DYNAMIC ENGINEERING

150 DuBois, Suite B & 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. 5.15.0-139-generic

Revision 01p4 5/9/25
Corresponding Hardware: Revision 06+
PMC 10-2015-0606/07
FLASH 0301

PMC-BiSerial-VI-UART
Linux Device Driver

Dynamic Engineering

150 DuBois, Suite B & 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).

The UART functionality is implemented in a Xilinx FPGA. It implements a PCI interface, FIFOs and protocol control/status for 8 channels. Each channel has separate 255 x 32 bit receive data and transmit data FIFOs.

New with Flash revision 3.1 is a programmable parallel port. The parallel port can be mapped in/out to replace unused UART ports. GPIO features including COS interrupts.

When the PmcBis6Uart Driver is installed a device file for the Parallel I/O port as well as a device file for each of the 8 UART I/O ports will be created.

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>” & “/dev/deGPIO_<x>”, where <x> can be replaced by the port number 0-8, 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.

In additional to standard operation mode, the driver supports both **DEBUG** and **TRACE** modes:

- **DEBUG** mode logs function or IOCTL failures to the kernel log, helping identify where failures occur.
- **TRACE** mode logs all IOCTL calls—regardless of success—along with their input or output data, depending on whether the call is a SET or GET. This is helpful when diagnosing unexpected behavior.

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.

To install the PmcBis6Uart driver, first extract the driver package "de_BiSerUartv1_0_4.zip" into your desired directory. Using the command prompt, navigate to the ".../de_BiSerUartv1_0_3_Par/Driver/build" directory, then run one of the following commands depending on the mode you want to driver to run in.

Enabling Modes

To enable these modes during driver compilation, use the following commands:

- Standard Operation mode:
  "sudo make"

- Enable DEBUG mode:
  "sudo make DEBUG_MODE=1"

- Enable TRACE mode:
  "sudo make TRACE_MODE="1

- Enable both DEBUG and TRACE modes:
  "sudo make DEBUG_MODE=1 TRACE_MODE=1"

**Ignore BTF generation, unless using vmlinux for additional debugging*

Once the build completes successfully, make the installation script executable by running

"sudo chmod +x Install"

and install the driver by executing the following command

"sudo ./Install"

Upon successful installation, the following messages will be printed to the screen.

"7 device Nodes created /dev/deUart_<0-7>"

"1 device Nodes created /dev/deGPIO_8"

To **uninstall** the driver, make the uninstall script executable with

"sudo chmod +x UnInstall"

Remove the driver by running

"sudo ./UnInstall"

Upon successful uninstallation, the following messages are printed to the screen.

"de_BiSerUart driver removed"

"device nodes removed"

Driver Software Description

The driver supports full duplex operation on all 8 channels.

The board's default configuration initializes all ports in UART mode upon startup. Individual ports may be reconfigured to operate in either UART or Parallel mode by setting the appropriate bits in the REG_PP_MUX register.
For details, refer to the IOCTL command: "IOCTL_PAR_GPIO_SET_PORTS".

A default UART 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 UART 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 UART 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 - UART

The HW and SW support 5 modes of UART 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.

Modes of Operation – Parallel

Each UART port, when configured in Parallel mode, contributes 4 data bit lanes to the overall parallel interface. With all 8 UART ports set to Parallel mode, this enables a maximum parallel bus width of 32 bits. The behavior and state of each bit lane are controlled through dedicated GPIO registers, accessible via corresponding SET and GET ioctl commands.

Each UART port's mode—UART or Parallel—is controlled by the first 8 bits of the REG_PP_MUX register. A bit value of 1 in this register sets the corresponding port to Parallel mode, while a value of 0 sets it to UART mode. This configuration is applied using the PAR_GPIO_SET_PORTS ioctl call.

Bit lanes are assigned sequentially based on which UART ports are active and configured for Parallel mode. If not all ports are in use or not all are configured as Parallel, gaps may appear in the bit stream. This arrangement allows flexible partial configurations but may require software awareness of active port mappings when interpreting parallel data.

Direction

Each bit in the parallel port can be configured as either an output, driving a signal from the board, or an input, receiving a signal from external sources.

Polarity

Each received bit can be individually configured to remain unchanged or to be inverted. This setting applies only to input data and does not affect output signals. Refer to the Filtered Data registers for more details.

