Linux Driver Manual

SpaceWire Monitor

Manual Revision 01p1
Revision Date 01/02/2023
Corresponding Hardware
Current Fab Number

Dynamic Engineering
150 DuBois St. Suite B/C
Santa Cruz, CA 95060
(831) 457-8891
www.dyneng.com
sales@dyneng.com
Est. 1988

SpaceWire Monitor

Cautions and Warnings

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 their 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 express written approval from the president of Dynamic Engineering.

Connection of incompatible hardware is likely to cause serious damage.

Table of Contents

Figures

No table of figures entries found.

Tables

Design Revision History

Table 1: Design Revision History

Revision	Date	Description

Manual Revision History

Table 2: Manual Revision History

Revision	Date	Description

NOTE: Dynamic Engineering has made every effort to ensure that this manual is accurate and complete; that being said, the company reserves the right to make improvements or changes to 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.

Product Description

The SpaceWire-Monitor (Monitor) is a 2-channel device used to monitor any two SpaceWire devices using two standard SpaceWire cables. This driver was developed as a Linux Kernel Driver using the standard Linux driver API (specifically developed on a 5.4.0-131-generic kernel version.

Each channel has a 64K internal FIFO plus a 512K external FIFO. While it is possible to write/read from the FIFO directly with the API, the driver implementation does require DMA capabilities upon loading, and standard read call utilizes DMA. Furthermore, as this device is a monitoring device, write functionality is not implemented (other than by using an IOCTL to write to the FIFO for test purposes).

Software Description

The Dynamic Engineering SpaceWire Monitor and SpaceWire IO card utilize the same Linux driver, version 1.1.4 or greater. The two kernel modules are built via compile time definitions contained within the Makefiles and driver for each of these kernel modules.

This package comes with a Monitor Test Suite (Test Suite), a basic data logger application (Logger), and a binary converter (Converter). All applications depend on the following files:

Table 3: Header Files

File Name	Description
de_Common.h	Defines several default loctl calls and data structures. It is shared with most of our Linux drivers.
de_SpwrCfg.h	Defines many of the data structures used by the loctl calls.
de_SpwrDrv.h	Provides register map, custom ioctl commands, and key data structures.

Test Suite

The Test Suite is used in-house to validate the hardware and provides a more extensive example of how to interact with the hardware. The Test Suite was written in C/C++ while the Logger is written in C. Like the logger described below, it can be used to log data and does so in binary form. The binary form allows disk writes to keep up with the speed of the data received by the device by coupling it with multithreading the writes. Importantly, it is assumed in the code that fwrite () is an atomic instruction and will not result in a race condition. Using the Test Suite, the user can select tests 8-11; the user will be prompted to enter file path or press enter for the default (using a custom file path allows writing to different disks).

When testing the functionality of the Monitor with the data tests (0-3), the Test Suite also assumes that the data coming over the wire matches the data sent by the SpaceWire testing app [this can be seen in tests.cpp **CreateComparisonData**(**uint32_t packet_len**), which mirrors the **de_loApp.c** data creation].

Logger

The Logger can be invoked for each channel to be monitored in separate terminal windows. The application continuously issues packet reads until interrupted via <ctrl-c>, or if any key is pressed. Packet data is written to a binary file, and each packet is preceded by a packet header, which contains a time-stamp, packet number, and the size of the packet. A binary to text conversion application is provided to convert the binary file to human readable format.

An optional third parameter, **disable_packet_mode**, allows Monitor to collect data ignoring packet boundaries. There are two reasons to execute in this mode. If channel being monitored is configured to operate with packet mode disabled, Monitor must execute in same mode, or no data will be captured. Running Monitor with packet mode disabled may enable the debug of improperly terminated packets assuming Tx data patterns are known.

