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

ip_gen

Linux Driver Documentation

Revision A

ip_gen
Linux Device Driver for an
Unknown IP Module

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

Table of Contents

Introduction

The ip_gen driver is a modular Linux driver for an unknown Industry Pack (IP) module. This driver can control any IP module by mapping the IO, MEM, and INT memory spaces so that they can be accessed by driver calls. When the IP carrier driver is loaded, it enumerates its IP bus by reading the ID proms of each installed IP, entering the results into the IP_CHILD_CONTEXT structure and cross-referencing this with the ip_device_table array in ip_device_table.h. The carrier driver makes the device type information available to both the user and the IP load script through the sys file system (/sys/bus/ip/devices/ip_***x_y***/devtype) where ***x*** is the zero-based carrier number and ***y*** is the zero-based ip carrier slot number. If no match is found for the data read or if a match is found, but the driver_exists field in the table is zero, the fields in IP_CHILD_CONTEXT are all set to zero and the ip_gen driver will be installed. When the ip_gen driver initializes, it checks the IP_CHILD_CONTEXT structure in each ip_device structure found. If all fields are zero, it has found an ip_gen device and will complete initialization and enter a pointer to the device structure into its device table. To access the hardware, a device node must be created in the /dev directory for each installed module. This is accomplished by the load_ip driver load script. A separate handle to each unknown IP can be obtained using open() calls (see below). IO Control calls, ioctl()'s are used to configure the module and write/read data to/from the IO space. Write and read calls are used to move data blocks in and out of the IP module MEM space.

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 hardware manual for the particular device being used.

Driver Installation

The source files for both the driver (ip_gen) and a test application for controlling a parallel TTL IP module (ip_gen_par_ttl_user_app) are provided in the driver delivery. Makefiles in each directory build and install the driver and test application with 'make' and 'make install' commands. A 'make clean' command will remove all executables and object files. Copy the source file directory tree to a directory where the driver and application code will be built and run ***make*** in the appropriate directories to build the driver and test application. Run ***make install*** in each directory to copy the ip_gen.ko file to the /lib/module/*version*/kernel/driver/misc/ directory and the test executable to the /usr/local/bin/ directory.

Load and unload scripts are provided to facilitate starting and stopping the carrier and IP module drivers. The load_ip script will load the ip carrier driver module, explore the /sys/bus/ip/devices/ device tree to discover which IP modules are installed, parse the /proc/devices file for the devices major numbers and create the required number of device nodes for the carrier. Then it loads the ip_gen driver and creates the appropriate number of ip_gen_***x*** device nodes as well (where ***x*** is the zero-based device number). Similarly the unload_ip script first unloads the installed IP module driver(s), and then unloads the carrier driver(s). These scripts and the ip_car_list and ip_driver_table files that list the possible ip carriers and correlate IP device names with driver names should be copied to /usr/local/bin/.

Driver Startup

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

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

Below is example code for opening a handle for unknown IP device ***dev_num***

```
#define NUM_DEVICES    5
#define HANDLE         long

HANDLE         hip_gen;
char           Name[INPUT_SIZE];
int            i, dev_num;

do
{
   printf("\nEnter target board number (starting with zero): \n");
   scanf("%d", &dev_num);
   if(dev_num < 0 || dev_num > NUM_DEVICES)
      printf("\nTarget board number %d out of range!\n", dev_num);
}
while(dev_num < 0 || dev_num > NUM_DEVICES);

sprintf(Name, "/dev/ip_gen_%d", dev_num);

hip_gen = open(Name , O_RDWR);

if(hip_gen < 2)
{
   printf("\n%sFAILED to open!\n", Name);
   return 1;
}
```

IO Controls

The driver uses ioctl() calls to configure the device and ascertain status information. The parameters passed to the ioctl() function include the handle obtained from the open() call, an integer command defined in the ip_gen_api.h file and an optional parameter used to pass data in and/or out of the device. The ioctl commands defined for the ip_gen driver are listed below.

IOCTL_IP_GEN_GET_INFO

Function: Returns the current driver version, instance number and location string.
Input: None
Output: DRIVER_IP_GEN_DEVICE_INFO structure
Notes: This call does not access the hardware, only driver parameters. Instance number is the zero-based count of ip_gen devices. The location string is a concatenation of the carrier name and IP slot number e.g. an IP in slot C on the second PCI3IP installed would have a location string of pci3ip_1_2. See ip_gen_api.h for the definition of DRIVER_IP_GEN_DEVICE_INFO.

IOCTL_IP_GEN_SET_MEM_OFFSET

Function: Sets the address offset into the MEM space for the next read or write call.
Input: offset (unsigned long integer)
Output: None
Notes: Sets the address offset into the IP MEM space for read and write operations.

