The application first writes to the specified port, and then reads received data. Data integrity is then validated.

Invocation parameters

The three smaller I/O application invocation paramaters are as follows:

dyn_io - 2 board test

```
./dyn_io 1 0 baud-rate frame_len num_iterations //(reader, port 0, board 1)
./dyn_io 0 8 baud-rate frame_len num_iterations //(writer, port 8, board 2)
```

The first parameter specifies reader/writer. The second parameter is port number, third parameter is baud-rate. Frame length is specified in bytes. Data is validated upon reception. Application will execute for num_iterations, or until terminated due to an error or interrupted via .

dyn_ioS - single board test

```
./dyn_ioS 0 baud-rate frame_len num_iterations //(port 0, board 1)
```

The first parameter specifies port. The second parameter is baud-rate followed by frame length in bytes. Data is validated upon reception. Application will executedfor num_iterations, or until terminated due to an error or interrupted via .

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-VI-UART user manual as appropriate (also referred to as the hardware manual).

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 or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

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
support@dyneng.com