Binary Converter

As both the Test Suite and the Logger create binary files (for both speed and disk space saving reasons), the files need to be converted to human-readable formats. This can be done using the binary converter software provided by **de_BinToTxt.c** by compiling the code and running the following command:

./dyn_bcnvrt <file to convert> <destination file>

Application Notes

The logging application has been tested with both channels operating at full rate (up to line rate of 200 MHz.) with packets of various sizes without packet drop. As the Monitor channel is not participating in flow control, the Monitor's FIFOs can possibly overrun if the host computer HDD or SSD is not fast enough to keep up with the packet receive rate. Depending on the customer packet traffic, a faster HDD or SSD may be required to ensure no packets are dropped.

In our full rate testing, we have determined that an HDD of 7200 RPN with a 32 MB cache or NVMe internal SSD will enable capture of back-to-back packets of any size (up to 256K) without packet drop.

Driver Installation

As our customers often use various implementations of the Linux kernel (including custom kernels), we do not provide a pre-built kernel driver binary for loading. Instead, we provide a make file that will, in most systems, build a kernel driver binary (.ko) file. To build the binary for the Monitor:

1. Navigate to build director
2. Command: **sudo make TYPE=mon**

NOTE: If the TYPE is not specified, a standard space-wire driver is built.

Once the driver has been built, the next step is to use the script provided to install the driver.

3. Command: **sudo ./Install_swMon**

The script should output a device number (e.g., 238). This script also creates the following device nodes to connect to software applications:

- **/dev/deSpwrMon_0**
- **/dev/deSpWrMon_1**

NOTE: The installation script does not permanently install the device driver. Again, as our clients all have very different implementations on Linux, we have found it more beneficial to keep the installation process simple and allow clients to customize how they would like to administer their third-party drivers.

If you wish to install the driver permanently for Ubuntu, you must look at the kernel you are using (uname -r), then, place the .ko file in:

- **/lib/modules/<your-kerne>/kernel/drivers/** and add edit the config file at:
- **/etc/modules-laod.d/modules.conf** to load your specific driver.

Driver Startup

Once the driver has been installed, connecting to the device is fairly simple from software. Indeed, one only needs to call the standard system command open() while passing the device node name:

- ("**dev/deSpWrMon_<x>"** and the **O_RDWR flag**)

This will return the file descriptor for that device (for an example see: **de_SpwrMonApi.cpp**).

Software Quick Start

Once the driver is installed and the software has been connected to the device using the standard open() system call, using the device is fairly straight forward. The driver follows a basic model of Open, Configure, Read/Write, Close.

To configure the device, the driver uses the ioctl command with the **DE_CONFIG_PT** flag and a **de_spwr_cfg_t** struct (defined in **de_SpwrCfg.h**). Within this struct, the user can set the op code (designating if the configuration is being read back or written to the device), the mode, the blocking timeout, the DMA priority, the mode (packet or not), and the auto_start (this is used to determine if, at any time of reading data, data is already being transmitted or not on the wire). For an example of use, see tests.cpp TestMonitoring() routine.

Once the device is configured, data can be extracted from the device using the standard read() system call. The key, however, is to make sure the buffer is large enough for the packet plus the header. Alternatively, you can set the read to read the max_packet size, and it will return with when the read either times out or a next packet received cannot fit in the buffer. Data in the buffer always follows the format [Header][Data]….[Header][Data]. Here the header is an array of 4x32bit unsigned integers, with the header[3] being the length data to follow. An example of parsing packets can be seen in: de_spwrMonitorApi.cpp AddPacketsToPacketList() routine.

IO Controls

In Addition to the above, the drivers use a few other IO Control calls (IOCTLs) to access the device. These are defined in:

- **de_SpwrDrv.h** and
- **de_Common.h**

They can be used by calling the system call: **iocIt()**

```
int ioctl(
  int          fd,       // file descriptor returned from open()
  unsigned long  request,    // Control code defined in API header file
  …,                  // pointer to structure, when required
);
```

