DYNAMIC ENGINEERING

150 DuBois, Suite C

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

Parallel-TTL-GPIO

Linux Documentation

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

Revision 01p1 7/19/21
Corresponding Hardware: Revision 02+
PMC 10-2007-0102, XMC 10-2012-0902
FLASH 0101

Parallel-TTL-GPIO
Linux Device Drivers for
PMC-Parallel-TTL-GPIO
XMC-Parallel-TTL-GPIO

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

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.

This product has been designed to operate with PMC/XMC carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.

Table of Contents

Introduction

The ParTtlGpio driver was developed on Ubuntu 18.04 with version 5.4.0-74-generic kernel.

PMC-Parallel-TTL-GPIO and XMC-Parallel-TTL-GPIO are supported with the same driver interface. “Parallel-TTL-GPIO” features a Spartan6 Xilinx FPGA to implement the PCI interface, FIFOs, and IO processing, control and status for 64 discrete IO. Each IO is a single ended signal with programmable 3.3V or 5V reference [one for all IO]. There is a programmable PLL with four clock outputs. PLL or the Oscillator can be used as the reference for the COS clock divider. Many COS frequencies are user selectable in this manner. An unusual feature is a standalone FIFO [8Kx32] with DMA in and out. The memory doesn’t “do anything” since it is not currently attached to input or output data. It is available for user purposes and to support future requirements.

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 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 is delivered with multiple example-tests. At the end of the UserAp menu is an item “Print Registers”. When executed – select the appropriate “test” number in the menu – the current contents of the registers are displayed. The structures for the Base and Base 1 registers are shown with the structure selection and current status. The remainder are shown as hex numbers. Easy way to check if GPIO is set-up the way you think it is.

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-TTL-GPIO or XMC-Parallel-TTL-GPIO user manual as appropriate (also referred to as the hardware 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 and install and uninstall scripts which install/uninstall the driver and create a device node(s).

de_ParTtlGpio.h.h and de_common.h are the C header files that define the Application Program Interface (API) for the ParTtlGpio driver. These files are required at compile time by any application that wish to interface with the driver and for compiling the driver.

NOTE: With the install script the driver (the .ko file) is not copied into the Linux Kernel tree and therefore will need to be reinstalled after reboot.

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 Parallel-TTL-GPIO 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. Type is set to 1 or 2 to show if PMC or XMC respectively.

```
// Board information
typedef struct de_rev {
  uint8_t       des_type;
  uint8_t       des_major;
  uint8_t       des_minor;
  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 the board.
Input: de_port_cfg_t structure
Output: de_port_cfg_t structure
Notes: This ioctl is used to configure the boards primary settings. As with the PLL above, this requires the de_op_t to say if the configuration is being read or written.

Note to use Read() or Write() calls the associated rd/wr_dma_enabled must be on as read and write use DMA by default.

```
// Board information
typedef struct de_port_cfg {
  de_op_t             op;
  unsigned long       blocking_to;  //if in non-blocking user to pick timeout in milliseconds
  unsigned long       master_int_on; //turns master int bit on/off
  unsigned long       half_div;
  unsigned long       use_pll; //user can set to true to use pll otherwise will use oscillator
  unsigned long       rd_dma_enabled; //must be enabled to use read functionality
  unsigned long       wr_dma_enabled; //must be enabled to use write functionality
  unsigned long       master_tx_enabled;
  unsigned long       vio_sel;
} de_port_cfg_t;
```

DE_CONFIG_GPIO

Function: Configures the GPIO pins on the device
Input: de_tty_gpio_cfg_t structure
Output: de_tty_gpio_cfg_t structure
Notes: This ioctl is used to configure the GPIO pins with a single ioctl. As with the PLL above, this requires the de_op_t to say if the configuration is being read or written. Each GPIO pin is divided into two 32-bit values.

```
// Board information
typedef struct de_tty_gpio_cfg {
  de_op_t      op;
  unsigned int  data_en_0;
  unsigned int  data_en_1;
  unsigned int  polarity_0;
  unsigned int  polarity_1;
  unsigned int  edge_level_0;
  unsigned int  edge_level_1;
  unsigned int  interrupt_en_0;
  unsigned int  interrupt_en_1;
  unsigned int  rising_edge_capture_0;
```

```
  unsigned int   rising_edge_capture_1;
  unsigned int   falling_edge_capture_0;
  unsigned int   falling_edge_capture_1;
  unsigned int   cos_rising_edge_capture_0;
  unsigned int   cos_rising_edge_capture_1;
  unsigned int   cos_falling_edge_capture_0;
  unsigned int   cos_falling_edge_capture_1;
  unsigned int   io_data_sync_unfiltered_0;
  unsigned int   io_data_sync_unfiltered_1;
  unsigned int   io_data_sync_polar_masked_0;
  unsigned int   io_data_sync_polar_masked_1;
} de_tty_gpio_cfg_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 5 elements, the first is the op code, the second is the base address (which for this card is always the enum value DE_REG_BASE from de_reg_off_t, but for cards with multiple ports may be another value. The third value is the value that is either read or to be written, the fourth value is the offset for the register (Here you can use the #defines from the de_ParTtlGpio.h header file, such as #define PAR_TTL_GPIO_BASE). The final element can be used to do a RMW mask.

```
typedef struct de_reg_cmd {
    de_op_t            op;
    de_reg_off_t       base; //base register for this card is always DE_REG_BASE
    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;
```

DE_RESET_IO

Function: Resets the IO and clears all FIFOs
Input: None
Output: None
Notes:

DE_WB_STATUS

Function: This ioctl attempts to clear any interrupts by writing back the values in the status register if (for some reason) the ISR did not clear the interrupts.
Input: None
Output: None
Notes: This is not normally used (it is primarily for test purposes).

DE_FORCE_INT

Function: This triggers the force interrupt.
Input: None
Output: None
Notes: This is used for testing purposes, but could be useful if wanting to test interrupts being triggered.

DE_SET_MAST_INT

Function: Allows user to set the master int reg
Input: Unsigned Int
Output: None
Notes: This can be used to set the master int register to 0x00 or 0x01 so that you can control the whether the device can trigger interrupts on the system.

Note: when using read/write calls the interrupt is automatically enabled during those calls as interrupts are required for DMA).

DE_SET_DATA_0

Function: Allows user to set the data_out_0 register for the GPIO
Input: Unsigned Long
Output: None
Notes: This can be used to set the lower 32-bit value of the data out registers for the GPIO

DE_SET_DATA_1

Function: Allows user to set the data_out_1 register for the GPIO
Input: Unsigned Long
Output: None
Notes: This can be used to set the upper 32-bit value of the data out registers for the GPIO

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/ de_ParTtlGpio_0”). The only configuration used in the open call that is supported is the O_NONBLOCK. if O_NONBLOCK is used the default timeout for read/write calls is 1 second, but this can be configured in the 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().

Write

Parallel-TTL-GPIO DMA data is written to the device (and out the port) using the Linux write() system call. Writes are executed using the standard Linux function write() and passing in the file descriptor to the device opened with open(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes.

Read

Parallel-TTL-GPIO DMA data is read from the device using the standard Linux read() command, and passing in the file descriptor to the device opened with open(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes.

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