

WIND RIVER

VxWorks®

DRIVERS API REFERENCE

6.6

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:
installDir\product_name\3rd_party_licensor_notice.pdf.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

Contents

This book provides reference entries that describe VxWorks drivers. For reference entries that describe the facilities available for VxWorks process-based application development, see the *VxWorks Application API Reference*. For reference entries that describe facilities available in the VxWorks kernel, see the *VxWorks Kernel API Reference*.

1. Libraries

This section provides reference entries for each of the VxWorks driver libraries, arranged alphabetically. Each entry lists the routines found in the library, including a one-line synopsis of each and a general description of their use.

Individual reference entries for each of the available functions in these libraries is provided in section 2.

2. Routines

This section provides reference entries for each of the routines found in the VxWorks driver libraries documented in section 1.

1

Libraries

ambaSio – ARM AMBA UART <i>tty</i> driver	5
amd8111LanEnd – END style AMD8111 LAN Ethernet driver	7
ataDrv – ATA/IDE and ATAPI CDROM (LOCAL and PCMCIA) disk device driver	10
ataShow – ATA/IDE (LOCAL and PCMCIA) disk device driver show routine	15
auEnd – END style Au MAC Ethernet driver	16
bcm1250MacEnd – END style BCM1250 MAC Ethernet driver	19
bio – buffer IO Implementation	20
cisLib – PCMCIA CIS library	21
cisShow – PCMCIA CIS show library	21
cmdLineBuild – source file included in sysLib.c for command-line builds	22
ctB69000Vga – a CHIPS B69000 initialization source module	22
dec21x40End – END-style DEC 21x40 PCI Ethernet network interface driver	24
device – Device Infrastructure Library	30
drvDownLoad – downloadable driver support module	31
el3c90xBEnd – END network interface driver for 3COM 3C90xB XL	31
elt3c509End – END network interface driver for 3COM 3C509	36
endLib – support library for END-based drivers	38
erfLib – Event Reporting Framework Library	39
erfShow – Event Reporting Framework Library Show routines	39
evbNs16550Sio – NS16550 serial driver for the IBM PPC403GA evaluation	39
fei82557End – END style Intel 82557 Ethernet network interface driver	40
g64120aMf – VxBus driver for galileo 64120a system controller	45
g64120aPci – VxBus Galileo 64120A PCI bus controller driver	45
gei82543End – Intel 82540/82541/82543/82544/82545/82546/ MAC driver	45
hwConfig – vxBus resource library	50
hwMemLib – hardware memory allocation library	50
i8250Sio – I8250 serial driver	51
iOlicomEnd – END style Intel Olicom PCMCIA network interface driver	51
iPIIX4 – low level initialization code for PCI ISA/IDE Xcelerator	54
isrDeferLib – ISR deferral library	58

ixp400I2c – Intel IXP400 I2C source file	58
ln97xEnd – END style AMD Am79C97X PCnet-PCI Ethernet driver	60
lptDrv – parallel chip device driver for the IBM-PC LPT	65
m8260SccEnd – END style Motorola MPC8260 network interface driver	66
m85xxCpu – device driver for remote Cpu on m85xx CPU with RapidIO	69
m85xxRio – RapidIO host controller in PowerPC 85XX CPU	69
mcf5475Pci – PCI host controller in Coldfire mcf5475/85 CPU	71
miiLib – Media Independent Interface library	71
motFcc2End – Second Generation Motorola FCC Ethernet network interface.	73
motFecEnd – END style Motorola FEC Ethernet network interface driver	82
ncr810Lib – NCR 53C8xx PCI SCSI I/O Processor (SIOP) library (SCSI-2)	89
ne2000End – NE2000 END network interface driver	90
nec765Fd – NEC 765 floppy disk device driver	92
ns16550Sio – NS 16550 UART <i>tty</i> driver	92
ns83902End – National Semiconductor DP83902A ST-NIC	93
pccardLib – PC CARD enabler library	95
pciAutoConfigLib – PCI bus scan and resource allocation facility	95
pciConfigLib – PCI Configuration space access support for PCI drivers	104
pciConfigShow – Show routines of PCI bus(IO mapped) library	114
pciIntLib – PCI Shared Interrupt support	115
pcic – Intel 82365SL PCMCIA host bus adaptor chip library	116
pcicShow – Intel 82365SL PCMCIA host bus adaptor chip show library	116
pcmciaLib – generic PCMCIA event-handling facilities	117
pcmciaShow – PCMCIA show library	118
pentiumPci – PCI host controller for Pentium CPU	118
ppc403Sio – ppc403GA serial driver	118
ppc440gpPci – PCI host controller in PowerPC 440GP CPU	119
ppc860Sio – Motorola MPC800 SMC UART serial driver	120
rm9000x2glSio – RM9000 <i>tty</i> driver	120
shSciSio – Hitachi SH SCI (Serial Communications Interface) driver	121
shScifSio – Renesas SH SCIF (Serial Communications Interface) driver	122
sioChanUtil – SIO Channel Utilities module	122
smEnd – END shared memory (SM) network interface driver	122
smEndShow – shared memory network END driver show routines	127
smcFdc37b78x – a superIO (fdc37b78x) initialization source module	128
sramDrv – PCMCIA SRAM device driver	130
sym895Lib – SCSI-2 driver for Symbios SYM895 SCSI Controller.	131
tcic – Databook TCIC/2 PCMCIA host bus adaptor chip driver	133
tcicShow – Databook TCIC/2 PCMCIA host bus adaptor chip show library	133
tffsConfig – TrueFFS configuration file for VxWorks	134
vgaInit – a VGA 3+ mode initialization source module	134
vxBus – vxBus subsystem source file	135
vxbArchAccess – vxBus access routines specific to an architecture	136
vxbAuxClkLib – vxBus auxiliary clock library	138
vxbCn3xxxTimer – Cn3xxxTimer driver for VxBus	139

vxbColdFireSio – coldfire vxBus Serial Communications driver	140
vxbDelayLib – vxBus delay function interfaces file	140
vxbDevTable – Support utils for table-based device enumeration	140
vxbDmaBufLib – buffer and DMA system for VxBus drivers	141
vxbDmaLib – vxBus DMA interfaces module	143
vxbEndQctrl – utility to change the RX job queue of a vxbus END device	143
vxbEpicIntCtr – Driver for Embedded Programmable Interrupt Controller	144
vxbHwifDebug – - HWIF Debug Utilities	147
vxbI8253Timer – driver for the intel 8253 timer	148
vxbI8259IntCtr – Intel 8259A Programmable Interrupt Controller driver	149
vxbIntCtrLib – - VxBus Interrupt Controller support library	150
vxbIntDynaCtrLib – - VxBus Dynamic Interrupt support library	151
vxbIntelTimestamp – Driver for the timestamp on the intel chip	151
vxbIoApicIntr – VxBus driver for the Intel IO APIC/xAPIC driver	152
vxbIxp400Sio – IXP400 UART <i>tty</i> driver	153
vxbIxp400Timer – ixp4xx processor timer library	154
vxbLoApicIntr – VxBus driver template	154
vxbLoApicTimer – Driver for the loApic timer	156
vxbM548xDma – Coldfire MCD slave DMA VxBus driver	157
vxbM54x5SliceTimer – timer driver for m54x5 Slice timers	158
vxbM85xxDecTimer – driver for Book E PowerPC decrementer	159
vxbM85xxFITimer – driver for PowerPC Book E Fixed Interval Timer(FIT)	159
vxbM85xxTimer – timer driver for m85xx PowerPC Book E Architecture	160
vxbM85xxWDTimer – driver for PowerPC Book E Watch Dog Timer(FIT)	162
vxbMc146818Rtc – driver for the MC146818 real time clock	162
vxbMipsCavIntCtr – interrupt controller for Cavium CN3xxx family	163
vxbMipsIntCtr – - generic interrupt controller for MIPS CPU	164
vxbMipsR4KTimer – MIPS R4000 on CPU Timer driver for VxBus	165
vxbMipsSbIntCtr – interrupt controller for BCM1480	166
vxbMpApic – VxBus driver for the Intel MP APIC/xAPIC (Advanced PIC)	166
vxbNs16550Sio – National Semiconductor 16550 Serial Driver	167
vxbOcteonSio – Octeon 16550ish Serial Driver	167
vxbOpenPicTimer – Driver for OpenPIC timers	168
vxbParamSys – vxBus parameter system	170
vxbPci – PCI bus support module for vxBus	171
vxbPciAccess – vxBus access routines for PCI bus type	174
vxbPlb – Processor Local Bus support module for vxBus	174
vxbPlbAccess – vxBus access routines for PLB	174
vxbPpcDecTimer – timer driver for the PowerPC decrementer	175
vxbPpcQuiccTimer – timer driver for quiccTimer	175
vxbPrimeCellSio – ARM AMBA UART <i>tty</i> driver for VxBus	176
vxbQeIntCtr – Quicc Engine interrupt controller driver	178
vxbQuiccIntCtr – Motorola MPC 83XX interrupt controller driver	178
vxbRapidIO – - RapidIO bus support module	179
vxbRapidIOCfgTable – RapidIO table-based device enumeration	179

vxbSb1DuartSio – BCM1250/1480 serial communications driver	180
vxbSb1Timer – Sb1Timer driver for VxBus	181
vxbSh7700Timer – SH77xx on-chip Timer driver for VxBus	181
vxbSh77xxPci – sh77xx PCI bus controller VxBus driver	183
vxbShScifSio – Renesas SuperH SCIF driver for VxBus	190
vxbShow – bus subsystem source file	191
vxbSmEnd – BSP configuration module for shared memory END driver	192
vxbSmSupport – support for shared memory	192
vxbSmcFdc37x – SMsC FDC37x Ultra I/O Configuration Driver for VxBus	193
vxbSysClkLib – vxBus system clock library	193
vxbTimerLib – vxBus timer interfaces module	194
vxbTimerStub – vxBus timer stub file	194
vxbTimestampLib – vxBus Timestamp library	195
vxbVxSimIpi – VxBus driver for Linux simulator IPI management	195
wancomEnd – END style Marvell/Galileo GT642xx Ethernet network interface driver	195
wdbEndPktDrv – END based packet driver for lightweight UDP/IP	198
wdbNetromPktDrv – NETROM packet driver for the WDB agent	198
wdbPipePktDrv – pipe packet driver for lightweight UDP/IP	199
wdbSlipPktDrv – a serial line packetizer for the WDB agent	201
wdbTsfsDrv – virtual generic file I/O driver for the WDB agent	201
wdbVioDrv – virtual <i>tty</i> I/O driver for the WDB agent	205
xbd – Extended Block Device Library	206
z8530Sio – Z8530 SCC Serial Communications Controller driver	206

ambaSio

NAME	ambaSio – ARM AMBA UART <i>tty</i> driver
ROUTINES	ambaDevInit() – initialise an AMBA channel ambaIntTx() – handle a transmitter interrupt ambaIntRx() – handle a receiver interrupt
DESCRIPTION	<p>This is the device driver for the Advanced RISC Machines (ARM) AMBA UART. This is a generic design of UART used within a number of chips containing (or for use with) ARM CPUs such as in the Digital Semiconductor 21285 chip as used in the EBSA-285 BSP.</p> <p>This design contains a universal asynchronous receiver/transmitter, a baud-rate generator, and an InfraRed Data Association (IrDa) Serial InfraRed (SiR) protocol encoder. The SiR encoder is not supported by this driver. The UART contains two 16-entry deep FIFOs for receive and transmit: if a framing, overrun or parity error occurs during reception, the appropriate error bits are stored in the receive FIFO along with the received data. The FIFOs can be programmed to be one byte deep only, like a conventional UART with double buffering, but the only mode of operation supported is with the FIFOs enabled.</p> <p>The UART design does not support the modem control output signals: DTR, RI and RTS. Moreover, the implementation in the 21285 chip does not support the modem control inputs: DCD, CTS and DSR.</p> <p>The UART design can generate four interrupts: Rx, Tx, modem status change and a UART disabled interrupt (which is asserted when a start bit is detected on the receive line when the UART is disabled). The implementation in the 21285 chip has only two interrupts: Rx and Tx, but the Rx interrupt is a combination of the normal Rx interrupt status and the UART disabled interrupt status.</p> <p>Only asynchronous serial operation is supported by the UART which supports 5 to 8 bit bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity. The default baud rate is determined by the BSP by filling in the AMBA_CHAN structure before calling ambaDevInit().</p> <p>The exact baud rates supported by this driver will depend on the crystal fitted (and consequently the input clock to the baud-rate generator), but in general, baud rates from about 300 to about 115200 are possible.</p> <p>In theory, any number of UART channels could be implemented within a chip. This driver has been designed to cope with an arbitrary number of channels, but at the time of writing, has only ever been tested with one channel.</p>
DATA STRUCTURES	
An AMBA_CHAN data structure is used to describe each channel, this structure is described in h/driv/sio/ambaSio.h .	

CALLBACKS Servicing a "transmitter ready" interrupt involves making a callback to a higher level library in order to get a character to transmit. By default, this driver installs dummy callback routines which do nothing. A higher layer library that wants to use this driver (e.g. **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK** ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

MODES This driver supports both polled and interrupt modes.

USAGE The driver is typically only called by the BSP. The directly callable routines in this module are **ambaDevInit()**, **ambaIntTx()** and **ambaIntRx()**.

The BSP's **sysHwInit()** routine typically calls **sysSerialHwInit()**, which initialises the hardware-specific fields in the **AMBA_CHAN** structure (e.g. register I/O addresses etc) before calling **ambaDevInit()** which resets the device and installs the driver function pointers. After this the UART will be enabled and ready to generate interrupts, but those interrupts will be disabled in the interrupt controller.

The following example shows the first parts of the initialisation:

```
#include "drv/sio/ambaSio.h"

LOCAL AMBA_CHAN ambaChan[N_AMBA_UART_CHANS];

void sysSerialHwInit (void)
{
    int i;

    for (i = 0; i < N_AMBA_UART_CHANS; i++)
    {
        ambaChan[i].regs = devParas[i].baseAdrs;
        ambaChan[i].baudRate = CONSOLE_BAUD_RATE;
        ambaChan[i].xtal = UART_XTAL_FREQ;

        ambaChan[i].levelRx = devParas[i].intLevelRx;
        ambaChan[i].levelTx = devParas[i].intLevelTx;

        /*
         * Initialise driver functions, getTxChar, putRcvChar and
         * channelMode, then initialise UART
         */
    }

    ambaDevInit (&ambaChan[i]);
}
```

The BSP's **sysHwInit2()** routine typically calls **sysSerialHwInit2()**, which connects the chip's interrupts via **intConnect()** (the two interrupts **ambaIntTx** and **ambaIntRx**) and enables those interrupts, as shown in the following example:

```
void sysSerialHwInit2 (void)
{
```

```
/* connect and enable Rx interrupt */

(void) intConnect (INUM_TO_IVEC(devParas[0].vectorRx),
                   ambaIntRx, (int) &ambaChan[0]);
intEnable (devParas[0].intLevelRx);

/* connect Tx interrupt */

(void) intConnect (INUM_TO_IVEC(devParas[0].vectorTx),
                   ambaIntTx, (int) &ambaChan[0]);
/*
 * There is no point in enabling the Tx interrupt, as it will
 * interrupt immediately and then be disabled.
 */
}
```

BSP

By convention all the BSP-specific serial initialisation is performed in a file called **sysSerial.c**, which is #include'd by **sysLib.c**. **sysSerial.c** implements at least four functions, **sysSerialHwInit()** **sysSerialHwInit2()**, **sysSerialChanGet()**, and **sysSerialReset()**. The first two have been described above, the others work as follows:

sysSerialChanGet is called by **usrRoot** to get the serial channel descriptor associated with a serial channel number. The routine takes a single parameter which is a channel number ranging between zero and **NUM_TTY**. It returns a pointer to the corresponding channel descriptor, **SIO_CHAN ***, which is just the address of the **AMBA_CHAN** structure.

sysSerialReset is called from **sysToMonitor()** and should reset the serial devices to an inactive state (prevent them from generating any interrupts).

INCLUDE FILES:

drv/sio/ambaSio.h **sioLib.h**

SEE ALSO:

Advanced RISC Machines AMBA UART (AP13) Data Sheet, Sheet," Digital Semiconductor EBSA-285 Evaluation Board Reference Manual.

INCLUDE FILES

none

amd8111LanEnd

NAME

amd8111LanEnd – END style AMD8111 LAN Ethernet driver

ROUTINES

amd8111LanEndLoad() – initialize the driver and device

amd8111LanDumpPrint() – Display statistical counters

amd8111LanErrCounterDump() – dump statistical counters

DESCRIPTION

This module implements the Advanced Micro Devices 8111 LAN END PCI Ethernet 32-bit network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These target-specific values and the external support routines are described below.

This driver supports multiple units per CPU. The driver can be configured to support big-endian or little-endian architectures. It contains error recovery code to handle known device errata related to DMA activity.

Some big-endian processors may be connected to a PCI bus through a host/PCI bridge which performs byte swapping during data phases. On such platforms, the controller need not perform byte swapping during a DMA access to memory shared with the host processor.

BOARD LAYOUT

This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The driver provides one standard external interface, **amd8111LanEndLoad()**. As input, this routine takes a string of colon-separated parameters. The parameters should be specified in hexadecimal (optionally preceded by **0x** or a minus sign **-**). The parameter string is parsed using **strtok_r()**.

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is:

unit:memAdrs:memSize:memWidth:offset:tdnum:rdnum:flags

unit

The unit number of the device. Unit numbers start at zero and increase for each device controlled by the same driver. The driver does not use this value directly. The unit number is passed through the MUX API where it is used to differentiate between multiple instances of a particular driver.

memAdrs

This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cache coherent memory for buffers and descriptors from the system memory pool. The PCnet device is a DMA type of device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the PCnet. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

memSize

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. The constant NONE can be used to indicate no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

memWidth

Some target hardware that restricts the shared memory region to a specific location also restricts the access width to this region by the CPU. On these targets, performing an access of an invalid width will cause a bus error.

This parameter can be used to specify the number of bytes of access width to be used by the driver during access to the shared memory. The constant NONE can be used to indicate no restrictions.

Current internal support for this mechanism is not robust; implementation may not work on all targets requiring these restrictions.

offset

This parameter specifies a memory alignment offset. Normally this parameter is zero except for architectures which can only access 32-bit words on 4-byte aligned address boundaries. For these architectures the value of this offset should be 2.

tdnum

This parameter specifies the number of Transmit Descriptors to allocate. Must be a power of 2.

rdnum

This parameter specifies the number of Receive Descriptors to allocate. Must be a power of 2.

flags

This parameter is used for future use. Currently its value should be zero.

PERFORMANCE

This driver has been empirically shown to give better performance when passed a cacheable region of memory for use by the driver provided the BSP supports caching.

EXTERNAL SUPPORT REQUIREMENTS

The BSP must provide the following function to perform BSP specific initialization:

```
IMPORT STATUS sysAmd8111LanInit (int unit, AMD8111_LAN_BOARD_INFO
*pBoard);
```

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 14240 bytes in text for a PENTIUM3 target
- 120 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1520 bytes for receive frames and transmit frames.

INCLUDE FILES none

SEE ALSO **muxLib**, **endLib**, **netBufLib**, the network stack documentation, "AMD-8111 HyperTransport I/O Hub Data Sheet"

ataDrv

NAME **ataDrv** – ATA/IDE and ATAPI CDROM (LOCAL and PCMCIA) disk device driver

ROUTINES

- ataDrv()** – Initialize the ATA driver
- ataXbdDevCreate()** – create an XBD device for a ATA/IDE disk
- ataDevCreate()** – create a device for a ATA/IDE disk
- ataBlkRW()** – read or write sectors to a ATA/IDE disk.
- atapiPktCmdSend()** – Issue a Packet command.
- atapiIoctl()** – Control the drive.
- atapiParamsPrint()** – Print the drive parameters.
- atapiCtrlMediumRemoval()** – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command
- atapiRead10()** – read one or more blocks from an ATAPI Device.
- atapiReadCapacity()** – issue a READ CD-ROM CAPACITY command to a ATAPI device
- atapiReadTocPmaAtip()** – issue a READ TOC command to a ATAPI device
- atapiScan()** – issue SCAN packet command to ATAPI drive.
- atapiSeek()** – issues a SEEK packet command to drive.
- atapiSetCDSpeed()** – issue SET CD SPEED packet command to ATAPI drive.
- atapiStopPlayScan()** – issue STOP PLAY/SCAN packet command to ATAPI drive.
- atapiStartStopUnit()** – Issues START STOP UNIT packet command
- atapiTestUnitRdy()** – issue a TEST UNIT READY command to a ATAPI drive
- ataCmd()** – issue a RegisterFile command to ATA/ATAPI device.
- ataInit()** – initialize ATA device.
- ataRW()** – read/write a data from/to required sector.
- ataDmaRW()** – read/write a number of sectors on the current track in DMA mode
- ataPiInit()** – init a ATAPI CD-ROM disk controller
- ataDevIdentify()** – identify device
- ataParamRead()** – Read drive parameters
- ataCtrlReset()** – reset the specified ATA/IDE disk controller
- ataStatusChk()** – Check status of drive and compare to requested status.
- atapiPktCmd()** – execute an ATAPI command with error processing
- atapiInit()** – init ATAPI CD-ROM disk controller
- ataXbdRawio()** – do raw I/O access
- ataRawio()** – do raw I/O access

DESCRIPTION**BLOCK DEVICE DRIVER:**

This is a Block Device Driver for ATA/ATAPI devices on IDE host controller. It also provides necessary functions to user for device and its features control which are not used or utilized by file system.

This driver provides standard Block Device Driver functions,(blkRd, blkWrt, ioctl, statusChk, and reset) for ATA and ATAPI devices separately as the scheme of implementation differs. These functions are implemented as **ataBlkRd()**, **ataBlkWrt()**, **ataBlkIoctl()**, **ataStatus()** and **ataReset()** for ATA devices and **atapiBlkRd()**, **atapiBlkWrt()**, **atapiBlkIoctl()**, **atapiStatusChk()** and **atapiReset()** for ATAPI devices. The Block Device Structure **BLK_DEV** is updated with these function pointers at initialization of the driver depending on the type of the device in function **ataDevCreate()**.

ataDrv(), a user callable function, initializes ATA/ATAPI devices present on the specified IDE controller(either primary or secondary), which must be called once for each controller, before usage of this driver, usually called from **usrRoot()** in **usrConfig.c**.

The routine **ataDevCreate()**, which is user callable function, is used to mount a logical drive on an ATAPI drive. This routine returns a pointer to **BLK_DEV** structure, which is used to mount the file system on the logical drive.

OTHER NECESSARY FUNCTIONS FOR USER:

There are various functions provided to user, which can be classified to different categories as device control function, device information functions and functions meant for packet devices.

Device Control Function:

atapiIoctl() function is used to control a device. Block Device Driver functions **ataBlkIoctl()** and **atapiBlkIoctl()** functions are also routed to this function. This function implements various control command functions which are not used by the I/O system (like power management feature set commands, host protected feature set commands, security feature set commands, media control functions etc).

Device Information Function:

In this category various functions are implemented depending on the information required. These functions return information required (like cylinder count, Head count, device serial number, device Type, etc) from the internal device structures.

Packet Command Functions:

Although Block Device Driver functions deliver packet commands using functions provided by **atapiLib.c** for required functionality. There are group of functions provided in this driver to user for ATAPI device, which implements packet commands for **CD_ROM** that comply to **ATAPI-SFF8020i** specification which are essentially required for CD ROM operation for file system. These functions are named after their command name (like for

REQUEST SENSE packet command **atapiReqSense()** function). To issue other packet commands **atapiPktCmdSend()** can be used.

This driver also provides a generic function **atapiPktCmdSend()** to issue a packet command to ATAPI devices, which can be utilized by user to issue packet command directly instead using the implemented functions also may be used to send new commands (may come in later specs) to device. User can issue any packet command using **atapiPktCmdSend()** function to the required device by passing its **BLK_DEV** structure pointer and pointer for ATAPI_CMD command packet.

typedef of ATAPI_CMD

```
typedef struct atapi_cmd
{
    UINT8      cmdPkt [ATAPI_MAX_CMD_LENGTH];
    char       **ppBuf;
    UINT32     bufLength;
    ATA_DATA_DIR direction;
    UINT32     desiredTransferSize;
    BOOL       dma;
    BOOL       overlap;
} ATAPI_CMD;
```

and **ATA_DATA_DIR** typedef is

```
typedef enum /* with respect to host/memory */
{
    NON_DATA, /* non data command */
    OUT_DATA, /* to drive from memory */
    IN_DATA   /* from drive to memory */
} ATA_DATA_DIR;
```

User is expected supposed to fill the **ATAPI_CMD** structure with required parameters of the packet and pass the **ATAPI_CMD** structure pointer to **atapiPktCmdSend()** function for command execution.

All the packet command functions require **ATA_DEV** structure to be passed, which alternatively a **BLK_DEV** Device Structure of the device. One should type convert the structure and the same **BLK_DEV** structure pointer to these functions.

The routine **ataPiRawio()** supports physical I/O access. The first argument is the controller number, 0 or 1; the second argument is drive number, 0 or 1; the third argument is a pointer to an **ATA_RAW** structure.

PARAMETERS:

The **ataPiDrv()** function requires a configuration flag as a parameter. The configuration flag is one of the following or Bitwise OR of any of the following combination:

configuration flag =

Transfer mode | Transfer bits | Transfer unit | Geometry parameters

Transfer mode

Description

Transfer Rate

ATA PIO DEF 0	PIO default mode
ATA PIO DEF 1	PIO default mode, no IORDY
ATA PIO 0	PIO mode 0
ATA PIO 1	PIO mode 1
ATA PIO 2	PIO mode 2
ATA PIO 3	PIO mode 3
ATA PIO 4	PIO mode 4
ATA PIO AUTO	PIO max supported mode
ATA DMA SINGLE 0	Single DMA mode 0
ATA DMA SINGLE 1	Single DMA mode 1
ATA DMA SINGLE 2	Single DMA mode 2
ATA DMA MULTI 0	Multi word DMA mode 0
ATA DMA MULTI 1	Multi word DMA mode 1
ATA DMA MULTI 2	Multi word DMA mode 2
ATA DMA ULTRA 0	Ultra DMA mode 0
ATA DMA ULTRA 1	Ultra DMA mode 1
ATA DMA ULTRA 2	Ultra DMA mode 2
ATA DMA ULTRA 3	Ultra DMA mode 3
ATA DMA ULTRA 4	Ultra DMA mode 4
ATA DMA ULTRA 5	Ultra DMA mode 5
ATA DMA AUTO	DMA max supported mode
Transfer bits	
ATA BITS 16	RW bits size, 16 bits
ATA BITS 32	RW bits size, 32 bits
Transfer unit	
ATA PIO SINGLE	RW PIO single sector
ATA PIO MULTI	RW PIO multi sector
Geometry parameters	
ATA GEO FORCE	set geometry in the table
ATA GEO PHYSICAL	set physical geometry
ATA GEO CURRENT	set current geometry

ISA SingleWord DMA mode is obsolete in ata-3.

The Transfer rates shown above are the Burst transfer rates. If ATA PIO AUTO is specified, the driver automatically chooses the maximum PIO mode supported by the device. If ATA DMA AUTO is specified, the driver automatically chooses the maximum Ultra DMA mode supported by the device and if the device doesn't support the Ultra DMA mode of data transfer, the driver chooses the best Multi Word DMA mode. If the device doesn't support the multiword DMA mode, driver chooses the best single word DMA mode. If the device doesn't support DMA mode, driver automatically chooses the best PIO mode. So it is recommended to specify the ATA DMA AUTO.

If ATA PIO MULTI is specified, and the device does not support it, the driver automatically chooses single sector or word mode. If ATA BITS 32 is specified, the driver uses 32-bit transfer mode regardless of the capability of the drive. The Single word DMA mode will not be supported by the devices compliant to ATA/ATAPI-5 or higher.

This driver supports UDMA mode data transfer from device to host, provided 80 conductor cable is used for required controller device. This check done at the initialisation of the device from the device parameters and if 80 conductor cable is connected then UDMA mode transfer is selected for operation subject to condition that required UDMA mode is supported by device as well as host. This driver follows ref-3 Chapter 4 "Determining a Drive's Transfer Rate Capability" to determine drives best transfer rate for all modes (ie UDMA, MDMA, SDMA and PIO modes).

The host IDE Bus master functions are to be mapped to following macro defined for various functionality in header file which are used in this driver.

ATA_HOST_CTRL_INIT - initialize the controller
ATA_HOST_DMA_ENGINE_INIT - initialize bus master DMA engine
ATA_HOST_DMA_ENGINE_START - Start bus master operation
ATA_HOST_DMA_ENGINE_STOP - Stop bus master operation
ATA_HOST_DMA_TRANSFER_CHK - check bus master data transfer complete
ATA_HOST_DMA_MODE_NEGOTIATE - get mode supported by controller
ATA_HOST_SET_DMA_RWMODE - set controller to required mode
ATA_HOST_CTRL_RESET - reset the controller

If **ATA_GEO_PHYSICAL** is specified, the driver uses the physical geometry parameters stored in the drive. If **ATA_GEO_CURRENT** is specified, the driver uses current geometry parameters initialized by BIOS. If **ATA_GEO_FORCE** is specified, the driver uses geometry parameters stored in **sysLib.c**.

The geometry parameters are stored in the structure table **ataTypes[]** in **sysLib.c**. That table has two entries, the first for drive 0, the second for drive 1. The members of the structure are:

```
int cylinders;           /* number of cylinders */  

int heads;               /* number of heads */  

int sectors;             /* number of sectors per track */  

int bytes;               /* number of bytes per sector */  

int precomp;              /* precompensation cylinder */
```

The driver supports two controllers and two drives on each. This is dependent on the configuration parameters supplied to **ataPiDrv()**.

SMP CONSIDERATIONS

Most of the processing in this driver occurs in the context of a dedicated task, and therefore is inherently SMP-safe. One area of possible concurrence occurs in the interrupt service routine, **ataIntr()**. An ISR-callable spin lock take/give pair has been placed around the code which acknowledges/clears the ATA controller's interrupt status register. If the BSP or application provides functions for **ataIntPreProcessing** or **ataIntPostProcessing**, consideration will have to be given to making these functions SMP-safe. Most likely, some portion(s) of these functions will need to be protected by a spin lock. The spin lock allocated for the controller can be used. Consult the SMP Migration Guide for hints.

References:

- 1) ATAPI-5 specification "T13-1321D Revision 1b, 7 July 1999"
- 2) ATAPI for CD-ROMs "SFF-8020i Revision 2.6, Jan 22,1996"
- 3) Intel 82801BA (ICH2), 82801AA (ICH), and 82801AB (ICH0) IDE Controller Programmer's Reference Manual, Revision 1.0 July 2000

Source of Reference Documents:

- 1) <ftp://ftp.t13.org/project/d1321r1b.pdf>
- 2) <http://www.bswd.com/sff8020i.pdf>

INCLUDE FILES

none

SEE ALSO*VxWorks Programmer's Guide: I/O System*

ataShow

NAME**ataShow** – ATA/IDE (LOCAL and PCMCIA) disk device driver show routine**ROUTINES**

- ataShowInit()** – initialize the ATA/IDE disk driver show routine
ataShow() – show the ATA/IDE disk parameters
ataDmaToggle() – turn on or off an individual controllers dma support
atapiCylinderCountGet() – get the number of cylinders in the drive.
atapiHeadCountGet() – get the number heads in the drive.
atapiDriveSerialNumberGet() – get the drive serial number.
atapiFirmwareRevisionGet() – get the firm ware revision of the drive.
atapiModelNumberGet() – get the model number of the drive.
atapiFeatureSupportedGet() – get the features supported by the drive.
atapiFeatureEnabledGet() – get the enabled features.
atapiMaxUDmaModeGet() – get the Maximum Ultra DMA mode the drive can support.
atapiCurrentUDmaModeGet() – get the enabled Ultra DMA mode.
atapiMaxMDmaModeGet() – get the Maximum Multi word DMA mode the drive supports.
atapiCurrentMDmaModeGet() – get the enabled Multi word DMA mode.
atapiMaxSDmaModeGet() – get the Maximum Single word DMA mode the drive supports
atapiCurrentSDmaModeGet() – get the enabled Single word DMA mode.
atapiMaxPioModeGet() – get the Maximum PIO mode that drive can support.
atapiCurrentPioModeGet() – get the enabled PIO mode.
atapiCurrentRwModeGet() – get the current Data transfer mode.
atapiDriveTypeGet() – get the drive type.
atapiVersionNumberGet() – get the ATA/ATAPI version number of the drive.

atapiRemovMediaStatusNotifyVerGet() – get the Media Stat Notification Version.
atapiCurrentCylinderCountGet() – get logical number of cylinders in the drive.
atapiCurrentHeadCountGet() – get the number of read/write heads in the drive.
atapiBytesPerTrackGet() – get the number of Bytes per track.
atapiBytesPerSectorGet() – get the number of Bytes per sector.
ataDumpPartTable() – dump the partition table from sector 0
ataDumptest() – a quick test of the dump functionality for ATA driver

DESCRIPTION	This library contains a driver show routine for the ATA/IDE (PCMCIA and LOCAL) devices supported on the IBM PC.
INCLUDE FILES	none

auEnd

NAME	auEnd – END style Au MAC Ethernet driver
ROUTINES	auEndLoad() – initialize the driver and device auInitParse() – parse the initialization string auDump() – display device status
DESCRIPTION	This module implements the Alchemey Semiconductor au on-chip ethernet MACs. The software interface to the driver is divided into three parts. The first part is the interrupt registers and their setup. This part is done at the BSP level in the various BSPs which use this driver. The second and third part are addressed in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the buffers. This driver is designed to be moderately generic. Though it currently is implemented on one processor, in the future it may be added to other Alchemey product offerings. Thus, it would be desirable to use the same driver with no source level changes. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These target-specific values and the external support routines are described below. This driver supports multiple units per CPU. The driver can be configured to support big-endian or little-endian architectures.
BOARD LAYOUT	This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The only external interface is the **auEndLoad()** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:devMemAddr:devIoAddr:enableAddr:vecNum:intLvl:offset :qtyCluster:flags

The **auEndLoad()** function uses **strtok()** to parse the string.