IOCTL Definitions

DE_GET_BD_INFO

- Functions: Gets the relevant board information for each port. This IOCTL takes a struct pointer as a parameter and then fills it.
- Input: **de_spwr_rev_t ***
- Output: **de_spwr_rev_t ***
- Notes:

```
typedef struct de_spwr_rev {
    /* FPGA design ID and revision          */
    uint8_t     des_major;
    /* Only applicable to BK versions, always 0
    *  for legacy versions                  */
    uint8_t     des_minor;
    uint8_t     des_type;
    /* bits 7-0 reflect user switch settings    */
    uint8_t     dips;
    uint16_t    max_speed[DE_NUM_SPWR_PTS];
    uint32_t    rx_fifo_len[DE_NUM_SPWR_PTS];
} __attribute__((packed)) de_spwr_rev_t;
```

DE_CONFIG_PT

- Function: Gets/Sets the port configuration
- Input: **de_spwr_cfg_t ***
- Output: **de_spwr_cfg_t ***
- Notes:

```
/* Ioctl/configuration operations */
typedef enum de_op {
   DE_GET_OP = 0,
   DE_SET_OP = 1,
   /* Read/Modify/Write used for register
   *  ioctls                       */
   DE_RMW_OP = 2,
} de_op_t;

/* Ioctl/configuration operations */
typedef enum de_op {
   DE_GET_OP = 0,
   DE_SET_OP = 1,
   /* Read/Modify/Write used for register
   *  ioctls                       */
   DE_RMW_OP = 2,
} de_op_t;

/*  DMA priorities for accessing
*   I/O FIFOs   */
typedef enum de_dma_pri {
   /* Default, R/W same priority   */
   DE_SAME_PRI = 0,
   /* DMA Reads prioritized        */
   DE_RD_PRI = 1,
```

```
    /* DMA Writes prioritized       */
    DE_WR_PRI =

/* Operating parameters per SpaceWire port                    */
typedef struct de_spwr_cfg {
    /* Of type de_opt_t                        */
    uint32_t        op;
    /* Of type de_opt_t, Wrap back or External I/O  */
    uint32_t        mode;
    /* Timeout in seconds for blocking reads
    *   If 0, then block forever    */
    uint32_t        blocking_rd_to;
    /*This stores the original number before conversion to jiffies */
    uint32_t        blocking_rd_to_sec_stored;
    /* Timeout in seconds for time code ioctl reads
    *  interrupt driven, if 0, block forever        */
    uint32_t        tc_rd_to;
    /* Of type de_dma_pri_t                      */
    uint8_t         dma_pri;
    /* Disable packet mode
    *  If this mode is utilized both transmit and receive ports must
    *  be operating in this mode                                  */
    uint8_t         disable_pkt_mode;
    /* Set to enable auto link start, this allows
    *  other end of link to initiate link start
    *  Note, this bit is overloaded when operating in monitor mode
    *  0 = Link down on start, 1 = link up on start               */
    uint8_t         auto_start;
     } __attribute__((packed)) de_spwr_cfg_t;
```

DE_GET_STATS

- Function: Gets Stats for the port
- Input: **de_spwr_stats_t ***
- Output: **de_spwr_stats_t ***
- Notes:

```
/* Stats maintained for each port */
typedef struct de_spwr_pt_stats {
    /* Current Link state, of de_lnk_stat_t     */
    uint32_t        cur_lnk_stat;
    /* Maximum rx or tx throughput */
    uint32_t        max_tput;
    uint32_t        purge_err_cnt;
    uint32_t        credit_err_cnt;
    uint32_t        esc_err_cnt;
    uint32_t        discon_err_cnt;
    uint32_t        parity_err_cnt;
    uint32_t        fifo_ovfl_cnt;
    uint64_t        rx_byte_cnt;
    uint64_t        tx_byte_cnt;
} __attribute__((packed)) de_spwr_pt_stats_t;
```

DE_REG

- Function: Gets/Sets specific register values at given level (port or base) and offset
- Input: **de_reg_cmd_t ***
- Output: **de_reg_cmd_t ***
- Notes:

```
typedef struct de_reg_cmd {
   /* DE_GET_OP, DE_SET_OP, DE_RMW_OP     */
   de_op_t        op;
   de_reg_off_t   base;
   /* Value read, or to be written        */
   unsigned int   val;
   /* Reg number or word offset from base  */
   unsigned int   reg;
   /* Mask if op is RMW                   */
   unsigned int   mask;
} de_reg_cmd_t;
```

Additional Software Information

Invocation of de_mon

./de_mon a | b 0 | 1

The first parameter specifies which channel to monitor (**a** for channel 0, **b** for channel 1). The second parameter specifies whether packets are currently being transmitted. 0 denotes packets are not being transmitted, 1 denotes packet transmission in progress.

