# Win10 Driver Manual

# SpaceWire Monitor

Manual Revision 01p1
Revision Date 06/07/2023

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

## Design Revision History

**Table 1: Design Revision History**

| Revision | Date | Description |
|---|---|---|
| | | |

## Manual Revision History

**Table 2: Manual Revision History**

| Revision | Date | Description |
|---|---|---|
| 1.1 | 6/7/23 | Original |

**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 two SpaceWire devices using two standard SpaceWire cables. This driver was developed as a Windows Kernel Driver using the KMDF.

Each channel has 64K internal FIFO plus a 512K external FIFO. While possible to write/read from the FIFO directly with the API, the driver implementation 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 uses the Win10 KMDV version 1.0 driver. This consists of two modules, a base driver and a channel driver module, that can be loaded using the .inf file.

This package comes with a Monitor User App which is used to test the hardware as well as provide an example of how to interface with the device through software:

**Table 3: Header Files**

| File Name | Description |
|---|---|
| ioctle.h | Defines serves as the primary API for the device and demonstrates how to use the IOCTL calls defined in the files below, as well as different options for reading from the device depending on the use case. |
| SpaceWireMonitorBasePublic.h | Defines many of the data structures used by the Ioctl calls for the base device. |
| SpaceWireMonitorChanPublic.h | Defines many of the data structures used by the Ioctl calls for the channel device. |
| DE/DE_WinDeviceApi.h | Defines helper functions that can be used to enumerate and open the devices, and simplifies calls to the windows framework for device interfacing. |

## Test Suite

The UserApp is used in-house to validate the hardware and provides a more extensive example of how to interact with the hardware. The UserApp was written in C/C++. Some of the test suite tests require a SpaceWireBK device installed in the same system, but those tests are automatically suppressed when this device is not present. The purpose of this is to test the HW within a single system and abstract away some of the complexity for the testing team.

## Logger

Logging the data can be done through the provided test but no data is sent through the application. The logging assumes that data is coming to the card through some other application or device and merely serves to capture the data as well as the packet descriptors and write that data to disk. The data is written in binary format, and the logger functions convert those binary files to readable txt files when logging is complete.

## Driver Installation

Driver Installation is simple, first right-click on the SpaceWireMonitorBase.inf file and click “install”, this will load the base driver automatically and enumerate the channels in the Window’s Device Manager. Once this is installed, follow the same step by right clicking on the SpaceWireMonitorChan.inf file and click “Install”.

Once the driver is installed on a system, the driver files are automatically copied to the “Window’s Store” (i.e., a special directory where windows stores driver files). The driver will automatically load every time the computer is booted.

## Uninstallation

Open the device manager and navigate to the “SpaceWireMon Device” element, expand the tree down by pressing the “>” arrow. There you will see the base and channel device. Right-click on the channel device and select “uninstall device” (IMPORTANT – once this uninstall is selected a window will pop up, check the box that says “Delete the driver software installed for this device” (this removes the driver from the windows store). For the second channel this pop-up will not offer the same box as the software is already removed from the store. Finally, do the same thing with the base driver – again remembering to check the box as the base driver is a separate piece of software that has been copied to the Window’s Store.