TARGET-SPECIFIC PARAMETERS*unit*

A convenient holdover from the former model. This parameter is used only in the string name for the driver.

devAddr

This parameter is the memory base address of the device registers in the memory map of the CPU. It indicates to the driver where to find the base MAC register.

devIoAddr

This parameter is the base address of the device registers for the dedicated DMA channel for the MAC device. It indicates to the driver where to find the DMA registers.

enableAddr

This parameter is the address MAC enable register. It is necessary to specify selection between MAC 0 and MAC 1.

vecNum

This parameter is the vector associated with the device interrupt. This driver configures the MAC device to generate hardware interrupts for various events within the device; thus it contains an interrupt handler routine. The driver calls **intConnect()** via the macro **SYS_INT_CONNECT()** to connect its interrupt handler to the interrupt vector generated as a result of the MAC interrupt.

intLvl

Some targets use additional interrupt controller devices to help organize and service the various interrupt sources. This driver avoids all board-specific knowledge of such devices. During the driver's initialization, the external routine **sysLanAuIntEnable()** is called to perform any board-specific operations required to allow the servicing of an interrupt. For a description of **sysLanAuIntEnable()**, see "External Support Requirements" below.

offset

This parameter specifies the offset from which the packet has to be loaded from the beginning of the device buffer. Normally this parameter is zero except for architectures which access long words only on aligned addresses. For these architectures the value of this offset should be 2.

qtyCluster

This parameter is used to explicitly allocate the number of clusters that will be allocated. This allows the user to suit the stack to the amount of physical memory on the board.

flags

This parameter is reserved for future use. Its value should be zero.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE(pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_LONG(pDrvCtrl, reg, data)
SYS_IN_LONG(pDrvCtrl, reg, data)
SYS_ENET_ADDR_GET(pDrvCtrl, pAddress)
sysLanAuIntEnable(pDrvCtrl->intLevel)
sysLanAuIntDisable(pDrvCtrl->intLevel)
sysLanAuEnetAddrGet(pDrvCtrl, enetAdrs)
```

There are default values in the source code for these macros. They presume memory mapped accesses to the device registers and the **intConnect()**, and **intEnable()** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, **SYS_INT_ENABLE** and **SYS_INT_DISABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine **intConnect()**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this routine is not implemented.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine **sysLanAuIntEnable()**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine **sysLanAuIntDisable()**.

The macro **SYS_ENET_ADDR_GET** is used get the ethernet hardware of the chip. This macro calls an external board level routine namely **sysLanAuEnetAddrGet()** to get the ethernet address.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore

- one interrupt vector
- 64 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1520 bytes for receive frames and transmit frames.

INCLUDES **end.h endLib.h etherMultiLib.h auEnd.h**

SEE ALSO **muxLib, endLib, netBufLib, VxWorks Device Driver Developer's Guide**

bcm1250MacEnd

NAME **bcm1250MacEnd** – END style BCM1250 MAC Ethernet driver

ROUTINES **bcm1250MacEndLoad()** – initialize the driver and device
bcm1250MacRxDmaShow() – display RX DMA register values
bcm1250MacTxDmaShow() – display TX DMA register values
bcm1250MacShow() – display the MAC register values
bcm1250MacPhyShow() – display the physical register values

DESCRIPTION This module implements the Broadcom BCM1250 on-chip ethernet MACs. The BCM1250 ethernet DMA has two channels, but this module only supports channel 0. The dual DMA channel feature is intended for packet classification and quality of service applications.

EXTERNAL INTERFACE

The only external interface is the **bcm1250MacEndLoad()** routine, which has the *initString* as its only parameter. The *initString* parameter must be a colon- delimited string in the following format:

unit:hwunit:vecnum:flags:numRds0:numTds0

TARGET-SPECIFIC PARAMETERS

unit

This parameter defines which ethernet interface is being loaded.

hwunit

This parameter is no longer used, but must be present so the string can be parsed properly. Its value should be zero.

vecnum

This parameter specifies the interrupt vector number. This driver configures the MAC device to generate hardware interrupts for various events within the device; thus it

contains an interrupt handler routine. The driver calls **bcm1250IntConnect()** to connect its interrupt handler to this interrupt vector.

flags

Device specific flags, for future use. Its value should be zero.

numRds0

This parameter specifies the number of receive DMA buffer descriptors for DMA channel 0.

numTds0

This parameter specifies the number of transmit DMA buffer descriptors for DMA channel 0.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 68 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1520 bytes for receive frames and and transmit frames.

INCLUDES **endLib.h etherMultiLib.h bcm1250MacEnd.h**

SEE ALSO **muxLib, endLib, netBufLib, VxWorks Device Driver Developer's Guide**

bio

NAME **bio** – buffer IO Implementation

ROUTINES **bioInit()** – initialize the bio library
bio_done() – terminates a bio operation
bio_alloc() – allocate memory blocks
bio_free() – free the bio memory

DESCRIPTION This library implements the buffer IO (BIO) library.

INCLUDE FILES **drv/xbd/bio.h, drv/xbd/xbd.h**

cisLib

NAME	cisLib – PCMCIA CIS library
ROUTINES	cisGet() – get information from a PC card's CIS cisFree() – free tuples from the linked list cisConfigregGet() – get the PCMCIA configuration register cisConfigregSet() – set the PCMCIA configuration register
DESCRIPTION	This library contains routines to manipulate the CIS (Configuration Information Structure) tuples and the card configuration registers. The library uses a memory window which is defined in pcmciaMemwin to access the CIS of a PC card. All CIS tuples in a PC card are read and stored in a linked list, cisTupleList . If there are configuration tuples, they are interpreted and stored in another link list, cisConfigList . After the CIS is read, the PC card's enabler routine allocates resources and initializes a device driver for the PC card. If a PC card is inserted, the CSC (Card Status Change) interrupt handler gets a CSC event from the PCMCIA chip and adds a cisGet() job to the PCMCIA daemon. The PCMCIA daemon initiates the cisGet() work. The CIS library reads the CIS from the PC card and makes a linked list of CIS tuples. It then enables the card. If the PC card is removed, the CSC interrupt handler gets a CSC event from the PCMCIA chip and adds a cisFree() job to the PCMCIA daemon. The PCMCIA daemon initiates the cisFree() work. The CIS library frees allocated memory for the linked list of CIS tuples.
INCLUDE FILES	none

cisShow

NAME	cisShow – PCMCIA CIS show library
ROUTINES	cisShow() – show CIS information
DESCRIPTION	This library provides a show routine for CIS tuples. This is provided for engineering debug use. This module uses floating point calculations. Any task calling cisShow() needs to have the VX_FP_TASK bit set in the task flags.
INCLUDE FILES	none

cmdLineBuild

NAME	cmdLineBuild – source file included in sysLib.c for command-line builds
ROUTINES	hardWareInterFaceInit() – Hardware Interface Pre-Kernel Initialization vxbDevInit() – HWIF Post-Kernel Init vxbDevConnect() – HWIF Post-Kernel Connection
DESCRIPTION	This library contains the hwif initialization for the command line build environment.
INCLUDE FILES	vxWorks.h config.h

ctB69000Vga

NAME	ctB69000Vga – a CHIPS B69000 initialization source module
ROUTINES	ctB69000VgaInit() – initializes the B69000 chip and loads font in memory.
DESCRIPTION	The 69000 is the first product in the CHIPS family of portable graphics accelerator product line that integrates high performance memory technology for the graphics frame buffer. Based on the proven HiQVideo graphics accelerator core, the 69000 combines state-of-the-art flat panel controller capabilities with low power, high performance integrated memory. The result is the start of a high performance, low power, highly integrated solution for the premier family of portable graphics products.

High Performance Integrated Memory

The 69000 is the first member of the HiQVideo family to provide integrated high performance synchronous DRAM (SDRAM) memory technology. Targeted at the mainstream notebook market, the 69000 incorporates 2MB of proprietary integrated SDRAM for the graphics/video frame buffer. The integrated SDRAM memory can support up to 83MHz operation, thus increasing the available memory bandwidth for the graphics subsystem. The result is support for additional high color / high resolution graphics modes combined with real-time video acceleration. This additional bandwidth also allows more flexibility in the other graphics functions intensely used in Graphical User Interfaces (GUIs) such as Microsoft Windows.

Frame-Based AGP Compatibility

The 69000 graphics is designed to be used with either 33MHz PCI, or with AGP as a frame-based AGP device, allowing it to be used with the AGP interface provided by the latest core logic chipsets.

HiQColor TM Technology

The 69000 integrates CHIPS breakthrough HiQColor technology. Based on the CHIPS proprietary TMED (Temporal Modulated Energy Distribution) algorithm, HiQColor technology is a unique process that allows the display of 16.7 million true colors on STN panels without using Frame Rate Control (FRC) or dithering. In addition, TMED also reduces the need for the panel tuning associated with current FRC-based algorithms. Independent of panel response, the TMED algorithm eliminates all of the flaws (such as shimmer, Mach banding, and other motion artifacts) normally associated with dithering and FRC. Combined with the new fast response, high-contrast, and low-crosstalk technology found in new STN panels, HiQColor technology enables the best display quality and color fidelity previously only available with TFT technology.

Versatile Panel Support

The HiQVideo family supports a wide variety of monochrome and color Single-Panel, Single-Drive (SS) and Dual-Panel, Dual Drive (DD), standard and high-resolution, passive STN and active matrix TFT/MIM LCD, and EL panels. With HiQColor technology, up to 256 gray scales are supported on passive STN LCDs. Up to 16.7M different colors can be displayed on passive STN LCDs and up to 16.7M colors on 24-bit active matrix LCDs.

The 69000 offers a variety of programmable features to optimize display quality. Vertical centering and stretching are provided for handling modes with less than 480 lines on 480-line panels. Horizontal and vertical stretching capabilities are also available for both text and graphics modes for optimal display of VGA text and graphics modes on 800x600, 1024x768 and 1280x1024 panels.

Television NTSC/PAL Flicker Free Output

The 69000 uses a flicker reduction process which makes text of all fonts and sizes readable by reducing the flicker and jumping lines on the display.

HiQVideo T Multimedia Support

The 69000 uses independent multimedia capture and display systems on-chip. The capture system places data in display memory (usually off screen) and the display system places the data in a window on the screen.

Low Power Consumption

The 69000 uses a variety of advanced power management features to reduce power consumption of the display sub-system and to extend battery life. Optimized for 3.3V operation, the 69000 internal logic, bus and panel interfaces operate at 3.3V but can tolerate 5V operation.

Software Compatibility / Flexibility

The HiQVideo controllers are fully compatible with the VGA standard at both the register and BIOS levels. CHIPS and third-party vendors supply a fully VGA compatible BIOS, end-user utilities and drivers for common application programs.

Acceleration for All Panels and All Modes

The 69000 graphics engine is designed to support high performance graphics and video acceleration for all supported display resolutions, display types, and color modes.

There is no compromise in performance operating in 8, 16, or 24 bpp color modes allowing true acceleration while displaying up to 16.7M colors.

USAGE This library provides initialization routines to configure CHIPS B69000 (VGA) in alphanumeric mode.

The functions addressed here include:

- Initialization of CHIPS B69000 IC.

USER INTERFACE

```
STATUS ctB69000VgaInit
(
    VOID
)
```

This routine will initialize the VGA card if present in PCI connector, sets up register set in VGA 3+ mode and loads the font in plane 2.

INCLUDE FILES NONE.

dec21x40End

NAME **dec21x40End** – END-style DEC 21x40 PCI Ethernet network interface driver

ROUTINES
endTok_r() – get a token string (modified version)
dec21x40EndLoad() – initialize the driver and device
dec21140SromWordRead() – read two bytes from the serial ROM
dec21x40PhyFind() – Find the first PHY connected to DEC MII port.
dec21145SPIReadBack() – Read all PHY registers out

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The driver provides one standard external interface, **dec21x40EndLoad()**. As input, this function expects a string of colon-separated parameters. The parameters should be specified as hexadecimal strings (optionally preceded by "0x" or a minus sign "-"). Although the parameter string is parsed using **endTok_r()**, each parameter is converted from string to binary by a call to:

```
strtoul(parameter, NULL, 16).
```

The format of the parameter string is:

```
"<deviceAddr>:<pciAddr>:<iVec>:<iLevel>:<numRds>:<numTds>:<
<memBase>:<memSize>:<userFlags>:<phyAddr>:<pPhyTbl>:<phyFlags>:<offset>:<
<loanBufs>:<drvFlags>"
```

TARGET-SPECIFIC PARAMETERS*deviceAddr*

This is the base address at which the hardware device registers are located.

pciAddr

This parameter defines the main memory address over the PCI bus. It is used to translate a physical memory address into a PCI-accessible address.

iVec

This is the interrupt vector number of the hardware interrupt generated by this Ethernet device. The driver uses `intConnect()` to attach an interrupt handler for this interrupt. The BSP can change this by modifying the global pointer `dec21x40IntConnectRtn` with the desired routines (usually `pciIntConnect`).

iLevel

This parameter defines the level of the hardware interrupt.

numRds

The number of receive descriptors to use. This controls how much data the device can absorb under load. If this is specified as NONE (-1), the default of 32 is used.

numTds

The number of transmit descriptors to use. This controls how much data the device can absorb under load. If this is specified as NONE (-1) then the default of 64 is used.

memBase

This parameter specifies the base address of a DMA-able cache-free pre-allocated memory region for use as a memory pool for transmit/receive descriptors and buffers including loaner buffers. If there is no pre-allocated memory available for the driver, this parameter should be -1 (NONE). In which case, the driver allocates cache safe memory for its use using `cacheDmaAlloc()`.

memSize

The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter. When specified this value must account for transmit/receive descriptors and buffers and loaner buffers

userFlags

User flags control the run-time characteristics of the Ethernet chip. Most flags specify non default CSR0 and CSR6 bit values. Refer to `dec21x40End.h` for the bit values of the flags and to the device hardware reference manual for details about device capabilities, CSR6 and CSR0.

phyAddr

This optional parameter specifies the address on the MII (Media Independent Interface) bus of a MII-compliant PHY (Physical Layer Entity). The module that is responsible for optimally configuring the media layer will start scanning the MII bus from the address in *phyAddr*. It will retrieve the PHY's address regardless of that, but, since the MII management interface, through which the PHY is configured, is a very slow one,

providing an incorrect or invalid address may result in a particularly long boot process. If the flag *DEC_USR_MII* is not set, this parameter is ignored.

pPhyTbl

This optional parameter specifies the address of a auto-negotiation table for the PHY being used. The user only needs to provide a valid value for this parameter if he wants to affect the order how different technology abilities are negotiated. If the flag *DEC_USR_MII* is not set, this parameter is ignored.

phyFlags

This optional parameter allows the user to affect the PHY's configuration and behaviour. See below, for an explanation of each MII flag. If the flag *DEC_USR_MII* is not set, this parameter is ignored.

offset

This parameter defines the offset which is used to solve alignment problem.

loanBufs

This optional parameter allows the user to select the amount of loaner buffers allocated for the driver's net pool to be loaned to the stack in receive operations. The default number of loaner buffers is 16. The number of loaner buffers must be accounted for when calculating the memory size specified by memSize.

drvFlags

This optional parameter allows the user to enable driver-specific features.

Device Type:

although the default device type is DEC 21040, specifying the **DECUSR21140** flag bit turns on DEC 21140 functionality.

Ethernet Address:

the Ethernet address is retrieved from standard serial ROM on both DEC 21040, and DEC 21140 devices. If the retrieve from ROM fails, the driver calls the **sysDec21x40EnetAddrGet()** BSP routine. Specifying **DECUSRXEA** flag bit tells the driver should, by default, retrieve the Ethernet address using the **sysDec21x40EnetAddrGet()** BSP routine.

Priority RX processing:

the driver programs the chip to process the transmit and receive queues at the same priority. By specifying **DECUSRBAR_RX**, the device is programmed to process receives at a higher priority.

TX poll rate:

by default, the driver sets the Ethernet chip into a non-polling mode. In this mode, if the transmit engine is idle, it is kick-started every time a packet needs to be transmitted. Alternatively, the chip can be programmed to poll for the next available transmit descriptor if the transmit engine is in idle state. The poll rate is specified by one of **DECUSRTAP_xxx** flags.

Cache Alignment:

the **DEC_USR_CAL_xxx** flags specify the address boundaries for data burst transfers.

DMA burst length:

the **DEC_USR_PBL_xxx** flags specify the maximum number of long words in a DMA burst.

PCI multiple read:

the **DEC_USR_RML** flag specifies that a device supports PCI memory-read-multiple.

Full Duplex Mode:

when set, the **DEC_USR_FD** flag allows the device to work in full duplex mode, as long as the PHY used has this capability. It is worth noting here that in this operation mode, the dec21x40 chip ignores the Collision and the Carrier Sense signals.

MII interface:

some boards feature an MII-compliant Physical Layer Entity (PHY). In this case, and if the flag **DEC_USR_MII** is set, then the optional fields *phyAddr*, *pPhyTbl*, and *phyFlags* may be used to affect the PHY's configuration on the network.

10Base-T Mode:

when the flag **DEC_USR_MII_10MB** is set, then the PHY will negotiate this technology ability, if present.

100Base-T Mode:

when the flag **DEC_USR_MII_100MB** is set, then the PHY will negotiate this technology ability, if present.

Half duplex Mode:

when the flag **DEC_USR_MII_HD** is set, then the PHY will negotiate this technology ability, if present.

Full duplex Mode:

when the flag **DEC_USR_MII_FD** is set, then the PHY will negotiate this technology ability, if present.

Auto-negotiation:

the driver's default behaviour is to enable auto-negotiation, as defined in "IEEE 802.3u Standard". However, the user may disable this feature by setting the flag **DEC_USR_MII_NO_AN** in the *phyFlags* field of the load string.

Auto-negotiation table:

the driver's default behaviour is to enable the standard auto-negotiation process, as defined in "IEEE 802.3u Standard". However, the user may wish to force the PHY to negotiate its technology abilities a subset at a time, and according to a particular order. The flag **DEC_USR_MII_AN_TBL** in the *phyFlags* field may be used to tell the driver that the PHY should negotiate its abilities as dictated by the

entries in the *pPhyTbl* of the load string. If the flag *DEC_USR_MII_NO_AN* is set, this parameter is ignored.

Link monitoring:

this feature enables the netTask to periodically monitor the PHY's link status for link down events. If any such event occurs, and if the flag *DEC_USR_MII_BUS_MON* is set, then a driver's optionally provided routine is executed, and the link is renegotiated.

Transmit threshold value:

the *DEC_USR_THR_XXX* flags enable the user to choose among different threshold values for the transmit FIFO. Transmission starts when the frame size within the transmit FIFO is larger than the threshold value. This should be selected taking into account the actual operating speed of the PHY. Again, see the device hardware reference manual for details.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions and provides a hook function:

sysLanIntEnable()

```
void sysLanIntEnable (int level)
```

This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level.

sysLanIntDisable()

```
void sysLanIntDisable (void)
```

This routine provides a target-specific interface for disabling Ethernet device interrupts.

sysDec21x40EnetAddrGet()

```
STATUS sysDec21x40EnetAddrGet (int unit, char *enetAdrs)
```

This routine provides a target-specific interface for accessing a device Ethernet address.

_func_dec21x40MediaSelect

```
FUNCPTR _func_dec21x40MediaSelect
```

If **_func_dec21x40MediaSelect** is NULL, this driver provides a default media-select routine that reads and sets up physical media using the configuration information from a Version 3 DEC Serial ROM. Any other media configuration can be supported by initializing **_func_dec21x40MediaSelect**, typically in **sysHwInit()**, to a target-specific media select routine.

A media select routine is typically defined as:

```
STATUS decMediaSelect
(
    DEC21X40_DRV_CTRL *      pDrvCtrl,    /* driver control */
    UINT *                   pCsr6Val   /* CSR6 return value */
)
{
```

```
    ...  
}
```

The *pDrvCtrl* parameter is a pointer to the driver control structure that this routine can use to access the Ethernet device. The driver control structure member **mediaCount**, is initialized to 0xff at startup, while the other media control members (**mediaDefault**, **mediaCurrent**, and **gprModeVal**) are initialized to zero. This routine can use these fields in any manner. However, all other driver control structure members should be considered read-only and should not be modified.

This routine should reset, initialize, and select an appropriate media. It should also write necessary the CSR6 bits (port select, PCS, SCR, and full duplex) to the memory location pointed to by *pCsr6Val*. The driver uses this value to program register CSR6. This routine should return **OK** or **ERROR**.

```
_func_dec21x40NanoDelay  
VOIDFUNC PTR _func_dec21x40NanoDelay
```

This driver uses a delay function that is dependent on the speed of the microprocessor. The delays generated by the generic driver delay function should be sufficient for most processors but are likely to cause some excessively slow functionality especially on the slower processors. On the other hand, insufficient delays generated on extremely fast processors may cause networking failures.

The variable *_func_dec21x40NanoDelay* may be used by the BSP to point to a function which will force a delay of a specified number of nanoseconds. The delay does not need to be very accurate but it must be equal to or greater than the requested amount. Typically *_func_dec21x40NanoDelay* will be initialized in *sysHwInit()* to a target-specific delay routine.

A 1nS delay routine is typically defined as:

```
void sysNanoDelay  
{  
    UINT32 nsec /* number of nanoseconds to delay */  
}  
{  
    volatile int delay;  
    volatile int i;  
  
    if (nsec < 100)  
        return; /* slow processor */  
  
    delay = FUDGE * nsec;  
    for (i=0; i<delay; i++)  
        ;  
}  
  
_func_dec21x40NanoDelay = sysNanoDelay;
```

The *nsec* parameter specifies the number of nanoseconds of delay to generate.

```
"_func_dec2114xIntAck"" 9 -1
VOIDFUNC PTR _func_dec2114xIntAck
```

This driver does acknowledge the LAN interrupts. However if the board hardware requires specific interrupt acknowledgement, not provided by this driver, the BSP should define such a routine and attach it to the driver via `_func_dec2114xIntAck`.

PCI ID VALUES	The dec21xxx series chips are now owned and manufactured by Intel. Chips may be identified by either PCI Vendor ID. ID value 0x1011 for Digital, or ID value 0x8086 for Intel. Check the Intel web site for latest information. The information listed below may be out of date.
----------------------	--

Chip	Vendor ID	Device ID
dec 21040	0x1011	0x0002
dec 21041	0x1011	0x0014
dec 21140	0x1011	0x0009
dec 21143	0x1011	0x0019
dec 21145	0x8086	0x0039

INCLUDE FILES	none
----------------------	------

SEE ALSO	<i>ifLib</i> , "DECchip 21040 Ethernet LAN Controller for PCI, ", "Digital Semiconductor 21140A PCI Fast Ethernet LAN Controller, ", "Using the Digital Semiconductor 21140A with Boot ROM, Serial ROM, and External Register: An Application Note", "Intel 21145 Phoneline/Ethernet LAN Controller Hardware Ref. Manual", "Intel 21145 Phoneline/Ethernet LAN Controller Specification Update"
-----------------	---

device

NAME	device – Device Infrastructure Library
-------------	---

ROUTINES	devInit() – initialize the device manager devAttach() – attach a device devDetach() – detach a device devMap() – map a device devMapUnsafe() – map a device unconditionally devUnmap() – unmap a device devUnmapUnsafe() – unmap a device unconditionally devName() – name a device
-----------------	--

DESCRIPTION	This library provides the interface for the device infrastructure.
--------------------	--

INCLUDE FILES	drv/manager/device.h
----------------------	-----------------------------

drvDownLoad

NAME	drvDownLoad -- downloadable driver support module
ROUTINES	drvUnloadAll() – unload all drivers which provide download/unload support
DESCRIPTION	This library implements the downloadable driver module support.
INCLUDE FILES	vxBus.h

el3c90xEnd

NAME	el3c90xEnd – END network interface driver for 3COM 3C90xB XL
ROUTINES	el3c90xEndLoad() – initialize the driver and device el3c90xInitParse() – parse the initialization string
DESCRIPTION	This module implements the device driver for the 3COM EtherLink XI and Fast EtherLink XL PCI network interface cards.

The 3c90x PCI ethernet controller is inherently little endian because the chip is designed to operate on a PCI bus which is a little endian bus. The software interface to the driver is divided into three parts. The first part is the PCI configuration registers and their set up. This part is done at the BSP level in the various BSPs which use this driver. The second and third part are dealt in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the buffers.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These target-specific values and the external support routines are described below.

This driver supports multiple units per CPU. The driver can be configured to support big-endian or little-endian architectures. It contains error recovery code to handle known device errata related to DMA activity.

Big endian processors can be connected to the PCI bus through some controllers which take care of hardware byte swapping. In such cases all the registers which the chip DMAs to, have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to be programmed to operate in little endian mode as it is on the PCI bus. If the CPU board hardware automatically swaps all the accesses to and from the PCI bus, then input and output byte stream need not be swapped.

The 3c90x series chips use a bus-master DMA interface for transferring packets to and from the controller chip. Some of the old 3c59x cards also supported a bus master mode, however for those chips you could only DMA packets to and from a contiguous memory buffer. For transmission this would mean copying the contents of the queued M_BLK chain into a an M_BLK cluster and then DMAing the cluster. This extra copy would sort of defeat the purpose of the bus master support for any packet that doesn't fit into a single M_BLK. By contrast, the 3c90x cards support a fragment-based bus master mode where M_BLK chains can be encapsulated using TX descriptors. This is also called the gather technique, where the fragments in an mBlk chain are directly incorporated into the download transmit descriptor. This avoids any copying of data from the mBlk chain.

NETWORK CARDS SUPPORTED

- 3Com 3c900-TPO 10Mbps/RJ-45
- 3Com 3c900-COMBO 10Mbps/RJ-45,AUI,BNC
- 3Com 3c905-TX 10/100Mbps/RJ-45
- 3Com 3c905-T4 10/100Mbps/RJ-45
- 3Com 3c900B-TPO 10Mbps/RJ-45
- 3Com 3c900B-COMBO 10Mbps/RJ-45,AUI,BNC
- 3Com 3c905B-TX 10/100Mbps/RJ-45
- 3Com 3c905B-FL/FX 10/100Mbps/Fiber-optic
- 3Com 3c980-TX 10/100Mbps server adapter
- Dell Optiplex GX1 on-board 3c918 10/100Mbps/RJ-45

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The only external interface is the **el3c90xEndLoad()** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

*unit:devMemAddr:devIoAddr:pciMemBase:vecNum:intLvl:memAdrs:
memSize:memWidth:flags:buffMultiplier*

The **el3c90xEndLoad()** function uses **strtok()** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit

A convenient holdover from the former model. This parameter is used only in the string name for the driver.

devMemAddr

This parameter in the memory base address of the device registers in the memory map of the CPU. It indicates to the driver where to find the register set. < This parameter should be equal to NONE if the device does not support memory mapped registers.

devIoAddr

This parameter in the IO base address of the device registers in the IO map of some CPUs. It indicates to the driver where to find the RDP register. If both *devIoAddr* and *devMemAddr* are given then the device chooses *devMemAddr* which is a memory mapped register base address. This parameter should be equal to NONE if the device does not support IO mapped registers.

pciMemBase

This parameter is the base address of the CPU memory as seen from the PCI bus. This parameter is zero for most intel architectures.

vecNum

This parameter is the vector associated with the device interrupt. This driver configures the LANCE device to generate hardware interrupts for various events within the device; thus it contains an interrupt handler routine. The driver calls **intConnect()** to connect its interrupt handler to the interrupt vector generated as a result of the LANCE interrupt. The BSP can use a different routine for interrupt connection by changing the point el3c90xIntConnectRtn to point to a different routine.

intLvl

Some targets use additional interrupt controller devices to help organize and service the various interrupt sources. This driver avoids all board-specific knowledge of such devices. During the driver's initialization, the external routine **sysEl3c90xIntEnable()** is called to perform any board-specific operations required to allow the servicing of a NIC interrupt. For a description of **sysEl3c90xIntEnable()**, see "External Support Requirements" below.

memAdrs

This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cache coherent memory for buffers and descriptors from the system pool. The 3C90x NIC is a DMA type of device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the NIC. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

memSize

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. The constant NONE can be used to indicate no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

memWidth

Some target hardware that restricts the shared memory region to a specific location also restricts the access width to this region by the CPU. On these targets, performing an access of an invalid width will cause a bus error.

This parameter can be used to specify the number of bytes of access width to be used by the driver during access to the shared memory. The constant NONE can be used to indicate no restrictions.

Current internal support for this mechanism is not robust; implementation may not work on all targets requiring these restrictions.

flags

This parameter is used for future use, currently its value should be zero.

buffMultiplier

This parameter is used increase the number of buffers allocated in the driver pool. If this parameter is -1 then a default multiplier of 2 is chosen. With a multiplier of 2 the total number of clusters allocated is 64 which is twice the cumulative number of upload and download descriptors. The device has 16 upload and 16 download descriptors. For example on choosing the buffer multiplier of 3, the total number of clusters allocated will be 96 ((16 + 16)*3). There are as many clBlks as the number of clusters. The number of mBlks allocated are twice the number of clBlks. By default there are 64 clusters, 64 clBlks and 128 mBlks allocated in the pool for the device. Depending on the load of the system increase the number of clusters allocated by incrementing the buffer multiplier.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE (pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE (pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_LONG(pDrvCtrl, reg, data)
SYS_IN_LONG(pDrvCtrl, reg, data)
SYS_DELAY (delay)
sysE13c90xIntEnable(pDrvCtrl->intLevel)
sysE13c90xIntDisable(pDrvCtrl->intLevel)
sysDelay (delay)
```

There are default values in the source code for these macros. They presume memory mapped accesses to the device registers and the normal **intConnect()**, and **intEnable()** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, **SYS_INT_ENABLE** and **SYS_INT_DISABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine **intConnect()**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns **OK**.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine **sysEl3c90xIntEnable()**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine **sysEl3c90xIntDisable()**.

The macro **SYS_DELAY** is used for a delay loop. It calls an external board level routine **sysDelay(delay)**. The granularity of delay is one microsecond.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 24072 bytes in text for a I80486 target
- 112 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1536 bytes for receive frames and transmit frames. There are 16 descriptors in the upload ring and 16 descriptors in the download ring. The buffer multiplier by default is 2, which means that the total number of clusters allocated by default are 64 ((upload descriptors + download descriptors)*2). There are as many clBlks as the number of clusters. The number of mBlks allocated are twice the number of clBlks. By default there are 64 clusters, 64 clBlks and 128 mBlks allocated in the pool for the device. Depending on the load of the system increase the number of clusters allocated by incrementing the buffer multiplier.

BIBLIOGRAPHY

3COM 3c90x and 3c90xB NICs Technical reference.

INCLUDES

end.h **endLib.h** **etherMultiLib.h** **el3c90xEnd.h**

SEE ALSO

muxLib, **endLib**, **netBufLib**, *VxWorks Device Driver Developer's Guide*.

elt3c509End

NAME	elt3c509End – END network interface driver for 3COM 3C509
ROUTINES	elt3c509Load() – initialize the driver and device elt3c509Parse() – parse the init string
DESCRIPTION	This module implements the 3COM 3C509 EtherLink III Ethernet network interface driver. This driver is designed to be moderately generic. Thus, it operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver load routine requires an input string consisting of several target-specific values. The driver also requires some external support routines. These target-specific values and the external support routines are described below.
BOARD LAYOUT	This device is on-board. No jumpering diagram is necessary.
EXTERNAL INTERFACE	<p>The only external interface is the elt3c509Load() routine, which expects the <i>initString</i> parameter as input. This parameter passes in a colon-delimited string of the format:</p> <p><i>unit:port:intVector:intLevel:attachementType:nRxFrames</i></p> <p>The elt3c509Load() function uses strtok() to parse the string.</p>
TARGET-SPECIFIC PARAMETERS	<p><i>unit</i></p> <p>A convenient holdover from the former model. This parameter is used only in the string name for the driver.</p> <p><i>intVector</i></p> <p>Configures the ELT device to generate hardware interrupts for various events within the device. Thus, it contains an interrupt handler routine. The driver calls intConnect() to connect its interrupt handler to the interrupt vector generated as a result of the ELT interrupt.</p> <p><i>intLevel</i></p> <p>This parameter is passed to an external support routine, sysEltIntEnable(), which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a ELT interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.</p> <p><i>attachmentType</i></p> <p>This parameter is used to select the transceiver hardware attachment. This is then used by the elt3c509BoardInit() routine to activate the selected attachment. elt3c509BoardInit() is called as a part of the driver's initialization.</p>

nRxFrames

This parameter is used as number of receive frames by the driver.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE(pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_WORD_STRING(pDrvCtrl, reg, pData, len)
SYS_IN_WORD_STRING(pDrvCtrl, reg, pData, len)

sysEltIntEnable(pDrvCtrl->intLevel)
sysEltIntDisable(pDrvCtrl->intLevel)
```

There are default values in the source code for these macros. They presume IO-mapped accesses to the device registers and the normal **intConnect()**, and **intEnable()** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, and **SYS_INT_ENABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine **intConnect()**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns **OK**.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine **sysEltIntEnable()**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine **sysEltIntDisable()**.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one interrupt vector
- 9720 bytes of text
- 88 bytes in the initialized data section (data)
- 0 bytes of bss

The driver requires 1520 bytes of preallocation for Transmit Buffer and 1520*nRxFrames of receive buffers. The default value of nRxFrames is 64 therefore total pre-allocation is (64 + 1)*1520.

TUNING HINTS	nRxFrames parameter can be used for tuning no of receive frames to be used for handling packet receive. More no. of these could help receiving more loaning in case of massive reception.
INCLUDES	end.h endLib.h etherMultiLib.h elt3c509End.h
SEE ALSO	muxLib , endLib , <i>VxWorks Device Driver Developer's Guide</i> .

endLib

NAME	endLib – support library for END-based drivers
ROUTINES	mib2Init() – initialize a MIB-II structure mib2ErrorAdd() – change a MIB-II error count endObjInit() – initialize an END_OBJ structure endObjFlagSet() – set the flags member of an END_OBJ structure endEtherAddressForm() – form an Ethernet address into a packet endEtherPacketDataGet() – return the beginning of the packet data endEtherPacketAddrGet() – locate the addresses in a packet endPollStatsInit() – initialize polling statistics updates endPoolCreate() – create a buffer pool for an END driver endPoolCreate() – create a jumbo buffer pool for an END driver endPoolDestroy() – destroy a pool created with endPoolCreate() endPoolTupleGet() – allocate an mBlk tuple for an END driver endPoolTupleFree() – free an mBlk tuple for an END driver endEtherCrc32BeGet() – calculate a big-endian CRC-32 checksum endEtherCrc32LeGet() – calculate a little-endian CRC-32 checksum endMcacheGet() – allocate an mBlk tuple from the mBlk recycle cache endMcachePut() – store an mBlk tuple in the mBlk recycle cache endMcacheFlush() – flush all tuples from the mBlk recycle cache
DESCRIPTION	This library contains support routines for Enhanced Network Drivers. These routines are common to ALL ENDs. Specialized routines should only appear in the drivers themselves. To use this feature, include the following component: INCLUDE_END
INCLUDE FILES	

erfLib

NAME	erfLib – Event Reporting Framework Library
ROUTINES	<code>erfLibInit()</code> – Initialize the Event Reporting Framework library <code>erfHandlerRegister()</code> – Registers an event handler for a particular event. <code>erfHandlerUnregister()</code> – Registers an event handler for a particular event. <code>erfCategoryAllocate()</code> – Allocates a User Defined Event Category. <code>erfTypeAllocate()</code> – Allocates a User Defined Type for this Category. <code>erfCategoryQueueCreate()</code> – Creates a Category Event Processing Queue. <code>erfCategoriesAvailable()</code> – Get the number of unallocated User Categories. <code>erfTypesAvailable()</code> – Get the number of unallocated User Types for a category. <code>erfEventRaise()</code> – Raises an event.
DESCRIPTION	This module provides an Event Reporting Framework for use by other libraries.
INCLUDE FILES	<code>erfLib.h</code> <code>erfLibP.h</code> <code>vxWorks.h</code> <code>errnoLib.h</code> <code>semLib.h</code> , <code>spinLockLib.h</code> <code>stdio.h</code> <code>stdlib.h</code> <code>string.h</code> <code>taskLib.h</code>

erfShow

NAME	erfShow – Event Reporting Framework Library Show routines
ROUTINES	<code>erfShow()</code> – Shows debug info for this library. <code>erfCategoriesAvailable()</code> – Get the maximum number of Categories. <code>erfCategoriesAvailable()</code> – Get the maximum number of Types. <code>erfDefaultQueueSizeGet()</code> – Get the size of the default queue.
DESCRIPTION	This module provides a Show routine for the an Event Reporting Framework.
INCLUDE FILES	<code>erfLib.h</code> <code>erfLibP.h</code> <code>vxWorks.h</code> <code>stdio.h</code> <code>stdlib.h</code> <code>errnoLib.h</code>

evbNs16550Sio

NAME	evbNs16550Sio – NS16550 serial driver for the IBM PPC403GA evaluation
ROUTINES	<code>evbNs16550HrdInit()</code> – initialize the NS 16550 chip

evbNs16550Int() – handle a receiver/transmitter interrupt for the NS 16550 chip

DESCRIPTION	This is the driver for the National NS 16550 UART Chip used on the IBM PPC403GA evaluation board. It uses the SCCs in asynchronous mode only.
USAGE	An EVBNS16550_CHAN structure is used to describe the chip. The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the register values in the EVBNS16550_CHAN structure (except the SIO_DRV_FUNCS) before calling evbNs16550HrdInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chip interrupt handler evbNs16550Int() via intConnect() .
 IOCTL FUNCTIONS	
	This driver responds to the same ioctl() codes as other serial drivers; for more information, see sioLib.h .
INCLUDE FILES	drv/sio/evbNs16550Sio.h

fei82557End

NAME	fei82557End – END style Intel 82557 Ethernet network interface driver
ROUTINES	fei82557EndLoad() – initialize the driver and device fei82557GetRUStatus() – Return the current RU status and int mask fei82557ShowRxRing() – Show the Receive ring fei82557DumpPrint() – Display statistical counters fei82557ErrCounterDump() – dump statistical counters
DESCRIPTION	This module implements an Intel 82557 and 82559 Ethernet network interface driver. (For the sake of brevity this document will only refer to the 82557.) This is a fast Ethernet PCI bus controller, IEEE 802.3 10Base-T and 100Base-T compatible. It also features a glueless 32-bit PCI bus master interface, fully compliant with PCI Spec version 2.1. An interface to MII compliant physical layer devices is built-in to the card. The 82557 Ethernet PCI bus controller also includes Flash support up to 1 MByte and EEPROM support, although these features are not dealt with in this driver. The 82557 establishes a shared memory communication system with the CPU, which is divided into three parts: the Control/Status Registers (CSR), the Command Block List (CBL) and the Receive Frame Area (RFA). The CSR is on chip and is either accessible with I/O or memory cycles, whereas the other structures reside on the host. The CSR is the main means of communication between the device and the host, meaning that the host issues commands through these registers while the chip posts status changes

in it, occurred as a result of those commands. Pointers to both the CBL and RFA are also stored in the CSR.

The CBL consists of a linked list of frame descriptors through which individual action commands can be performed. These may be transmit commands as well as non-transmit commands, e.g. Configure or Multicast setup commands. While the CBL list may function in two different modes, only the simplified memory mode is implemented in the driver.

The RFA consists of a pair of linked list rings, the Receive Frame Descriptor (RFD) ring and the Receive Buffer Descriptor (RBD) ring. The RFDs hold the status of completed DMAs. The RBDs hold the pointers to the DMA buffers, referred to as clusters.

When the device is initialized or restarted it is passed a pointer to an RFD. This RFD is considered to be the "first" RFD. This RFD holds a pointer to one of the RBDs. This RBD is then considered the "first" RBD. All other RFDs only have a NULL RBD pointer, actually 0xffffffff. Once the device is started the rings are traversed by the device independently.

Either descriptor type RFD or RBD can have a bit set in it to indicate that it is the End of the List (EL). This is initially set in the RBD descriptor immediately before the first RBD. This acts as a stop which prevents the DMA engine from wrapping around the ring and encountering a used descriptor. This is an unallowable condition and results in the device stopping operation without an interrupt or any indication of failure. When the EL RBD is encountered the device goes into the receive stall state. The driver must then restart the device. To reduce, if not eliminate, the occurrence of this costly, time consuming operation, the driver continually advances the EL to the last cleared RBD. Then when the driver services an incoming frame it clears the RFD RBD pair and advances the EL. If the driver is not able to service an incoming frame, because of a shortage of resources such as clusters, the driver will throw that frame away and clear the RFD RBD pair and advance EL.

Because the rings are independently traversed by the device it is imperative that they be kept in sync. Unfortunately, there is no indication from one or the other as to which descriptor it is paired with. It is left to the driver to keep track of which descriptor goes with its counter part. If this synchronization is lost then the performance of the driver will be greatly impaired or worse. To keep this synchronization this driver embeds the RBD descriptors in tags. To do this it utilizes memory that would otherwise have been wasted. The DMA engine purportedly works most efficiently when the descriptors are on a 32 byte boundary. The descriptors are only 16 bytes so there are 16 bytes to work with. The RBD_TAGS have as their first 16 bytes the RBD itself, then it holds the RFD pointer to its counter part, a pointer to itself, a 16 bit index, a 16 bit next index, and 4 bytes of spare. This arrangement allows the driver to traverse only the RBD ring and discover the corresponding RFD through the RBD_TAG and guaranteeing synchronization.

The driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, this driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The driver provides the standard external interface, **fei82557EndLoad()**, which takes a string of colon separated parameters. The parameters should be specified in hexadecimal, optionally preceded by "0x" or a minus sign "-".

The parameter string is parsed using **strtok_r()** and each parameter is converted from a string representation to binary by a call to **strtoul(parameter, NULL, 16)**.

The format of the parameter string is:

"memBase:memSize:nTfds:nRfds:flags:offset:maxRxFrames: clToRfdRatio:nClusters"

In addition, the two global variables **feiEndIntConnect** and **feiEndIntDisconnect** specify respectively the interrupt connect routine and the interrupt disconnect routine to be used depending on the BSP. The former defaults to **intConnect()** and the user can override this to use any other interrupt connect routine (say **pciIntConnect()**) in **sysHwInit()** or any device specific initialization routine called in **sysHwInit()**. Likewise, the latter is set by default to **NULL**, but it may be overridden in the BSP in the same way.

TARGET-SPECIFIC PARAMETERS

memBase

This parameter is passed to the driver via **fei82557EndLoad()**.

The Intel 82557 device is a DMA-type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the 82557.

This parameter can be used to specify an explicit memory region for use by the 82557. This should be done on targets that restrict the 82557 to a particular memory region. Since use of this parameter indicates that the device has limited access to this specific memory region all buffers and descriptors directly accessed by the device (RFDs, RBDs, CFDs, and clusters) must be carved from this region. Since the transmit buffers must reside in this region the driver will revert to using simple mode buffering for transmit meaning that zero copy transmit is not supported. This then requires that there be enough space for clusters to be attached to the CFDs. The minimum memory requirement is for 32 bytes for all descriptors plus at least two 1536 byte clusters for each RFD and one 1536 byte cluster for each CFD. Also, it should be noted that this memory must be non-cached.

The constant **NONE** can be used to indicate that there are no memory limitations, in which case the driver will allocate cache aligned memory for its use using **memalign()**.

memSize

The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as **NONE** (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of descriptors and clusters specified. The amount of memory

allocated must be enough to hold the RFDs, RBDs, CFDs and clusters. The minimum memory requirement is for 32 bytes each for all descriptors, 32 bytes each for alignment of the descriptor types (RFDs, RBDs, and CFDs), plus at least two 1536 byte clusters for each RFD and one 1536 byte cluster for each CFD. Otherwise the End Load routine will return **ERROR**. The number of clusters can be specified by either passing a value in the nCluster parameter, in which case the nCluster value must be at least nRFDs * 2, or by setting the cluster to RFD ratio (clToRfdRatio) to a number equal or greater than 2.

nTfds

This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is less than two, a default of 64 is used.

nRfds

This parameter specifies the number of receive descriptors to be allocated. If this parameter is less than two, a default of 128 is used.

flags

User flags may control the run-time characteristics of the Ethernet chip. Not implemented.

offset

Offset used to align IP header on word boundary for CPUs that need long word aligned access to the IP packet (this will normally be zero or two). This parameter is optional, the default value is zero.

deviceID

This parameter is used to indicate the specific type of device being used, the 82557 or subsequent. This is used to determine if features which were introduced after the 82557 can be used. The default is the 82557. If this is set to any value other than ZERO (0), NONE (-1), or FEI82557_DEVICE_ID (0x1229) it is assumed that the device will support features not in the 82557.

maxRxFrames

This parameter limits the number of frames the receive handler will service in one pass. It is intended to prevent the tNetTask from hogging the CPU and starving applications. This parameter is optional, the default value is nRFDs * 2.

clToRfdRatio

Cluster To RFD Ratio sets the number of clusters as a ratio of nRFDs. The minimum setting for this parameter is 2. This parameter is optional, the default value is 5.

nClusters

Number of clusters to allocate. This value must be at least nRFD * 2. If this value is set then the *clToRfdRatio* is ignored. This parameter is optional, the default is nRFDs * clToRfdRatio.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