./de_mon a | b 0 | 1 1

The last optional parameter disables packet mode. All data received by Monitor is captured irrespective of packet boundaries if enabled on channel being monitored.

The application will write packet data to a binary file named **monitor_a.dat** for channel a, or **monitor_b.dat** for channel b. The file will be resident in the same directory as the Monitor application. Upon completion of capture (application terminated via <ctrl-c>), the file can be converted to a text file by invoking the conversion application.

./ de_BinToTxt bin_file out_file

Command line view while monitor is running:

```
xxxx@dyneng1:~/Dyneng/de_SpWrMon$ ./dyn_mon a 0
File write len = 511K bytes
Configured /dev/deSpWrMon_0
Monitor a::cumm pkt bytes 532000000 wait_cnt 0
```

Notes: 1) cumm pkt bytes is the stored size of the binary file. 2) For this example, the Monitor application captured 1 million 512-byte packets.

Command line view after ^c stops monitor application which captured two 128 byte packets:

xxxx@dyneng1:~/Dyneng/de_SpWrMon$ ./dyn_mon a 0

```
File write len = 511K bytes
Configured /dev/deSpWrMon_0
^CMonitor a::cumm pkt bytes 296 wait_cnt 0
                Cur link state  = Link Down
                Rx byte cnt     = 256
                Tx byte cnt     = 0
                Rx packet cnt  = 2
                Rx max packets        = 1
                Rx max bytes  = 128
                Max loop        = 2
                Drop cnt        = 0
Closed file 0
Channel a cummulative byte count 296 wait_cnt 0
xxxx@dyneng1:~/Dyneng/de_SpWrMon$
```

Notes: 1) For this example the Monitor application captured two 128 byte packets. 2) Cumulative byte count is the bytes needed to store captured data and time stamps in binary format.

Text file result of two 128 byte packets captured by Monitor application. Text file is created by converting binary file monitor0_a.dat written by Monitor application to a readable text file (montior0_a.txt) using the ./dyn_bcnvrt program provided.

TS is time stamp of format sec.ns

```
TS:37107.968356917::Packet Number:0::Packet Length:128
a0 b0 c0 00 a0 b0 c0 01 a0 b0 c0 02 a0 b0 c0 03 a0 b0 c0 04 a0 b0 c0 05 a0 b0 c0 06 a0 b0
c0 07 a0 b0 c0 08 a0 b0 c0 09 a0 b0 c0 0a a0 b0 c0 0b a0 b0 c0 0c a0 b0 c0 0d a0 b0 c0 0e
a0 b0 c0 0f a0 b0 c0 10 a0 b0 c0 11 a0 b0 c0 12 a0 b0 c0 13 a0 b0 c0 14 a0 b0 c0 15 a0 b0
c0 16 a0 b0 c0 17 a0 b0 c0 18 a0 b0 c0 19 a0 b0 c0 1a a0 b0 c0 1b a0 b0 c0 1c a0 b0 c0 1d
a0 b0 c0 1e a0 b0 c0 1f

TS:37107.968436231::Packet Number:1::Packet Length:128
0d 0e 0f 00 0d 0e 0f 01 0d 0e 0f 02 0d 0e 0f 03 0d 0e 0f 04 0d 0e 0f 05 0d 0e 0f 06 0d 0e
0f 07 0d 0e 0f 08 0d 0e 0f 09 0d 0e 0f 0a 0d 0e 0f 0b 0d 0e 0f 0c 0d 0e 0f 0d 0d 0e 0f 0e
0d 0e 0f 0f 0d 0e 0f 10 0d 0e 0f 11 0d 0e 0f 12 0d 0e 0f 13 0d 0e 0f 14 0d 0e 0f 15 0d 0e
0f 16 0d 0e 0f 17 0d 0e 0f 18 0d 0e 0f 19 0d 0e 0f 1a 0d 0e 0f 1b 0d 0e 0f 1c 0d 0e 0f 1d
0d 0e 0f 1e 0d 0e 0f 1f
```