Edge Level

Each bit in the parallel port input can be configured as either edge-sensitive or level-sensitive. When a bit is set, it is treated as edge-sensitive, responding only to transitions; when cleared, it is treated as level-sensitive, responding to the current logic level. This configuration affects only input data and has no impact on output behavior.

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 struct containing Xilinx flash rev (maj/min), type id, and 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 for a RMW mask def 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

DE_REG_WAIT

Function: Registers the calling process to wait for an interrupt.
Input: None
Output: None
Notes: This IOCTL sets an internal flag indicating that the calling process is waiting for an interrupt from the device. When an interrupt occurs on the parallel bus, the driver first checks the device's interrupt status registers to determine whether to run the interrupt service routine (ISR). After the ISR completes, it checks the waiting flag for the registered instance. If the flag is set, the driver signals the waiting process and clears the flag to indicate the interrupt has been acknowledged. This ensures that only processes actively waiting are notified after a valid interrupt

DE_WAIT_INT

Function: Waits for an interrupt event to be signaled by the driver.
Input: None
Output: None
Notes: This IOCTL blocks the calling process until either the interrupt wait flag is cleared (signaled by the driver after an interrupt) or the specified timeout period elapses. Internally, a semaphore is used to implement the wait. If the flag is not cleared within the given time, the function returns, allowing the calling application to handle the timeout condition as needed.

PAR_GPIO_GET_AND_CLEAR_ISR_STATUS

Function: Returns and clears the interrupt status and registers.
Input: None
Output: **PAR_TTL_GPIO_ISR_STAT** structure
Notes: Since the interrupt service routine may have fired multiple times, this returns the cumulative values, or-ed together, of interrupt status and registers read in the interrupt service routine. This IOCTL will clear the stored values.

```
typedef struct _PAR_GPIO_ISR_STAT {
        ULONG    InterruptStatus;
        ULONG    RisingData;
        ULONG    FallingData;
        ULONG    FilteredData;
        ULONG    DirectData;
} PAR_GPIO_ISR_STAT, * PPAR_GPIO_ISR_STAT;
```

PAR_GPIO_SET_PORTS

Function: Select mode of operation for each port. UART or Parallel.
Input: Struct - PAR_GPIO_TYPE
Output: None
Notes: Default is UART operation, HW resets to this state and driver initializes as well.

PAR_GPIO_GET_PORTS

Function: Reads and returns a single 32-bit data word from the Direction Register.
Input: None
Output: Stuct - PAR_GPIO_TYPE
Notes:

```
typedef struct _PAR_GPIO_TYPE {
        PortType    PORT1;              // Set Port to UART or Parallel
        PortType    PORT2;              // Set Port to UART or Parallel
        PortType    PORT3;              // Set Port to UART or Parallel
        PortType    PORT4;              // Set Port to UART or Parallel
        PortType    PORT5;              // Set Port to UART or Parallel
        PortType    PORT6;              // Set Port to UART or Parallel
        PortType    PORT7;              // Set Port to UART or Parallel
        PortType    PORT8;              // Set Port to UART or Parallel
} PAR_GPIO_TYPE, * PPAR_GPIO_TYPE;
```

PAR_GPIO_SET_MINTEN

Function: Enable or Disable Master Interrupt Enable for Parallel Port operation
Input: Struct - PAR_GPIO_MINT_EN
Output: None
Notes: The user app must call IOCTL_PAR_GPIO_GET_AND_CLEAR_ISR_STATUS after enabling master interrupts to clear out any older unprocessed interrupt status bit. No effect on UART interrupts. Separate enables for Rising, Falling, Level types. ISR returns with Rising and Falling status captured and cleared plus interrupts returned to the enabled state. Level based interrupts are left not enabled requiring the user to re-enable when the IO is in the correct pre-trigger state.

```
typedef struct _PAR_GPIO_MINT_EN {
        // TRUE = Enabled, Default is Disabled
        BOOLEAN  MasterCosRintEn; // Master for Rising Interrupts
        BOOLEAN  MasterCosFintEn; // Master for Falling Interrupts
        BOOLEAN  MasterCosLintEn; // Master for Level Interrupts
} PAR_GPIO_MINT_EN, * PPAR_GPIO_MINT_EN;
```

PAR_GPIO_SET_DATA_OUT

Function: Writes a single 32-bit data-word to the Data Transmit Register
Input: ULONG
Output: None
Notes: IOCTL_PAR_GPIO_SET_DIR must also be set to make this value the output value.