```
STATUS sys557Init (int unit, FEI_BOARD_INFO *pBoard)
```

This routine performs any target-specific initialization required before the 82557 device is initialized by the driver. The driver calls this routine every time it wants to [re]initialize the device. This routine returns **OK**, or **ERROR** if it fails.

SYSTEM RESOURCE USAGE

The driver uses **cacheDmaMalloc()** to allocate memory to share with the 82557. The size of this area is affected by the configuration parameters specified in the **fei82557EndLoad()** call.

Either the shared memory region must be non-cacheable, or else the hardware must implement bus snooping. The driver cannot maintain cache coherency for the device because fields within the command structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

TUNING HINTS

The adjustable parameters are:

The number of TFDs and RFDs that will be created at run-time. These parameters are given to the driver when **fei82557EndLoad()** is called. There is one TFD and one RFD associated with each transmitted frame and each received frame respectively. For memory-limited applications, decreasing the number of TFDs and RFDs may be desirable. Increasing the number of TFDs will provide no performance benefit after a certain point. Increasing the number of RFDs will provide more buffering before packets are dropped. This can be useful if there are tasks running at a higher priority than tNetTask.

The maximum receive frames *maxRxFrames*. This parameter will allow the driver to service fixed amount of incoming traffic before forcing the receive handler to relinquish the CPU. This prevents the possible scenario of the receive handler starving the application.

The parameters *clToRfdRatio* and *nClusters* control the number of clusters created which is the major portion of the memory allocated by the driver. For memory-limited applications, decreasing the number clusters may be desirable. However, this also will probably result in performance degradation.

ALIGNMENT

Some architectures do not support unaligned access to 32-bit data items. On these architectures (eg ARM and MIPs), it will be necessary to adjust the offset parameter in the load string to realign the packet. Failure to do so will result in received packets being absorbed by the network stack, although transmit functions should work **OK**. Also, some architectures do not support SNOOPING, for these architectures the utilities FLUSH and INVALIDATE are used for cache coherency of DMA buffers (clusters). These utilities depend on the buffers being cache line aligned and being cache line multiple. Therefore, if memory for these buffers is pre-allocated then it is imperative that this memory be cache line aligned and being cache line multiple.

INCLUDE FILES

none

SEE ALSO

ifLib, Intel 82557 User's Manual, Intel 32-bit Local Area Network (LAN) Component User's Manual

g64120aMf

NAME

g64120aMf – VxBus driver for galileo 64120a system controller

ROUTINES

g64120aMfRegister() – register g64120aMf driver

g64120aMfpDrvCtrlShow() – show pDrvCtrl for g64120a system controller

DESCRIPTION

This is the VxBus driver for the Galileo 64120a system controller.

Note that this is a multi-function device. Therefore, this driver manages the subordinate functions. However, each subordinate function has its own driver. So, for example, the PCI controller on the 64120a has a driver called **g64120aPci**.

INCLUDE FILES

none

g64120aPci

NAME

g64120aPci – VxBus Galileo 64120A PCI bus controller driver

ROUTINES

g64120aPciRegister() – register g64120aPci driver

DESCRIPTION

This module provides VxBus driver support for the PCI bridge in the Marvell/Galileo GT64120A system controller. This bridge is used most frequently on MIPS Malta boards. This driver support for PCI memory mapped and I/O mapped register access, PCI configuration access and PCI bus-master DMA.

INCLUDE FILES

none

SEE ALSO

vxBus, **vxbDmaBufLib**

gei82543End

NAME

gei82543End – Intel 82540/82541/82543/82544/82545/82546/ MAC driver

ROUTINES	gei82543EndLoad() – initialize the driver and device gei82543RegGet() – get the specified register value in 82543 chip gei82543RegSet() – set the specified register value gei82543LedOn() – turn on LED gei82543LedOff() – turn off LED gei82543PhyRegGet() – get the register value in PHY gei82543PhyRegSet() – set the register value in PHY gei82543TbiCompWr() – enable/disable the TBI compatibility workaround gei82543Unit() – return a pointer to the END_DEVICE for a gei unit
DESCRIPTION	<p>The gei82543End driver supports Intel PRO1000 T/F/XF/XT/MT/MF adaptors. These adaptors use Intel 82543GC/82544GC/EI/82540/82541/82545/82546EB/ Gigabit Ethernet controllers. The 8254x are highly integrated, high-performance LAN controllers for 1000/100/10Mb/s transfer rates. They provide 32/64 bit 33/66Mhz interfaces to the PCI bus with 32/64 bit addressing and are fully compliant with PCI bus specification version 2.2. The 82544, 82545 and 82546 also provide PCI-X interface.</p> <p>The 8254x controllers implement all IEEE 802.3 receive and transmit MAC functions. They provide a Ten-Bit Interface (TBI) as specified in the IEEE 802.3z standard for 1000Mb/s full-duplex operation with 1.25 GHz Ethernet transceivers (SERDES), as well as a GMII interface as specified in IEEE 802.3ab for 10/100/1000 BASE-T transceivers, and also an MII interface as specified in IEEE 802.3u for 10/100 BASE-T transceivers.</p> <p>The 8254x controllers offer auto-negotiation capability for TBI and GMII/MII modes and also support IEEE 802.3x compliant flow control. This driver supports the checksum offload features of the 8254x family as follows:</p>

--- Chip ---	---- TX offload capabilities ----	---- RX offload capabilities
82543	TCP/IPv4, UDP/IPv4, IPv4	TCP/IPv4, UDP/IPv4
82544	TCP/IPv4, UDP/IPv4, IPv4	TCP/IPv4, UDP/IPv4, IPv4
8254[5601]	TCP/IPv4, UDP/IPv4, IPv4 TCP/IPv6, UDP/IPv6	TCP/IPv4, UDP/IPv4, IPv4 TCP/IPv6, UDP/IPv6

For the 82540/82541/82545/82546, the driver supports the transport checksum over IPv6 on receive via the packet checksum feature. (The RX IPv6 checksum offload apparently does not function on these chips as documented.) To avoid doing additional work massaging the packet checksum value when the received packets might not be destined for this target, receive checksum offload of TCP or UDP over IPv6 is attempted only when the IPv6 header follows immediately after a 14-byte ethernet header, there are no IPv6 extension headers, and there is no excess padding after the end of the IPv6 payload.

Although these devices also support other features such as jumbo frames, and provide flash support up to 512KB and EEPROM support, this driver does NOT support these features.

The 8254x establishes a shared memory communication system with the CPU, which is divided into two parts: the control/status registers and the receive/transmit descriptors/buffers. The control/status registers are on the 8254x chips and are only accessible with PCI or PCI-X memory cycles, whereas the other structures reside on the

host. The buffer size can be programmed between 256 bytes to 16k bytes. This driver uses the receive buffer size of 2048 bytes for an MTU of 1500.

The Intel PRO/1000 F/XF/MF adapters only implement the TBI mode of the 8254x controller with built-in SERDESs in the adaptors.

The Intel PRO/1000 T adapters based on 82543GC implement the GMII mode with a Gigabit Ethernet Transceiver (PHY) of MARVELL's Alaska 88E1000/88E1000S. However, the PRO/1000 XT/MT adapters based on 82540/82544/82545/82546 use the built-in PHY in controllers.

The driver on the current release supports both GMII mode for Intel PRO1000T/XT/MT adapters and TBI mode for Intel PRO1000 F/XF/MF adapters. However, it requires the target-specific initialization code -- `sys543BoardInit()` -- to distinguish these kinds of adapters by PCI device IDs.

EXTERNAL INTERFACE

The driver provides the standard external interface, `gei82543EndLoad()`, which takes a string of colon separated parameters. The parameter string is parsed using `strtok_r()` and each parameter is converted from a string representation to a binary.

The format of the parameter string is:

`"memBase:memSize:nRxDes:nTxDes:flags:offset:mtu"`

TARGET-SPECIFIC PARAMETERS

memBase

This parameter is passed to the driver via `gei82543EndLoad()`.

The 8254x is a DMA-type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the 8254x.

This parameter can be used to specify an explicit memory region for use by the 8254x chip. This should be done on targets that restrict the 8254x to a particular memory region. The constant **NONE** can be used to indicate that there are such memory, in which case the driver will allocate cache safe memory for its use using `cacheDmaAlloc()`.

memSize

The memory size parameter specifies the size of the pre-allocated memory region. The driver checks the size of the provided memory region is adequate with respect to the given number of transmit Descriptor and Receive Descriptor.

nRxDes

This parameter specifies the number of transmit descriptors to be allocated. If this number is 0, a default value of 24 will be used.

nTxDes

This parameter specifies the number of receive descriptors to be allocated. If this parameter is 0, a default of 24 is used.

flags

This parameter is provided for user to customize this device driver for their application.

GEI_END_SET_TIMER (0x01): a timer will be started to constantly free back the loaned transmit mBlks.

GEI_END_SET_RX_PRIORITY (0x02): packet transfer (receive) from device to host memory will have higher priority than the packet transfer (transmit) from host memory to device in the PCI bus. For end-station application, it is suggested to set this priority in favor of receive operation to avoid receive overrun. However, for routing applications, it is not necessary to use this priority. This option is only for 82543-based adapters.

GEI_END_FREE_RESOURCE_DELAY (0x04): when transmitting larger packets, the driver will hold mblk(s) from the network stack and return them after the driver has completed transmitting the packet, and either the timer has expired or there are no more available descriptors. If this option is not used, the driver will free mblk(s) when ever the packet transmission is done. This option will place greater demands on the network pool and should only be used in systems which have sufficient memory to allocate a large network pool. It is not advised for the memory-limited target systems.

GEI_END_TBI_COMPATIBILITY (0x200): if this driver enables the workaround for TBI compatibility HW bugs (#define INCLUDE_TBI_COMPATIBLE), user can set this bit to enable a software workaround for the well-known TBI compatibility HW bug in the Intel PRO1000 T adapter. This bug is only occurred in the copper-and-82543-based adapter, and the link partner has advertised only 1000Base-T capability.

GEI_END_USER_MEM_FOR_DESC_ONLY (0x400): User can provide memory for this driver through the shMemBase and shMemSize in the load string. By default, this memory is used for TX/RX descriptors and RX buffer. However, if this flag is set, that memory will be only used for TX/RX descriptors, and the driver will malloc other memory for RX buffers and maintain cache coherency for RX buffers. It is user's responsibility to maintain the cache coherence for memory they provided.

GEI_END_FORCE_FLUSH_CACHE: Set this flag to force flushing the data cache for transmit data buffers even when bus snooping is enabled on the target.

GEI_END_FORCE_INVALIDATE_CACHE: Set this flag to force invalidating the data cache for receive data buffers even when bus snooping is enabled on the target.

offset

This parameter is provided for the architectures which need DWORD (4 byte) alignment of the IP header. In that case, the value of OFFSET should be two, otherwise, the default value is zero.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

```
STATUS sys82543BoardInit (int unit, ADAPTOR_INFO *pBoard)
```

This routine performs some target-specific initialization such as EEPROM validation and obtaining ETHERNET address and initialization control words

(ICWs) from EEPROM. The routine also initializes the adaptor-specific data structure. Some target-specific functions used later in driver operation are hooked up to that structure. It's strongly recommended that users provide a delay function with higher timing resolution. This delay function will be used in the PHY's read/write operations if GMII is used. The driver will use **taskDelay()** by default if user can NOT provide any delay function, and this will probably result in very slow PHY initialization process. The user should also specify the PHY's type of MII or GMII. This routine returns **OK**, or **ERROR** if it fails.

SYSTEM RESOURCE USAGE

The driver uses **cacheDmaMalloc()** to allocate memory to share with the 8254xGC. The size of this area is affected by the configuration parameters specified in the **gei82543EndLoad()** call.

Either the shared memory region must be non-cacheable, or else the hardware must implement bus snooping. The driver cannot maintain cache coherency for the device because fields within the command structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

SYSTEM TUNING HINTS

Significant performance gains may be had by tuning the system and network stack. This may be especially necessary for achieving gigabit transfer rates.

Increasing the network stack's pools are strongly recommended. This driver borrows mblk's from the network stack to accelerate packet transmitting. Theoretically, the number borrowed clusters could be the same as the number of the device's transmit descriptors. However, if the network stack has fewer available clusters than available transmit descriptors then this will result in reduced throughput. Therefore, increasing the network stack's number of clusters relative to the number of transmit descriptors will increase bandwidth. Of course this technique will eventually reach a point of diminishing return. There are actually several sizes of clusters available in the network pool. Increasing any or all of these cluster sizes will result in some increase in performance. However, increasing the 2048-byte cluster size will likely have the greatest impact since this size will hold an entire MTU and header.

Increasing the number of receive descriptors and clusters may also have positive impact.

Increasing the buffer size of sockets can also be beneficial. This can significantly improve performance for a target system under higher transfer rates. However, it should be noted that large amounts of unread buffers idling in sockets reduces the resources available to the rest of the stack. This can, in fact, have a negative impact on bandwidth. One method to

reduce this effect is to carefully adjust application tasks' priorities and possibly increase number of receive clusters.

Callback functions defined in the **sysGei82543End.c** can be used to dynamically and/or statically change the internal timer registers such as ITR, RADV, and RDTR to reduce RX interrupt rate.

INCLUDE FILES	none
----------------------	------

SEE ALSO	muxLib , endLib , RS-82543GC GIGABIT ETHERNET CONTROLLER NETWORKING DEVELOPER'S MANUAL
-----------------	--

hwConfig

NAME	hwConfig – vxBus resource library
-------------	--

ROUTINES	hcfDeviceGet() – get the HCF_DEVICE pointer devResourceGet() – find vxBus resource devResourceIntrGet() – find vxBus interrupt resources devResourceIntrShow() – show interrupt values for specified device hcfResourceShow() – show values for specified resource hcfResourceDevShow() – show the device and resource values hcfResourceAllShow() – show all devices and resource values
-----------------	---

DESCRIPTION	This library contains the functions which handle the vxBus hardware resources.
--------------------	--

INCLUDE FILES	hwConf.h
----------------------	-----------------

hwMemLib

NAME	hwMemLib – hardware memory allocation library
-------------	--

ROUTINES	hwMemLibInit() – initialize hardware memory allocation library hwMemPoolCreate() – create or add a memory pool for driver memory allocation hwMemAlloc() – allocate a buffer from the hardware memory pool hwMemFree() – return buffer to the hardware memory pool hwMemShow() –
-----------------	--

DESCRIPTION	This library implements the hardware interface memory management.
--------------------	---

INCLUDE FILES hwMemLib.h

1

i8250Sio

NAME i8250Sio – I8250 serial driver**ROUTINES**
i8250HrdInit() – initialize the chip
i8250Int() – handle a receiver/transmitter interrupt**DESCRIPTION** This is the driver for the Intel 8250 UART Chip used on the PC 386. It uses the SCCs in asynchronous mode only.**USAGE** An I8250_CHAN structure is used to describe the chip. The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the register values in the I8250_CHAN structure (except the SIO_DRV_FUNCS) before calling i8250HrdInit(). The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupt handler (i8250Int) via intConnect().**IOCTL FUNCTIONS**

This driver responds to all the same ioctl() codes as a normal serial driver; for more information, see the comments in **sioLib.h**. As initialized, the available baud rates are 110, 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400.

This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS. The function HUPCL(hang up on last close) is available.

INCLUDE FILES drv/sio/i8250Sio.h

iOlicomEnd

NAME iOlicomEnd – END style Intel Olicom PCMCIA network interface driver**ROUTINES**
iOlicomEndLoad() – initialize the driver and device
iOlicomIntHandle() – interrupt service for card interrupts**BOARD LAYOUT** The device resides on a PCMCIA card and is soft configured. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the END external interface with the following exceptions. The only external interface is the **iOlicomEndLoad()** routine. All of the parameters are passed as strings in a colon (:) separated list to the load function as an initString. The **iOlicomEndLoad()** function uses **strtok()** to parse the string.

The string contains the target specific parameters like this:

```
"io_baseA:attr_baseA:mem_baseA:io_baseB:attr_baseB:mem_baseB: \
ctrl_base:intVectA:intLevelA:intVectB:intLevelB: \
txBdNum:rxBdNum:pShMem:shMemSize"
```

TARGET-SPECIFIC PARAMETERS

I/O base address A

This is the first parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA I/O space for socket A.

Attribute base address A

This is the second parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA attribute space for socket A. On the PID board, this should be the offset of the beginning of the attribute space from the beginning of the memory space.

Memory base address A

This is the third parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA memory space for socket A.

I/O base address B

This is the fourth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA I/O space for socket B.

Attribute base address B

This is the fifth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA attribute space for socket B. On the PID board, this should be the offset of the beginning of the attribute space from the beginning of the memory space.

Memory base address B

This is the sixth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA memory space for socket B.

PCMCIA controller base address

This is the seventh parameter passed to the driver init string. This parameter indicates the base address of the Vadem PCMCIA controller.

interrupt vectors and levels

These are the eighth, ninth, tenth and eleventh parameters passed to the driver init string.

The mapping of IRQs generated at the Card/PCMCIA level to interrupt levels and vectors is system dependent. Furthermore the slot holding the PCMCIA card is not initially known. The interrupt levels and vectors for both socket A and socket B must be passed to **iOlicomEndLoad()**, allowing the driver to select the required parameters later.

number of transmit and receive buffer descriptors

These are the twelfth and thirteenth parameters passed to the driver init string.

The number of transmit and receive buffer descriptors (BDs) used is configurable by the user upon attaching the driver. There must be a minimum of two transmit and two receive BDs, and there is a maximum of twenty transmit and twenty receive BDs. If this parameter is "NULL" a default value of 16 BDs will be used.

offset

This is the fourteenth parameter passed to the driver in the init string.

This parameter defines the offset which is used to solve alignment problem.

base address of buffer pool

This is the fifteenth parameter passed to the driver in the init string.

This parameter is used to notify the driver that space for the transmit and receive buffers need not be allocated, but should be taken from a private memory space provided by the user at the given address. The user should be aware that memory used for buffers must be 4-byte aligned but need not be non-cacheable. If this parameter is "NONE", space for buffers will be obtained by calling **malloc()** in **iOlicomEndLoad()**.

mem size of buffer pool

This is the sixteenth parameter passed to the driver in the init string.

The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter.

Ethernet address

This parameter is obtained from the Card Information Structure on the Olicom PCMCIA card.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support function:

void sysLanIntEnable (int level)

This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level. This routine is called each time that the **iOlicomStart()** routine is called.

void sysLanIntDisable (int level)

This routine provides a target-specific interface for disabling Ethernet device interrupts. The driver calls this routine from the **iOlicomStop()** routine each time a unit is disabled.

```
void sysBusIntAck(void)
```

This routine acknowledge the interrupt if it's necessary.

INCLUDE FILES	none
----------------------	------

SEE ALSO	muxLib , endLib , <i>Intel 82595TX ISA/PCMCIA High Integration Ethernet Controller User Manual</i> , <i>Vadem VG-468 PC Card Socket Controller Data Manual</i> .
-----------------	--

iPIIX4

NAME	iPIIX4 – low level initialization code for PCI ISA/IDE Xcelerator
-------------	---

ROUTINES	iPIIX4Init() – initialize PIIX4 iPIIX4KbdInit() – initializes the PCI-ISA/IDE bridge iPIIX4FdInit() – initializes the floppy disk device iPIIX4AtaInit() – low level initialization of ATA device iPIIX4IntrRoute() – Route PIRQ[A:D] iPIIX4GetIntr() – give device an interrupt level to use
-----------------	--

DESCRIPTION	The 82371AB PCI ISA IDE Xcelerator (PIIX4) is a multi-function PCI device implementing a PCI-to-ISA bridge function, a PCI IDE function, a Universal Serial Bus host/hub function, and an Enhanced Power Management function. As a PCI-to-ISA bridge, PIIX4 integrates many common I/O functions found in ISA-based PC systems—two 82C37 DMA Controllers, two 82C59 Interrupt Controllers, an 82C54 Timer/Counter, and a Real Time Clock. In addition to compatible transfers, each DMA channel supports Type F transfers. PIIX4 also contains full support for both PC/PCI and Distributed DMA protocols implementing PCI-based DMA. The Interrupt Controller has Edge or Level sensitive programmable inputs and fully supports the use of an external I/O Advanced Programmable Interrupt Controller (APIC) and Serial Interrupts. Chip select decoding is provided for BIOS, Real Time Clock, Keyboard Controller, second external microcontroller, as well as two Programmable Chip Selects.
--------------------	---

PIIX4 is a multi-function PCI device that integrates many system-level functions. PIIX4 is compatible with the PCI Rev 2.1 specification, as well as the IEEE 996 specification for the ISA (AT) bus.

PCI to ISA/EIO Bridge

PIIX4 can be configured for a full ISA bus or a subset of the ISA bus called the Extended IO (**EIO**) bus. The use of the **EIO** bus allows unused signals to be configured as general purpose inputs and outputs. PIIX4 can directly drive up to five ISA slots without external data or address buffering. It also provides byte-swap logic, I/O recovery support, wait-state generation, and SYCLK generation. X-Bus chip selects are provided for Keyboard Controller, BIOS, Real Time Clock, a second microcontroller, as

well as two programmable chip selects. PIIX4 can be configured as either a subtractive decode PCI to ISA bridge or as a positive decode bridge. This gives a system designer the option of placing another subtractive decode bridge in the system (e.g., an Intel 380FB Dock Set).

IDE Interface (Bus Master capability and synchronous DMA Mode)

The fast IDE interface supports up to four IDE devices providing an interface for IDE hard disks and CD ROMs. Each IDE device can have independent timings. The IDE interface supports PIO IDE transfers up to 14 Mbytes/sec and Bus Master IDE transfers up to 33 Mbytes/sec. It does not consume any ISA DMA resources. The IDE interface integrates 16x32-bit buffers for optimal transfers.

PIIX4's IDE system contains two independent IDE signal channels. They can be configured to the standard primary and secondary channels (four devices) or primary drive 0 and primary drive 1 channels (two devices). This allows flexibility in system design and device power management.

Compatibility Modules

The DMA controller incorporates the logic of two 82C37 DMA controllers, with seven independently programmable channels. Channels [0:3] are hardwired to 8-bit, count-by-byte transfers, and channels [5:7] are hardwired to 16-bit, count-by-word transfers. Any two of the seven DMA channels can be programmed to support fast Type-F transfers. The DMA controller also generates the ISA refresh cycles.

The DMA controller supports two separate methods for handling legacy DMA via the PCI bus. The PC/PCI protocol allows PCI-based peripherals to initiate DMA cycles by encoding requests and grants via three PC/PCI REQ#/GNT# pairs. The second method, Distributed DMA, allows reads and writes to 82C37 registers to be distributed to other PCI devices. The two methods can be enabled concurrently. The serial interrupt scheme typically associated with Distributed DMA is also supported.

The timer/counter block contains three counters that are equivalent in function to those found in one 82C54 programmable interval timer. These three counters are combined to provide the system timer function, refresh request, and speaker tone. The 14.31818-MHz oscillator input provides the clock source for these three counters.

PIIX4 provides an ISA-Compatible interrupt controller that incorporates the functionality of two 82C59 interrupt controllers. The two interrupt controllers are cascaded so that 14 external and two internal interrupts are possible. In addition, PIIIX4 supports a serial interrupt scheme. PIIIX4 provides full support for the use of an external IO APIC.

Enhanced Universal Serial Bus (USB) Controller

The PIIIX4 USB controller provides enhanced support for the Universal Host Controller Interface (UHCI). This includes support that allows legacy software to use a USB-based keyboard and mouse.

RTC

PIIX4 contains a Motorola MC146818A-compatible real-time clock with 256 bytes of battery-backed RAM. The real-time clock performs two key functions: keeping track of the time of day and storing system data, even when the system is powered down. The RTC operates on a 32.768-kHz crystal and a separate 3V lithium battery that provides up to 7 years of protection.

The RTC also supports two lockable memory ranges. By setting bits in the configuration space, two 8-byte ranges can be locked to read and write accesses. This prevents unauthorized reading of passwords or other system security information. The RTC also supports a date alarm, that allows for scheduling a wake up event up to 30 days in advance, rather than just 24 hours in advance.

GPIO and Chip Selects

Various general purpose inputs and outputs are provided for custom system design. The number of inputs and outputs varies depending on PIIX4 configuration. Two programmable chip selects are provided which allows the designer to place devices on the X-Bus without the need for external decode logic.

Pentium and Pentium II Processor Interface

The PIIX4 CPU interface allows connection to all Pentium and Pentium II processors. The Sleep mode for the Pentium II processors is also supported.

Enhanced Power Management

PIIX4's power management functions include enhanced clock control, local and global monitoring support for 14 individual devices, and various low-power (suspend) states, such as Power-On Suspend, Suspend-to-DRAM, and Suspend-to-Disk. A hardware-based thermal management circuit permits software-independent entrance to low-power states. PIIX4 has dedicated pins to monitor various external events (e.g., interfaces to a notebook lid, suspend/resume button, battery low indicators, etc.). PIIX4 contains full support for the Advanced Configuration and Power Interface (ACPI) Specification.

System Management Bus (SMBus)

PIIX4 contains an SMBus Host interface that allows the CPU to communicate with SMBus slaves and an SMBus Slave interface that allows external masters to activate power management events.

Configurability

PIIX4 provides a wide range of system configuration options. This includes full 16-bit I/O decode on internal modules, dynamic disable on all the internal modules, various peripheral decode options, and many options on system configuration.

USAGE

This library provides low level routines for PCI - ISA bridge initialization, and PCI interrupts routing. There are many functions provided here for enabling different logical devices existing on ISA bus.

The functions addressed here include:

- Creating a logical device using an instance of physical device on PCI bus and initializing internal database accordingly.
- Initializing keyboard (logical device number 11) on PIIX4.
- Initializing floppy disk drive (logical device number 5) on PIIX4.
- Initializing ATA device (IDE interface) on PIIX4.
- Route IRQ[A:D] from PCI expansion slots on given PIIX4.
- Get interrupt level for a given device on PCI expansion slot.

USER INTERFACE STATUS iPIIX4Init
(
)

The above mentioned routine does locates and initializes the PIIX4.

STATUS iPIIX4KbdInit
(
)

The above mentioned routine does keyboard specific initialization on PIIX4.

STATUS iPIIX4FdInit
(
)

The above mentioned routine does floppy disk specific initialization on PIIX4.

STATUS iPIIX4AtaInit
(
)

The above mentioned routine does ATA device specific initialization on PIIX4.

STATUS iPIIX4IntrRoute
(
int pintx, char irq
)

The above mentioned routines routes IRQ[A:D] to interrupt routing state machine embedded in PIIX4 and makes them level triggered. This routine should be called early in boot process.

int iPIIX4GetIntr
(
int pintx
)

This above mentioned routine returns the interrupt level of a PCI interrupt previously set by iPIIX4IntrRoute.

INCLUDE FILES iPIIX4.h

isrDeferLib

NAME	isrDeferLib – ISR deferral library
ROUTINES	<p>isrDeferLibInit() – Initialize the ISR deferral library.</p> <p>isrDeferQueueGet() – Get a deferral queue</p> <p>isrDeferJobAdd() – Add a job to the deferral queue</p> <p>isrDeferIsrReroute() – Reroute deferral work to a new CPU in the system.</p>
DESCRIPTION	<p>This module is used by interrupt service routines to defer interrupt processing from interrupt context to task context. Device drivers use this library by allocating a deferral task, and then enqueueing work on the task's work queue. This library is designed to work with VxBus-compliant device drivers.</p> <p>This module operates in two distinct modes depending on the module's configuration. If the mode is configured for "per-CPU" deferral tasks, this module creates (as needed) a single deferral task on each CPU in the system. This single queue handles all deferral operations performed by device drivers that are servicing their interrupts on that CPU.</p> <p>If the mode is configured for "per-ISR" deferral tasks, this module creates a unique deferral task for each requester, and sets the CPU affinity for the created task to the requested CPU index.</p> <p>Device drivers call isrDeferQueueGet to gain access to a (possibly shared) deferral task. To enqueue work onto the deferral task, a device driver's interrupt service routine initializes the data fields of an ISR_DEFER_JOB structure, and then calls isrDeferJobAdd. It is the responsibility of the driver to ensure that any deferred work enqueued by the driver has been completed before reusing the ISR_DEFER_JOB structure to enqueue additional work.</p> <p>SMP-aware device drivers are informed by the VxBus subsystem when their interrupts have been rerouted to a new CPU in the system. At this point, the driver can call isrDeferIsrReroute to exchange its currently-held deferral queue for a new queue.</p>
INCLUDE FILES	isrDeferLib.h

ixp400I2c

NAME	ixp400I2c – Intel IXP400 I2C source file
ROUTINES	<p>ixp400I2CStart (Control signal)() – initiate an I2C bus transfer</p> <p>ixp400I2CStop (Control signal)() – terminate an I2C bus transfer</p> <p>ixp400I2CAckSend (Control signal)() – send an acknowledgement</p> <p>ixp400I2CAckReceive (Control Signal)() – get an acknowledgement</p>

ixp400I2CByteTransmit() – transmit a byte on the I2C bus
ixp400I2CByteReceive() – receive a byte on the I2C bus
ixp400I2CWriteTransfer() – write to a slave device
ixp400I2CReadTransfer() – read from a slave device

DESCRIPTION This is a driver for the Intel IXP400 I2C bus protocol. When the **INCLUDE_I2C_HW_CTRL** BSP configuration constant is defined, the routines in this module support the IXP465 I2C controller. Otherwise, this is a software only implementation which uses two pins on the IXP400 GPIO Controller.

USAGE An example read from an I2C device using the individual protocol functions is shown below:

```
...
{
if (OK == ixp400I2CStart ())
{
    ixp400I2CByteTransmit ((devAddr & IXP400_I2C_WRITE_MSK));
    ixp400I2CAckReceive();

    ixp400I2CByteTransmit (offset);
    ixp400I2CAckReceive();

    /* Switch to read mode */

    IXP400_I2C_SCL_SET_HIGH;
    IXP400_I2C_SDA_SET_HIGH;

    if(ixp400I2CStart() != OK)
    {
        ixp400I2CStop ();
        return ERROR;
    }

    ixp400I2CByteTransmit ((devAddr | IXP400_I2C_READ_FLAG));
    ixp400I2CAckReceive ();

    for (byteCnt = 0; byteCnt<num; ++byteCnt)
    {
        ixp400I2CByteReceive (Buf);
        ++Buf;

        /* Prevent giving an ACK on the last byte */

        if (byteCnt < (num - 1))
            ixp400I2CAckSend ();
    }

    ixp400I2CStop ();
}
}
```

INCLUDE FILES	ixp400I2c.h
SEE ALSO	" <i>ixp400 Data Sheet</i> ,"

In97xEnd

NAME	In97xEnd – END style AMD Am79C97X PCnet-PCI Ethernet driver
ROUTINES	In97xEndLoad() – initialize the driver and device In97xInitParse() – parse the initialization string
DESCRIPTION	This module implements the Advanced Micro Devices Am79C970A, Am79C971, Am79C972, and Am79C973 PCnet-PCI Ethernet 32-bit network interface driver. The PCnet-PCI ethernet controller is inherently little-endian because the chip is designed to operate on a PCI bus which is a little-endian bus. The software interface to the driver is divided into three parts. The first part is the PCI configuration registers and their set up. This part is done at the BSP level in the various BSPs which use this driver. The second and third part are dealt with in the driver. The second part of the interface is comprised of the I/O control registers and their programming. The third part of the interface is comprised of the descriptors and the buffers. This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These target-specific values and the external support routines are described below. This driver supports multiple units per CPU. The driver can be configured to support big-endian or little-endian architectures. It contains error recovery code to handle known device errata related to DMA activity. Some big-endian processors may be connected to a PCI bus through a host/PCI bridge which performs byte swapping during data phases. On such platforms, the PCnet-PCI controller need not perform byte swapping during a DMA access to memory shared with the host processor.
BOARD LAYOUT	This device is on-board. No jumpering diagram is necessary.
EXTERNAL INTERFACE	The driver provides one standard external interface, In97xEndLoad() . As input, this routine takes a string of colon-separated parameters. The parameters should be specified in hexadecimal (optionally preceded by 0x or a minus sign -). The parameter string is parsed using strtok_r() .

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is:

```
<unit:devMemAddr:devIoAddr:pciMemBase:vecNum:intLvl:  
memAdrs:memSize:memWidth:csr3b:offset:flags>
```

unit

The unit number of the device. Unit numbers start at zero and increase for each device controlled by the same driver. The driver does not use this value directly. The unit number is passed through the MUX API where it is used to differentiate between multiple instances of a particular driver.

devMemAddr

This parameter is the memory mapped I/O base address of the device registers in the memory map of the CPU. The driver will locate device registers as offsets from this base address.

The PCnet presents two registers to the external interface, the RDP (Register Data Port) and RAP (Register Address Port) registers. This driver assumes that these two registers occupy two unique addresses in a memory space that is directly accessible by the CPU executing this driver. The driver assumes that the RDP register is mapped at a lower address than the RAP register; the RDP register is therefore derived from the "base address." This is a required parameter.

devIoAddr

This parameter specifies the I/O base address of the device registers in the I/O map of some CPUs. It indicates to the driver where to find the RDP register. This parameter is no longer used, but is retained so that the load string format will be compatible with legacy initialization routines. The driver will always use memory mapped I/O registers specified via the *devMemAddr* parameter.

pciMemBase

This parameter is the base address of the host processor memory as seen from the PCI bus. This parameter is zero for most Intel architectures.

vecNum

This parameter is the vector associated with the device interrupt. This driver configures the PCnet device to generate hardware interrupts for various events within the device; thus it contains an interrupt handler routine. The driver calls **pciIntConnect()** to connect its interrupt handler to the interrupt vector generated as a result of the PCnet interrupt.

intLvl

Some targets use additional interrupt controller devices to help organize and service the various interrupt sources. This driver avoids all board-specific knowledge of such devices. During the driver's initialization, the external routine **sysLan97xIntEnable()** is called to perform any board-specific operations required to allow the servicing of a PCnet interrupt. For a description of **sysLan97xIntEnable()**, see "External Support Requirements" below.

memAdrs

This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cache coherent memory for buffers and descriptors from the system memory pool. The PCnet device is a DMA type of device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the PCnet. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

memSize

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. The constant NONE can be used to indicate no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

memWidth

Some target hardware that restricts the shared memory region to a specific location also restricts the access width to this region by the CPU. On these targets, performing an access of an invalid width will cause a bus error.

This parameter can be used to specify the number of bytes of access width to be used by the driver during access to the shared memory. The constant NONE can be used to indicate no restrictions.

Current internal support for this mechanism is not robust; implementation may not work on all targets requiring these restrictions.

csr3b

The PCnet-PCI Control and Status Register 3 (CSR3) controls, among other things, big-endian and little-endian modes of operation. When big-endian mode is selected, the PCnet-PCI controller will swap the order of bytes on the AD bus during a data phase on access to the FIFOs only: AD[31:24] is byte 0, AD[23:16] is byte 1, AD[15:8] is byte 2 and AD[7:0] is byte 3. In order to select the big-endian mode, set this parameter to (0x0004). Most implementations, including natively big-endian host architectures, should set this parameter to (0x0000) in order to select little-endian access to the FIFOs, as the driver is currently designed to perform byte swapping as appropriate to the host architecture.

offset

This parameter specifies a memory alignment offset. Normally this parameter is zero except for architectures which can only access 32-bit words on 4-byte aligned address boundaries. For these architectures the value of this offset should be 2.

flags

This parameter is used for future use. Currently its value should be zero.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires five externally defined support functions that can be customized by modifying global pointers. The function pointer types and default "bindings" are specified

below. To change the defaults, the BSP should create an appropriate routine and set the function pointer before first use. This would normally be done within `sysHwInit2()`.

Note that all of the pointers to externally defined functions *must* be set to a valid executable code address. Also, note that `sysLan97xIntEnable()`, `sysLan97xIntDisable()`, and `sysLan97xEnetAddrGet()` must be defined in the BSP. This was done so that the driver would be compatible with initialization code and support routines in existing BSPs.

The function pointer convention has been introduced to facilitate future driver versions that do not explicitly reference a named BSP-defined function. Among other things, this would allow a BSP designer to define, for example, one `endIntEnable()` routine to support multiple END drivers.

In97xIntConnect