## API

The primary API for the device can be located in the Ioctl.h file, and there are some additional ioctl calls that can be found in the SpaceWireMonitorBase/ChanPublic.h files. One of the most useful of those is the IOCTL_DE_REG_CMD which takes a DE_REG_CMD as both input and output and allows you to read/write to any register using the offsets defined in SpaceWireMonitorChanPublic.h.

```
/**
 * @brief Get information for the channel
 * @param handle handle to monitor channel
 * @param chan_device_info struct filled by routine
 * @return true on success
*/
bool GetChanInfo(HANDLE handle, SPWRMON_CHAN_DRIVER_DEVICE_INFO & chan_device_info);

/**
 * @brief Get the channel configuration
 * @param handle handle to monitor channel
 * @param config struct filled by routine
 * @return true on success
*/

bool GetChanConfig(HANDLE handle, SPWRMON_CHAN_CONFIG& config);

/**
 * @brief Set the channel configuration
 * @param handle handle to monitor channel
 * @param config struct filled by routine
 * @return true on success
*/
bool SetChanConfig(HANDLE handle, SPWRMON_CHAN_CONFIG& config);

/**
 * @brief Reads packet length from the channels packet length fifo
 * @param handle handle to monitor channel
 * @param val ULONG filled by routine
 * @return true on success
*/
bool ReadPacketLenFifo(HANDLE handle, ULONG &val);

/**
 * @brief Resests all the channels fifos
 * @param handle handle to monitor channel
 * @return true on success
*/
bool ResetFifos(HANDLE handle);

/**
 * @brief Gets the channels fifo levels
 * @param handle handle to monitor channel
 * @param val SPWRMON_CHAN_FIFO_COUNTS struct filled by routine
 * @return true on success
*/
bool GetFifoCounts(HANDLE handle, SPWRMON_CHAN_FIFO_COUNTS& val);

/**
 * @brief Writes data to the channels data fifo - used for testing
 * @param handle handle to monitor channel
 * @param val ULONG written to fifo
 * @return true on success
*/
bool WriteDataFifo(HANDLE handle, ULONG& val);

/**
 * @brief Reads data from the channels data fifo
 * @param handle handle to monitor channel
 * @param val ULONG filled by routine
 * @return true on success
*/
bool ReadDataFifo(HANDLE handle, ULONG& val);

/**
 * @brief Disables the monitor - monitor is enabled by configuring the channel
 * @param handle handle to monitor channel
 * @return true on success
*/
bool DisableMonitor(HANDLE handle);

/**
 * @brief Forces an interrupt on the channel (this call already enables the interrupt in the driver)
 * @param handle handle to monitor channel
 * @return true on success
*/
bool ForceInterrupt(HANDLE handle);
```

```
/**
 * @brief Register an event to be signaled when an interrupt occurs
 * @param handle handle to monitor channel
 * @param event object to be signaled
 * @return true on success
*/
bool RegisterMonEvent(HANDLE handle, HANDLE * event_handle);

/**
 * @brief Enables the master interrupt for the channel
 * @param handle handle to monitor channel
 * @return true on success
*/
bool EnableInterrupt(HANDLE handle);

/**
 * @brief Disables the master interrupt for the channel
 * @param handle handle to monitor channel
 * @return true on success
*/
bool DisableInterrupt(HANDLE handle);

/**
 * @brief Reads all data on the device, if small packets are being monitored
 *        this can drop packets because of DMA overhead
 * @param handle handle to monitor channel
 * @param data empty vector, cleared, resized, and filled by routine
 * @return bytes read or negative on error
*/
uint32_t ReadDevice(HANDLE handle, std::vector<char>& data);

/**
 * @brief Reads all data on the device, if small packets are being monitored
 *        this can drop packets because of DMA overhead
 * @param handle handle to monitor channel
 * @param data buffer pointer filled by routine
 * @param len of desired data - routine will not read more than this but may read less
 * @return bytes read or negative on error
*/
uint32_t ReadDevice(HANDLE handle, char * data, uint32_t len);

/**
 * @brief Reads all data on the device, but will optimize read size to avoid dropping packets.
 *        This can be slower due to the optimization and timeout required for the optimization.
 * @param handle handle to monitor channel
 * @param data empty vector, cleared, resized, and filled by routine
 * @param size of desired data - routine will not read more than this but may read less
 * @param timeout_ms timeout in milliseconds if less data than size is read
 * @return bytes read or negative on error
*/
uint32_t ReadDeviceDMAOptimized(HANDLE handle, std::vector<char>& data, ULONG size, ULONG timeout_ms);
```

```
/**
 * @brief Reads any available packet lens, and associates descriptor with timestamp.
 * @param handle handle to monitor channel
 * @param packet_lens vector of packet lens filled by routine, will add data to existing data (not
clear)
 * @param timestamps vector of timestamps filled by routine, will add data to existing data (not clear)
 * @return number of descriptors read or negative on error
*/
uint32_t ReadPacketLensWithTimestamps(HANDLE handle, std::vector<ULONG>& packet_lens,
std::vector<std::chrono::steady_clock::time_point>&timestamps);
```

## 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) |