Command line view after enter stops monitor application with packet mode disabled which captured 256 bytes (channel a transmitted four, 64 byte packets):

Note: Packet count is not displayed when Monitor invoked with packet mode disabled.

```
xxxx@dyneng1:~/Dyneng/de_SpWrMon$ ./dyn_mon a 0 1
File write len = 511K bytes
Configured /dev/deSpWrMon_0
Monitor a::cumm pkt bytes 304 wait_cnt 0

                Cur link state    = Link Down
                Rx byte cnt     = 256
                Rx max bytes      = 128
```

```
                Max loop        = 3
                Drop cnt        = 0
Closed file 0
Channel a cummulative byte count 304 wait_cnt 0
```

Resulting text file when running with packet mode disabled (four 64 byte packets transmitted):

Note: First two packets, packet boundaries happened to be preserved, last two packets were combined into one.
Disregard packet numbers and boundaries in this mode, only inspect data.
TS is time stamp of format sec.ns

```
TS:5179.822099121::Packet Number:0::Packet Length:64
a0 b0 c0 00 a0 b0 c0 01 a0 b0 c0 02 a0 b0 c0 03 a0 b0 c0 04 a0 b0 c0 05 a0 b0 c0 06 a0 b0
c0 07 a0 b0 c0 08 a0 b0 c0 09 a0 b0 c0 0a a0 b0 c0 0b a0 b0 c0 0c a0 b0 c0 0d a0 b0 c0 0e
a0 b0 c0 0f

TS:5179.822099121::Packet Number:1::Packet Length:64
0d 0e 0f 00 0d 0e 0f 01 0d 0e 0f 02 0d 0e 0f 03 0d 0e 0f 04 0d 0e 0f 05 0d 0e 0f 06 0d 0e
0f 07 0d 0e 0f 08 0d 0e 0f 09 0d 0e 0f 0a 0d 0e 0f 0b 0d 0e 0f 0c 0d 0e 0f 0d 0d 0e 0f 0e
0d 0e 0f 0f

TS:5179.822099121::Packet Number:2::Packet Length:128
a0 b0 c0 00 a0 b0 c0 01 a0 b0 c0 02 a0 b0 c0 03 a0 b0 c0 04 a0 b0 c0 05 a0 b0 c0 06 a0 b0
c0 07 a0 b0 c0 08 a0 b0 c0 09 a0 b0 c0 0a a0 b0 c0 0b a0 b0 c0 0c a0 b0 c0 0d a0 b0 c0 0e
a0 b0 c0 0f 0d 0e 0f 00 0d 0e 0f 01 0d 0e 0f 02 0d 0e 0f 03 0d 0e 0f 04 0d 0e 0f 05 0d 0e
0f 06 0d 0e 0f 07 0d 0e 0f 08 0d 0e 0f 09 0d 0e 0f 0a 0d 0e 0f 0b 0d 0e 0f 0c 0d 0e 0f 0d
0d 0e 0f 0e 0d 0e 0f 0f
```

Debug

If an error is encountered while running the Monitor, please perform the following steps. This information is required by Technical Support to resolve any issues.

1) Execute modinfo de_SpwrDrv.ko
   Driver version as well as other information will be displayed.
   Note the version.
2) Execute dmesg
   This command will display error messages logged by the driver. Dynamic Engineering drivers are quite verbose logging errors and cause.
   Take a screen shot of the output.
3) Contact Technical Support with information collected in steps 1 and 2.