```
IMPORT STATUS (* ln97xIntConnect)
(
    VOIDFUNC PTR * vector,      /* interrupt vector to attach to      */
    VOIDFUNC PTR   routine,     /* routine to be called             */
    int           parameter    /* parameter to be passed to routine */
);

/* default setting */

ln97xIntConnect = pciIntConnect;
```

The `ln97xIntConnect` pointer specifies a function used to connect the driver interrupt handler to the appropriate vector. By default it is the `pciIntLib` routine `pciIntConnect()`.

In97xIntDisconnect

```
IMPORT STATUS (* ln97xIntDisconnect)
(
    VOIDFUNC PTR * vector,      /* interrupt vector to attach to      */
    VOIDFUNC PTR   routine,     /* routine to be called             */
    int           parameter    /* routine parameter                 */
);

/* default setting */

ln97xIntDisconnect = pciIntDisconnect2;
```

The `ln97xIntDisconnect` pointer specifies a function used to disconnect the interrupt handler prior to unloading the driver. By default it is the `pciIntLib` routine `pciIntDisconnect2()`.

In97xIntEnable

```
IMPORT STATUS (* ln97xIntEnable)
(
    int level                  /* interrupt level to be enabled */
);

/* default setting */

ln97xIntEnable = sysLan97xIntEnable;
```

The **ln97xIntEnable** pointer specifies a function used to enable the interrupt level for the END device. It is called once during initialization. By default it is a BSP routine named **sysLan97xIntEnable()**. The implementation of this routine can vary between architectures, and even between BSPs for a given architecture family. Generally, the parameter to this routine will specify an interrupt *level* defined for an interrupt controller on the host platform. For example, MIPS and PowerPC BSPs may implement this routine by invoking the WRS **intEnable()** library routine. WRS Intel Pentium BSPs may implement this routine via **sysIntEnablePIC()**.

In97xIntDisable

```
IMPORT STATUS (* ln97xIntDisable)
(
    int level           /* interrupt level to be disabled */
);

/* default setting */

ln97xIntDisable = sysLan97xIntDisable;
```

The **ln97xIntDisable** pointer specifies a function used to disable the interrupt level for the END device. It is called during stop. By default it is a BSP routine named **sysLan97xIntDisable()**. The implementation of this routine can vary between architectures, and even between BSPs for a given architecture family. Generally, the parameter to this routine will specify an interrupt *level* defined for an interrupt controller on the host platform. For example, MIPS and PowerPC BSPs may implement this routine by invoking the WRS **intDisable()** library routine. WRS Intel Pentium BSPs may implement this routine via **sysIntDisablePIC()**.

In97xEnetAddrGet

```
IMPORT STATUS (* ln97xEnetAddrGet)
(LN_97X_DRV_CTRL * pDrvCtrl, char * pStationAddr);

/* default setting */

ln97xEnetAddrGet = sysLan97xEnetAddrGet;
```

The **ln97xEnetAddrGet** pointer specifies a function used to get the Ethernet (IEEE station) address of the device. By default it is a BSP routine named **sysLan97xEnetAddrGet()**.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 14240 bytes in text for a PENTIUM3 target
- 120 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1520 bytes for receive frames and transmit frames.

INCLUDE FILES

none

SEE ALSO

muxLib, **endLib**, **netBufLib**, *VxWorks Device Driver Developer's Guide*, *PCnet-PCI II Single-Chip Full-Duplex Ethernet Controller for PCI Local Bus Product*, *PCnet-FAST Single-Chip Full-Duplex 10/100 Mbps Ethernet Controller for PCI Local Bus Product*

IptDrv

NAME

IptDrv – parallel chip device driver for the IBM-PC LPT

ROUTINES

IptDrv() – initialize the LPT driver
IptDevCreate() – create a device for an LPT port
IptShow() – show LPT statistics

DESCRIPTION

This is the basic driver for the LPT used on the IBM-PC. If the component **INCLUDE_LPT** is enabled, the driver initializes the LPT port on the PC.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, two routines must be called directly: **IptDrv()** to initialize the driver, and **IptDevCreate()** to create devices.

There are one other callable routines: **IptShow()** to show statistics. The argument to **IptShow()** is a channel number, 0 to 2.

Before the driver can be used, it must be initialized by calling **IptDrv()**. This routine should be called exactly once, before any reads, writes, or calls to **IptDevCreate()**. Normally, it is called from **usrRoot()** in **usrConfig.c**. The first argument to **IptDrv()** is a number of channels, 0 to 2. The second argument is a pointer to the resource table. Definitions of members of the resource table structure are:

```
int ioBase;           /* IO base address */  
int intVector;       /* interrupt vector */  
int intLevel;        /* interrupt level */  
BOOL autofeed;       /* TRUE if enable autofeed */  
int busyWait;         /* loop count for BUSY wait */  
int strobeWait;       /* loop count for STROBE wait */  
int retryCnt;         /* retry count */  
int timeout;          /* timeout second for syncSem */
```

IOCTL FUNCTIONS

This driver responds to two functions: **LPT_SETCONTROL** and **LPT_GETSTATUS**. The argument for **LPT_SETCONTROL** is a value of the control register. The argument for **LPT_GETSTATUS** is a integer pointer where a value of the status register is stored.

INCLUDE FILES none**SEE ALSO** the VxWorks programmer guides.

m8260SccEnd

NAME **m8260SccEnd** – END style Motorola MPC8260 network interface driver

ROUTINES **m8260SccEndLoad()** – initialize the driver and device

BOARD LAYOUT This device is on-chip. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard END external interface. The only external interface is the **motSccEndLoad()** routine. The parameters are passed into the **motSccEndLoad()** function as a single colon-delimited string. The **motSccEndLoad()** function uses **strtok()** to parse the string, which it expects to be of the following format:

unit:motCpmAddr:ivec:sccNum:txBdNum:rxBdNum:txBdBase:rxBdBase:bufBase

TARGET-SPECIFIC PARAMETERS

unit

A convenient holdover from the former model. This parameter is used only in the string name for the driver.

motCpmAddr

Indicates the address at which the host processor presents its internal memory (also known as the dual ported RAM base address). With this address, and the SCC number (see below), the driver is able to compute the location of the SCC parameter RAM and the SCC register map, and, ultimately, to program the SCC for proper operations. This parameter should point to the internal memory of the processor where the SCC physically resides. This location might not necessarily be the Dual-Port RAM of the microprocessor configured as master on the target board.

ivec

This driver configures the host processor to generate hardware interrupts for various events within the device. The interrupt-vector offset parameter is used to connect the

driver's ISR to the interrupt through a call to the VxWorks system function **intConnect()**.

sccNum

This driver is written to support multiple individual device units. Thus, the multiple units supported by this driver can reside on different chips or on different SCCs within a single host processor. This parameter is used to explicitly state which SCC is being used (SCC1 is most commonly used, thus this parameter most often equals "1").

txBdNum and *rxBdNum*

Specify the number of transmit and receive buffer descriptors (BDs). Each buffer descriptor resides in 8 bytes of the processor's dual-ported RAM space, and each one points to a 1520 byte buffer in regular RAM. There must be a minimum of two transmit and two receive BDs. There is no maximum, although more than a certain amount does not speed up the driver and wastes valuable dual-ported RAM space. If any of these parameters is "NULL", a default value of "32" BDs is used.

txBdBase and *rxBdBase*

Indicate the base location of the transmit and receive buffer descriptors (BDs). They are offsets, in bytes, from the base address of the host processor's internal memory (see above). Each BD takes up 8 bytes of dual-ported RAM, and it is the user's responsibility to ensure that all specified BDs fit within dual-ported RAM. This includes any other BDs the target board might be using, including other SCCs, SMCs, and the SPI device. There is no default for these parameters. They must be provided by the user.

bufBase

Tells the driver that space for the transmit and receive buffers need not be allocated but should be taken from a cache-coherent private memory space provided by the user at the given address. The user should be aware that memory used for buffers must be 4-byte aligned and non-cacheable. All the buffers must fit in the given memory space. No checking is performed. This includes all transmit and receive buffers (see above). Each buffer is 1520 bytes. If this parameter is "NONE", space for buffers is obtained by calling **cacheDmaMalloc()** in **motSccEndLoad()**.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions:

sysXxxEnetEnable()

This is either **sys360EnetEnable()** or **sysSccEnetEnable()**, based on the actual host processor being used. See below for the actual prototypes. This routine is expected to handle any target-specific functions needed to enable the Ethernet controller. These functions typically include enabling the Transmit Enable signal (TENA) and connecting the transmit and receive clocks to the SCC. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per unit, from the **motCpmEndLoad()** routine.

sysXxxEnetDisable()

This is either **sys360EnetDisable()** or **sysSccEnetDisable()**, based on the actual host processor being used. See below for the actual prototypes. This routine is expected to handle any target-specific functions required to disable the Ethernet controller. This usually involves disabling the Transmit Enable (TENA) signal. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine from the **motCpmEndStop()** routine each time a unit is disabled.

sysXxxEnetAddrGet()

This is either **sys360EnetAddrGet()** or **sysSccEnetAddrGet()**, based on the actual host processor being used. See below for the actual prototypes. The driver expects this routine to provide the six-byte Ethernet hardware address that is used by this unit. This routine must copy the six-byte address to the space provided by *addr*. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine, once per unit, from the **motSccEndLoad()** routine.

In the case of the CPU32, the prototypes of the above mentioned support routines are as follows:

```
STATUS sys360EnetEnable (int unit, UINT32 regBase)
void sys360EnetDisable (int unit, UINT32 regBase)
STATUS sys360EnetAddrGet (int unit, u_char * addr)
```

In the case of the PPC860, the prototypes of the above mentioned support routines are as follows:

```
STATUS sysSccEnetEnable (int unit)
void sysSccEnetDisable (int unit)
STATUS sysSccEnetAddrGet (int unit, UINT8 * addr)
```

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 0 bytes in the initialized data section (data)
- 1272 bytes in the uninitialized data section (BSS)

The data and BSS sections are quoted for the CPU32 architecture and could vary for other architectures. The code size (text) varies greatly between architectures, and is therefore not quoted here.

If the driver allocates the memory to share with the Ethernet device unit, it does so by calling the **cacheDmaMalloc()** routine. For the default case of 32 transmit buffers, 32 receive buffers, and 16 loaner buffers (this is not configurable), the total size requested is 121,600 bytes. If a non-cacheable memory region is provided by the user, the size of this region should be this amount, unless the user has specified a different number of transmit or receive BDs.

This driver can operate only if this memory region is non-cacheable or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the buffers are asynchronously modified by both the driver and the device, and these fields might share the same cache line. Additionally, the chip's dual-ported RAM must be declared as non-cacheable memory where applicable (for example, when attached to a 68040 processor). For more information, see the *Motorola MC68EN360 User's Manual*, *Motorola MPC860 User's Manual*, *Motorola MPC821 User's Manual* *Motorola MPC8260 User's Manual*

INCLUDE FILES	none
----------------------	------

m85xxCpu

NAME	m85xxCpu – device driver for remote Cpu on m85xx CPU with RapidIO
ROUTINES	m85xxCpuRegister() – register m85xxCpu driver
DESCRIPTION	This is the VxBus driver for the M85XX_CPU
INCLUDE FILES	none

m85xxRio

NAME	m85xxRio – RapidIO host controller in PowerPC 85XX CPU
ROUTINES	m85xxRioRegister() – register PowerPC 85xx rapidIO with bus subsystem m85xxRioRegDbgRead() – read rapidIO registers m85xxRioStatusShow() – show state of RapidIO interface displayRapidIOCfgRegs() – displays RIO registers given a base address
DESCRIPTION	This is the RAPIDIO driver to support configuration of the Serial RapidIO device on the MPC8548 chip. It has been tested for configuring another agent serial rapidio 8548 device although may be developed to support other chips easily none were available for testing.
	This is a vxbus driver which needs specific configuration information provided by the board support package. This is done via the hwconf.c file in the BSP. An example is below:

```
const struct hcResource m85xxRio0Resources[] = {  
    { "regBase", HCF_RES_INT, {(void *)RAPIDIO_BASE} },
```

```
{ "deviceBase", HCF_RES_INT, {(void *) (RAPIDIO_ADRS + 0x0000000)}},  
{ "deviceSize", HCF_RES_INT, {(void *) (RAPIDIO_SIZE - 0x0000000)}},  
{ "rioBusAdrs", HCF_RES_INT, {(void *) RAPIDIO_BUS_ADRS }},  
{ "rioBusSize", HCF_RES_INT, {(void *) RAPIDIO_BUS_SIZE }},  
{ "tgtIf", HCF_RES_INT, {(void *) LAWAR_TGTIF_RAPIDIO }},  
{ "localMemAdrs", HCF_RES_INT, {(void *) LOCAL_MEM_LOCAL_ADRS}}}  
}; #define m85xxRio0Num NELEMENTS(m85xxRio0Resources)
```

On top of this each CPU needs configuration information passed in:

```
const struct hcfResource m85xxCPU0Resources[] = {  
    { "regBase", HCF_RES_INT, {(void *) RAPIDIO_BASE }},  
    { "targetID", HCF_RES_INT, {(void *) 0 }},  
    { "outboundWindow0", HCF_RES_INT, {(void *) RIO_CHANNEL_RESERVED }},  
    { "outboundWindow1", HCF_RES_INT, {(void *) RIO_CHANNEL_MAINT }},  
    { "outboundWindow2", HCF_RES_INT, {(void *) RIO_CHANNEL_CFG }},  
    { "outboundWindow3", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow4", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow5", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow6", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow7", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow8", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "inboundWindow0", HCF_RES_INT, {(void *) RIO_CHANNEL_RESERVED }},  
    { "inboundWindow1", HCF_RES_INT, {(void *) RIO_CHANNEL_SM }},  
    { "inboundWindow2", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "inboundWindow3", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "inboundWindow4", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }}  
}; #define m85xxCPU0Num NELEMENTS(m85xxCPU0Resources)  
  
const struct hcfResource m85xxCPU1Resources[] = {  
    { "regBase", HCF_RES_INT, {(void *) RAPIDIO_BASE }},  
    { "targetID", HCF_RES_INT, {(void *) 0x9 }},  
    { "hopCount", HCF_RES_INT, {(void *) 0x0 }},  
    { "outboundWindow0", HCF_RES_INT, {(void *) RIO_CHANNEL_RESERVED }},  
    { "outboundWindow1", HCF_RES_INT, {(void *) RIO_CHANNEL_SM }},  
    { "outboundWindow2", HCF_RES_INT, {(void *) RIO_CHANNEL_TAS_SET }},  
    { "outboundWindow3", HCF_RES_INT, {(void *) RIO_CHANNEL_TAS_CLEAR }},  
    { "outboundWindow4", HCF_RES_INT, {(void *) RIO_CHANNEL_DOORBELL }},  
    { "outboundWindow5", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow6", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow7", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "outboundWindow8", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "inboundWindow0", HCF_RES_INT, {(void *) RIO_CHANNEL_RESERVED }},  
    { "inboundWindow1", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }},  
    { "inboundWindow2", HCF_RES_INT, {(void *) RIO_CHANNEL_UNRESERVED }}  
}; #define m85xxCPU1Num NELEMENTS(m85xxCPU1Resources)
```

```
{ "inboundWindow3", HCF_RES_INT, {(void *)RIO_CHANNEL_UNRESERVED }},
{ "inboundWindow4", HCF_RES_INT, {(void *)RIO_CHANNEL_UNRESERVED }}

}; #define m85xxCPU1Num NELEMENTS(m85xxCPU1Resources)
```

INCLUDE FILES	none
---------------	------

mcf5475Pci

NAME	mcf5475Pci – PCI host controller in Coldfire mcf5475/85 CPU
ROUTINES	mcf5475PciRegister() – register Coldfire mcf5475 host controller sysPciHostBridgeInit() – Initialize the PCI Host Bridge
DESCRIPTION	This module provides VxBus driver support for the PCI bridge in the coldfire onchip system controller. This driver support for PCI memory mapped and I/O mapped register access, PCI configuration access.
INCLUDE FILES	none
SEE ALSO	vxBus , vxbDmaBufLib

miiLib

NAME	miiLib – Media Independent Interface library
ROUTINES	miiPhyInit() – initialize and configure the PHY devices miiPhyUnInit() – uninitialized a PHY miiAnCheck() – check the auto-negotiation process result miiPhyOptFuncMultiSet() – set pointers to MII optional registers handlers miiPhyOptFuncSet() – set the pointer to the MII optional registers handler miiLibInit() – initialize the MII library miiLibUnInit() – uninitialized the MII library miiShow() – show routine for MII library miiRegsGet() – get the contents of MII registers
DESCRIPTION	This module implements a Media Independent Interface (MII) library.

The MII is an inexpensive and easy-to-implement interconnection between the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) media access controllers and the Physical Layer Entities (PHYs).

The purpose of this library is to provide Ethernet drivers in VxWorks with a standardized, MII-compliant, easy-to-use interface to various PHYs. In other words, using the services of this library, network drivers will be able to scan the existing PHYs, run diagnostics, electrically isolate a subset of them, negotiate their technology abilities with other link-partners on the network, and ultimately initialize and configure a specific PHY in a proper, MII-compliant fashion.

In order to initialize and configure a PHY, its MII management interface has to be used. This is made up of two lines: management data clock (MDC) and management data input/output (MDIO). The former provides the timing reference for transfer of information on the MDIO signal. The latter is used to transfer control and status information between the PHY and the MAC controller. For this transfer to be successful, the information itself has to be encoded into a frame format, and both the MDIO and MDC signals have to comply with certain requirements as described in the 802.3u IEEE Standard.

Since no assumption can be made as to the specific MAC-to-MII interface, this library expects the driver's writer to provide it with specialized read and write routines to access that interface. See EXTERNAL SUPPORT REQUIREMENTS below.

```
miiPhyUnInit (), miiLibInit (), miiLibUnInit (), miiPhyOptFuncSet ().

STATUS      miiLibInit (void);
STATUS      miiLibUnInit (void);
```

EXTERNAL SUPPORT REQUIREMENTS

phyReadRtn

```
STATUS      phyReadRtn (DRV_CTRL * pDrvCtrl, UINT8 phyAddr,
                      UINT8 phyReg, UINT16 * value);
```

This routine is expected to perform any driver-specific functions required to read a 16-bit word from the *phyReg* register of the MII-compliant PHY whose address is specified by *phyAddr*. Reading is performed through the MII management interface.

phyWriteRtn

```
STATUS      phyWriteRtn (DRV_CTRL * pDrvCtrl, UINT8 phyAddr,
                        UINT8 phyReg, UINT16 value);
```

This routine is expected to perform any driver-specific functions required to write a 16-bit word to the *phyReg* register of the MII-compliant PHY whose address is specified by *phyAddr*. Writing is performed through the MII management interface.

phyDelayRtn

```
STATUS      phyDelayRtn (UINT32 phyDelayParm);
```

This routine is expected to cause a limited delay to the calling task, no matter whether this is an active delay, or an inactive one. *miiPhyInit()* calls this routine on several occasions throughout the code with *phyDelayParm* as parameter. This represents the

granularity of the delay itself, whereas the field *phyMaxDelay* in **PHY_INFO** is the maximum allowed delay, in *phyDelayParm* units. The minimum elapsed time (*phyMaxDelay * phyDelayParm*) must be 5 seconds.

The user should be aware that some of these events may take as long as 2-3 seconds to be completed, and he should therefore tune this routine and the parameter *phyMaxDelay* accordingly.

If the related field *phyDelayRtn* in the **PHY_INFO** structure is initialized to NULL, no delay is performed.

phyLinkDownRtn

```
STATUS phyLinkDownRtn (DRV_CTRL *);
```

This routine is expected to take any action necessary to re-initialize the media interface, including possibly stopping and restarting the driver itself. It is called when a link down event is detected for any active PHY, with the pointer to the relevant driver control structure as only parameter.

To use this feature, include the following component: **INCLUDE_MIILIB**

INCLUDE FILES

none

SEE ALSO

IEEE 802.3.2000 Standard

motFcc2End

NAME

motFcc2End – Second Generation Motorola FCC Ethernet network interface.

ROUTINES

motFccEndLoad() – initialize the driver and device
motFccDumpRxRing() – Show the Receive Ring details
motFccDumpTxRing() – Show the Transmit Ring details
motFccMiiShow() – debug function to show the Mii settings in the Phy Info structure
motFccMibShow() – Debug Function to show MIB statistics.
motFccShow() – Debug Function to show driver specific control data.
motFccIramShow() – Debug Function to show FCC CP internal ram parameters.
motFccPramShow() – Debug Function to show FCC CP parameter ram.
motFccEramShow() – Debug Function to show FCC CP ethernet parameter ram.
motFccDrvShow() – debug function to show FCC parameter ram addresses, initial BD and cluster settings

DESCRIPTION

This module implements a Motorola Fast Communication Controller (FCC) Ethernet network interface driver. This is a second generation driver that is based on the original **motFccEnd.c**. It differs from the original in initialization, performance, features and SPR fixes.

The driver "load string" interface differs from its predecessor. A parameter that contains a pointer to a predefined array of function pointers was added to the end of the load string. This array replaces multiple individual function pointers for dual ported RAM allocation, MII access, duplex control, and heartbeat and disconnect functionality; it is described more fully below. The array simplifies updating the driver and BSP code independently.

Performance of the driver has been greatly enhanced. A layer of unnecessary queuing was removed. Time-critical functions were re-written to be more fluid and efficient. The driver's work load is distributed between the interrupt and the net job queue. Only one `netJobAdd()` call is made per interrupt. Multiple events pending are sent as a single net job.

A new Generic MIB interface has been implemented.

Several SPRs, written against the original motFccEnd driver and previous motFcc2End versions, are fixed.

The FCC supports several communication protocols. This driver supports the FCC operating in Ethernet mode, which is fully compliant with the IEEE 802.3u 10Base-T and 100Base-T specifications.

The FCC establishes a shared memory communication system with the CPU, which may be divided into three parts: a set of Control/Status Registers (CSR) and FCC-specific parameters, the buffer descriptors (BD), and the data buffers.

Both the CSRs and the internal parameters reside in the MPC8260's internal RAM. They are used for mode control, and to extract status information of a global nature. For instance, by programming these registers, the driver can specify which FCC events should generate an interrupt, whether features like the promiscuous mode or the heartbeat are enabled, and so on. Pointers to both the Transmit Buffer Descriptors ring (TBD) and the Receive Buffer Descriptors ring (RBD) are stored in the internal parameter RAM. The latter also includes protocol-specific parameters, such as the individual physical address of the station and the maximum receive frame length.

The BDs are used to pass data buffers and related buffer information between the hardware and the software. They may reside either on the 60x bus, or on the CPM local bus. They include local status information, and a pointer to the receive or transmit data buffers. These buffers are located in external memory, and may reside on the 60x bus, or the CPM local bus (see below).

This driver is designed to be moderately generic. Without modification, it can operate across all the FCCs in the MPC8260, regardless of where the internal memory base address is located. To achieve this goal, this driver must be given several target-specific parameters and some external support routines. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

BOARD LAYOUT

This device is on-board. No jumper diagram is necessary.

EXTERNAL INTERFACE

The driver provides the standard external interface, **motFccEnd2Load()**, which takes a string of parameters delineated by colons. The parameters should be specified in hexadecimal, optionally preceded by "0x" or a minus sign "-".

The parameter string is parsed using **strtok_r()** and each parameter is converted from a string representation to binary by a call to **strtoul(parameter, NULL, 16)**.

The format of the parameter string is:

```
"unit:immrVal:fccNum:bdBase:bdSize:bufBase:bufSize:fifoTxBase:fifoRxBase:  
:tbdNum:rbdNum:phyAddr:phyDefMode:phyAnOrderTbl: userFlags:function  
table(:maxRxFrames)"
```

TARGET-SPECIFIC PARAMETERS*unit*

This driver is written to support multiple individual device units. This parameter is used to explicitly state which unit is being used. Default is unit 0.

immrVal

Indicates the address at which the host processor presents its internal memory (also known as the internal RAM base address). With this address, and the *fccNum* value (see below), the driver is able to compute the location of the FCC parameter RAM, and, ultimately, to program the FCC for proper operation.

fccNum

This driver is written to support multiple individual device units. This parameter is used to explicitly state which FCC is being used.

bdBase

The Motorola Fast Communication Controller is a DMA-type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the FCC.

This parameter tells the driver that space for both the TBDs and the RBDs need not be allocated, but should be taken from a cache-coherent private memory space provided by the user at the given address. TBDs and RBDs are both 8 bytes each, and individual descriptors must be 8-byte aligned. The driver requires that an additional 8 bytes be provided for alignment, even if *bdBase* is aligned to begin with.

If this parameter is "NONE", space for buffer descriptors is obtained by calling **cacheDmaMalloc()** in **motFccInitMem()**.

bdSize

The *bdSize* parameter specifies the size of the pre-allocated memory region for the BDs. If *bdBase* is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks that the size of the provided memory region is adequate with respect to the given number of Transmit Buffer Descriptors and Receive Buffer Descriptors (plus an additional 8 bytes for alignment).

bufBase

This parameter tells the driver that space for data buffers need not be allocated but should be taken from a cache-coherent private memory space provided at the given address. The memory used for buffers must be 32-byte aligned and non-cacheable. The FCC poses one more constraint, in that DMA cycles may occur even when all the incoming data have already been transferred to memory. This means at most 32 bytes of memory at the end of each receive data buffer may be overwritten during reception. The driver pads that area out, thus consuming some additional memory.

If this parameter is "NONE", space for buffer descriptors is obtained by calling **memalign()** in **motFccInitMem()**.

bufSize

The *bufSize* parameter specifies the size of the pre-allocated memory region for data buffers. If *bufBase* is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks that the size of the provided memory region is adequate with respect to the given number of Receive Buffer Descriptors and a non-configurable number of transmit buffers (**MOT_FCC_TX_CL_NUM**). All the above should fit in the given memory space. This area should also include room for buffer management structures.

fifoTxBase

Indicate the base location of the transmit FIFO, in internal memory. The user does not need to initialize this parameter. The default value (see **MOT_FCC_FIFO_TX_BASE**) is highly optimized for best performance. However, if the user wishes to reserve that very area in internal RAM for other purposes, this parameter may be set to a different value.

If *fifoTxBase* is specified as NONE (-1), the driver uses the default value.

fifoRxBase

Indicate the base location of the receive FIFO, in internal memory. The user does not need to initialize this parameter. The default value (see **MOT_FCC_FIFO_RX_BASE**) is highly optimized for best performance. However, if the user wishes to reserve that very area in internal RAM for other purposes, this parameter may be set to a different value.

If *fifoRxBase* is specified as NONE (-1), the driver uses the default value.

tbdNum

This parameter specifies the number of transmit buffer descriptors (TBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space. If this parameter is less than a minimum number specified in **MOT_FCC_TBD_MIN**, or if it is "NULL", a default value of **MOT_FCC_TBD_DEF_NUM** is used. This parameter should always be an even number since each packet the driver sends may consume more than a single TBD.

rbdNum

This parameter specifies the number of receive buffer descriptors (RBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space, and each one points to a buffer in external RAM. If this parameter is less than a minimum number specified

in MOT_FCC_RBD_MIN, or if it is "NULL", a default value of MOT_FCC_RBD_DEF_NUM is used. This parameter should always be an even number.

phyAddr

This parameter specifies the logical address of a MII-compliant physical device (PHY) that is to be used as a physical media on the network. Valid addresses are in the range 0-31. There may be more than one device under the control of the same management interface. The default physical layer initialization routine scans the whole range of PHY devices starting from the one in *phyAddr*. If this parameter is "MII_PHY_NULL", the default physical layer initialization routine finds out the PHY actual address by scanning the whole range. The one with the lowest address is chosen.

phyDefMode

This parameter specifies the operating mode that is set up by the default physical layer initialization routine in case all the attempts made to establish a valid link failed. If that happens, the first PHY that matches the specified abilities is chosen to work in that mode, and the physical link is not tested.

phyAnOrderTbl

This parameter may be set to the address of a table that specifies the order how different subsets of technology abilities, if enabled, should be advertised by the auto-negotiation process. Unless the flag MOT_FCC_USR_PHY_TBL is set in the userFlags field of the load string, the driver ignores this parameter.

The user does not normally need to specify this parameter, since the default behaviour enables auto-negotiation process as described in IEEE 802.3u.

userFlags

This field enables the user to give some degree of customization to the driver.

MOT_FCC_USR_PHY_NO_AN: The default physical layer initialization routine exploits the auto-negotiation mechanism as described in the IEEE Std 802.3u, to bring a valid link up. According to it, all the link partners on the media take part in the negotiation process, and the highest priority common denominator technology ability is chosen. To prevent auto-negotiation from occurring, set this bit in the user flags.

MOT_FCC_USR_PHY_TBL: In the auto-negotiation process, PHYs advertise all their technology abilities at the same time, and the result is that the maximum common denominator is used. However, this behaviour may be changed, and the user may affect the order how each subset of PHY's abilities is negotiated. Hence, when the **MOT_FCC_USR_PHY_TBL** bit is set, the default physical layer initialization routine looks at the motFccAnOrderTbl[] table and auto-negotiate a subset of abilities at a time, as suggested by the table itself. It is worth noticing here, however, that if the **MOT_FCC_USR_PHY_NO_AN** bit is on, the above table is ignored.

MOT_FCC_USR_PHY_NO_FD: The PHY may be set to operate in full duplex mode, provided it has this ability, as a result of the negotiation with other link partners. However, in this operating mode, the FCC ignores the collision detect and carrier sense

signals. To prevent negotiating full duplex mode, set the **MOT_FCC_USR_PHY_NO_FD** bit in the user flags.

MOT_FCC_USR_PHY_NO_HD: The PHY may be set to operate in half duplex mode, provided it has this ability, as a result of the negotiation with other link partners. To prevent negotiating half duplex mode, set the **MOT_FCC_USR_PHY_NO_HD** bit in the user flags.

MOT_FCC_USR_PHY_NO_100: The PHY may be set to operate at 100Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. To prevent negotiating 100Mbit/s speed, set the **MOT_FCC_USR_PHY_NO_100** bit in the user flags.

MOT_FCC_USR_PHY_NO_10: The PHY may be set to operate at 10Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. To prevent negotiating 10Mbit/s speed, set the **MOT_FCC_USR_PHY_NO_10** bit in the user flags.

MOT_FCC_USR_PHY_ISO: Some boards may have different PHYs controlled by the same management interface. In some cases, it may be necessary to electrically isolate some of them from the interface itself, in order to guarantee a proper behaviour on the medium layer. If the user wishes to electrically isolate all PHYs from the MII interface, the **MOT_FCC_USR_PHY_ISO** bit should be set. The default behaviour is to not isolate any PHY on the board.

MOT_FCC_USR_LOOP: When the **MOT_FCC_USR_LOOP** bit is set, the driver configures the FCC to work in internal loopback mode, with the TX signal directly connected to the RX. This mode should only be used for testing.

MOT_FCC_USR_RMON: When the **MOT_FCC_USR_RMON** bit is set, the driver configures the FCC to work in RMON mode, thus collecting network statistics required for RMON support without the need to receive all packets as in promiscuous mode.

MOT_FCC_USR_BUF_LBUS: When the **MOT_FCC_USR_BUF_LBUS** bit is set, the driver configures the FCC to work as though the data buffers were located in the CPM local bus.

MOT_FCC_USR_BD_LBUS: When the **MOT_FCC_USR_BD_LBUS** bit is set, the driver configures the FCC to work as though the buffer descriptors were located in the CPM local bus.

MOT_FCC_USR_HBC: If the **MOT_FCC_USR_HBC** bit is set, the driver configures the FCC to perform heartbeat check following end of transmission and the HB bit in the status field of the TBD is set if the collision input does not assert within the heartbeat window. The user does not normally need to set this bit.

Function

This is a pointer to the structure **FCC_END_FUNCS**. The structure contains mostly FUNCPTRs that are used as a communication mechanism between the driver and the BSP. If the pointer contains a NULL value, the driver uses system default

functions for the m82xxDpram DPRAM allocation and, obviously, the driver does not support BSP function calls for heartbeat errors, disconnect errors, and PHY status changes that are hardware specific.

```
FUNCPTR miiPhyInit;
```

BSP Mii/Phy Init Function

This function pointer is initialized by the BSP and called by the driver to initialize the MII driver. The driver sets up its PHY settings and then calls this routine. The BSP is responsible for setting BSP specific PHY parameters and then calling the miiPhyInit. The BSP is responsible to set up any call to an interrupt. See miiPhyInit below.

```
FUNCPTR miiPhyInt;
```

Driver Function for BSP to Call on a Phy Status Change

This function pointer is initialized by the driver and called by the BSP. The BSP calls this function when it handles a hardware MII specific interrupt. The driver initializes this to the function motFccPhyLSCInt. The BSP may or may not choose to call this function. It will depend if the BSP supports an interrupt driven PHY. The BSP can also set up the **miiLib** driver to poll. In this case the miiPhy driver calls this function. See **miiLib** for details. Note: Not calling this function when the PHY duplex mode changes results in a duplex mis-match. This causes TX errors in the driver and a reduction in throughput.

```
FUNCPTR miiPhyBitRead;
```

MII Bit Read Function

This function pointer is initialized by the BSP and called by the driver. The driver calls this function when it needs to read a bit from the MII interface. The MII interface is hardware specific.

```
FUNCPTR miiPhyBitWrite;
```

MII Bit Write Function

This function pointer is initialized by the BSP and called by the driver. The driver calls this function when it needs to write a bit to the MII interface. This MII interface is hardware specific.

```
FUNCPTR miiPhyDuplex;
```

Duplex Status Call Back

This function pointer is initialized by the BSP and called by the driver. The driver calls this function to obtain the status of the duplex setting in the PHY.

```
FUNCPTR miiPhySpeed;
```

Speed Status Call Back

This function pointer is initialized by the BSP and called by the driver. The driver calls this function to obtain the status of the speed setting in the PHY. This interface is hardware specific.

```
FUNCPTR hbFail;
```

HeartBeat Fail Indicator

This function pointer is initialized by the BSP and called by the driver. The driver calls this function to indicate an FCC heartbeat error.

```
FUNCPTR intDisc;
```

Disconnect Function

This function pointer is initialized by the BSP and called by the driver. The driver calls this function to indicate an FCC disconnect error.

```
FUNCPTR dpramFree;
```

DPRAM Free routine

This function pointer is initialized by the BSP and called by the driver. The BSP allocates memory for the BDs from this pool. The Driver must free the BD area using this function.

```
FUNCPTR dpramFccMalloc;
```

DPRAM FCC Malloc routine

This function pointer is initialized by the BSP and called by the driver. The Driver allocates memory from the FCC specific POOL using this function.

```
FUNCPTR dpramFccFree;
```

DPRAM FCC Free routine

This function pointer is initialized by the BSP and called by the driver. The Driver frees memory from the FCC specific POOL using this function.

maxRxFrames

The *maxRxFrames* parameter is optional. It limits the number of frames the receive handler services in one pass. It is intended to prevent the tNetTask from monopolizing the CPU and starving applications. The default value is nRFDs * 2.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions.

sysFccEnetEnable()

```
STATUS sysFccEnetEnable (UINT32 immrVal, UINT8 fccNum);
```

This routine is expected to handle any target-specific functions needed to enable the FCC. These functions typically include setting the Port B and C on the MPC8260 so that the MII interface may be used. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine, once per device, from the **motFccStart()** routine.

sysFccEnetDisable()

```
STATUS sysFccEnetDisable (UINT32 immrVal, UINT8 fccNum);
```

This routine is expected to perform any target specific functions required to disable the MII interface to the FCC. This involves restoring the default values for all the Port B and C signals. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine from the **motFccStop()** routine each time a device is disabled.

sysFccEnetAddrGet()

```
STATUS sysFccEnetAddrGet (int unit, UCHAR *address);
```

The driver expects this routine to provide the six-byte Ethernet hardware address that is used by this device. This routine must copy the six-byte address to the space provided by *enetAddr*. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine, once per device, from the **motFccEndLoad()** routine.

```
STATUS sysFccMiiBitWr (UINT32 immrVal, UINT8 fccNum, INT32 bitVal);
```

This routine is expected to perform any target specific functions required to write a single bit value to the MII management interface of a MII-compliant PHY device. The MII management interface is made up of two lines: management data clock (MDC) and management data input/output (MDIO). The former provides the timing reference for transfer of information on the MDIO signal. The latter is used to transfer control and status information between the PHY and the FCC. For this transfer to be successful, the information itself has to be encoded into a frame format, and both the MDIO and MDC signals have to comply with certain requirements as described in the 802.3u IEEE Standard. There is not built-in support in the FCC for the MII management interface. This means that the clocking on the MDC line and the framing of the information on the MDIO signal have to be done in software. Hence, this routine is expected to write the value in *bitVal* to the MDIO line while properly sourcing the MDC clock to a PHY, for one bit time.