PAR_GPIO_GET_DATA_OUT

Function: Reads and returns a single 32-bit data word from the Data Register.
Input: None
Output: LONG
Notes: This is the register read-back and will match the SET data. Use Direct or Filtered registers to obtain the state of the IO.

PAR_GPIO_SET_DIR

Function: Writes a single 32-bit data-word to the Direction Register
Input: ULONG
Output: None
Notes: Setting a '1' in this register will make this bit an output. Setting a '0' will make the bit an input.

PAR_GPIO_GET_DIR

Function: Reads and returns a single 32-bit data word from the Direction Register.
Input: None
Output: ULONG

PAR_GPIO_SET_POL

Function: Writes a single 32-bit data-word to the Polarity Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit will be inverted. This only affects input data, not output data. See the Filtered Data registers.

PAR_GPIO_GET_POL

Function: Reads and returns a single 32-bit data word from the Polarity Register.
Input: None
Output: ULONG
Notes:

PAR_GPIO_SET_EDGE_LEVEL

Function: Writes a single 32-bit data-word to the EdgeLevel Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit will be treated as edge sensitive. Only affects input side data, not the driven data. For each bit cleared, the data is treated as level sensitive.

PAR_GPIO_GET_EDGE_LEVEL

Function: Writes a single 32-bit data-word to the EdgeLevel Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit will be treated as edge sensitive. Only affects input side data, not the driven data. For each bit cleared, the data is treated as level sensitive.

PAR_GPIO_SET_INT_EN

Function: Writes a single 32-bit data-word to the Interrupt Enable Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit the associated interrupt will be enabled. Used for both Level and Edge defined processing. See Rising and Falling for additional options.

PAR_GPIO_GET_INT_EN

Function: Reads and returns a single 32-bit data word from the Interrupt Enable
Input: None
Output: ULONG
Notes:

PAR_GPIO_READ_DIRECT

Function: Reads and returns a single 32-bit data word from the IO port.
Input: None
Output: ULONG
Notes: Direct data is synchronized but not filtered in any way. Get the state of the IO (whether defined as output or input).

PAR_GPIO_READ_FILTERED

Function: Reads and returns a single 32-bit data word from the IO port after
Input: None
Output: ULONG
Notes: Direct data is synchronized but not filtered in any way. Get the state of the IO (whether defined as output or input).

PAR_GPIO_SET_COS_RISING_STAT

Function: Writes a single 32-bit data-word to the Rising Status Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding bit in the Rising Status Register is cleared. If interrupts are being used, the COS Rising value will be captured and the register bits will be automatically cleared in the interrupt service routine. The value can be retrieved with the IOCTL_PAR_GPIO_GET_AND_CLEAR_ISR_STATUS.

PAR_GPIO_GET_COS_RISING_STAT

Function: Reads and returns a single 32-bit data word from the Rising Status Register.
Input: None
Output: ULONG
Notes: When an IO bit programmed as Edge and Rising transitions from low to high the status bit is set. If the corresponding Interrupt Enable is also set an interrupt is generated. Clear by writing back with the bit(s) set. If interrupts are being used, the COS Rising value will be captured and the register bits will be automatically cleared in the interrupt service routine. The value captured can be retrieved with the IOCTL_PAR_GPIO_GET_AND_CLEAR_ISR_STATUS.

PAR_GPIO_SET_COS_FALLING_STAT

Function: Writes a single 32-bit data-word to the Falling Status Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding bit in the Falling Status Register is cleared. If interrupts are being used, the COS Falling value will be captured and the register bits will be automatically cleared in the interrupt service routine. The value captured can be retrieved with the IOCTL_PAR_GPIO_GET_AND_CLEAR_ISR_STATUS.

PAR_GPIO_GET_COS_FALLING_STAT

Function: Reads and returns a single 32-bit data word from the Falling Status Register.
Input: None
Output: ULONG
Notes: When an IO bit programmed as Edge and Falling transitions from High to Low the status bit is set. If the corresponding Interrupt Enable is also set an interrupt is generated. Clear by writing back with the bit(s) set. If interrupts are being used, the COS Falling value will be captured and the register bits will be automatically cleared in the interrupt service routine. The value captured can be retrieved with the IOCTL_PAR_GPIO_GET_AND_CLEAR_ISR_STATUS.

PAR_GPIO_SET_COS_RISING_EN

Function: Writes a single 32-bit data-word to the Rising Enable Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding IO bit is enabled to be captured for rising edge transitions.