IOCTL_IP_GEN_GET_MEM_OFFSET

Function: Returns the address offset into the MEM space.
Input: None
Output: offset (unsigned long integer)
Notes: Returns the value set in the previous call.

IOCTL_IP_GEN_PUT_IO_DATA

Function: Writes data to the IP IO space.
Input: IO_ACCESS structure
Output: None
Notes: The IO_ACCESS structure has three fields: Address, Data and Length. Address is the offset into the mapped IO space for the module, Data is the value to write to that address and Length is the size of the data (1, 2 or 4 bytes). If Length is set to 4, the carrier will perform two successive 16-bit writes to consecutive 16-bit addresses otherwise the a single access is made.

IOCTL_IP_GEN_GET_IO_DATA

Function: Reads data from the IP IO space.
Input: IO_ACCESS structure
Output: IO_ACCESS structure
Notes: The Address and Length fields must be filled in prior to making this call. When the structure is returned, the Data field will contain the data value read at the requested address.

IOCTL_IP_GEN_SET_IP_CONTROL

Function: Sets the configuration of the IP slot.
Input: Register configuration (unsigned long integer)
Output: None
Notes: Controls the IP clock speed IP data controls and IP interrupt enables for the IP slot that the board occupies. See the bit definitions in the ip_gen_api.h header file for information on the register controls.

IOCTL_IP_GEN_GET_IP_CONTROL

Function: Returns the configuration of the IP slot.
Input: None
Output: Register configuration (unsigned long integer)
Notes: Returns the slot configuration register value for the IP slot that the board occupies. See the bit definitions in the ip_gen_api.h header file for information on decoding this value.

IOCTL_IP_GEN_SET_BASE_CONFIG

Function: Sets configuration parameters in the IP-QuadUART base control register.
Input: IP_GEN_BASE_CONFIG structure
Output: None
Notes: Selects the reference clock source(s) for the UART channels and the bus timeout interrupt enable state. See ip_gen_api.h for the definition of IP_GEN_BASE_CONFIG.

IOCTL_IP_GEN_WAIT_ON_INTERRUPT

Function: Causes an entry to be placed in the interrupt wait queue.
Input: Delay value to wait in jiffies.
Output: None
Notes: This call is used to implement a user defined interrupt service routine. It will return when an interrupt occurs or when the delay time specified expires. If the delay is set to zero, the call will wait indefinitely. The delay time is dependent on the platform setting for jiffy, which could be anything from 10 milliseconds to less than 1 millisecond.

IOCTL_IP_GEN_ENABLE_INTERRUPT

Function: Enables the IP slot interrupts.
Input: None
Output: None
Notes: Sets both of the IP slot interrupt enables, leaving all other bit values in the IP slot control register the same. This call is used in the user interrupt processing function to re-enable the interrupts after they were disabled in the driver interrupt service routine. This allows that function to enable the interrupts without knowing the particulars of the other configuration bits.

IOCTL_IP_GEN_DISABLE_INTERRUPT

Function: Disables the IP slot interrupts.
Input: None
Output: None
Notes: Clears both of the IP slot interrupt enables, leaving all other bit values in the IP slot control register the same. This call is used when interrupt processing is no longer desired.

IOCTL_IP_GEN_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted for the IP slot. This call is used for development, to test interrupt processing.

IOCTL_IP_GEN_GET_ISR_STATUS

Function: Returns the InterruptStatus, InterruptVector, and TimedOut status read in the last interrupt service routine.
Input: None
Output: IP_GEN_INT_STAT structure
Notes: The InterruptStatus field contains information on the interrupt source IP_INT0, IP_INT1, Force Interrupt or Bus Error from the last driver interrupt service routine execution. The interrupt vector read in the last ISR is also returned as well as the time-out status. See ip_gen_api.h for the definition of IP_GEN_INT_STAT and information on the InterruptStatus bits.

write

Data to be written to the IP MEM space uses a write() call starting at the memory offset specified with the SET_MEM_OFFSET call. The user supplies the device handle, a pointer to the buffer containing the data, and the number of bytes to write. The driver takes advantage of the carrier’s 32-bit double-write capability to load two 16-bit IP bus words with a single PCI write until less than four bytes remain in the buffer, which are completed with byte writes.

read

Data to be read from the IP MEM space uses a read() call starting from the memory offset specified with the SET_MEM_OFFSET call. The user supplies the device handle, a pointer to the buffer to store the data in, and the number of bytes to read. As with the write call above the driver takes advantage of the carrier 32-bit double-read capability to read two 16-bit IP bus words with a single PCI read until less than four bytes remain to be read, which are completed with byte reads.

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
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891
(831) 457-4793 fax
support@dyneng.com