```
STATUS sysFccMiiBitRd (UINT32 immrVal, UINT8 fccNum, INT8 * bitVal);
```

This routine is expected to perform any target specific functions required to read a single bit value from the MII management interface of a MII-compliant PHY device. The MII management interface is made up of two lines: management data clock (MDC) and management data input/output (MDIO). The former provides the timing reference for transfer of information on the MDIO signal. The latter is used to transfer control and status information between the PHY and the FCC. For this transfer to be successful, the information itself has to be encoded into a frame format, and both the MDIO and MDC signals have to comply with certain requirements as described in the 802.3u IEEE Standard. There is not built-in support in the FCC for the MII management interface. This means that the clocking on the MDC line and the framing of the information on the MDIO signal have to be done in software. Hence, this routine is expected to read the value from the MDIO line in *bitVal*, while properly sourcing the MDC clock to a PHY, for one bit time.

SYSTEM RESOURCE USAGE

If the driver allocates the memory for the BDs to share with the FCC, it does so by calling the **cacheDmaMalloc()** routine. If this region is provided by the user, it must be from non-cacheable memory.

This driver can operate only if this memory region is non-cacheable or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the BDs are asynchronously modified by both the driver and the device, and these fields share the same cache line.

If the driver allocates the memory for the data buffers to share with the FCC, it does so by calling the memalign() routine. The driver does not need to use cache-safe memory for data buffers, since the host CPU and the device are not allowed to modify buffers asynchronously. The related cache lines are flushed or invalidated as appropriate.

TUNING HINTS	The only adjustable parameters are the number of TBDs and RBDs that are created at run-time. These parameters are given to the driver when motFccEndLoad() is called. There is one RBD associated with each received frame, whereas a single transmit packet frequently uses more than one TBD. For memory-limited applications, decreasing the number of RBDs may be desirable. Decreasing the number of TBDs below a certain point results in substantial performance degradation, and is not recommended. Increasing the number of buffer descriptors can boost performance.
---------------------	--

SPECIAL CONSIDERATIONS

INCLUDE FILES	none
SEE ALSO	ifLib , <i>MPC8260 Fast Ethernet Controller (Supplement to the MPC860 User's Manual)</i> , <i>Motorola MPC860 User's Manual</i>

motFecEnd

NAME	motFecEnd – END style Motorola FEC Ethernet network interface driver
ROUTINES	motFecEndLoad() – initialize the driver and device
DESCRIPTION	This module implements a Motorola Fast Ethernet Controller (FEC) network interface driver. The FEC is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. Hardware support of the Media Independent Interface (MII) is built in the chip. The FEC establishes a shared memory communication system with the CPU, which is divided into two parts: the Control/Status Registers (CSR), and the buffer descriptors (BD). The CSRs reside in the MPC860T Communication Controller's internal RAM. They are used for mode control and to extract status information of a global nature. For instance, the types of events that should generate an interrupt, or features like the promiscuous mode or the max receive frame length may be set programming some of the CSRs properly. Pointers to both the Transmit Buffer Descriptors ring (TBD) and the Receive Buffer Descriptors ring (RBD) are also stored in the CSRs. The CSRs are located in on-chip RAM and must be accessed using the big-endian mode.

The BDs are used to pass data buffers and related buffer information between the hardware and the software. They reside in the host main memory and basically include local status information and a pointer to the actual buffer, again in external memory.

This driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

For versions of the MPC860T starting with revision D.4 and beyond the functioning of the FEC changes slightly. An additional bit has been added to the Ethernet Control Register (ECNTRL), the FEC PIN MUX bit. This bit must be set prior to issuing commands involving the other two bits in the register (ETHER_EN, RESET). The bit must also be set when either of the other two bits are being utilized. For versions of the 860T prior to revision D.4, this bit should not be set.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

The driver provides the standard external interface, **motFecEndLoad()**, which takes a string of colon-separated parameters. The parameters should be specified in hexadecimal, optionally preceded by "0x" or a minus sign "-".

The parameter string is parsed using **strtok_r()** and each parameter is converted from a string representation to binary by a call to **strtoul(parameter, NULL, 16)**.

The format of the parameter string is:

```
"motCpmAddr:ivec:bufBase:bufSize:fifoTxBase:fifoRxBase  
:tbdNum:rbdNum:phyAddr:isoPhyAddr:phyDefMode:userFlags :clockSpeed"
```

TARGET-SPECIFIC PARAMETERS

motCpmAddr

Indicates the address at which the host processor presents its internal memory (also known as the dual ported RAM base address). With this address, the driver is able to compute the location of the FEC parameter RAM, and, ultimately, to program the FEC for proper operations.

ivec

This driver configures the host processor to generate hardware interrupts for various events within the device. The interrupt-vector offset parameter is used to connect the driver's ISR to the interrupt through a call to the VxWorks system function **intConnect()**. It is also used to compute the interrupt level (0-7) associated with the FEC interrupt (one of the MPC860T SIU internal interrupt sources). The latter is given as a parameter to **intEnable()**, in order to enable this level interrupt to the PPC core.

bufBase

The Motorola Fast Ethernet Controller is a DMA-type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the FEC.

This parameter tells the driver that space for both the TBDs and the RBDs needs not be allocated but should be taken from a cache-coherent private memory space provided by the user at the given address. The user should be aware that memory used for buffers descriptors must be 8-byte aligned and non-cacheable. All the buffer descriptors should fit in the given memory space.

If this parameter is "NONE", space for buffer descriptors is obtained by calling `cacheDmaMalloc()` in `motFecEndLoad()`.

bufSize

The memory size parameter specifies the size of the pre-allocated memory region. If `bufBase` is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of Transmit Buffer Descriptors and Receive Buffer Descriptors.

fifoTxBase

Indicate the base location of the transmit FIFO, in internal memory. The user does not need to initialize this parameter, as the related FEC register defaults to a proper value after reset. The specific reset value is microcode dependent. However, if the user wishes to reserve some RAM for other purposes, he may set this parameter to a different value. This should not be less than the default.

If `fifoTxBase` is specified as NONE (-1), the driver ignores it.

fifoRxBase

Indicate the base location of the receive FIFO, in internal memory. The user does not need to initialize this parameter, as the related FEC register defaults to a proper value after reset. The specific reset value is microcode dependent. However, if the user wishes to reserve some RAM for other purposes, he may set this parameter to a different value. This should not be less than the default.

If `fifoRxBase` is specified as NONE (-1), the driver ignores it.

tbdNum

This parameter specifies the number of transmit buffer descriptors (TBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space, and each one points to a 1536-byte buffer again in external RAM. If this parameter is less than a minimum number specified in the macro `MOT_FEC_TBD_MIN`, or if it is "NULL", a default value of 64 is used. This default number is kept deliberately high, since each packet the driver sends may consume more than a single TBD. This parameter should always equal a even number.

rbdNum

This parameter specifies the number of receive buffer descriptors (RBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space, and each one points to a 1536-byte buffer again in external RAM. If this parameter is less than a minimum number specified in the macro `MOT_FEC_RBD_MIN`, or if it is "NULL", a default value of 48 is used. This parameter should always equal a even number.

phyAddr

This parameter specifies the logical address of a MII-compliant physical device (PHY) that is to be used as a physical media on the network. Valid addresses are in the range 0-31. There may be more than one device under the control of the same management interface. If this parameter is "NULL", the default physical layer initialization routine will find out the PHY actual address by scanning the whole range. The one with the lowest address will be chosen.

isoPhyAddr

This parameter specifies the logical address of a MII-compliant physical device (PHY) that is to be electrically isolated by the management interface. Valid addresses are in the range 0-31. If this parameter equals 0xff, the default physical layer initialization routine will assume there is no need to isolate any device. However, this parameter will be ignored unless the **MOT_FEC_USR_PHY_ISO** bit in the *userFlags* is set to one.

phyDefMode

This parameter specifies the operating mode that will be set up by the default physical layer initialization routine in case all the attempts made to establish a valid link failed. If that happens, the first PHY that matches the specified abilities will be chosen to work in that mode, and the physical link will not be tested.

userFlags

This field enables the user to give some degree of customization to the driver, especially as regards the physical layer interface.

clockSpeed

This field enables the user to define the speed of the clock being used to drive the interface. The clock speed is used to derive the MII management interface clock, which cannot exceed 2.5 MHz. *clockSpeed* is optional in BSPs using clocks that are 50 MHz or less, but it is required in faster designs to ensure proper MII interface operation.

MOT_FEC_USR_PHY_NO_AN: the default physical layer initialization routine will exploit the auto-negotiation mechanism as described in the IEEE Std 802.3, to bring a valid link up. According to it, all the link partners on the media will take part to the negotiation process, and the highest priority common denominator technology ability will be chosen. If the user wishes to prevent auto-negotiation from occurring, he may set this bit in the user flags.

MOT_FEC_USR_PHY_TBL: in the auto-negotiation process, PHYs advertise all their technology abilities at the same time, and the result is that the maximum common denominator is used. However, this behaviour may be changed, and the user may affect the order how each subset of PHY's abilities is negotiated. Hence, when the **MOT_FEC_USR_PHY_TBL** bit is set, the default physical layer initialization routine will look at the *motFecPhyAnOrderTbl[]* table and auto-negotiate a subset of abilities at a time, as suggested by the table itself. It is worth noticing here, however, that if the **MOT_FEC_USR_PHY_NO_AN** bit is on, the above table will be ignored.

MOT_FEC_USR_PHY_NO_FD: the PHY may be set to operate in full duplex mode, provided it has this ability, as a result of the negotiation with other link partners.

However, in this operating mode, the FEC will ignore the collision detect and carrier sense signals. If the user wishes not to negotiate full duplex mode, he should set the **MOT_FEC_USR_PHY_NO_FD** bit in the user flags.

MOT_FEC_USR_PHY_NO_HD: the PHY may be set to operate in half duplex mode, provided it has this ability, as a result of the negotiation with other link partners. If the user wishes not to negotiate half duplex mode, he should set the **MOT_FEC_USR_PHY_NO_HD** bit in the user flags.

MOT_FEC_USR_PHY_NO_100: the PHY may be set to operate at 100Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. If the user wishes not to negotiate 100Mbit/s speed, he should set the **MOT_FEC_USR_PHY_NO_100** bit in the user flags.

MOT_FEC_USR_PHY_NO_10: the PHY may be set to operate at 10Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. If the user wishes not to negotiate 10Mbit/s speed, he should set the **MOT_FEC_USR_PHY_NO_10** bit in the user flags.

MOT_FEC_USR_PHY_ISO: some boards may have different PHYs controlled by the same management interface. In some cases, there may be the need of electrically isolating some of them from the interface itself, in order to guarantee a proper behaviour on the medium layer. If the user wishes to electrically isolate one PHY from the MII interface, he should set the **MOT_FEC_USR_PHY_ISO** bit and provide its logical address in the *isoPhyAddr* field of the load string. The default behaviour is to not isolate any PHY on the board.

MOT_FEC_USR_SER: the user may set the **MOT_FEC_USR_SER** bit to enable the 7-wire interface instead of the MII which is the default.

MOT_FEC_USR_LOOP: when the **MOT_FEC_USR_LOOP** bit is set, the driver will configure the FEC to work in loopback mode, with the TX signal directly connected to the RX. This mode should only be used for testing.

MOT_FEC_USR_HBC: if the **MOT_FEC_USR_HBC** bit is set, the driver will configure the FEC to perform heartbeat check following end of transmission and the HB bit in the status field of the TBD will be set if the collision input does not assert within the heartbeat window (also see *_func_motFecHbFail*, below). The user does not normally need to set this bit.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions:

sysFecEnetEnable()

```
STATUS sysFecEnetEnable (UINT32 motCpmAddr) ;
```

This routine is expected to handle any target-specific functions needed to enable the FEC. These functions typically include setting the Port D on the 860T-based board so that the MII interface may be used, and also disabling the IRQ7 signal. This routine is

expected to return **OK** on success, or **ERROR**. The driver calls this routine, once per device, from the **motFecEndLoad()** routine.

sysFecEnetDisable()

```
STATUS sysFecEnetDisable (UINT32 motCpmAddr);
```

This routine is expected to perform any target specific functions required to disable the MII interface to the FEC. This involves restoring the default values for all the Port D signals. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine from the **motFecEndStop()** routine each time a device is disabled.

sysFecEnetAddrGet()

```
STATUS sysFecEnetAddrGet (UINT32 motCpmAddr, UCHAR * enetAddr);
```

The driver expects this routine to provide the six-byte Ethernet hardware address that is used by this device. This routine must copy the six-byte address to the space provided by *enetAddr*. This routine is expected to return **OK** on success, or **ERROR**. The driver calls this routine, once per device, from the **motFecEndLoad()** routine.

_func_motFecPhyInit

```
FUNCPTR _func_motFecPhyInit
```

This driver sets the global variable **_func_motFecPhyInit** to the MII-compliant media initialization routine **motFecPhyInit()**. If the user wishes to exploit a different way to configure the PHY, he may set this variable to his own media initialization routine, typically in **sysHwInit()**.

_func_motFecHbFail

```
FUNCPTR _func_motFecPhyInit
```

The FEC may be configured to perform heartbeat check following end of transmission, and to generate an interrupt, when this event occurs. If this is the case, and if the global variable **_func_motFecHbFail** is not **NULL**, the routine referenced to by **_func_motFecHbFail** is called, with a pointer to the driver control structure as parameter. Hence, the user may set this variable to his own heart beat check fail routine, where he can take any action he sees appropriate. The default value for the global variable **_func_motFecHbFail** is **NULL**.

SYSTEM RESOURCE USAGE

If the driver allocates the memory to share with the Ethernet device, it does so by calling the **cacheDmaMalloc()** routine. For the default case of 64 transmit buffers and 48 receive buffers, the total size requested is 912 bytes, and this includes the 16-byte alignment requirement of the device. If a non-cacheable memory region is provided by the user, the size of this region should be this amount, unless the user has specified a different number of transmit or receive BDs.

This driver can operate only if this memory region is non-cacheable or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the BDs are asynchronously modified by both the driver and the device, and these fields might share the same cache line.

Data buffers are instead allocated in the external memory through the regular memory allocation routine (`memalign`), and the related cache lines are then flushed or invalidated as appropriate. The user should not allocate memory for them.

TUNING HINTS

The only adjustable parameters are the number of TBDs and RBDs that will be created at run-time. These parameters are given to the driver when `motFecEndLoad()` is called. There is one RBD associated with each received frame whereas a single transmit packet normally uses more than one TBD. For memory-limited applications, decreasing the number of RBDs may be desirable. Decreasing the number of TBDs below a certain point will provide substantial performance degradation, and is not recommended. An adequate number of loaning buffers are also pre-allocated to provide more buffering before packets are dropped, but this is not configurable.

The relative priority of the netTask and of the other tasks in the system may heavily affect performance of this driver. Usually the best performance is achieved when the netTask priority equals that of the other applications using the driver.

SPECIAL CONSIDERATIONS

Due to the FEC8 errata in the document: "MPC860 Family Device Errata Reference" available at the Motorola web site, the number of receive buffer descriptors (RBD) for the FEC (see `configNet.h`) is kept deliberately high. According to Motorola, this problem was fixed in Rev. B3 of the silicon. In memory-bound applications, when using the above mentioned revision of the MPC860T processor, the user may decrease the number of RBDs to fit his needs.

DIFFERENCES BETWEEN PPC FEC AND COLDFIRE FEC

This driver now supports the FEC as implemented in the Coldfire range of processors (MCF5272 and MCF5282). The Coldfire implementation differs from the PPC in the following ways:

- The register addresses are different, and not in the same order in memory.
- Limited throughput full-duplex 100Mbps operation. External bus use is the limiting factor.
- Only big-endian mode is supported for buffer descriptors and buffers.
- Separate interrupt vectors for Rx, Tx and non-time-critical interrupts. It is assumed that these three interrupts occupy three consecutive vectors starting at iVec.
- Interrupt priority is set in the interrupt controller. This is handled by the BSP.
- The formula for calculating the MDC frequency. On the MCF5282, $MDCfreq = SYSfreq/4*div$ (where div = value written to the `MII_SPEED` field of the `MII_SPEED` register).

INCLUDE FILES

none

SEE ALSO

ifLib, *MPC860T Fast Ethernet Controller (Supplement to the MPC860 User's Manual)*, *Motorola MPC860 User's Manual*

ncr810Lib

NAME

ncr810Lib – NCR 53C8xx PCI SCSI I/O Processor (SIOP) library (SCSI-2)

ROUTINES

ncr810CtrlCreate() – create a control structure for the NCR 53C8xx SIOP
ncr810CtrlInit() – initialize a control structure for the NCR 53C8xx SIOP
ncr810SetHwRegister() – set hardware-dependent registers for the NCR 53C8xx SIOP
ncr810Show() – display values of all readable NCR 53C8xx SIOP registers

DESCRIPTION

This is the I/O driver for the NCR 53C8xx PCI SCSI I/O Processors (SIOP), supporting the NCR 53C810 and the NCR 53C825 SCSI controllers. It is designed to work with **scsiLib** and **scsi2Lib**. This driver runs in conjunction with a script program for the NCR 53C8xx controllers. These scripts use DMA transfers for all data, messages, and status. This driver supports cache functions through **cacheLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly. **ncr810CtrlCreate()** creates a controller structure and **ncr810CtrlInit()** initializes it. The NCR 53C8xx hardware registers need to be configured according to the hardware implementation. If the default configuration is not correct, the routine **ncr810SetHwRegister()** must be used to properly configure the registers.

PCI MEMORY ADDRESSING

The global variable **ncr810PciMemOffset** was created to provide the BSP with a means of changing the **VIRT_TO_PHYS** mapping without changing the functions in the **cacheFuncs** structures. In generating physical addresses for DMA on the PCI bus, local addresses are passed through the function **CACHE_DMA_VIRT_TO_PHYS** and then the value of **ncr810PciMemOffset** is added. For backward compatibility, the initial value of **ncr810PciMemOffset** comes from the macro **PCI_TO_MEM_OFFSET** defined in **ncr810.h**.

I/O MACROS

All device access for input and output is done via macros which can be customized for each BSP. These routines are **NCR810_IN_BYTE**, **NCR810_OUT_BYTE**, **NCR810_IN_16**, **NCR810_OUT_16**, **NCR810_IN_32** and **NCR810_OUT_32**. By default, these are defined as generic memory references.

INCLUDE FILES

ncr810.h, **ncr810Script.h** and **scsiLib.h**

SEE ALSO

scsiLib, **scsi2Lib**, **cacheLib**, *SYM53C825 PCI-SCSI I/O Processor Data Manual*, *SYM53C810 PCI-SCSI I/O Processor Data Manual*, *NCR 53C8XX Family PCI-SCSI I/O Processors Programming Guide*, the VxWorks programmer guides.

ne2000End

NAME	ne2000End – NE2000 END network interface driver
ROUTINES	ne2000EndLoad() – initialize the driver and device
DESCRIPTION	This module implements the NE2000 Ethernet network interface driver.
EXTERNAL INTERFACE	
	The only external interface is the ne2000EndLoad() routine, which expects the <i>initString</i> parameter as input. This parameter passes in a colon-delimited string of the format: <i>unit:adrs:vecNum:intLvl:byteAccess:usePromNetAddr:offset</i> The ne2000EndLoad() function uses strtok() to parse the string.
TARGET-SPECIFIC PARAMETERS	
<i>unit</i>	A convenient holdover from the former model. This parameter is used only in the string name for the driver.
<i>adrs</i>	Tells the driver where to find the ne2000.
<i>vecNum</i>	Configures the ne2000 device to generate hardware interrupts for various events within the device. Thus, it contains an interrupt handler routine. The driver calls sysIntConnect() to connect its interrupt handler to the interrupt vector generated as a result of the ne2000 interrupt.
<i>intLvl</i>	This parameter is passed to an external support routine, sysLanIntEnable() , which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a ne2000 interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.

byteAccess

Tells the driver the NE2000 is jumpered to operate in 8-bit mode. Requires that **SYS_IN_WORD_STRING()** and **SYS_OUT_WORD_STRING()** be written to properly access the device in this mode.

usePromEnetAddr

Attempt to get the ethernet address for the device from the on-chip (board) PROM attached to the NE2000. Will fall back to using the BSP-supplied ethernet address if this parameter is 0 or if unable to read the ethernet address.

offset

Specifies the memory alignment offset.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_IN_CHAR(pDrvCtrl, reg, pData)
SYS_OUT_CHAR(pDrvCtrl, reg, pData)
SYS_IN_WORD_STRING(pDrvCtrl, reg, pData)
SYS_OUT_WORD_STRING(pDrvCtrl, reg, pData)
```

These macros allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine **intConnect()**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns **OK**.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. By default this is the routine **sysLanIntEnable()**, defined in the module **sysLib.o**.

The macro **SYS_ENET_ADDR_GET** is used to get the ethernet address (MAC) for the device. The single argument to this routine is the **END_DEVICE** pointer. By default this routine copies the ethernet address stored in the global variable **ne2000EndEnetAddr** into the **END_DEVICE** structure.

The macros **SYS_IN_CHAR**, **SYS_OUT_CHAR**, **SYS_IN_WORD_STRING** and **SYS_OUT_WORD_STRING** are used for accessing the ne2000 device. The default macros map these operations onto **sysInByte()**, **sysOutByte()**, **sysInWordString()** and **sysOutWordString()**.

INCLUDES

end.h **endLib.h** **etherMultiLib.h**

SEE ALSO

muxLib, **endLib**, *VxWorks Device Driver Developer's Guide*.

nec765Fd

NAME	nec765Fd – NEC 765 floppy disk device driver
ROUTINES	<code>fdDrv()</code> – initialize the floppy disk driver <code>fdDevCreate()</code> – create a device for a floppy disk <code>fdRawio()</code> – provide raw I/O access
DESCRIPTION	This is the driver for the NEC 765 Floppy Chip used on the PC 386/486.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, two routines must be called directly: `fdDrv()` to initialize the driver, and `fdDevCreate()` to create devices. Before the driver can be used, it must be initialized by calling `fdDrv()`. This routine should be called exactly once, before any reads, writes, or calls to `fdDevCreate()`. Normally, it is called from `usrRoot()` in `usrConfig.c`.

The routine `fdRawio()` allows physical I/O access. Its first argument is a drive number, 0 to 3; the second argument is a type of diskette; the third argument is a pointer to the `FD_RAW` structure, which is defined in `nec765Fd.h`.

Interleaving is not supported when the driver formats.

Two types of diskettes are currently supported: 3.5" 2HD 1.44MB and 5.25" 2HD 1.2MB. You can add additional diskette types to the `fdTypes[]` table in `sysLib.c`.

The `BLK_DEV` `bd_mode` field will reflect the disk's write protect tab.

INCLUDE FILES	none
SEE ALSO	the VxWorks programmer guides.

ns16550Sio

NAME	ns16550Sio – NS 16550 UART <i>tty</i> driver
ROUTINES	<code>ns16550DevInit()</code> – initialize an NS16550 channel <code>ns16550IntWr()</code> – handle a transmitter interrupt <code>ns16550IntRd()</code> – handle a receiver interrupt <code>ns16550IntEx()</code> – miscellaneous interrupt processing <code>ns16550Int()</code> – interrupt level processing

DESCRIPTION	This is the driver for the NS16552 DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a complete modem control capability.
	A NS16550_CHAN structure is used to describe the serial channel. This data structure is defined in ns16550Sio.h .
	Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control.
USAGE	The BSP's sysHwInit() routine typically calls sysSerialHwInit() , which creates the NS16550_CHAN structure and initializes all the values in the structure (except the SIO_DRV_FUNCS) before calling ns16550DevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() (either the single interrupt ns16550Int or the three interrupts ns16550IntWr , ns16550IntRd , and ns16550IntEx).
	This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS. The function HUPCL(hang up on last close) is available. When hardware flow control is enabled, the signals RTS and DTR are set TRUE and remain set until a HUPCL is performed.
INCLUDE FILES	drv/sio/ns16552Sio.h

ns83902End

NAME ns83902End – National Semiconductor DP83902A ST-NIC

ROUTINES **ns83902EndLoad()** – initialize the driver and device
ns83902RegShow() – prints the current value of the NIC registers

EXTERNAL INTERFACE

The only external interface is the **ns83902EndLoad()** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

"baseAdrs:intVec:intLvl:dmaPort:bufSize:options"

The **ns83902EndLoad()** function uses **strtok()** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit

A convenient holdover from the former model. This parameter is used only in the string name for the driver.

baseAdrs

Base address at which the NIC hardware device registers are located.

vecNum

This is the interrupt vector number of the hardware interrupt generated by this Ethernet device.

intLvl

This parameter defines the level of the hardware interrupt.

dmaPort

Address of the DMA port used to transfer data to the host CPU.

bufSize

Size of the NIC buffer memory in bytes.

options

Target specific options:

bit0 - wide (0: byte, 1: word)

bit1 - register interval (0: 1byte, 1: 2 bytes)

EXTERNAL SUPPORT REQUIREMENTS

This driver requires four external support functions, and provides a hook function:

void sysLanIntEnable (int level)

This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level.

void sysLanIntDisable (void)

This routine provides a target-specific interface for disabling Ethernet device interrupts.

STATUS sysEnetAddrGet (int unit, char *enetAdrs)

This routine provides a target-specific interface for accessing a device Ethernet address.

sysNs83902DelayCount

This variable is used to introduce at least a 4 bus cycle (BSCK) delay between successive NIC chip selects.

SYSTEM RESOURCE USAGE

This driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector

INCLUDE FILES

none

SEE ALSO

muxLib, DP83902A ST-NIC Serial Interface Controller for Twisted Pair

pccardLib

NAME	pccardLib – PC CARD enabler library
ROUTINES	pccardMount() – mount a DOS file system pccardMkfs() – initialize a device and mount a DOS file system pccardAtaEnabler() – enable the PCMCIA-ATA device pccardSramEnabler() – enable the PCMCIA-SRAM driver pccardEltEnabler() – enable the PCMCIA Etherlink III card pccardTffsEnabler() – enable the PCMCIA-TFFS driver
DESCRIPTION	This library provides generic facilities for enabling PC CARD. Each PC card device driver needs to provide an enabler routine and a CSC interrupt handler. The enabler routine must be in the pccardEnabler structure. Each PC card driver has its own resource structure, xxResources . The ATA PC card driver resource structure is ataResources in sysLib , which also supports a local IDE disk. The resource structure has a PC card common resource structure in the first member. Other members are device-driver dependent resources. The PCMCIA chip initialization routines tcicInit() and pcicInit() are included in the PCMCIA chip table pcmciaAdapter . This table is scanned when the PCMCIA library is initialized. If the initialization routine finds the PCMCIA chip, it registers all function pointers of the PCMCIA_CHIP structure. A memory window defined in pcmciaMemwin is used to access the CIS of a PC card through the routines in cisLib .
INCLUDE FILES	none
SEE ALSO	pcmciaLib , cisLib , tcic , pcic

pciAutoConfigLib

NAME	pciAutoConfigLib – PCI bus scan and resource allocation facility
ROUTINES	pciAutoConfigLibInit() – initialize PCI autoconfig library pciAutoCfg() – Automatically configure all nonexcluded PCI headers pciAutoCfgCtl() – set or get pciAutoConfigLib options pciAutoDevReset() – quiesce a PCI device and reset all writeable status bits pciAutoBusNumberSet() – set the primary, secondary, and subordinate bus number pciAutoFuncDisable() – disable a specific PCI function pciAutoFuncEnable() – perform final configuration and enable a function pciAutoGetNextClass() – find the next device of specific type from probe list

pciAutoRegConfig() – assign PCI space to a single PCI base address register
pciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge
pciAutoAddrAlign() – align a PCI address and check boundary conditions
pciAutoConfig() – automatically configure all nonexcluded PCI headers (obsolete)

DESCRIPTION	This library provides a facility for automated PCI device scanning and configuration on PCI-based systems. Modern PCI based systems incorporate many peripherals and may span multiple physical bus segments, and these bus segments may be connected via PCI-to-PCI Bridges. Bridges are identified and properly numbered before a recursive scan identifies all resources on the bus implemented by the bridge. Post-scan configuration of the subordinate bus number is performed. Resource requirements of each device are identified and allocated according to system resource pools that are specified by the BSP Developer. Devices may be conditionally excluded, and interrupt routing information obtained via optional routines provided by the BSP Developer.
--------------------	--

GENERAL ALGORITHM

The library must first be initialized by a call to **pciAutoConfigLibInit()**. The return value, pCookie, must be passed to each subsequent call from the library. Options can be set using the function **pciAutoCfgCtl()**. The available options are described in the documentation for **pciAutoCfgCtl()**.

After initialization of the library and configuration of any options, autoconfiguration takes place in two phases. In the first phase, all devices and subordinate busses in a given system are scanned and each device that is found causes an entry to be created in the **Probelist** or list of devices found during the probe/configuration process.

In the second phase each device that is on the Probelist is checked to see if it has been excluded from automatic configuration by the BSP developer. If a particular function has not been excluded, then it is first disabled. The Base Address Registers of the particular function are read to ascertain the resource requirements of the function. Each resource requirement is checked against available resources in the applicable pool based on size and alignment constraints.

After all functions on the Probelist have been processed, each function and its appropriate Memory or I/O decoder(s) are enabled for operation.

HOST BRIDGE DETECTION/CONFIGURATION

Note that the PCI Host Bridge is automatically excluded from configuration by the autoconfig routines, as it is often already configured as part of the system bootstrap device configuration.

PCI-PCI BRIDGE DETECTION/CONFIGURATION

Busses are scanned by first writing the primary, secondary, and subordinate bus information into the bridge that implements the bus. Specifically, the primary and

secondary bus numbers are set to their corresponding value, and the subordinate bus number is set to 0xFF, because the final number of sub-busses is not known. The subordinate bus number is later updated to indicate the highest numbered sub-bus that was scanned once the scan is complete.

GENERIC DEVICE DETECTION/CONFIGURATION

The autoconfiguration library creates a list of devices during the process of scanning all of the busses in a system. Devices with vendor IDs of 0xFFFF and 0x0000 are skipped. Once all busses have been scanned, all non-excluded devices are then disabled prior to configuration.

Devices that are not excluded will have Resources allocated according to Base Address Registers that are implemented by the device and available space in the applicable resource pool. PCI **Natural** alignment constraints are adhered to when allocating resources from pools.

Also initialized are the cache line size register and the latency timer. Bus mastering is unconditionally enabled.

If an interrupt assignment routine is registered, then the interrupt pin register of the PCI Configuration space is passed to this routine along with the bus, device, and function number of the device under consideration.

There are two different schemes to determine when the BSP interrupt assignment routine is called by autoconfig. The call is done either only for bus-0 devices or for all devices depending upon how the autoIntRouting is set by the BSP developer (see the section "INTERRUPT ROUTING ACROSS PCI-TO-PCI BRIDGES" below for more details).

The interrupt level number returned by this routine is then written into the interrupt line register of the PCI Configuration Space for subsequent use by device drivers. If no interrupt assignment routine is registered, 0xFF is written into the interrupt line register, specifying an unknown interrupt binding.

Lastly, the functions are enabled with what resources were able to be provided from the applicable resource pools.

RESOURCE ALLOCATION

Resource pools include the 32-bit Prefetchable Memory pool, the 32-bit Non-prefetchable Memory ("MemIO") pool, the 32-bit I/O pool, and the 16-bit I/O allocation pool. The allocation in each pool begins at the specified base address and progresses to higher numbered addresses. Each allocated address adheres to the PCI **natural** alignment constraints of the given resource requirement specified in the Base Address Register.

DATA STRUCTURES

Data structures are either allocated statically or allocated dynamically, depending on the value of the build macro **PCI_AUTO_STATIC_LIST**, discussed below. In either case, the structures are initialized by the call to **pciAutoConfigLibInit()**.

For ease of upgrading from the older method which used the **PCI_SYSTEM** structure, the option **PCI_SYSTEM_STRUCT_COPY** has been implemented. See the in the documentation for **pciAutoCfgCtl()** for more information.

PCI RESOURCE POOLS

Resources used by **pciAutoConfigLib** can be divided into two groups.

The first group of information is the Memory and I/O resources, that are available in the system and that autoconfig can use to allocate to functions. These resource pools consist of a base address and size. The base address specified here should be the address relative to the PCI bus. Each of these values in the **PCI_SYSTEM** data structure is described below:

pciMem32

Specifies the 32-bit prefetchable memory pool base address. Normally, this is given by the BSP constant **PCI_MEM_ADRS**. It can be set with the **pciAutoCfgCtl()** command **PCI_MEM32_LOC_SET**.

pciMem32Size

Specifies the 32-bit prefetchable memory pool size. Normally, this is given by the BSP constant **PCI_MEM_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_MEM32_SIZE_SET**.

pciMemIo32

Specifies the 32-bit non-prefetchable memory pool base address. Normally, this is given by the BSP constant **PCI_MEMIO_ADRS**. It can be set with the **pciAutoCfgCtl()** command **PCI_MEMIO32_LOC_SET**.

pciMemIo32Size

Specifies the 32-bit non-prefetchable memory pool size. Normally, this is given by the BSP constant **PCI_MEMIO_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_MEMIO32_SIZE_SET**.

pciIo32

Specifies the 32-bit I/O pool base address. Normally, this is given by the BSP constant **PCI_IO_ADRS**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO32_LOC_SET**.

pciIo32Size

Specifies the 32-bit I/O pool size. Normally, this is given by the BSP constant **PCI_IO_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO32_SIZE_SET**.

pciIo16

Specifies the 16-bit I/O pool base address. Normally, this is given by the BSP constant **PCI_ISA_IO_ADDR**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO16_LOC_SET**.

pciIo16Size

Specifies the 16-bit I/O pool size. Normally, this is given by the BSP constant **PCI_ISA_IO_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO16_SIZE_SET**.

PREFETCH MEMORY ALLOCATION

The pciMem32 pointer is assumed to point to a pool of prefetchable PCI memory. If the size of this pool is non-zero, then prefetch memory will be allocated to devices that request it given that there is enough memory in the pool to satisfy the request, and the host bridge or PCI-to-PCI bridge that implements the bus that the device resides on is capable of handling prefetchable memory. If a device requests it, and no prefetchable memory is available or the bridge implementing the bus does not handle prefetchable memory then the request will be attempted from the non-prefetchable memory pool.

PCI-to-PCI bridges are queried as to whether they support prefetchable memory by writing a non-zero value to the prefetchable memory base address register and reading back a non-zero value. A zero value would indicate the bridge does not support prefetchable memory.

BSP-SPECIFIC ROUTINES

Several routines can be provided by the BSP Developer to customize the degree to which the system can be automatically configured. These routines are normally put into a file called **sysBusPci.c** in the BSP directory. The trivial cases of each of these routines are shown in the USAGE section below to illustrate the API to the BSP Developer.

DEVICE INCLUSION

Specific devices other than bridges can be excluded from auto configuration and either not used or manually configured later. For more information, see the **PCI_INCLUDE_FUNC_SET** section in the documentation for **pciAutoCfgCtl()**.

INTERRUPT ASSIGNMENT

Interrupt assignment can be specified by the BSP developer by specifying a routine for **pciAutoConfigLib** to call at the time each device or bridge is configured. See the **PCI_INT_ASSIGN_FUNC_SET** section in the documentation for **pciAutoCfgCtl()** for more information.

INTERRUPT ROUTING ACROSS PCI-TO-PCI BRIDGES

PCI autoconfig allows use of two interrupt routing strategies for handling devices that reside across a PCI-to-PCI Bridge. The BSP-specific interrupt assignment routine described in the above section is called for all devices that reside on bus 0. For devices residing across a PCI-to-PCI bridge, one of two supported interrupt routing strategies may be selected by setting the **PCI_AUTO_INT_ROUTE_SET** command using **pciAutoCfgCtl()** to the boolean value **TRUE** or **FALSE**:

TRUE

If automatic interrupt routing is set to **TRUE**, then autoconfig only calls the BSP interrupt routing routine for devices on bus number 0. If a device resides on a higher numbered bus, then a cyclic algorithm is applied to the IRQs that are routed through the bridge. The algorithm is based on computing a **route offset** that is the device number modulo 4 for every bridge device that is traversed. This offset is used with the

device number and interrupt pin register of the device of interest to compute the contents of the interrupt line register.

FALSE

If automatic interrupt routing is set to **FALSE**, then autoconfig calls the BSP interrupt assignment routine to do all interrupt routing regardless of the bus on which the device resides. The return value represents the contents of the interrupt line register in all cases.

BRIDGE CONFIGURATION

The BSP developer may wish to perform configuration of bridges before and/or after the normal configuration of the bus they reside on. Two routines can be specified for this purpose.

The bridge pre-configuration pass initialization routine is provided so that the BSP Developer can initialize a bridge device prior to the configuration pass on the bus that the bridge implements.

The bridge post-configuration pass initialization routine is provided so that the BSP Developer can initialize the bridge device after the bus that the bridge implements has been enumerated.

These routines are configured by calling **pciAutoCfgCtl()** with the command **PCI_BRIDGE_PRE_CONFIG_FUNC_SET** and the command **PCI_BRIDGE_POST_CONFIG_FUNC_SET**, respectively.

HOST BRIDGE CONFIGURATION

The PCI Local Bus Specification, rev 2.1 does not specify the content or initialization requirements of the configuration space of PCI Host Bridges. Due to this fact, no host bridge specific assumptions are made by autoconfig and any PCI Host Bridge initialization that must be done before either scan or configuration of the bus must be done in the BSP. Comments illustrating where this initialization could be called in relation to invoking the **pciAutoConfig()** routine are in the USAGE section below.

LIBRARY CONFIGURATION MACROS

The following four macros can be defined by the BSP Developer in **config.h** to govern the operation of the autoconfig library.

PCI_AUTO_MAX_FUNCTIONS

Defines the maximum number of functions that can be stored in the probe list during the autoconfiguration pass. The default value for this define is 32, but this may be overridden by defining **PCI_AUTO_MAX_FUNCTIONS** in **config.h**.

PCI_AUTO_STATIC_LIST

If defined, then a statically allocated array of size **PCI_AUTO_MAX_FUNCTION** instances of the **PCI_LOC** structure will be instantiated.

PCI_AUTO_RECLAIM_LIST

This define may only be used if **PCI_AUTO_STATIC_LIST** is not defined. If defined, this allows the autoconfig routine to perform a **free()** operation on a dynamically allocated probe list. Note that if **PCI_AUTO_RECLAIM_LIST** is defined and **PCI_AUTO_STATIC_LIST** is also, a compiler error will be generated.