PAR_GPIO_GET_COS_RISING_EN

Function: Reads and returns a single 32-bit data word from the Rising Enable Register.
Input: None
Output: ULONG
Notes: Register read, will match current register value.

PAR_GPIO_SET_COS_FALLING_EN

Function: Writes a single 32-bit data-word to the Falling Enable Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding IO bit is enabled to be captured for falling edge transitions.

PAR_GPIO_GET_COS_FALLING_EN

Function: Reads and returns a single 32-bit data word from the Falling Enable Register.
Input: None
Output: ULONG
Notes: Register read, will match current register value.

PAR_GPIO_SET_HALFDIV

Function: Writes a single 32-bit data-word to the Rising Enable Register
Input: ULONG
Output: None
Notes: Write to this register to define divider to apply to COS reference clock selected. COS clock is Reference / 2N where N= 16 bits. Set upper bits to 0.

PAR_GPIO_GET_HALFDIV

Function: Reads and returns a single 32-bit data word from the HalfDiv Register
.***Input:*** None
Output: ULONG
Notes: Write to this register to define divider to apply to COS reference clock selected. COS clock is Reference / 2N where N= 16 bits. Set upper bits to 0.

PAR_GPIO_SET_TERM

Function: Writes a single 32-bit data-word to the Termination Register
.***Input:*** ULONG
Output: None
Notes: Setting a '1' in this register will terminate this bit. Setting a '0' will disable termination on this bit. Normally, bits programmed as inputs are terminated. Check your system design as the termination may be supplied in the cable.

PAR_GPIO_GET_TERM

Function: Reads and returns a single 32-bit data word from the Termination Register.
.***Input:*** None
Output: ULONG
Notes:

Open()

UART

All ioctl, read, write, and close operations for UART ports use the file descriptor (fd) returned from a standard open() system call. This open() must be passed the UART device node (e.g., "/dev/deUart_n"). The only supported flag during open is O_NONBLOCK, which enables non-blocking I/O behavior.
However, the effective blocking behavior can be controlled using the DE_CONFIG_PT ioctl, by setting the blocking_to parameter in the provided configuration structure. This allows the driver to block internally for a defined timeout even when O_NONBLOCK is used. This behavior is tied to the dyneng_UART_open() function, defined in the driver's file_operations struct for UART devices.

Parallel

The Parallel port is opened similarly using the open() system call with the appropriate device node (e.g., "/dev/deGPIO_8").
Unlike UART, the Parallel port driver does **not** implement read() or write() operations. Instead, interaction is handled via ioctl calls defined in dyneng_PAR_ioctl().

Close()

For both UART and Parallel devices, the close() system call is used to release the file descriptor returned by open(). This is handled in the driver via the release function in their respective file_operations structures (dyneng_UART_release or dyneng_PAR_release).

Read() and Write()

UART

Data is transmitted or received using the standard Linux read() and write() system calls. These calls interface with the hardware via the driver's dyneng_read() and dyneng_write() functions. The maximum buffer size for a read or write is constrained by the DE_MAX_FRAME macro, and potentially limited further based on the current **mode of operation** selected through configuration.
If errors occur during read or write operations, it is recommended to inspect the kernel log using dmesg to identify the cause. The driver includes detailed debug messages, especially useful when debug or trace modes are enabled.

Parallel

The Parallel port does not support read() or write() operations. All data exchange or configuration must be performed through ioctl calls.

File Operations Structs

```
struct file_operations dyneng_UART_fops = {
   .owner              = THIS_MODULE,
   .unlocked_ioctl     = dyneng_ioctl,
   .read               = dyneng_read,
   .write              = dyneng_write,
   .open               = dyneng_UART_open,
   .release            = dyneng_UART_release,
};


struct file_operations dyneng_PAR_fops = {
   .owner              = THIS_MODULE,
   .unlocked_ioctl     = dyneng_PAR_ioctl,
   .open               = dyneng_PAR_open,
   .release            = dyneng_PAR_release,
};
```

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.

UserAp Installation

To install the UserAp, create and move to a build directory inside the UserAp folder using the following command.

"mkdir build"

"cd build"

Geneate the Makefile by calling cmake on the UserApp directory containing the file CMakeLists.txt.

"cmake .."

Create the executable file by calling.

"sudo make:

Upon completion an executable called "biserialVIUart_user_app_cpp" will be present inside the build directory.

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 B & C
Santa Cruz, CA 95060
831-457-8891
support@dyneng.com