Warranty and Repair

Please refer to the warranty page on our website for the warranty and options that are currently offered.

www.dyneng.com/warranty

Service Policy

Before returning a product for repair, verify to the best of your ability, that the suspected unit is as fault. Then call the Dynamic Engineering Customer Service Department for a Return Material Authorization (RMA) number. Carefully package the product, in the original packaging if possible, and ship prepaid and insured with the RMA number clearly written on the outside of the package. Include a return address and the telephone number of a technical contact. For out-of-warranty repairs, a purchase order for repair charges must accompany the return. Dynamic Engineering will not be responsible for damages due to improper packaging of returned items. For service on Dynamic Engineering products not purchased directly from Dynamic Engineering, contact your reseller. Products returned to Dynamic Engineering for repair by anyone other than the original customer will be treated as out-of-warranty.

Out-of-Warranty Repairs

Out-of-warranty repairs will be billed on a material and labor basis. Customer approval will be obtained before repairing any item if the repair charges will exceed one half of the list price for one of that kind of unit. Return transportation and insurance will be billed as part of the repair in addition to the minimum RMA charge.

Contact:

Customer Service Department
Dynamic Engineering
150 DuBois St. Suite B/C
Santa Cruz, CA 95005
(831) 457-8891
support@dyneng.com

Glossary

Baud	Used as the bit period when talking about UARTs; Not strictly correct, but is the common usage when talking about UARTs.
CardID	Unique number assigned to a design to distinguish between all designs of a particular vendor
CFM	Cubic feet per minute
FIFO	First In First Out memory
Flash	Non-volatile memory used on Dynamic Engineering boards to store FPGA configurations or BIOS
JTAG	Joint Test Action Group – a standard used to control serial data transfer for test and programming operations.
LFM	Linear feet per minute
LVDS	Low Voltage Differential Signaling
MUX	Multiplexor – multiple signals multiplexed to one with a selection mechanism to control which path is active.
Packed	When UART characters are always sent/received in groups of four, allowing full use of host bus/FIFO bandwidth.
Packet	Group of characters transferred. When the characteristics of the group of characters is known, the data can be stored in packets and transferred as such; the system is optimized as a result. Any number of characters can be transferred.
PCI	Peripheral Component Interconnect – parallel bus from host to this device
PIM	PMC Interface Module (PIM). Provides rear I/O in cPCI systems. Mounts to PIM Carrier
PIM Carrier	PIM Mounting Device. Mounts on rear of cPCI backplane.
PMC	PCI Mezzanine Card – establishes common connectors, connections, size and other mechanical features.
TAP	Test Access Port – basically a multi-state port that can be controlled with JTAG [TMS, TDI, TDO, TCK]. The TAP States are the states in the State Machine that are controlled by the commands received over the JTAG link.
TCK	Test Clock provides synchronization for the TDI, TDO, and TMS signals

TDI	Test Data in – this serial line provides the data input to the device controlled by the TMS commands. For example, the data to program the FLASH comes on the TDI line while the commands to the state machine to move through the necessary states comes over TMS. Rising edge of TCK valid.
TDO	Test Data Out is the shifted data out. Valid on the falling edge of the TCK. Not all states output data.
TMS	Test Mode State – this serial line provides the state switching controls. '1' indicates to move to the next state, '0' means stay put in cases where delays can happen; otherwise, 0,2 are used to choose which branch to take. Due to the complexity of state manipulation, the instructions are usually precompiled. Rising edge of TCK valid.
UART	Universal Asynchronous Receiver Transmitter. Common serialized data transfer with start bit, stop bit, optional parity, optional 7/8 bit data. Can be over any electrical interface. RS232 and RS422 are most common.
Unpacked	When UART characters are sent on an unknown basis requiring single character storage and transfer over the host bus
VendorID	Manufacturers number for PCI/PCIe boards. DCBA is Dynamic Engineering's VendorID
VME	Versa Module European
VPX	Family of standards based on the VITA 46.0
XMC	Switched mezzanine card (PMC with PCIe)