USAGE

The following code sample illustrates the usage of the **PCI_SYSTEM** structure and invocation of the autoconfig library. Note that the example BSP-specific routines are merely stubs. The code in each routine varies by BSP and application.

```
#include "pciAutoConfigLib.h"

LOCAL PCI_SYSTEM sysParams;

void sysPciAutoConfig (void)
{
    void * pCookie;

    /* initialize the library */
    pCookie = pciAutoConfigLibInit(NULL);

    /* 32-bit Prefetchable Memory Space */

    pciAutoCfgCtl(pCookie, PCI_MEM32_LOC_SET, PCI_MEM_ADRS);
    pciAutoCfgCtl(pCookie, PCI_MEM32_SIZE_SET, PCI_MEM_SIZE);

    /* 32-bit Non-prefetchable Memory Space */

    pciAutoCfgCtl(pCookie, PCI_MEMIO32_LOC_SET, PCI_MEMIO_ADRS);
    pciAutoCfgCtl(pCookie, PCI_MEMIO32_SIZE_SET, PCI_MEMIO_SIZE);

    /* 16-bit ISA I/O Space */

    pciAutoCfgCtl(pCookie, PCI_IO16_LOC_SET, PCI_ISA_IO_ADRS);
    pciAutoCfgCtl(pCookie, PCI_IO16_SIZE_SET, PCI_ISA_IO_SIZE);

    /* 32-bit PCI I/O Space */

    pciAutoCfgCtl(pCookie, PCI_IO32_LOC_SET, PCI_IO_ADRS);
    pciAutoCfgCtl(pCookie, PCI_IO32_SIZE_SET, PCI_IO_SIZE);

    /* Configuration space parameters */

    pciAutoCfgCtl(pCookie, PCI_MAX_BUS_SET, 0);
    pciAutoCfgCtl(pCookie, PCI_MAX_LAT_ALL_SET, PCI_LAT_TIMER);
    pciAutoCfgCtl(pCookie, PCI_CACHE_SIZE_SET,
                  (_CACHE_ALIGN_SIZE / 4));

    /*
     * Interrupt routing strategy
     * across PCI-to-PCI Bridges
     */
}

pciAutoCfgCtl(pCookie, PCI_AUTO_INT_ROUTE_SET, TRUE);
```

```
/* Device inclusion and interrupt routing routines */

pciAutoCfgCtl(pCookie, PCI_INCLUDE_FUNC_SET,
               sysPciAutoconfigInclude);
pciAutoCfgCtl(pCookie, PCI_INT_ASSIGN_FUNC_SET,
               sysPciAutoconfigIntrAssign);

/*
 * PCI-to-PCI Bridge Pre-
 * and Post-enumeration init
 * routines
 */

pciAutoCfgCtl(pCookie, PCI_BRIDGE_PRE_CONFIG_FUNC_SET,
               sysPciAutoconfigPreEnumBridgeInit);
pciAutoCfgCtl(pCookie, PCI_BRIDGE_POST_CONFIG_FUNC_SET,
               sysPciAutoconfigPostEnumBridgeInit);

/*
 * Perform any needed PCI Host Bridge
 * Initialization that needs to be done
 * before pciAutoConfig is invoked here
 * utilizing the information in the
 * newly-populated sysParams structure.
 */

pciAutoCfg (&sysParams);

/*
 * Perform any needed post-enumeration
 * PCI Host Bridge Initialization here.
 * Information about the actual configuration
 * from the scan and configuration passes
 * can be obtained using the assorted
 * PCI_*_GET commands to pciAutoCfgCtl().
 */

}

/*
 * Local BSP-Specific routines
 * supplied by BSP Developer
 */
STATUS sysPciAutoconfigInclude
(
    PCI_SYSTEM * pSys,           /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,             /* pointer to function in question */
)
{
    UINT devVend                /* deviceID/vendorID of device */
)
{
    return OK; /* Autoconfigure all devices */
}
```

```
UCHAR sysPciAutoconfigIntrAssign
{
    PCI_SYSTEM * pSys,                         /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,                           /* pointer to function in question */
}
UCHAR pin                                     /* contents of PCI int pin register
*/
{
    return (UCHAR)0xff;
}

void sysPciAutoconfigPreEnumBridgeInit
(
    PCI_SYSTEM * pSys,                         /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,                           /* pointer to function in question */
)
UINT devVend                                /* deviceID/vendorID of device */
{
{
    return;
}

void sysPciAutoconfigPostEnumBridgeInit
(
    PCI_SYSTEM * pSys,                         /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,                           /* pointer to function in question */
)
UINT devVend                                /* deviceID/vendorID of device */
{
{
    return;
}
```

CONFIGURATION SPACE PARAMETERS

The cache line size register specifies the cacheline size in longwords. This register is required when a device can generate a memory write and Invalidate bus cycle, or when a device provides cacheable memory to the system.

Note that in the above example, the macro `_CACHE_ALIGN_SIZE` is utilized. This macro is implemented for all supported architectures and is located in the `architecture.h` file in `.../target/h/arch/architecture`. The value of the macro indicates the cache line size in bytes for the particular architecture. For example, the PowerPC architecture defines this macro to be 32, while the ARM 810 defines it to be 16. The PCI cache line size field and the `cacheSize` element of the `PCI_SYSTEM` structure expect to see this quantity in longwords, so the byte value must be divided by 4.

LIMITATIONS

The current version of the autoconfig facility does not support 64-bit prefetchable memory behind PCI-to-PCI bridges, but it does support 32-bit prefetchable memory.

The autoconfig code also depends upon the BSP Developer specifying resource pools that do not conflict with any resources that are being used by statically configured devices.

INCLUDE FILES **pciAutoConfigLib.h**

SEE ALSO "PCI Local Bus Specification, Revision 2.1, June 1, 1996", "PCI Local Bus PCI to PCI Bridge Architecture Specification, Revision 1.0, April 5, 1994"

pciConfigLib

NAME **pciConfigLib** – PCI Configuration space access support for PCI drivers

ROUTINES **pciConfigLibInit()** – initialize the configuration access-method and addresses
pciFindDevice() – find the nth device with the given device & vendor ID
pciFindClass() – find the nth occurrence of a device by PCI class code.
pciDevConfig() – configure a device on a PCI bus
pciConfigBdfPack() – pack parameters for the Configuration Address Register
pciConfigExtCapFind() – find extended capability in ECP linked list
pciConfigInByte() – read one byte from the PCI configuration space
pciConfigInWord() – read one word from the PCI configuration space
pciConfigInLong() – read one longword from the PCI configuration space
pciConfigOutByte() – write one byte to the PCI configuration space
pciConfigOutWord() – write one 16-bit word to the PCI configuration space
pciConfigOutLong() – write one longword to the PCI configuration space
pciConfigModifyLong() – Perform a masked longword register update
pciConfigModifyWord() – Perform a masked longword register update
pciConfigModifyByte() – Perform a masked longword register update
pciSpecialCycle() – generate a special cycle with a message
pciConfigForEachFunc() – check condition on specified bus
pciConfigReset() – disable cards for warm boot

DESCRIPTION This module contains routines to support accessing the PCI bus Configuration Space. The library is PCI Revision 2.1 compliant.

In general, functions in this library should not be called from the interrupt level, (except **pciInt()**) because Configuration Space access, which is slow, should be limited to initialization only.

The functions addressed here include:

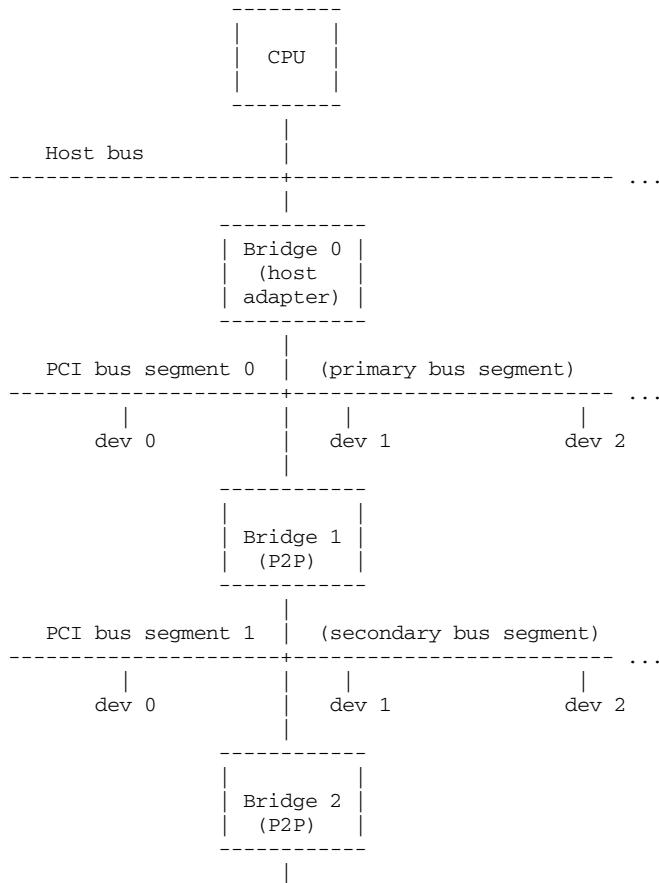
- Initialization of the library.
- Locating a device by Device ID and Vendor ID.
- Locating a device by Class Code.
- Generation of Special Cycles.
- Accessing Configuration Space structures.

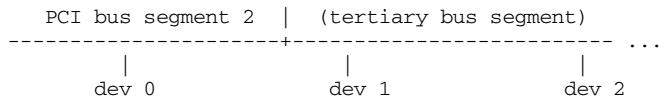
PCI BUS CONCEPTS

The PCI bus is an unterminated, high impedance CMOS bus using reflected wave signalling as opposed to incident wave. Because of this, the PCI bus is physically limited in length and the number of electrical loads that can be supported. Each device on the bus represents one load, including adapters and bridges.

To accomodate additional devices, the PCI standard allows multiple PCI buses to be interconnected via PCI-to-PCI bridge (PPB) devices to form one large bus. Each constituent bus is referred to as a bus segment and is subject to the above limitations.

The bus segment accessible from the host bus adapter is designated the primary bus segment (see figure). Progressing outward from the primary bus (designated segment number zero from the PCI architecture point of view) are the secondary and tertiary buses, numbered as segments one and two, respectively. Due to clock skew concerns and propagation delays, practical PCI bus architectures do not implement bus segments beyond the tertiary level.





For further details, refer to the PCI to PCI Bridge Architecture Specification.

I/O MACROS AND CPU ENDIAN-ness

PCI bus I/O operations must adhere to little endian byte ordering. Thus if an I/O operation larger than one byte is performed, the lower I/O addresses contain the least significant bytes of the multi-byte quantity of interest.

For architectures that adhere to big-endian byte ordering, byte-swapping must be performed. The architecture-specific byte-order translation is done as part of the I/O operation in the following routines: `sysPciInByte`, `sysPciInWord`, `sysPciInLong`, `sysOutPciByte`, `sysPciOutWord`, `sysPciOutLong`. The interface to these routines is mediated by the following macros:

PCI_IN_BYTE

read a byte from PCI I/O Space

PCI_IN_WORD

read a word from PCI I/O Space

PCI_IN_LONG

read a longword from PCI I/O Space

PCI_OUT_BYTE

write a byte from PCI I/O Space

PCI_OUT_WORD

write a word from PCI I/O Space

PCI_OUT_LONG

write a longword from PCI I/O Space

By default, these macros call the appropriate PCI I/O routine, such as `sysPciInWord`. For architectures that do not require byte swapping, these macros simply call the appropriate default I/O routine, such as `sysInWord`. These macros may be redefined by the BSP if special processing is required.

INITIALIZATION

`pciConfigLibInit()` should be called before any other `pciConfigLib` functions. Generally, this is performed by `sysHwInit()`.

After the library has been initialized, it may be utilized to find devices, and access PCI configuration space.

Any PCI device can be uniquely addressed within Configuration Space by the **geographic** specification of a Bus segment number, Device number, and a Function number (BDF). The configuration registers of a PCI device are arranged by the PCI standard according to a Configuration Header structure. The BDF triplet specifies the location of the header structure of one device. To access a configuration register, its location in the header must be

given. The location of a configuration register of interest is simply the structure member offset defined for the register. For further details, refer to the PCI Local Bus Specification, Revision 2.1. Refer to the header file **pciConfigLib.h** for the defined standard configuration register offsets.

The maximum number of Type-1 Configuration Space buses supported in the 2.1 Specifications is 256 (0x00 - 0xFF), far greater than most systems currently support. Most buses are numbered sequentially from 0. An optional define called **PCI_MAX_BUS** may be declared in **config.h** to override the default definition of 256. Similarly, the default number of devices and functions may be overridden by defining **PCI_MAX_DEV** and/or **PCI_MAX_FUNC**. Note that the number of devices applies only to bus zero, all others being restricted to 16 by the 2.1 spec.

ACCESS MECHANISM 1

This is the preferred access mechanism for a PC-AT class machines. It uses two standard PCI I/O registers to initiate a configuration cycle. The type of cycle is determined by the Host-bridge device based on the devices primary bus number. If the configuration bus number matches the primary bus number then a type 0 configuration cycle occurs. Otherwise a type 1 cycle is generated. This is all transparent to the user.

The two arguments used for mechanism 1 are the CAR register address which by default is **PCI_CONFIG_ADDR** (0xCF8), and the CDR register address which is normally **PCI_CONFIG_DATA** (0xCFC).

e.g.

```
pciConfigLibInit (PCI_MECHANISM_1, PCI_CONFIG_ADDR,  
                  PCI_CONFIG_DATA, NULL);
```

ACCESS MECHANISM 2

This is the non-preferred legacy mechanism for PC-AT class machines. The three arguments used for mechanism 2 are the CSE register address which by default is **PCI_CONFIG_CSE** (0xCF8), and the Forward register address which is normally **PCI_CONFIG_FORWARD** (0xCFA), and the configuration base address which is normally **PCI_CONFIG_BASE** (0xC000).

e.g.

```
pciConfigLibInit (PCI_MECHANISM_2, PCI_CONFIG_CSE,  
                  PCI_CONFIG_FORWARD, PCI_CONFIG_BASE);
```

ACCESS MECHANISM 0

We have added a non-standard access method that we call method 0. Selecting method 0 installs user supplied read and write routines to actually handle configuration read and writes (32 bit accesses only). The BSP will supply pointers to these routines as arguments 2 and 3 (read routine is argument 2, write routine is argument 3). A user provided special cycle routine is argument 4. The special cycle routine is optional and a NULL pointer should be used if the special cycle routine is not provided by the BSP.

All accesses are expected to be 32 bit accesses with these routines. The code in this library will perform bit manipulation to emulate byte and word operations. All routines return **OK** to indicate successful operation and **ERROR** to indicate failure.

Initialization examples using special access method 0:

```
pciConfigLibInit (PCI_MECHANISM_0, myReadRtn,
                  myWriteRtn, mySpecialRtn);
-or-
pciConfigLibInit (PCI_MECHANISM_0, myReadRtn,
                  myWriteRtn, NULL);
```

The calling convention for the user read routine is:

```
STATUS myReadRtn (int bus, int dev, int func,
                   int reg, int size, void * pResult);
```

The calling convention for the user write routine is:

```
STATUS myWriteRtn (int bus, int dev, int func,
                   int reg, int size, UINT32 data);
```

The calling convention for the optional special cycle routine is:

```
STATUS mySpecialRtn (int bus, UINT32 data);
```

In the Type-1 method, PCI Configuration Space accesses are made by the sequential access of two 32-bit hardware registers: the Configuration Address Register (CAR) and the Configuration Data Register (CDR). The CAR is written to first with the 32-bit value designating the PCI bus number, the device on that bus, and the offset to the configuration register being accessed in the device. The CDR is then read or written, depending on whether the register of interest is to be read or written. The CDR access may be 8-bits, 16-bits, or 32-bits in size. Both the CAR and CDR are mapped by the standard to predefined addresses in the PCI I/O Space: CAR = 0xCF8 and CDR = 0xCFC.

The Type-2 access method maps any one configuration header into a fixed 4K byte window of PCI I/O Space. In this method, any PCI I/O Space access within the range of 0xC000 to 0xCFFF will be translated to a Configuration Space access. This access method utilizes two 8-bit hardware registers: the Configuration Space Enable register (CSE) and the Forward register (CFR). Like the CAR and CDR, these registers occupy preassigned PCI I/O Space addresses: CSE = 0xCF8, CFR = 0xCFA. The CSE specifies the device to be accessed and the function within the device. The CFR specifies the bus number on which the device of interest resides. The access sequence is 1) write the bus number to CFR, 2) write the device location information to CSE, and 3) perform an 8-bit, 16-bit, or 32-bit read or write at an offset into the PCI I/O Space starting at 0xC000. The offset specifies the configuration register within the configuration header which now appears in the 4K byte Configuration Space window.

SPECIAL STATUS BITS

Be careful to not use `pciConfigOutWord`, `pciConfigOutByte`, `pciConfigModifyWord`, or `pciConfigModifyByte` for modifying the Command and status register

(PCI_CFG_COMMAND). The bits in the status register are reset by writing a 1 to them. For each of the listed functions, it is possible that they will emulate the operation by reading a 32 bit quantity, shifting the new data into the proper byte lane and writing back a 32 bit value.

Improper use may inadvertently clear all error conditions indications if the user tries to update the command bits. The user should insure that only full 32 bit operations are performed on the command/status register. Use pciConfigInLong to read the Command/Status reg, mask off the status bits, mask or insert the command bit changes and then use pciConfigOutLong to rewrite the Command/Status register. Use of pciConfigModifyLong is okay if the status bits are rewritten as zeroes.

```
/*
 * This example turns on the write invalidate enable bit in the Command
 * register without clearing the status bits or disturbing other
 * command bits.
 */

pciConfigInLong (bus, dev, func, PCI_CFG_COMMAND, &temp);
temp &= 0x0000ffff;
temp |= PCI_CMD_WI_ENABLE;
pciConfigOutLong (bus, dev, func, PCI_CFG_COMMAND, temp);

/* -or- include 0xfffff0000 in the bit mask for ModifyLong */

pciConfigModifyLong (bus, dev, func, PCI_CFG_COMMAND,
                     (0xfffff0000 | PCI_CMD_WI_ENABLE), PCI_CMD_WI_ENABLE);
```

The above warning applies to any configuration register containing write 1 to clear bits.

PCI DEVICE LOCATION

After the library has been initialized, the Configuration Space of any PCI device may be accessed after first locating the device.

Locating a device is accomplished using either **pciFindDevice()** or **pciFindClass()**. Both routines require an index parameter indicating which instance of the device should be returned, since multiple instances of the same device may be present in a system. The instance number is zero-based.

pciFindDevice() accepts the following parameters:

vendorId

The vendor ID of the device.

deviceId

The device ID of the device.

index

The instance number.

pciFindClass() simply requires a class code and the index:

classCode

The 24-bit class of the device.

index

The instance number.

In addition, both functions return the following parameters by reference:

pBusNo

Where to return bus segment number containing the device.

pDeviceNo

Where to return the device ID of the device.

pFuncNo

Where to return the function number of the device.

These three parameters, Bus segment number, Device number, and Function number (BDF), provide a means to access the Configuration Space of any PCI device.

PCI BUS SPECIAL CYCLE GENERATION

The PCIbus Special Cycle is a cycle used to broadcast data to one or many devices on a target PCI bus. It is common, for example, for Intel x86-based systems to broadcast to PCI devices that the system is about to go into a halt or shutdown condition.

The special cycle is initiated by software. Utilizing CSAM-1, a 32-bit write to the configuration address port specifying the following

Bus Number

The PCI bus of interest.

Device Number

Set to all 1's (01Fh).

Function Number

Set to all 1's (07d).

Configuration Register Number

Zeroed.

The **pciSpecialCycle()** function facilitates generation of a Special Cycle by generating the correct address data noted above. The data passed to the function is driven onto the bus during the Special Cycle's data phase. The parameters to the **pciSpecialCycle()** function are:

busNo

Bus on which Special Cycle is to be initiated.

message

Data driven onto AD[31:0] during the Special Cycle.

PCI DEVICE CONFIGURATION SPACE ACCESS

The routines `pciConfigInByte()`, `pciConfigInWord()`, `pciConfigInLong()`, `pciConfigOutByte()`, `pciConfigOutWord()`, and `pciConfigOutLong()` may be used to access the Configuration Space of any PCI device, once the library has been properly initialized. It should be noted that, if no device exists at the given BDF address, the resultant behavior of the Configuration Space access routines is to return a value with all bits set, as set forth in the PCI bus standard.

In addition to the BDF numbers obtained from the `pciFindXxx` functions, an additional parameter specifying an offset into the PCI Configuration Space must be specified when using the access routines. VxWorks includes defined offsets for all of the standard PCI Configuration Space structure members as set forth in the PCI Local Bus Specification 2.1 and the PCI Local Bus PCI to PCI Bridge Architecture Specification 1.0. The defined offsets are all prefixed by "PCI_CFG_". For example, if Vendor ID information is required, `PCI_CFG_VENDOR_ID` would be passed as the offset argument to the access routines.

In summary, the pci configuration space access functions described above accept the following parameters.

Input routines:

busNo

Bus segment number on which the device resides.

deviceNo

Device ID of the device.

funcNo

Function number of the device.

offset

Offset into the device configuration space.

pData

Where to return the data.

Output routines:

busNo

Bus segment number on which the device resides.

deviceNo

Device ID of the device.

funcNo

Function number of the device.

offset

Offset into the device configuration space.

Data

Data to be written.

PCI CONFIG SPACE OFFSET CHECKING

PciConfigWordIn(), **pciConfigWordOut()**, **pciConfigLongIn()**, and **pciConfigLongOut()** check the offset parameter for proper offset alignment. Offsets should be multiples of 4 for longword accesses and multiples of 2 for word accesses. Misaligned accesses will not be performed and **ERROR** will be returned.

The previous default behaviour for this library was to not check for valid offset values. This has been changed and checks are now done by default. These checks exist to insure that the user gets the correct data using the correct configuration address offsets. The user should define **PCI_CONFIG_OFFSET_NOCHECK** to achieve the older behaviour. If user code behaviour changes, the user should investigate why and fix the code that is calling into this library with invalid offset values.

PCI DEVICE CONFIGURATION

The function **pciDevConfig()** is used to configure PCI devices that require no more than one Memory Space and one I/O Space. According to the PCI standard, a device may have up to six 32-bit Base Address Registers (BARs) each of which can have either a Memory Space or I/O Space base address. In 64-bit PCI devices, the registers double up to give a maximum of three 64-bit BARs. The 64-bit BARs are not supported by this function nor are more than one 32-bit BAR of each type, Memory or I/O.

The **pciDevConfig()** function sets up one PCI Memory Space and/or one I/O Space BAR and issues a specified command to the device to enable it. It takes the following parameters:

pciBusNo

PCI bus segment number.

pciDevNo

PCI device number.

pciFuncNo

PCI function number.

devIoBaseAdrs

Base address of one IO-mapped resource.

devMemBaseAdrs

Base address of one memory-mapped resource.

command

Command to issue to device after configuration.

UNIFORM DEVICE ACCESS

The function **pciConfigForEachFunc()** is used to perform some action on every device on the bus. This does a depth-first recursive search of the bus and calls a specified routine for each function it finds. It takes the following parameters:

bus

The bus segment to start with. This allows configuration on and below a specific place in the bus hierarchy.

recurse

A boolean argument specifying whether to do a recursive search or to do just the specified bus.

funcCheckRtn

A user supplied function which will be called for each PCI function found. It must return STATUS. It takes four arguments: "bus", "device", "function", and a user-supplied arg "pArg". The typedef **PCI_FOREACH_FUNC** is defined in **pciConfigLib.h** for these routines. Note that it is possible to apply "funcCheckRtn" only to devices of a specific type by querying the device type for the class code. Similarly, it is possible to exclude bridges or any other device type using the same mechanism.

pArg

The fourth argument to funcCheckRtn.

SYSTEM RESET

The function **pciConfigReset()** is useful at the time of a system reset. When doing a system reset, the devices on the system should be disabled so that they do not write to RAM while the system is trying to reboot. The function **pciConfigReset()** can be installed using **rebootHookAdd()**, or it can be called directly from **sysToMonitor()** or elsewhere in the BSP. It accepts one argument for compatibility with **rebootHookAdd()**:

startType

Ignored.

Note that this function disables all access to the PCI bus except for the use of PCI config space. If there are devices on the PCI bus which are required to reboot, then those devices must be re-enabled after the call to **pciConfigReset()** or the system will not be able to reboot.

USAGE

The following code sample illustrates the usage of this library. Initialization of the library is performed first, then a sample device is found and initialized.

```
#include "drv/pci/pciConfigLib.h"

#define PCI_ID_LN_DEC21140      0x00091011

IMPORT pciInt();
LOCAL VOID deviceIsr(int);

int      param;
STATUS   result;
int      pciBusNo;        /* PCI bus number */
int      pciDevNo;        /* PCI device number */
int      pciFuncNo;       /* PCI function number */

/*
 * Initialize module to use CSAM-1
 * (if not performed in sysHwInit())
 *
```

```

        */

        if (pciConfigLibInit (PCI_MECHANISM_1,
                            PCI_PRIMARY_CAR,
                            PCI_PRIMARY_CDR,
                            0)
            != OK)
        {
            sysToMonitor (BOOT_NO_AUTOBOOT);
        }

/*
 * Find a device by its device ID, and use the
 * Bus, Device, and Function number of the found
 * device to configure it, using pciDevConfig(). In
 * this case, the first instance of a DEC 21040
 * Ethernet NIC is searched for. If the device
 * is found, the Bus, Device Number, and Function
 * Number are fed to pciDevConfig, along with the
 * constant PCI_IO_LN2_ADRS, which defines the start
 * of the I/O space utilized by the device. The
 * device and its I/O space is then enabled.
 *
*/
if (pciFindDevice (PCI_ID_LN_DEC21040 & 0xFFFF,
                   (PCI_ID_LN_DEC21040 >> 16) & 0xFFFF,
                   0,
                   &pciBusNo,
                   &pciDevNo,
                   &pciFuncNo)
    != ERROR)
{
    (void)pciDevConfig (pciBusNo, pciDevNo, pciFuncNo,
                        PCI_IO_LN2_ADRS,
                        NULL,
                        (PCI_CMD_MASTER_ENABLE |
                        PCI_CMD_IO_ENABLE));
}

```

INCLUDE FILES **pciConfigLib.h**

SEE ALSO *PCI Local Bus Specification, Revision 2.1, June 1, 1996 , PCI Local Bus PCI to PCI Bridge Architecture Specification, Revision 1.0, April 5, 1994"*

pciConfigShow

NAME **pciConfigShow** – Show routines of PCI bus(IO mapped) library

ROUTINES	<code>pciDeviceShow()</code> – print information about PCI devices <code>pciHeaderShow()</code> – print a header of the specified PCI device <code>pciFindDeviceShow()</code> – find a PCI device and display the information <code>pciFindClassShow()</code> – find a device by 24-bit class code <code>pciConfigStatusWordShow()</code> – show the decoded value of the status word <code>pciConfigCmdWordShow()</code> – show the decoded value of the command word <code>pciConfigFuncShow()</code> – show configuration details about a function <code>pciConfigTopoShow()</code> – show PCI topology
DESCRIPTION	This module contains show routines to see all devices and bridges on the PCI bus. This module works in conjunction with <code>pciConfigLib.o</code> . There are two ways to find out an empty device. <ul style="list-style-type: none">- check Master Abort bit after the access.- check whether the read value is 0xffff. It uses the second method, since I didn't see the Master Abort bit of the host/PCI bridge changing.

INCLUDE FILES

pcilntLib

NAME	<code>pcilntLib</code> – PCI Shared Interrupt support
ROUTINES	<code>pcilntLibInit()</code> – initialize the <code>pcilntLib</code> module <code>pcilnt()</code> – interrupt handler for shared PCI interrupt. <code>pcilntConnect()</code> – connect the interrupt handler to the PCI interrupt. <code>pcilntDisconnect()</code> – disconnect the interrupt handler (OBSOLETE) <code>pcilntDisconnect2()</code> – disconnect an interrupt handler from the PCI interrupt.
DESCRIPTION	This component is PCI Revision 2.1 compliant. The functions addressed here include: <ul style="list-style-type: none">- Initialize the library.- Connect a shared interrupt handler.- Disconnect a shared interrupt handler.- Master shared interrupt handler. Shared PCI interrupts are supported by three functions: <code>pcilnt()</code> , <code>pcilntConnect()</code> , and <code>pcilntDisconnect2()</code> . <code>pcilntConnect()</code> adds the specified interrupt handler to the link list and <code>pcilntDisconnect2()</code> removes it from the link list. The master interrupt handler

pcInt() executes these interrupt handlers in the link list for a PCI interrupt. Each interrupt handler must check the device dependent interrupt status bit to determine the source of the interrupt, since it simply execute all interrupt handlers in the link list. **pcInt()** should be attached by **intConnect()** function in the BSP initialization with its parameter. The parameter is an vector number associated to the PCI interrupt.

INCLUDE FILES

pcic

NAME	pcic – Intel 82365SL PCMCIA host bus adaptor chip library
ROUTINES	pcicInit() – initialize the PCIC chip
DESCRIPTION	This library contains routines to manipulate the PCMCIA functions on the Intel 82365 series PCMCIA chip. The following compatible chips are also supported: <ul style="list-style-type: none">- Cirrus Logic PD6712/20/22- Vadem VG468- VLSI 82c146- Ricoh RF5C series
	The initialization routine pcicInit() is the only global function and is included in the PCMCIA chip table pcmciaAdapter . If pcicInit() finds the PCIC chip, it registers all function pointers of the PCMCIA_CHIP structure.
INCLUDE FILES	none

pcicShow

NAME	pcicShow – Intel 82365SL PCMCIA host bus adaptor chip show library
ROUTINES	pcicShow() – show all configurations of the PCIC chip
DESCRIPTION	This is a driver show routine for the Intel 82365 series PCMCIA chip. pcicShow() is the only global function and is installed in the PCMCIA chip table pcmciaAdapter in pcmciaShowInit() .
INCLUDE FILES	none

pcmciaLib

NAME	pcmciaLib – generic PCMCIA event-handling facilities
ROUTINES	pcmciaInit() – initialize the PCMCIA event-handling package pcmciad() – handle task-level PCMCIA events
DESCRIPTION	This library provides generic facilities for handling PCMCIA events.

USER-CALLABLE ROUTINES

Before the driver can be used, it must be initialized by calling **pcmciaInit()**. This routine should be called exactly once, before any PC card device driver is used. Normally, it is called from **usrRoot()** in **usrConfig.c**.

The **pcmciaInit()** routine performs the following actions:

- Creates a message queue.
- Spawns a PCMCIA daemon, which handles jobs in the message queue.
- Finds out which PCMCIA chip is installed and fills out the **PCMCIA_CHIP** structure.
- Connects the CSC (Card Status Change) interrupt handler.
- Searches all sockets for a PC card. If a card is found, it:
 - gets CIS (Card Information Structure) information from a card
 - determines what type of PC card is in the socket
 - allocates a resource for the card if the card is supported
 - enables the card
 - Enables the CSC interrupt.

The CSC interrupt handler performs the following actions:

- Searches all sockets for CSC events.
- Calls the PC card's CSC interrupt handler, if there is a PC card in the socket.
- If the CSC event is a hot insertion, it asks the PCMCIA daemon to call **cisGet()** at task level. This call reads the CIS, determines the type of PC card, and initializes a device driver for the card.
- If the CSC event is a hot removal, it asks the PCMCIA daemon to call **cisFree()** at task level. This call de-allocates resources.

INCLUDE FILES	none
----------------------	------

pcmciaShow

NAME	pcmciaShow – PCMCIA show library
ROUTINES	pcmciaShowInit() – initialize all show routines for PCMCIA drivers pcmciaShow() – show all configurations of the PCMCIA chip
DESCRIPTION	This library provides a show routine that shows the status of the PCMCIA chip and the PC card.
INCLUDE FILES	none

pentiumPci

NAME	pentiumPci – PCI host controller for Pentium CPU
ROUTINES	pentiumPciRegister() – register Pentium PCI host bridge device driver pentiumPciMmuMapAdd() – memory map sysPhysMemDesc pentiumPciPhysMemHandle() – configure PCI memory for a device pentiumPciPhysMemShow() – display sysPhysMemDesc entries sysPciHostBridgeInit() – initialize the PCI Host Bridge pentiumPciRegAddrShow() – debug routine to print register addresses pentiumPciBusDevGett() – find bus controller pentiumPciAllHeaderShow() – show PCI header for specified device pciAllHeaderShow() – show PCI header for all devices
DESCRIPTION	This library contains the support routines for Pentium PCI host controllers on the vxBus.
INCLUDE FILES	pciBusLib.h

ppc403Sio

NAME	ppc403Sio – ppc403GA serial driver
ROUTINES	ppc403DummyCallback() – dummy callback routine ppc403DevInit() – initialize the serial port unit ppc403IntWr() – handle a transmitter interrupt

ppc403IntRd() – handle a receiver interrupt
ppc403IntEx() – handle error interrupts

DESCRIPTION	This is the driver for PPC403GA serial port on the on-chip peripheral bus. The SPU (serial port unit) consists of three main elements: receiver, transmitter, and baud-rate generator. For details, refer to the <i>PPC403GA Embedded Controller User's Manual</i> .
USAGE	A PPC403_CHAN structure is used to describe the chip. This data structure contains the single serial channel. The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the PPC403_CHAN structure (except the SIO_DRV_FUNCS) before calling ppc403DevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chip interrupt routines ppc403IntWr() and ppc403IntRd() via intConnect() .
IOCTL FUNCTIONS	This driver responds to the same ioctl() codes as other SIO drivers; for more information, see sioLib.h .
INCLUDE FILES	drv/sio/ppc403Sio.h

ppc440gpPci

NAME	ppc440gpPci – PCI host controller in PowerPC 440GP CPU
ROUTINES	ppc440gpPciRegister() – register PowerPC 440GP host bridge ibmPciConfigWrite() – write to one PCI configuration register location ibmPciConfigRead() – reads one PCI configuration space register ibmPciSpecialCycle() – generates a PCI special cycle ppc440gpPciHostBridgeInit() – Initialize the PCI-X Host Bridge sysPciHostBridgeInit() – Initialize the PCI-X Host Bridge ppc440gpPciRegAddrShow() – debug routine to print register addresses
DESCRIPTION	This module provides VxBus driver support for the onchip 440GP PCI bridge controller. This driver support for PCI memory mapped and I/O mapped register access, PCI configuration access.
INCLUDE FILES	none
SEE ALSO	vxBus , vxbDmaBufLib

ppc860Sio

NAME	ppc860Sio – Motorola MPC800 SMC UART serial driver
ROUTINES	<code>ppc860DevInit()</code> – initialize the SMC <code>ppc860Int()</code> – handle an SMC interrupt
DESCRIPTION	This is the driver for the SMCs in the internal Communications Processor (CP) of the Motorola MPC68860/68821. This driver only supports the SMCs in asynchronous UART mode.
USAGE	A PPC800SMC_CHAN structure is used to describe the chip. The BSP's <code>sysHwInit()</code> routine typically calls <code>sysSerialHwInit()</code> , which initializes all the values in the PPC800SMC_CHAN structure (except the <code>SIO_DRV_FUNCS</code>) before calling <code>ppc860DevInit()</code> . The BSP's <code>sysHwInit2()</code> routine typically calls <code>sysSerialHwInit2()</code> which connects the chip's interrupts via <code>intConnect()</code> .
INCLUDE FILES	<code>drv/sio/ppc860Sio.h</code>

rm9000x2glSio

NAME	rm9000x2glSio – RM9000 <i>tty</i> driver
ROUTINES	<code>rm9000x2glDevInit()</code> – intialize an NS16550 channel <code>rm9000x2glIntWr()</code> – handle a transmitter interrupt <code>rm9000x2glIntRd()</code> – handle a receiver interrupt <code>rm9000x2glIntEx()</code> – miscellaneous interrupt processing <code>rm9000x2glInt()</code> – interrupt level processing <code>rm9000x2glIntMod()</code> – interrupt level processing
DESCRIPTION	This is the driver for the RM9000x2gl DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a complete modem control capability. A RM9000x2gl_CHAN structure is used to describe the serial channel. This data structure is defined in <code>rm9000x2glSio.h</code> . Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control.

This driver is a modification of the WindRiver **ns16550Sio.c** driver any changes to this driver should also be reflected in the **ns16550Sio.c** driver.

USAGE	The BSP's sysHwInit() routine typically calls sysSerialHwInit() , which creates the RM9000x2gl_CHAN structure and initializes all the values in the structure (except the SIO_DRV_FUNCS) before calling rm9000x2glDevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() (either the single interrupt rm9000x2glInt or the three interrupts rm9000x2glIntWr , rm9000x2glIntRd , and rm9000x2glIntEx). This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS. The function HUPCL(hang up on last close) is available. When hardware flow control is enabled, the signals RTS and DTR are set TRUE and remain set until a HUPCL is performed.
INCLUDE FILES	drv/sio/rm9000x2glSio.h

shSciSio

NAME	shSciSio – Hitachi SH SCI (Serial Communications Interface) driver
ROUTINES	shSciDevInit() – initialize a on-chip serial communication interface shSciIntRcv() – handle a channel's receive-character interrupt. shSciIntTx() – handle a channels transmitter-ready interrupt. shSciIntErr() – handle a channel's error interrupt.
DESCRIPTION	This is the driver for the Hitachi SH series on-chip SCI (Serial Communication Interface). It uses the SCI in asynchronous mode only.
USAGE	A SCI_CHAN structure is used to describe the chip. The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the SCI_CHAN structure (except the SIO_DRV_FUNCS) before calling shSciDevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() .
INCLUDE FILES	drv/sio/shSciSio.h sioLib.h

shScifSio

NAME	shScifSio – Renesas SH SCIF (Serial Communications Interface) driver
ROUTINES	shScifDevInit() – initialize a on-chip serial communication interface shScifIntRcv() – handle a channel's receive-character interrupt. shScifIntTx() – handle a channels transmitter-ready interrupt. shScifIntErr() – handle a channel's error interrupt.
DESCRIPTION	This is the driver for the Renesas SH series on-chip SCIF (Serial Communication Interface with FIFO). It uses the SCIF in asynchronous mode only.
USAGE	A SCIF_CHAN structure is used to describe the chip. The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the SCIF_CHAN structure (except the SIO_DRV_FUNCS) before calling shSciDevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() .
INCLUDE FILES	drv/sio/shSciSio.h drv/sio/shScifSio.h sioLib.h

sioChanUtil

NAME	sioChanUtil – SIO Channel Utilities module
ROUTINES	sioNextChannelNumberAssign() – assign a new serial channel number sysSerialChanGet() – get the SIO_CHAN device associated with a serial channel sysSerialChanConnect() – connect the SIO_CHAN device sysSerialConnectAll() – connect all SIO_CHAN devices
DESCRIPTION	This library is the SIO channel utilities module
INCLUDE FILES	sioLib.h sioChanUtil.h

smEnd

NAME	smEnd – END shared memory (SM) network interface driver
-------------	---

ROUTINES **smEndLoad()** – attach the SM interface to the MUX, initialize driver and device

DESCRIPTION This module implements the VxWorks shared memory (SM) Enhanced Network Driver (END).

This driver is designed to be moderately generic, operating unmodified across most targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters are detailed below.

There are no user-callable routines.

This driver is layered between the shared memory packet library and the MUX modules. The SM END gives CPUs sharing common memory the ability to communicate using Internet Protocol (IP).

Sending of multiple frames (mBlk chains) is supported but only single frames can be received as there is not yet a **netBufLib** support routine to do so.

I/O CONTROL CODES

The standard END commands implemented are:

Command	Data	Function
EIOCSADDR	char *	set SM device address
EIOCGADDR	char *	get SM device address
EIOCSFLAGS	int	set SM device flags
EIOCGFLAGS	int	get SM device flags
EIOCGMWIDTH	int *	get memory width (always 0)
EIOCMULTIADD	--	[not supported]
EIOCMULTIDEL	--	[not supported]
EIOCMULTIGET	--	[not supported]
EIOCPOOLLSTART	N/A	start polled operation
EIOCPOOLLSTOP	N/A	stop polled operation
EIOCGMIB2	M2_INTERFACETBL *	return MIB2 information
EIOCGFBUF	int	return minimum First Buffer for chaining
EIOCGHDRLEN	int *	get ether header length

The driver-specific commands implemented are:

Command	Data	Function
SMIOCGMCPYRTN	FUNCPTR *	get mblk copy routine pointer
SMIOCSMCPYRTN	FUNCPTR	set mblk copy routine pointer
SMIOCGCCPYRTN	FUNCPTR *	get chained mblk copy routine pointer
SMIOCSCCPYRTN	FUNCPTR	set chained mblk copy routine pointer

MUX INTERFACE The interfaces into this module from the MUX module follow.

smEndLoad

Called by the MUX, the routine initializes and attaches this shared memory network interface driver to the MUX. It is the only globally accessible entry into this driver. This routine typically gets called twice per SM interface and accepts a pointer to a string of initialization parameters. The first call to this routine will be made with an empty string. This action signals the routine to return a device name, not to load and initialize the driver. The second call will be with a valid parameter string, signalling that the driver is to be loaded and initialized with the parameter values in the string. The shared memory region must have been setup and initialized (via smPktSetup) prior to calling **smEndLoad()**. Although initialized, no devices will become active until **smEndStart()** is called.

The following routines are all local to this driver but are listed in the driver entry function table:

smEndUnload()

Called by the MUX, this routine stops all associated devices, frees driver resources, and prepares this driver to be unloaded. If required, calls to **smEndStop()** will be made to all active devices.

smEndStart()

Called by the MUX, the routine starts this driver and device(s). The routine activates this driver and its device(s). The activities performed are dependent upon the selected mode of operation, interrupt or polled.

smEndStop()

Called by the MUX, the routine stops this driver by inactivating the driver and its associated device(s). Upon completion of this routine, this driver is left in the same state it was just after **smEndLoad()** execution.

smEndRecv()

This routine is not called from the MUX. It gets called from this drivers interrupt service routine (ISR) to process input shared memory packets. It then passes them on to the MUX.

smEndSend()

Called by the MUX, this routine sends a packet via shared memory.

smEndPollRec()

Called by the MUX, this routine polls the shared memory region designated for this CPU to determine if any new packet buffers are available to be read. If so, it reads the packet into the supplied mBlk and returns **OK** to the MUX. If the packet is too big for the mBlk or if no packets are available, **EAGAIN** is returned. If the device is not in polled mode, **EIO** is returned.

smEndPollSend()

Called by the MUX, this routine does a polled send of one packet to shared memory. Because shared memory buffers act as a message queue, this routine will attempt to put the polled mode packet at the head of the list of buffers. If no free buffers are available,

the buffer currently appearing first in the list is overwritten with the packet. This routine returns OK or an error code directly, not through errno. It does not free the mBlk it is passed under any circumstances, that being the responsibility of the caller.

smEndIoctl()

Called by the MUX, the routine accesses the control routines for this driver.

smEndMCastAddrAdd()

Called by the MUX, this routine adds an address to a device's multicast address list.

smEndMCastAddrDel()

Called by the MUX, this routine deletes an address from a device's multicast address list.

smEndMCastAddrGet()

Called by the MUX, this routine gets the multicast address list maintained for a specified device.

The following routines do not require shared memory specific logic so the default END library routines are referenced in the function table:

endEtherAddressForm()

Called by the MUX, this routine forms an address by adding appropriate link-level (shared memory) information to a specified mBlk in preparation for transmission.

endEtherPacketDataGet()

Called by the MUX, this routine derives the protocol specific data within a specified mBlk by stripping the link-level (shared memory) information from it. The resulting data are copied to another mBlk.

endEtherPacketAddrGet()

Called by the MUX, this routine extracts address information from one mBlk, ignoring all other data. Each source and destination address is written to its own mBlk. For ethernet packets, this routine produces two output mBlks (an address pair). However, for non-ethernet packets, up to four mBlks (two address pairs) may be produced; two for an intermediate address pair and two more for the terminal address pair.

OPTIONAL EXTERNAL SUPPORT

The following routine(s) may be optionally provided for this module at run time via the associated IOCTL codes:

smEndCopyRtn()

```
int smEndCopyRtn (void* source, void* destination, UINT numBytes);
```

A function hook to allow the BSP to specify how data are copied between mBlks and SM packets. The default is **bcopy()**. Any function specified must have the same type, number, and order of input and output arguments. The following IOCTL codes apply:

```
SMIOCGMCPYRTN - get mblk copy routine pointer  
SMIOCSMCPYRTN - set mblk copy routine pointer
```

For example:

```

void    myDmaCopyFunc (u_char *, u_char *, unsigned);
int     smFd;           /* SM file descriptor */
STATUS result;

...
result = ioctl (smFd, SMIOCSMCPYRTN, (int)myDmaCopyFunc);
...

```

smEndMblkCopyRtn()

```
int smEndMblkCopyRtn (M_BLK_ID, char *, FUNCPTR);
```

A function hook to allow the BSP to specify how frames (mblk chains) are copied to and from SM packets. The default is **netMblkToBufCopy()**, a unidirectional copy. Any function specified must have the same type, number, and order of input and output arguments. The following IOCTL codes apply:

```
SMIOCGCCPYRTN - get chained mblk copy routine pointer
SMIOCSCCPYRTN - set chained mblk copy routine pointer
```

For example:

```

int    myDmaMblkCopyFunc (M_BLK_ID pFrame, char * pBuf, UINT copyDirection);
int    smFd;           /* SM file descriptor */
STATUS result;

...
result = ioctl (smFd, SMIOCSCCPYRTN, (int)myDmaMblkCopyFunc);
...

```

TARGET-SPECIFIC PARAMETERS

These parameters are input to this driver in an ASCII string format, using colon delimited values, via the **smEndLoad()** routine. Each parameter has a preselected radix in which it is expected to be read as shown below.

Parameter	Radix	Use
SM_UNIT	10	Unit number assigned to shared memory device
SM_NET_DEV_NAME	--	String literal name of shared memory device
SM_ANCHOR_ADRS	16	SM anchor region address within SM address space
SM_MEM_ADRS	16	Shared memory address
SM_MEM_MEM_SIZE	16	Shared memory network size in bytes.
SM_TAS_TYPE	10	Used by the master CPU when building SM.
		Test-and-set type (SM_TAS_HARD or SM_TAS_SOFT)
SM_CPUS_MAX	10	Maximum number of CPUs supported in SM (0 = default)
SM_MASTER_CPU	10	Master CPU#
SM_LOCAL_CPU	10	This board's CPU number (NONE = use sysProcNumGet)
SM_PKTS_SIZE	10	Max number of data bytes per shared memory packet (0 = default)

Parameter	Radix	Use
SM_MAX_INPUT_PKTS	10	Max number of queued receive packets for this CPU (0 = default)
SM_INT_TYPE	10	Interrupt method (SM_INT_MAILBOX/_BUS/_NONE)
SM_INT_ARG1	16	1st interrupt argument
SM_INT_ARG2	16	2nd interrupt argument
SM_INT_ARG3	16	3rd interrupt argument
SM_NUM_MBLKS	16	Number of mBlks in driver memory pool (if < 16, a default value is used)
SM_NUM_CBLKS	16	Number of mBlks in driver memory pool (if < 16, a default value is used)

ISR LIMITATIONS

Because this driver may be used in systems without chaining of interrupts, and there can be two or more SM subnets using the same type of SM interrupt, all shared memory subnets are serviced each time there is an interrupt by the SM interrupt service routine (ISR) **smEndIsr()**. This is NOT optimal and does waste some time but is required due to the lack of guaranteed SM interrupt chaining.

When and if interrupt chaining becomes a guaranteed feature for all SM interrupt types, the ISR can be optimized.

MESSAGE LIMITATIONS

This driver does not support multicast messages or multicast operations.

INCLUDES **smEnd.h**

SEE ALSO **muxLib, endLib**

smEndShow

NAME **smEndShow** – shared memory network END driver show routines

ROUTINES **smNetShow()** – show information about a shared memory network

DESCRIPTION This library provides show routines for the shared memory network interface END driver. The **smNetShow()** routine is provided as a diagnostic aid to show current shared memory network status.

INCLUDE FILES **smPktLib.h**

SEE ALSO the network stack programmer's guide.

smcFdc37b78x

NAME	smcFdc37b78x – a superIO (fdc37b78x) initialization source module
ROUTINES	smcFdc37b78xDevCreate() – set correct IO port addresses for Super I/O chip smcFdc37b78xInit() – initializes Super I/O chip Library smcFdc37b78xKbdInit() – initializes the keyboard controller
DESCRIPTION	The FDC37B78x with advanced Consumer IR and IrDA v1.0 support incorporates a keyboard interface, real-time clock, SMSC's true CMOS 765B floppy disk controller, advanced digital data separator, 16 byte data FIFO, two 16C550 compatible UARTs, one Multi-Mode parallel port which includes ChiProtect circuitry plus EPP and ECP support, on-chip 12 mA AT bus drivers, and two floppy direct drive support, soft power management and SMI support and Intelligent Power Management including PME and SCI/ACPI support. The true CMOS 765B core provides 100% compatibility with IBM PC/XT and PC/AT architectures in addition to providing data overflow and underflow protection. The SMSC advanced digital data separator incorporates SMSC's patented data separator technology, allowing for ease of testing and use. Both on-chip UARTs are compatible with the NS16C550. The parallel port, the IDE interface, and the game port select logic are compatible with IBM PC/AT architecture, as well as EPP and ECP. The FDC37B78x incorporates sophisticated power control circuitry (PCC) which includes support for keyboard, mouse, modem ring, power button support and consumer infrared wake-up events. The PCC supports multiple low power down modes. The FDC37B78x provides features for compliance with the "Advanced Configuration and Power Interface Specification" (ACPI). These features include support of both legacy and ACPI power management models through the selection of SMI or SCI. It implements a power button override event (4 second button hold to turn off the system) and either edge triggered interrupts. The FDC37B78x provides support for the ISA Plug-and-Play Standard (Version 1.0a) and provides for the recommended functionality to support Windows95, PC97 and PC98. Through internal configuration registers, each of the FDC37B78x's logical device's I/O address, DMA channel and IRQ channel may be programmed. There are 480 I/O address location options, 12 IRQ options or Serial IRQ option, and four DMA channel options for each logical device.
USAGE	This library provides routines to initialize various logical devices on superIO chip (fdc37b78x). The functions addressed here include:

- Creating a logical device and initializing internal database accordingly.
- Enabling as many device as permitted by this facility by single call. The user of this facility can selectively initialize a set of devices on superIO chip.
- Initializing keyboard by sending commands to its controller embedded in superIO chip.

CHANGING I/O PORT ADDRESSES

This library allows users to change superIO's config, index, and data I/O port addresses. The default I/O port addresses are defined in **target/h/drv/smcFdc37b78x.h** file. These mnemonics can be overridden by defining in architecture related BSP header file. These default setting can also be changed on-the-fly by passing in a pointer of type **SMCFDC37B78X_IOPORTS** with different I/O port addresses. If not redefined, they take their default values as defined in **smcFdc37b78x.h** file.

SMCFDC37B78X_CONFIG_PORT

Defines the config I/O port for SMC-FDC37B78X superIO chip.

SMCFDC37B78X_INDEX_PORT

Defines the index I/O port for SMC-FDC37B78X superIO chip.

SMCFDC37B78X_DATA_PORT

Defines the data I/O port for SMC-FDC37B78X superIO chip.

USER INTERFACE

```
VOID smcFdc37b78xDevCreate
(
    SMCFDC37B78X_IOPORTS *smcFdc37b78x_iop
)
```

This is a very first routine that should be called by the user of this library. This routine sets up IO port address that will subsequently be used later on. The IO PORT setting could either be overridden by redefining **SMCFDC37B78X_CONFIG_PORT**, **SMCFDC37B78X_INDEX_PORT** and **SMCFDC37B78X_DATA_PORT** or on-the-fly by passing in a pointer of type **SMCFDC37B78X_IOPORTS**.

```
VOID smcFdc37b78xInit
(
    int devInitMask
)
```

This is routine intakes device initialization mask and initializes only those devices that are requested by user. Device initialization mask holds bitwise ORed values of all devices that are requested by user to enable on superIO device.

The mnemonics that are supported in current version of this facility are:

SMCFDC37B78X_COM1_EN

Use this mnemonic to enable COM1 only.

SMCFDC37B78X_COM2_EN

Use this mnemonic to enable COM2 only.

SMCFDC37B78X_LPT1_EN

Use this mnemonic to enable LPT1 only.

SMCFDC37B78X_KBD_EN

Use this mnemonic to enable KBD only.

SMCFDC37B78X_FDD_EN

Use this mnemonic to enable FDD only.

The above mentioned can be bitwise ORed to enable more than one device at a time. e.g. if you want COM1 and COM2 to be enable on superIO chip call:

```
smcFdc37b78xInit (SMCFDC37B78X_COM1_EN | SMCFDC37B78X_COM2_EN);
```

The prerequisites for above mentioned call, superIO chip library should be initialized using **smcFdc37b78xDevCreate()** with parameter as per user's need.

```
STATUS smcFdc37b78xKbdInit
(
VOID
)
```

This routine sends some keyboard commands to keyboard controller embedded in superIO chip. Call to this function is required for proper functioning of keyboard driver.

INCLUDE FILES **smcFdc37b78x.h**

sramDrv

NAME **sramDrv** – PCMCIA SRAM device driver

ROUTINES

- sramDrv()** – install a PCMCIA SRAM memory driver
- sramMap()** – map PCMCIA memory onto a specified ISA address space
- sramDevCreate()** – create a PCMCIA memory disk device

DESCRIPTION This is a device driver for the SRAM PC card. The memory location and size are specified when the "disk" is created.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, two routines must be called directly: **sramDrv()** to initialize the driver, and **sramDevCreate()** to create block devices. Additionally, the **sramMap()** routine is called directly to map the PCMCIA memory onto the ISA address space. Note that this routine does not use any mutual exclusion or synchronization mechanism; thus, special care must be taken in the multitasking environment.

Before using this driver, it must be initialized by calling **sramDrv()**. This routine should be called only once, before any reads, writes, or calls to **sramDevCreate()** or **sramMap()**. It can be called from **usrRoot()** in **usrConfig.c** or at some later point.

INCLUDE FILES

none

SEE ALSO

the VxWorks programmer guides.

sym895Lib

NAME

sym895Lib – SCSI-2 driver for Symbios SYM895 SCSI Controller.

ROUTINES

sym895CtrlCreate() – create a structure for a SYM895 device
sym895CtrlInit() – initialize a SCSI Controller Structure
sym895SetHwOptions() – sets the Sym895 chip Options
sym895Intr() – interrupt service routine for the SCSI Controller
sym895Show() – display values of all readable SYM 53C8xx SIOP registers
sym895GPIOConfig() – configures general purpose pins GPIO 0-4
sym895GPIOCtrl() – controls general purpose pins GPIO 0-4
sym895Loopback() – this routine performs loopback diagnostics on 895 chip

DESCRIPTION

The SYM53C895 PCI-SCSI I/O Processor (SIOP) brings Ultra2 SCSI performance to Host adapter, making it easy to add a high performance SCSI Bus to any PCI System. It supports Ultra-2 SCSI rates and allows increased SCSI connectivity and cable length Low Voltage Differential (LVD) signaling for SCSI. This driver runs in conjunction with SCRIPTS Assembly program for the Symbios SCSI controllers. These scripts use DMA transfers for all data, messages, and status transfers.

For each controller device a manager task is created to manage SCSI threads on each bus. A SCSI thread represents each unit of SCSI work.

This driver supports multiple initiators, disconnect/reconnect, tagged command queuing and synchronous data transfer protocol. In general, the SCSI system and this driver will automatically choose the best combination of these features to suit the target devices used. However, the default choices may be over-ridden by using the function "**scsiTargetOptionsSet()**" (see **scsiLib**).

Scatter / Gather memory support: Scatter-Gather transfers are used when data scattered across memory must be transferred across the SCSI bus together with out CPU intervention. This is achieved by a chain of block move script instructions together with the support from the driver. The driver is expected to provide a set of addresses and byte counts for the SCRIPTS code. However there is no support as such from vxworks SCSI Manager for this kind of data transfers. So the implementation, as of today, is not completely integrated with vxworks, and assumes support from SCSI manager in the form of array of

pointers. The macro **SCATTER_GATHER** in **sym895.h** is thus not defined to avoid compilation errors.

Loopback mode allows 895 chip to control all SCSI signals, regardless of whether it is in initiator or target role. This mode insures proper SCRIPTS instructions fetches and data paths. SYM895 executes initiator instructions through the SCRIPTS, and the target role is implemented in **sym895Loopback** by asserting and polling the appropriate SCSI signals in the SOCL, SODL, SBCL, and SBDL registers.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly **sym895CtrlCreate()** to create a controller structure, and **sym895CtrlInit()** to initialize it. If the default configuration is not correct, the routine **sym895SetHwRegister()** must be used to properly configure the registers.

Critical events, which are to be logged anyway irrespective of whether debugging is being done or not, can be logged by using the **SCSI_MSG** macro.

PCI MEMORY ADDRESSING

The global variable **sym895PciMemOffset** was created to provide the BSP with a means of changing the **VIRT_TO_PHYS** mapping without changing the functions in the **cacheFuncs** structures. In generating physical addresses for DMA on the PCI bus, local addresses are passed through the function **CACHE_DMA_VIRT_TO_PHYS** and then the value of **sym895PciMemOffset** is added. For backward compatibility, the initial value of **sym895PciMemOffset** comes from the macro **PCI_TO_MEM_OFFSET**.

INTERFACE

The BSP must connect the interrupt service routine for the controller device to the appropriate interrupt system. The routine to be called is **sym895Intr()**, and the argument is the pointer to the controller device pSiop. i.e.

```
pSiop = sym895CtrlCreate (...);  
intConnect (xxxx, sym895Intr, pSiop);  
sym895CtrlInit (pSiop, ...);
```

HARDWARE ACCESS

All hardware access is to be done through macros. The default definition of the **SYM895_REGx_READ()** and **SYM895_REGx_WRITE()** macros (where x stands for the width of the register being accessed) assumes an I/O mapped model. Register access mode can be set to either IO/memory using **SYM895_IO_MAPPED** macro in **sym895.h**. The macros can be redefined as necessary to accommodate other models, and situations where timing and write pipe considerations need to be addressed. In IO mapped mode, BSP routines **sysInByte()**, **sysOutByte()** are used for accessing SYM895 registers. If these standard calls are not supported, the calls supported by respective BSP are to be mapped to these standard calls. Memory mapped mode makes use of pointers to register offsets.

The macro **SYM895_REGx_READ(pDev, reg)** is used to read a register of width x bits. The two arguments are the device structure pointer and the register offset.

The macro SYM895_REGx_WRITE(pDev, reg,data) is used to write data to the specified register address. These macros presume memory mapped I/O by default. Both macros can be redefined to tailor the driver to some other I/O model.

The global variable sym895Delaycount provides the control count for the sym895's delay loop. This variable is global in order to allow BSPs to adjust its value if necessary. The default value is 10 but it may be set to a higher value as system clock speeds dictate.

INCLUDE FILES scsiLib.h, sym895.h, and sym895Script.c

SEE ALSO scsiLib, scsi2Lib, cacheLib, SYM53C895 PCI-SCSI I/O Processor Data Manual Version 3.0, Symbios Logic PCI-SCSI I/O Processors Programming Guide Version 2.1

tcic

NAME tcic – Databook TCIC/2 PCMCIA host bus adaptor chip driver

ROUTINES tcicInit() – initialize the TCIC chip

DESCRIPTION This library contains routines to manipulate the PCMCIA functions on the Databook DB86082 PCMCIA chip.

The initialization routine **tcicInit()** is the only global function and is included in the PCMCIA chip table **pcmciaAdapter**. If **tcicInit()** finds the TCIC chip, it registers all function pointers of the **PCMCIA_CHIP** structure.

INCLUDE FILES none

tcicShow

NAME tcicShow – Databook TCIC/2 PCMCIA host bus adaptor chip show library

ROUTINES tcicShow() – show all configurations of the TCIC chip

DESCRIPTION This is a driver show routine for the Databook DB86082 PCMCIA chip. **tcicShow()** is the only global function and is installed in the PCMCIA chip table **pcmciaAdapter** in **pcmciaShowInit()**.

INCLUDE FILES none

tffsConfig

NAME	tffsConfig – TrueFFS configuration file for VxWorks
ROUTINES	tffsShowAll() – show device information on all socket interfaces tffsShow() – show device information on a specific socket interface tffsBootImagePut() – write to the boot-image region of the flash device
DESCRIPTION	This source file, with the help of sysTffs.c , configures TrueFFS for VxWorks. The functions defined here are generic to all BSPs. To include these functions in the BSP-specific module, the BSP's sysTffs.c file includes this file. Within the sysTffs.c file, define statements determine which functions from the tffsConfig.c file are ultimately included in TrueFFS. The only externally callable routines defined in this file are tffsShow() , tffsShowAll() , and tffsBootImagePut() . You can exclude the show utilities if you edit config.h and <code>define INCLUDE_SHOW_ROUTINES</code> . You can exclude tffsBootImagePut() if you edit sysTffs.c and <code>define INCLUDE_TFFS_BOOT_IMAGE</code> . (If you find these utilities are missing and you want them included, edit config.h and <code>define INCLUDE_SHOW_ROUTINES</code> and <code>INCLUDE_TFFS_BOOT_IMAGE</code> .) If you wish to include only the TrueFFS specific show routines you could define <code>INCLUDE_TFFS_SHOW</code> instead of <code>INCLUDE_SHOW_ROUTINES</code> in config.h . However, for the most part, these externally callable routines are only a small part of the TrueFFS configuration needs handled by this file. The routines internal to this file make calls into the MTDs and translation layer modules of TrueFFS. At link time, resolving the symbols associated with these calls pulls MTD and translation layer modules into VxWorks. However, each of these calls to the MTDs and the translation layer modules is only conditionally included. The constants that control the includes are defined in sysTffs.c . To exclude an MTD or translation layer module, you edit sysTffs.c , <code>define INCLUDE_MTD_*</code> and <code>INCLUDE_TRANSLATION_LAYER_*</code> , and rebuild sysTffs.o . These constants are described in the reference entry for sysTffs .
INCLUDE FILES	stdcomp.h

vgaInit

NAME	vgaInit – a VGA 3+ mode initialization source module
ROUTINES	vgaInit() – initializes the VGA chip and loads font in memory.

DESCRIPTION	This library provides initialization routines to configure VGA in 3+ alphanumeric mode.
	The functions addressed here include:
	- Initialization of the VGA specific register set.
USER INTERFACE	<pre>STATUS vgaInit (VOID)</pre>
	This routine will initialize the VGA specific register set to bring a VGA card in VGA 3+ mode and loads the font in plane 2.
REFERENCES	<i>Programmer's Guide to the EGA, VGA, and Super VGA Cards - Ferraro. Programmer's Guide to PC & PS/2 Video Systems - Richard Wilton.</i>
INCLUDE FILES	NONE.

vxBus

NAME	vxBus – vxBus subsystem source file
ROUTINES	<code>vxLibInit()</code> – initialize vxBus library <code>vxInit()</code> – initialize vxBus <code>vxDevInitInternal()</code> – second-pass initialization of devices <code>vxDevConnectInternal()</code> – third-pass initialization of devices <code>vxLibError()</code> – handle error conditions <code>vxDevRegister()</code> – register a device driver <code>vxDrvRescan()</code> – rescan all orphans to match against driver <code>vxDeviceDriverRelease()</code> – turn an instance into an orphan <code>vxDriverUnregister()</code> – remove a device driver from the bus subsystem <code>vxBusTypeRegister()</code> – register a bus type <code>vxBusTypeUnregister()</code> – unregister a bus type <code>vxDeviceAnnounce()</code> – announce device discovery to bus subsystem <code>vxDevRemovalAnnounce()</code> – announce device removal to bus subsystem <code>vxBusAnnounce()</code> – announce bus discovery to bus subsystem <code>vxDevParent()</code> – find parent device <code>vxDevPath()</code> – trace from device to nexus <code>vxDevMethodGet()</code> – find entry point of method <code>vxDevIterate()</code> – perform specified action for each device <code>vxDeviceMethodRun()</code> – run method on device <code>vxDevMethodRun()</code> – run method on devices <code>vxSubDevAction()</code> – perform an action on all devs on bus controller

vxbResourceFind() – find and allocate a VxBus resource
vxbDevError() – driver does not support specified functionality
nullDrv() – optional driver functionality not present
noDev() – optional driver functionality not present
vxbBusTypeString() – retrieve bus type string
vxbAccessMethodGet() – find specific method for accessing device
vxbVolRegWrite() – volatile register writes
vxbIntConnect() – connect device's interrupt
vxbIntDisconnect() – disconnect device's interrupt
vxbIntAcknowledge() – Acknowledge device's interrupt
vxbIntEnable() – Enable device's interrupt
vxbIntDisable() – disable device's interrupt
vxbIntVectorGet() – get device's interrupt vector
vxbIntToCpuRoute() – reroute all interrupts for a specified CPU
vxbIntReroute() – Route specified interrupt to destination CPU
vxbDevStructAlloc() – allocate VX_B_DEVICE structure
vxbDevStructFree() – free VX_B_DEVICE structure
vxbInstUnitSet() – set the unit number
vxbInstUnitGet() – get the unit number
vxbNextUnitGet() – get the next available unit number for a driver
vxbLockTake() – take a VxBus lock
vxbLockGive() – release a VxBus lock
vxbLockInit() – initialize a VxBus lock

DESCRIPTION This library contains the support routines for the VxBus subsystem.

INCLUDE FILES **vxBus.h** **vxbPlbLib.h**

vxbArchAccess

NAME **vxbArchAccess** – VxBus access routines specific to an architecture

ROUTINES

- _archRegProbe()** – probe a register on the device
- _archRegisterRead8()** – read 8-bit value from a register
- _archRegisterRead16()** – read 16-bit value from a register
- _archRegisterRead32()** – read 32-bit value from a register
- _archRegisterRead64()** – read 64-bit value from a register
- _archRegisterWrite8()** – write 8-bits to a register
- _archRegisterWrite16()** – write 16-bits to a register
- _archRegisterWrite32()** – write 32-bits to a register
- _archRegisterWrite64()** – write 64-bits to a register
- _archVolatileRegisterWrite()** – write to a volatile register

_archVolatileRegisterRead() – read from a volatile register
_archOptRegWr64_00() – write 64 bits to a mem space register
_archOptRegWr64_07() – swap 64 bit data and write to a mem space register
_archOptRegWr64_20() – write 64 bits to an IO space register
_archOptRegWr64_27() – swap and write 64 bits to an IO space register
_archOptRegWr64_10() – write 64 bits to a mem space register and flush data
_archOptRegWr64_17() – swap, write 64 bits to a mem space register & flush data
_archOptRegWr64_30() – write 64 bits to a IO space register and flush data
_archOptRegWr64_37() – swap, write 64 bits to a IO space register & flush data
_archOptRegWr32_00() – write 32 bits to mem space register
_archOptRegWr32_03() – swap the data and write 32 bits to mem space register
_archOptRegWr32_20() – write 32 bits to an IO space register
_archOptRegWr32_23() – swap and write 32 bits to an IO space register
_archOptRegWr32_10() – write 32 bits to an mem space register and flush
_archOptRegWr32_10() – swap & write 32 bits to an mem space register and flush
_archOptRegWr32_30() – write 32 bits to an IO space register and flush
_archOptRegWr32_33() – swap and write 32 bits to an IO space register & flush
_archOptRegWr16_00() – write 16 bits to a mem space register
_archOptRegWr16_01() – swap and write 16 bits to a mem space register
_archOptRegWr16_20() – write 16 bits to an IO space register
_archOptRegWr16_21() – swap and write 16 bits to an IO space register
_archOptRegWr16_10() – write 16 bits to a mem space register and flush data
_archOptRegWr16_11() – swap, write 16 bits to mem space register & flush data
_archOptRegWr16_30() – write 16 bits to IO space register & flush data
_archOptRegWr16_31() – swap & write 16 bits to IO space register & flush data
_archOptRegWr8_00() – write 8 bits to mem space register
_archOptRegWr8_20() – write 8 bits to IO space register
_archOptRegWr8_10() – write 8 bits to mem space register and flush data
_archOptRegWr8_30() – write 8 bits to IO space register and flush data
_archOptRegWrRd64_00() – write 64 bits to a mem space register and read back
_archOptRegWrRd64_07() – swap 64 bit data and write & read from mem space
_archOptRegWrRd64_20() – write 64 bits to an IO space register and read back
_archOptRegWrRd64_27() – swap, write 64 bits & read from an IO space register
_archOptRegWrRd64_10() – write 64 bits to mem space register, flush data & read
_archOptRegWrRd64_17() – swap, write 64 bits to memspace, flush data & read
_archOptRegWrRd64_30() – write 64 bits to IO space register, flush and read
_archOptRegWrRd64_37() – swap, write 64 bits to IO space, flush & read data
_archOptRegWrRd32_00() – write 32 bits to mem space register & read back
_archOptRegWrRd32_03() – swap, write 32 bits to mem space register & read
_archOptRegWrRd32_20() – write 32 bits to an IO space register & read back
_archOptRegWrRd32_23() – swap, write 32 bits to an IO space register & read
_archOptRegWrRd32_10() – write 32 bits to memspace register, flush & read
_archOptRegWrRd32_13() – swap, write 32 bits to memspace, flush & read
_archOptRegWrRd32_30() – write 32 bits to an IO space register, flush & read
_archOptRegWrRd32_33() – swap, write 32 bits to IO space, flush & read

`_archOptRegWrRd16_00()` – write 16 bits to a mem space register & read back
`_archOptRegWrRd16_01()` – swap, write 16 bits to a mem space register & read
`_archOptRegWrRd16_20()` – write 16 bits to an IO space register & read
`_archOptRegWrRd16_21()` – swap, write 16 bits to an IO space register & read
`_archOptRegWrRd16_10()` – write 16 bits to memspace, flush data & read
`_archOptRegWrRd16_11()` – swap, write 16 bits to memspace, flush & read data
`_archOptRegWrRd16_30()` – write 16 bits to IO space, flush & read back data
`_archOptRegWrRd16_31()` – swap, write 16 bits to IO space, flush data & read
`_archOptRegWrRd8_00()` – write 8 bits to mem space register and read back data
`_archOptRegWrRd8_20()` – write 8 bits to IO space register and read back
`_archOptRegWrRd8_10()` – write 8 bits to mem space register, flush data & read
`_archOptRegWrRd8_30()` – write 8 bits to IO space register, flush data & read
`_archOptRegRd64_00()` – read 64 bits from mem space register
`_archOptRegRd64_07()` – read 64 bits from mem space register and swap data
`_archOptRegRd64_20()` – read 64 bits from IO space register
`_archOptRegRd64_27()` – read 64 bits from IO space register and swap data
`_archOptRegRd32_00()` – read 32 bits from mem space register
`_archOptRegRd32_03()` – read 32 bits from mem space register and swap data
`_archOptRegRd32_20()` – read 32 bits from IO space register
`_archOptRegRd32_23()` – read 32 bits from IO space register and swap data
`_archOptRegRd16_00()` – read 16 bits from mem space register
`_archOptRegRd16_01()` – read 16 bits from mem space register and swap data
`_archOptRegRd16_20()` – read 16 bits from IO space register
`_archOptRegRd16_21()` – read 16 bits from IO space register and swap data
`_archOptRegRd8_00()` – read 8 bits from mem space register
`_archOptRegRd8_20()` – read 8 bits from IO space register
`vxbRegMap()` – obtain an access handle for a given register mapping

DESCRIPTION	This library contains the support routines for architecture dependent access on the vxBus.
INCLUDE FILES	<code>vxbAccess.h</code> <code>vxbArchAccess.h</code>

vxbAuxClkLib

NAME	<code>vxbAuxClkLib</code> – vxBus auxiliary clock library
ROUTINES	<code>vxbAuxClkLibInit()</code> – initialize the auxiliary clock library <code>sysAuxClkHandleGet()</code> – get the timer handle for auxiliary clock <code>sysAuxClkConnect()</code> – connect a routine to the auxiliary clock interrupt <code>sysAuxClkDisable()</code> – turn off auxiliary clock interrupts <code>sysAuxClkEnable()</code> – turn on auxiliary clock interrupts <code>sysAuxClkRateGet()</code> – get the auxiliary clock rate

sysAuxClkRateSet() – set the auxiliary clock rate
vxbAuxClkShow() – show the auxiliary clock information

DESCRIPTION	This module implements the auxiliary clock library interfaces.
INCLUDE FILES	none

vxbCn3xxxTimer

NAME	vxbCn3xxxTimer – Cn3xxxTimer driver for VxBus
ROUTINES	vxbCn3xxxTimerDrvRegister() – register cn3xxx timer driver
DESCRIPTION	<p>This is the vxBus compliant timer driver which implements the functionality specific to timers on Cavium CN3xxx Core.</p> <p>The cn3xxx timer has 23-bit counter and clock upto a frequency of 1MHz. Hence it allows a maximum count of 8388608. It can operate in two modes:</p> <ul style="list-style-type: none">i) One shot mode - The timer counts down to 0 and stops.ii) Repeated mode - The timer counts done to 0 and gets automatically reloaded <p>A timer specific data structure (struct cn3xxxTimerData) is maintained within this driver. This is given as pCookie in cn3xxxTimerAllocate and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.</p> <p>The driver implements all the vxBus driver specific initialization routines like cn3xxxTimerInstInit(), cn3xxxTimerInstInit2() and cn3xxxTimerInstConnect().</p> <p>A variable of type struct cn3xxxTimerData is allocated for a timer device and stored in pInst->pDrvCtrl in cn3xxxTimerInstInit()</p> <p>cn3xxxTimerInstInit2() hooks an ISR to be used for the timer device and also hooks the decrementeer exception handler.</p> <p>A method for methodId VXB_METHOD_TIMER_FUNC_GET is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.</p> <p>The driver is SMP-Aware. It uses ISR-Task Callable spinlock instead of inlocks for locking of interrupt. Hence it will work on SMP Environment.</p>
INCLUDE FILES	none

vxbColdFireSio

NAME	vxbColdFireSio – coldfire vxBus Serial Communications driver
ROUTINES	coldFireSioRegister() – register coldFire sio driver
DESCRIPTION	This is the driver for the UART contained in the ColdFire Microcontroller.
INCLUDE FILES	none

vxbDelayLib

NAME	vxbDelayLib – vxBus delay function interfaces file
ROUTINES	vxbDelayLibInit() – initialize the delay library vxbDelay() – delay for a moment vxbMsDelay() – delay for delayTime milliseconds vxbUsDelay() – delay for delayTime microseconds vxbDelayTimersShow() – show the delay timers information
DESCRIPTION	This file implements the vxbus delay functionalities like <code>vxbDelay()</code> , <code>vxbMsDelay()</code> and <code>vxbUsDelay()</code> . All the delay functions are busy wait functions. <code>vxbDelay()</code> gives a delay of 1 ms <code>vxbMsDelay()</code> is used to give a delay in units of milliseconds <code>vxbUsDelay()</code> is used to give a delay in units of microseconds
INCLUDE FILES	none

vxbDevTable

NAME	vxbDevTable – Support utils for table-based device enumeration
ROUTINES	vxbDevTblEnumerate() – enumerate VxBus devices from hwconf device table
DESCRIPTION	This library contains the utility functions used for the table based device enumeration.
INCLUDE FILES	<code>vxBus.h</code>

vxbDmaBufLib

NAME	vxbDmaBufLib – buffer and DMA system for VxBus drivers
ROUTINES	<code>vxbDmaBufInit()</code> – initialize buffer and DMA system <code>vxbDmaBufTagParentGet()</code> – retrieve parent DMA tag <code>vxbDmaBufTagCreate()</code> – create a DMA tag <code>vxbDmaBufTagDestroy()</code> – destroy a DMA tag <code>vxbDmaBufMapCreate()</code> – create/allocate a DMA map <code>vxbDmaBufMapDestroy()</code> – release a DMA map <code>vxbDmaBufMemAlloc()</code> – allocate DMA-able memory <code>vxbDmaBufMemFree()</code> – release DMA-able memory <code>vxbDmaBufMapLoad()</code> – map a virtual buffer <code>vxbDmaBufMapMblkLoad()</code> – map a virtual buffer with mBlk <code>vxbDmaBufMapIoVecLoad()</code> – map a virtual buffer with scatter/gather <code>vxbDmaBufMapUnload()</code> – unmap/destroy a previous virtual buffer mapping <code>vxbDmaBufMapFlush()</code> – flush DMA Map cache <code>vxbDmaBufMapInvalidate()</code> – invalidate DMA Map cache <code>vxbDmaBufFlush()</code> – partial cache flush for DMA map <code>vxbDmaBufInvalidate()</code> – partial cache invalidate for DMA map <code>vxbDmaBufSync()</code> – do cache flushes or invalidates
DESCRIPTION	<p>This module provides support for driver DMA. The organization of this follows the FreeBSD bus_dma to some extent.</p> <p>DMA devices are supported by use of a DMA Tag. Each DMA tag corresponds to a set of restrictions on DMA. These restrictions include information such as alignment, transaction size, minimum and maximum address ranges, and so on. Tags are created with the <code>vxbDmaBufTagCreate()</code> function.</p> <p>Once a DMA tag is available, the driver should create DMA maps to describe pending operations. A map is created when a buffer address (or a set of buffer addresses) is provided to this module. The map contains the physical addresses representing the buffer which the driver can use to fill in to descriptor fields. Each map usually represents a single pending transfer. Scatter/gather is supported both with the UIO structure or with mBlks.</p> <p>Buffers may be cached, and that is handled by the use of flush and invalidate operations, <code>vxbDmaBufMapFlush()</code> and <code>vxbDmaBufMapInvalidate()</code>. Before initiating a write operation, the driver must call the flush function. After a buffer has been filled by the DMA device, the driver must call the invalidate function before passing the data to the OS/middleware module (e.g. the network stack or filesystem). The <code>vxbDmaBufSync()</code> operation, with the same semantics as the FreeBSD version, is available to help ease porting of FreeBSD drivers to VxWorks.</p> <p>Device descriptors are special in several ways. Memory for the descriptors must be allocated somehow, but DMA restrictions are still in place for the descriptors, since both the</p>

CPU and the device access the data contained in the descriptors. This module provides a memory allocation function, `vxbDmaBufMemAlloc()`, intended primarily for use as descriptors. In addition, descriptors require DMA maps in order for the driver to program the descriptor location into device register(s). However, flush and invalidate operations should be performed on an individual descriptor, and not on the entire descriptor table. Drivers may choose to have a map for each descriptor, or a single map for all descriptors. If a single map is used, flush and invalidate operations can be performed on a range within a DMA map with the `vxbDmaBufFlush()` and `vxbDmaBufInvalidate()` functions, which accept offset and size arguments.

Typical calling sequences include:

descriptors:

```
vxbDmaBufTagCreate()  
vxbDmaBufMemAlloc()  
vxbDmaBufFlush() and vxbDmaBufInvalidate()
```

buffers:

during initialization:

```
vxbDmaBufTagCreate()  
vxbDmaBufMapCreate()
```

during write setup

```
vxbDmaBufMapLoad() / vxbDmaBufMapMblkLoad() / vxbDmaBufMapIoVecLoad()  
vxbDmaBufSync(..., VXB_DMABUF_SYNC_PREWRITE)
```

during receive interrupt handling

```
vxbDmaBufSync(..., VXB_DMABUF_SYNC_POSTREAD)  
/* in rfa_addbuf() */  
vxbDmaBufMapLoad()  
vxbDmaBufSync(..., VXB_DMABUF_SYNC_PREREAD | VXB_DMABUF_SYNC_PREWRITE)
```

Current implementation notes:

This library is intended to support bounce buffers, where the device reads and writes to one area of memory, but the OS/middleware module uses a different region. However, this functionality is not currently implemented.

In the normal case, allocation for `vxbDmaBufMemAlloc()` is currently performed with `malloc()` from the system memory pool. If the `VXB_DMABUF_NOCACHE` flag is specified in the `vxbDmaBufTagCreate()` call, then `cacheDmaMalloc()` is used. This has several implications. First, `vxbDmaBufMemAlloc()` cannot be called until the system memory

pool has been initialized, which occurs before the VxBus init2 routine. So **vxbDmaBufMemAlloc()** cannot be used in the driver's probe or init1 routines. Second, some hardware platforms cannot use memory from the system memory pool (whether cached or not). Therefore, this module allows BSP hooks for the BSP to override the memory allocation and free routines. The routines must take the **VXB_DEVICE_ID** and size as arguments to the allocation routine, and the **VXB_DEVICE_ID** and address as arguments to the free routine. If provided, the BSP routines must allocate buffers for all devices, though internally they can default to **malloc()** and **free()** for everything except the special-requirements devices. Once installed, the BSP hooks must not be changed.

INCLUDE FILES**vxbDmaBufLib.h**

vxbDmaLib

NAME**vxbDmaLib** – vxBus DMA interfaces module**ROUTINES**

vxbDmaLibInit() – initialize the VxBus slave DMA library
vxbDmaChanAlloc() – allocate and initialize a DMA channel
vxbDmaChanFree() – free a DMA channel

DESCRIPTION

This library provides the DMA specific interfaces which can be used by device drivers to perform DMA operations.

INCLUDE FILES

none

vxbEndQctrl

NAME**vxbEndQctrl** – utility to change the RX job queue of a vxbus END device**ROUTINES**

vxbEndQctrlInit() – initialize the vxbEndQctrl library
vxbEndQnumSet() – set the network job queue for a VxBus END device

DESCRIPTION

This module provides convenient functions to change the job queue used by an END device to deliver received packets to the MUX (and to attached network protocols). In SMP systems, assigning different interfaces to different job queues may improve performance for some network loads.

This component may be added to the VxWorks image by including the component **INCLUDE_VXBUS_END_QCTRL**.

By default, a VxBus END device will post work to the job queue serviced by the network daemon tNet0, i.e. to network job queue number 0. To set the job queue used by a VxBus END device to a different job queue, use the routine **vxbEndRxQnumByNameSet()**. For instance, to set the interface *gei2* to use job queue number 1 (serviced by tNet1), call:

```
vxbEndRxQnumByNameSet ("gei", 2, 1);
```

INCLUDE FILES **hwif/util/vxbEndQctrl.h**

vxbEpicIntCtlr

NAME	vxbEpicIntCtlr – Driver for Embedded Programmable Interrupt Controller
ROUTINES	vxbEpicIntCtlrRegister() – register epicIntCtlr driver vxbEpicIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller vxbEpicSharedMsgHandler() – Handles the interrupt and calls the relevant ISRs
DESCRIPTION	<p>This module implements the Embedded Programmable Interrupt Controller (EPIC) driver for the MPC8548.</p> <p>The EPIC is an integrated interrupt controller in the 8548 which provides following major capabilities:</p> <ul style="list-style-type: none"> Support for twelve external interrupt sources and thirty-two internal interrupt sources Support for connection of external interrupt controller device e.g. 8259 like as implemented on a WinBond chip 12 external interrupt sources 32 internal interrupt sources 16 programmable interrupt priority levels Fully-nested interrupt delivery Spurious vector generation Route to critical interrupt destinations Route to external pin Configurable sense and polarity at initialization and runtime Inter-processor interrupts <p>The current implementation of this EPIC controller does not support the following features or mode of operations:</p>

PIC global timers

Messaging interrupts

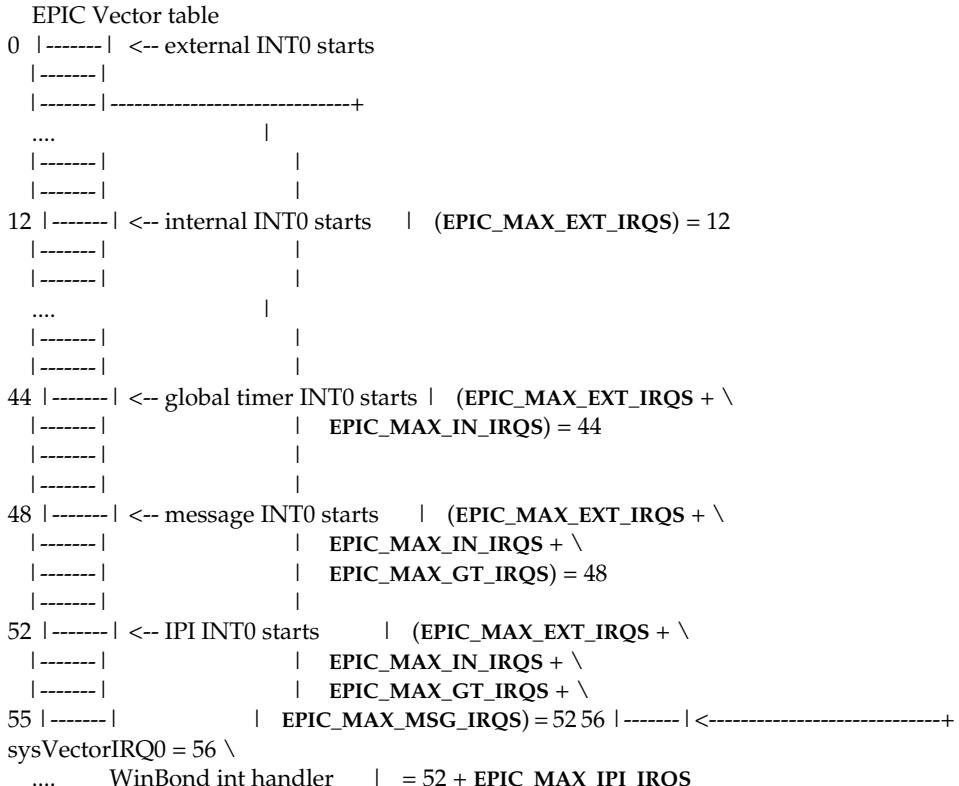
EPIC features are customized by writing into general control registers or into interrupt level specific registers (IVPRs).

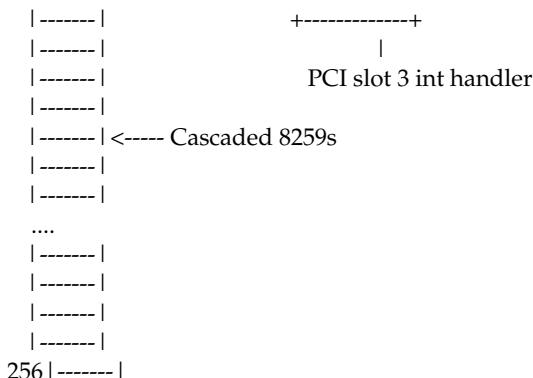
This driver allows a basic interface to the EPIC such as initializing it, setting interrupt vectors, priorities, level/edge sense and interrupt polarities, as well as enabling and disabling specific interrupts.

This driver implements a complete interrupt architecture system, complete with vector table.

Since interrupt vectors can be shared, this driver does provide for overloading of interrupt routines (i.e. there is a list of interrupt routines for each interrupt vector (level)). To service a vector requires that all connected interrupt routines be called in order of their connection.

The following diagram shows an example of how interrupts can be configured in this system.





The driver is designed to put external interrupts at the beginning of the vector table. As a result, devices that route their interrupt to the EPIC on the MPC8540 does not need to translate the vector number. Therefore, the macros `IVEC_TO_INUM(x)` and `INUM_TO_IVEC(x)` are not necessary. For some existing drivers, it may be necessary to use the following defines:

```
#undef INUM_TO_IVEC  
#define INUM_TO_IVEC(x) (x)  
#undef IVEC_TO_INUM  
#define IVEC_TO_INUM(x) (x)
```

If there are other devices in the system capable of generating their own vectors then we presume that an appropriate interrupt handler is created and attached to the vector associated with the correct IRQ number. That interrupt handler would get a new vector directly from the device and then call all of the handlers attached to that new vector. Vector information is stored in a linked list of `INT_HANDLER_DESC` structures. The `vxbEpicIntTbl` array contains a pointer to the first entry for each vector.

INITIALIZATION

This driver is initialized from the BSP, usually as part of `sysHwInit()`. The first routine to be called is `vxbSysEpicInit()`. The routine resets the global configuration register and resets the vxbEpic registers to default values.

The second routine to be called is **vxbSysEpicIntrInit()**. This routine takes no arguments. This routine allocates the vector table and initializes the chips to a default state. All individual interrupt sources are disabled. Each has to be individually enabled by **intEnable()** before it will be unmasked and allowed to generate an interrupt.

CRITICAL INTERRUPT

To enable the EPIC to handle also critical interrupt, or if a normal interrupt is to be rerouted to the critical input pin, add the following to the hcfResource table for the epic in `hwconf.c`:

```
const struct hcfResource epic0Resources[] = {  
    /* ... */  
    {"crtEnable",     HCF_RES_INT,  {(void *)TRUE}},
```

```
/* ... */  
};
```

The critical interrupt handler uses information from the summary registers CISR0 and CISR1. The EPIC does not manage critical interrupts and hence Iack or EOI do not apply. It was seen that the summary registers go through a transient state before settling on the result. This causes spurious interrupts to be generated, and the vectors being called. A typical behavior is the printout of "uninitialized PIC interrupt vector 0xXX". This is observed only when at least one source has been routed to critical pin.

CUSTOMIZING THIS DRIVER

The BSP can change the default polarity and sensitivity for the external interrupt and the internal interrupt independently. They are:

```
exSense    /* default to EPIC_SENSE_LVL */  
exPolar    /* default to EPIC_INT_ACT_HIGH */  
inPolar    /* default to EPIC_INT_ACT_HIGH */
```

The default can be overridden by adding the above as parameters in **hwconf.c**. For example,

```
const struct hcfResource epic0Resources[] = {  
    /* ... */  
    { "exPolar", HCF_RES_INT,  {(void *)EPIC_INT_ACT_LOW} },  
    /* ... */  
};
```

The available options are:

```
exSense    EPIC_SENSE_LVL, EPIC_SENSE_EDG  
xxPolar   EPIC_INT_ACT_LOW, EPIC_INT_ACT_HIGH /*if level-sense */  
           EPIC_INT_EDG_NEG, EPIC_INT_EDG_POS /*if edge-sense */
```

The macros **CPU_INT_LOCK()** and **CPU_INT_UNLOCK** provide the access to the CPU level interrupt lock/unlock routines. We presume that there is a single interrupt line to the CPU. By default these macros call **intLock()** and **intUnlock()** respectively.

INCLUDE FILES

vx~~b~~HwifDebug

NAME vx~~b~~HwifDebug -- HWIF Debug Utilities

ROUTINES

- hwifDevRegRead()** – read device register
- hwifDevRegWrite()** – write device register
- hwifDevRegWrite64()** – write device register
- hwifDevCfgRead()** – read from device configuration space
- hwifDevCfgWrite()** – write device configuration space

hwifDevCfgWrite64() – write device register

DESCRIPTION This library implements the debug functions for the hwif modules

INCLUDE FILES **vxBusLib.h**

vxbI8253Timer

NAME **vxbI8253Timer** – driver for the intel 8253 timer

ROUTINES **vxbI8253TimerDrvRegister()** – registers the driver for i8253 timer

DESCRIPTION This is the vxBus compliant timer driver which interacts with the Intel 8253 timer.

A timer specific data structure (struct i8253TimerData) is maintained for each of the 3 timers supported by this driver. An array of such data structures is stored in struct i8253Data;

The driver implements all the vxBus driver specific initialization routines like i8253TimerInstInit (), i8253TimerInstInit2 () and i8253TimerInstConnect ().

A variable of type **struct i8253Data** is allocated for the instance and stored in **pInst->pDrvCtrl** in i8253TimerInstInit ()

i8253TimerInstInit2 () hooks the ISRs to be used.

A method for methodId **vxbTimerFuncGet_desc** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device.

TARGET-SPECIFIC PARAMETERS

The parameters are retrieved from **hwconf.c** in the target BSP.

regBase

i8253 base address. The parameter type is **HCF_RES_INT**.

countRegAddr

count register address The parameter type is **HCF_RES_INT**.

irq

Interrupt vector number for the timer interrupt source. The parameter type is **HCF_RES_INT**.

irqLevel

Interrupt number for the timer interrupt source. The parameter type is **HCF_RES_INT**.

clkFreq

clock frequency of i8253 timer. The parameter type is **HCF_RES_INT**.

clkRateMin

Specify the minimum clock rate(in ticks/sec) which can be supported by the i8253 timer. The parameter type is **HCF_RES_INT**.

clkRateMax

Specify the maximum clock rate(in ticks/sec) which can be supported by the i8253 timer. The parameter type is **HCF_RES_INT**.

INCLUDE FILES

none

vxBI8259IntCtlr

NAME

vxBI8259IntCtlr – Intel 8259A Programmable Interrupt Controller driver

ROUTINES

vxBI8259IntCtlrRegister() – registers I8259 driver with vxBus
vxBI8259IntEnablePIC() – enable a PIC interrupt level
vxBI8259IntDisablePIC() – disables a PIC interrupt level
vxBI8259IntEOI() – to send an EOI signal

DESCRIPTION

This module is a driver for the Intel 8259A PIC (Programmable Interrupt Controller). The Intel 8259A handles up to 8 vectored priority interrupts for the CPU. It is cascade-able for up to 64 vectored priority interrupts. Hence we can cascade 8 Intel 8259 Controllers together to obtain 64 vectored priority interrupts.

The 8259A accepts two types of command words generated by the CPU:

1. Initialization Command Words (ICWs): Before normal operation can begin, each 8259A in the system must be brought to a starting point by a sequence of 2 to 4 bytes timed by WR pulses. There are 4 ICWs.
2. Operation Command Words (OCWs): These are the command words which command the 8259A to operate in various interrupt modes.

Communicating with I8259A

Communication with the 8259A is facilitated by sending various commands over the bus to the two I/O ports (port A and port B) of the controller. ICW1 is sent over Port A and ICW2, ICW3 and ICW4 is send over portB

The 8259A PIC can be controlled by sending specific operation command words (OCWs) to either port B (OCW-1) or port A (OCW-2 and OCW-3).

Intel 8259 PIC can work in following modes: a. Fully nested mode : b. Rotating priority mode c. Special mask mode d. Polled mode For more information on these modes, please refer to Intel 8259 PIC (Programmable Interrupt Controller Manual).

INITIALIZATION

This driver is initialized from the BSP, usually as part of `sysHwInit()`. The driver registers its self with `vxBus` using the `vxBusDevRegister()` function. During initialization, the interrupt controller is initialized by issuing the ICW commands.

The interrupt controller drive also registers following method: i) Method to connect an ISR to an interrupt pin ii) Method to disconnect an ISR from an interrupt pin iii) Method to enable interrupt iv) Method to disable interrupt v) Method to acknowledge the interrupt

CASCADING OF I8259 INTERRUPT CONTROLLER

The 8259A can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels. The master controls the slaves through the 3 line cascade bus. The cascade bus acts like chip selects to the slaves during the INTA sequence.

LEGACY SUPPORT This driver provides legacy support to older I8259 PIC. It maintains some global pointers which get populated only if legacy support is defined.

PLEASE NOTE THAT WE NEED TO REMOVE THIS LEGACY SUPPORT ONCE ALL THE DRIVERS**ARE SUCCESSFULLY PORTED TO VXBUS**

INCLUDE FILES `vxWorks.h`, `spinLockLib.c`, `vxBus.h`, `hwConf.h`, `vxblntCtlr.h`, `vxblntCtlrLib.h`,
`vxBAccess.h`, `vxBusLib.h`

vxblntCtlrLib

NAME `vxblntCtlrLib` -- VxBus Interrupt Controller support library

ROUTINES

- `intCtlrStrayISR()` – flag stray interrupt
- `intCtlrChainISR()` – process interrupt chain
- `intCtlrTableCreate()` – Create table structure
- `intCtlrTableUserSet()` – Fill in ISR table entry: driver name, unit, index
- `intCtlrTableArgGet()` – Fetch ISR argument from ISR table entry
- `intCtlrTableIsrGet()` – Fetch ISR function pointer from ISR table entry
- `intCtlrTableFlagsSet()` – Set ISR flags in ISR table entry
- `intCtlrTableFlagsGet()` – Fetch ISR flags from ISR table entry
- `isrRerouteNotify()` – Notify driver that ISR has been rerouted
- `intCtlrISRDisable()` – Disable specified ISR
- `intCtlrISREnable()` – Enable specified ISR
- `intCtlrISRRemove()` – Remove ISR from specified pin
- `intCtlrISRAAdd()` – Install ISR at specified pin
- `intCtlrPinFind()` – Retrieve input pin associated with device
- `intCtlrHwConfGet()` – get interrupt input pin configuration from BSP
- `intCtlrHwConfShow()` – print interrupt input pin configuration

DESCRIPTION	This is the library file for vxBus Interrupt Controller Drivers. This file provides the interfaces that are used by vxBus Interrupt Controller Drivers.
INCLUDE FILES	vxbIntCtlrLib.h, vxbIntrCtlr.h

vxbIntDynaCtlrLib

NAME	vxbIntDynaCtlrLib -- VxBus Dynamic Interrupt support library
ROUTINES	vxbMsiConnect() – calls vxbIntDynaConnect if available vxbIntDynaConnect() – Initializes/connects the dynamic interrupt vxbIntDynaVecAlloc() – Finds the the next vector for dynamic interrupt vxbIntDynaVecOwnerFind() – Finds the owner of dynnamic vector alloc vxbIntDynaVecDevMultiProgram() – program multiple vectors vxbIntDynaVecMultiConnect() – connect to multiple vectors vxbIntDynaCtlrInit() – Intializes function pointers to enable library use.
DESCRIPTION	This is the library file for vxbus dynamic interrupt support.
INCLUDE FILES	vxbIntCtlrLib.h, vxbIntrCtlr.h, vxbIntDynaCtlrLib.c

vxbIntelTimestamp

NAME	vxbIntelTimestamp – Driver for the timestamp on the intel chip
ROUTINES	vxbIaTimestampDrvRegister() – register intel timestamp driver
DESCRIPTION	
NOTE	It is strongly recommended that 32-bit software only operate the timer in 32-bit mode. A race condition arises if the 32-bit CPU reads the 64-bit register using two seperate 32-bit reads.
	This is the vxBus compliant timer driver which interacts with the timestamp on the intel chip.
	The timestamp details are maintained in struct iaTimestampData .
	The driver implements all the vxBus driver specific initialization routines like iaTimestampInstInit() , iaTimestampInstInit2() and iaTimestampInstConnect() .

A variable of type **struct iaTimestampData** is allocated for the instance and stored in **pInst->pDrvCtrl** in **iaTimestampInstInit()**

A method for methodId **vxbTimerFuncGet_desc** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device.

TARGET-SPECIFIC PARAMETERS

The following parameter is retrieved from **hwconf.c** in the target BSP.

clkFreq

Specify the clock frequency of the timestamp timer. The parameter type is **HCF_RES_INT**.

INCLUDE FILES	none
----------------------	------

vxbIoApicIntr

NAME	vxbIoApicIntr – VxBus driver for the Intel IO APIC/xAPIC driver
ROUTINES	vxbIoApicIntrDrvRegister() – register ioApic driver vxbIoApicIntrDataShow() – show IO APIC register data acquired by vxBus vxbIoApicIntrShowAll() – show All IO APIC registers
DESCRIPTION	<p>This is the VxBus driver for the Io APIC Interrupt Driver</p> <p>This module is a driver for the IO APIC/xAPIC (Advanced Programmable Interrupt Controller) for P6 (PentiumPro, II, III) family processors and P7 (Pentium4) family processors. The IO APIC/xAPIC is included in the Intel's system chip set, such as ICH2. Software intervention may be required to enable the IO APIC/xAPIC in some chip sets.</p> <p>The 8259A interrupt controller is intended for use in a uni-processor system, IO APIC can be used in either a uni-processor or multi-processor system. The IO APIC handles interrupts very differently than the 8259A. Briefly, these differences are:</p> <ul style="list-style-type: none"> - Method of Interrupt Transmission. The IO APIC transmits interrupts through a 3-wire bus and interrupts are handled without the need for the processor to run an interrupt acknowledge cycle. - Interrupt Priority. The priority of interrupts in the IO APIC is independent of the interrupt number. For example, interrupt 10 can be given a higher priority than interrupt 3. - More Interrupts. The IO APIC supports a total of 24 interrupts. <p>The IO APIC unit consists of a set of interrupt input signals, a 24-entry by 64-bit Interrupt Redirection Table, programmable registers, and a message unit for sending and receiving APIC messages over the APIC bus or the Front-Side (system) bus. IO devices inject</p>

interrupts into the system by asserting one of the interrupt lines to the IO APIC. The IO APIC selects the corresponding entry in the Redirection Table and uses the information in that entry to format an interrupt request message. Each entry in the Redirection Table can be individually programmed to indicate edge/level sensitive interrupt signals, the interrupt vector and priority, the destination processor, and how the processor is selected (statically and dynamically). The information in the table is used to transmit a message to other APIC units (via the APIC bus or the Front-Side (system) bus).

IO APIC is used in the Symmetric IO Mode (define SYMMETRIC_IO_MODE in the BSP). The base address of IO APIC is determined in `loApicInit()` and stored in the global variable `ioApicBase` and `ioApicData`. `ioApicInit()` initializes the IO APIC with information stored in `ioApicRed0_15` and `ioApicRed16_23`. The `ioApicRed0_15` is the default lower 32 bit value of the redirection table entries for IRQ 0 to 15 that are edge triggered positive high, and the `ioApicRed16_23` is for IRQ 16 to 23 that are level triggered positive low. `ioApicRedSet()` and `ioApicRedGet()` are used to access the redirection table. `ioApicEnable()` enables the IO APIC or xAPIC. `ioApicIrqSet()` set the specific IRQ to be delivered to the specific Local APIC.

INCLUDE FILES

none

vxblxp400Sio

NAME`vxblxp400Sio` – IXP400 UART *tty* driver**ROUTINES**

`ixp400SioRegister()` – register `ixp400Sio` driver
`ixp400SioDevInit()` – initialise a UART channel
`ixp400SioIntWr()` – handle a transmitter interrupt
`ixp400SioIntErr()` – handle an error interrupt
`ixp400SioInt()` – interrupt level processing
`ixp400SioStatsShow()` – display UART error statistics.

DESCRIPTION

This is the driver for the Intel IXP400 internal UART. This device comprises an asynchronous receiver/transmitter, a baud rate generator, and some modem control capability.

An IXP400_SIO_CHAN structure is used to describe the serial channel. This data structure is defined in `ixp400Sio.h`.

Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control.

USAGE

This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS.

INCLUDE FILES **ixp400Sio.h**

SEE ALSO *ixp400 Data Sheet*

vxbIxpxp400Timer

NAME **vxbIxpxp400Timer** – ixp4xx processor timer library

ROUTINES **ixp400TimerDrvRegister()** – register ixp400 timer driver

DESCRIPTION This is the vxBus compliant timer driver which implements the functionality specific to ixp400 timers.

A timer-specific data structure (struct ixp400TimerData) is maintained within this driver. This is given as **pCookie** in **ixp400TimerAllocate()** and is used from then on by the Timer Abstraction Layer or application to communicate with this driver.

The driver implements all the vxBus driver-specific initialization routines like **ixp400TimerInstInit()**, **ixp400TimerInstInit2()** and **ixp400TimerInstConnect()**.

A variable of type **struct ixp400TimerData** is allocated for a timer device and stored in **pInst->pDrvCtrl** in **ixp400TimerInstInit()**

ixp400TimerInstInit2() hooks an ISR to be used for the timer device.

A method for methodId **VXB_METHOD_TIMER_FUNC_GET** is implemented in this driver and is used by the Timer Abstraction Layer or application to retrieve the characteristics of a timer device. A pointer to **struct vxbTimerFunctionality** is allocated by the Timer Abstraction Layer or application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.

INCLUDE FILES none

vxbLoApicIntr

NAME **vxbLoApicIntr** – VxBus driver template

ROUTINES **vxbLoApicIntrDrvRegister()** – register loApic driver

vxbLoApicIpiGen() – Generate specified IPI

vxbLoApicIpiConnect() – Connect ISR to specified IPI

vxbLoApicIpiEnable() – Enable ISR for specified IPI
vxbLoApicIpiDisconn() – Disconnect ISR from specified IPI
vxbLoApicIpiDisable() – Disable specified IPI
vxbLoApicIpiPrioGet() – Find priority of specified IPI
vxbLoApicIpiPrioSet() – Set priority of specified IPI
vxbLoApicStatusShow() – show Local APIC TMR, IRR, ISR registers
vxbLoApicIntrShow() – show Local APIC registers

DESCRIPTION	This is the VxBus driver for the Local APIC Interrupt Driver This module is a driver for the local APIC/xAPIC (Advanced Programmable Interrupt Controller) in P6 (PentiumPro, II, III) family processors and P7 (Pentium4) family processors. The local APIC/xAPIC is included in selected P6 (PentiumPro, II, III) and P7 (Pentium4) family processors. Beginning with the P6 family processors, the presence or absence of an on-chip local APIC can be detected using the CPUID instruction. When the CPUID instruction is executed, bit 9 of the feature flags returned in the EDX register indicates the presence (set) or absence (clear) of an on-chip local APIC. The local APIC performs two main functions for the processor: - It processes local external interrupts that the processor receives at its interrupt pins and local internal interrupts that software generates. - In multiple-processor systems, it communicates with an external IO APIC chip. The external IO APIC receives external interrupt events from the system and interprocessor interrupts from the processors on the system bus and distributes them to the processors on the system bus. The IO APIC is part of Intel's system chip set. The local APIC controls the dispatching of interrupts (to its associated processor) that it receives either locally or from the IO APIC. It provides facilities for queuing, nesting and masking of interrupts. It handles the interrupt delivery protocol with its local processor and accesses to APIC registers, and also manages interprocessor interrupts and remote APIC register reads. A timer on the local APIC allows local generation of interrupts, and local interrupt pins permit local reception of processor-specific interrupts. The local APIC can be disabled and used in conjunction with a standard 8259A style interrupt controller. Disabling the local APIC can be done in hardware for the Pentium processors or in software for the P6 and P7 (Pentium4) family processors. The local APIC in the Pentium4 processors (called the xAPIC) is an extension of the local APIC found in the P6 family processors. The primary difference between the APIC architecture and xAPIC architecture is that with Pentium4 processors, the local xAPICs and IO xAPIC communicate with one another through the processors system bus; whereas, with the P6 family processors, communication between the local APICs and the IO APIC is handled through a dedicated 3-wire APIC bus. Also, some of the architectural features of the local APIC have been extended and/or modified in the local xAPIC. The base address of the local APIC and IO APIC is taken from the MP configuration table (see Intel MP Specification Version 1.4) or the IA32_APIC_BASE MSR. It uses
--------------------	---

LOAPIC_BASE and **IOAPIC_BASE** defined in the BSP, if it is not able to find the addresses. This driver contains three routines for use. They are:

loApicInit() initializes the Local APIC for the interrupt mode chosen. **loApicEnable()** enables or disables the Local APIC. **loApicIpi()** delivers the inter processor interrupt to the specified local APIC.

Local APIC is used in the Virtual Wire Mode (define **VIRTUAL_WIRE_MODE** in the BSP) and the Symmetric IO Mode (define **SYMMETRIC_IO_MODE** in the BSP), but not in the PIC Mode which is the default interrupt mode and uses 8259A PIC.

Virtual Wire Mode is one of three interrupt modes defined by the MP specification. In this mode, interrupts are generated by the 8259A equivalent PICs, but delivered to the Boot Strap Processor by the local APIC that is programmed to act as a "virtual Wire"; that is, the local APIC is logically indistinguishable from a hardware connection. This is a uniprocessor compatibility mode.

Symmetric IO Mode is one of three interrupt modes defined by the MP specification. In this mode, the local and IO APICs are fully functional, and interrupts are generated and delivered to the processors by the APICs. Any interrupt can be delivered to any processor. This is the only multiprocessor interrupt mode.

The local and IO APICs support interrupts in the range of 32 to 255. Interrupt priority is implied by its vector, according to the following relationship: "priority = vector / 16".

Here the quotient is rounded down to the nearest integer value to determine the priority, with 1 being the lowest and 15 is the highest. Because vectors 0 through 31 are reserved for exclusive use by the processor, the priority of user defined interrupts range from 2 to 15. A value of 15 in the Interrupt Class field of the Task Priority Register (TPR) will mask off all interrupts, which require interrupt service.

The P6 family processor's local APIC includes an in-service entry and a holding entry for each priority level. To avoid losing interrupts, software should allocate no more than 2 interrupt vectors per priority. The Pentium4 processor expands this support of all acceptance of two interrupts per vector rather than per priority level.

INCLUDE FILES **vxCpuLib.h, vxIpiLib.h, vxbIntCtlrLib.h, vxbIntDynaCtlrLib.h**

vxbLoApicTimer

NAME	vxbLoApicTimer – Driver for the loApic timer
ROUTINES	vxbLoApicTimerDrvRegister() – register loApic timer driver
DESCRIPTION	This is the vxBus compliant timer driver which interacts with the loApic timer.

The timer details of loApic timer is maintained in **struct loApicTimerData**.

The driver implements all the vxBus driver specific initialization routines like **loApicTimerInstInit()**, **loApicTimerInstInit2()** and **loApicTimerInstConnect()**.

A variable of type **struct loApicTimerData** is allocated for the instance and stored in **pInst->pDrvCtrl** in **loApicTimerInstInit()**

loApicTimerInstInit2() hooks an ISR to be used.

A method for methodId **vxbTimerFuncGet_desc** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device.

TARGET-SPECIFIC PARAMETERS

The parameters are retrieved from **hwconf.c** in the target BSP.

regBase

loApic timer register base address. The parameter type is **HCF_RES_INT**.

irq

Interrupt vector number for the timer interrupt source. The parameter type is **HCF_RES_INT**.

irqLevel

Interrupt number for the timer interrupt source. The parameter type is **HCF_RES_INT**.

clkFreq

clock frequency of loApic timer. The parameter type is **HCF_RES_INT**.

clkRateMin

Specify the minimum clock rate(in ticks/sec) which can be supported by loApic timer. The parameter type is **HCF_RES_INT**.

clkRateMax

Specify the maximum clock rate(in ticks/sec) which can be supported by loApic timer. The parameter type is **HCF_RES_INT**.

INCLUDE FILES

none

vx~~b~~**M548xDma**

NAME **vx~~b~~**M548xDma**** – Coldfire MCD slave DMA VxBus driver

ROUTINES **m548xDmaDrvRegister()** – register coldFire DMA driver

DESCRIPTION	This module implements a VxBus slave DMA driver front-end for the Coldfire Multi-Channel DMA controller. The DMA controller is needed in order to support the on-board FEC ethernet ports.
	The MCD controller is driven by microcode, which is loaded from and manipulated using an API library provided by Freescale. The library provides the ability to start, stop, pause and resume DMA operations. For the FEC, once a DMA channel is started, it continues to run until the DMA task is killed. The MCD controller has a single interrupt dedicated to it, which fires whenever a transfer completes. The dma_utils module determines which channel triggered the interrupt and invokes the callback which has been associated with that channel. Note that callbacks occur in interrupt context, and as a resume, so do the DMA completion routines registered via this module.
BOARD LAYOUT	The Multi-Channel DMA controller is internal to the Coldfire CPU.
EXTERNAL INTERFACE	
INCLUDE FILES	none
SEE ALSO	vxBus

vxbM54x5SliceTimer

NAME	vxbM54x5SliceTimer – timer driver for m54x5 Slice timers
ROUTINES	m54x5TimerDrvRegister() – register coldFire timer driver
DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality specific to m54x5 Slice timers
	A timer specific data structure (struct m54x5TimerData) is maintained within this driver. This is given as pCookie in m54x5TimerAllocate and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.
	The driver implements all the vxBus driver specific initialization routines like m54x5TimerInstInit(), m54x5TimerInstInit2() and m54x5TimerInstConnect().
	A variable of type struct m54x5TimerData is allocated for a timer device and stored in pInst->pDrvCtrl in m54x5TimerInstInit()
	m54x5TimerInstInit2() hooks an ISR to be used for the timer device.
	A method for methodId VXB_METHOD_TIMER_FUNC_GET is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer

Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.

INCLUDE FILES

none

vxbM85xxDecTimer

NAME

vxbM85xxDecTimer – driver for Book E PowerPC decrementer

ROUTINES**DESCRIPTION**

This is the vxBus compliant timer driver which implements the functionality specific to the Book E PowerPC decrementer.

The Book E PowerPc specific decrementer is a enhanced one and it differs from legacy PPC Decrementer. The Book E decrementer supports auto reload feature and provides seperate registers for interrupt handling.

Decrmenter specific routines in this driver are appropriated from **m85xxTimer.c**.

The timer specific data structures are declared in **vxbM85xxTimer.h** and its populated by **m85xxTimerInstInit()** during systme boot up, defined in **vxbM85xxTimer.c**.

m85xx Timer instance will allocate an array of per Timer data structure which contains the members specific to the timers.

A pointer to the this timer data structure is recieved (*pCookie) by the timer driver functions.

The Decremeter is initialized as one of two timers registered by the m85xxTimer instance.

This source file provides the Decrementer functionality specific routines to Timer abstraction layer.

Note: This source file is not an independent one which needs to build along with other m85xx timer driver files.

INCLUDE FILES

none

vxbM85xxFITimer

NAME

vxbM85xxFITimer – driver for PowerPC Book E Fixed Interval Timer(FIT)

ROUTINES

DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality specific to the Book E PowerPC Fixed Interval Timer(FIT). The Book E PowerPc specific FIT gives time periods in a fixed range. The FIT specific routines in this driver are appropriated from m85xxTimer.c . A common timer specific data structure is declared in vxbM85xxTimer.h , which consists of members required by the timer like Count, timerState etc., The timer data structure is populated by m85xxTimerInstInit() during system boot up, as defined in vxbM85xxTimer.c . FIT is initialized as one of two timers registered by the m85xxTimer instance. This source file provides the FIT functionality specific routines to Timer abstraction layer. Note: This source file is not an independent one which needs to build along with other m85xx timer driver files.
INCLUDE FILES	none

vxbM85xxTimer

NAME	vxbM85xxTimer – timer driver for m85xx PowerPC Book E Architecture
ROUTINES	m85xxTimerDrvRegister() – register m85xx timer driver
DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality specific to the PowerPC Book E Timer Device. The Book E Timer Device consists of three timer devices with single time base. 1. Decrementer 2. Fixed Interval Timer (FIT) 3. Watch Dog Timer All the 3 timer devices uses the same Timer Registers (i.e.Control and Status registers) for enabling, count loading and interrupt handling etc., Hence this driver implements three timer functionalities in a single timer device instance. The timer specific functions are defined in three seperate source files named vxbM85xxDecTimer.c , vxbM85xxFITimer.c and vxbM85xxWDTimer.c

A device specific instance data structure (struct m85xxTimerInstance), is maintained within this driver, which holds pointer to device instance , no.of timers in the instance and timer functionality specific data structure (struct m85xxTimerData)

A pointer to the allocated instance structure is stored in **pInst->pDrvCtrl** during **m85xxTimerInstInit()**.

The driver implements all the vxBus driver specific initialization routines like **m85xxTimerInstInit()**, **m85xxTimerInstInit2()** and **m85xxTimerInstConnect()**.

m85xxTimerInstInit1() allocates memory for timer instance data structure and fills the timer functionality specific parameters by retrieving it from **hwconf.c**

m85xxTimerInstInit2() attempts to connect the driver to the vxBus interrupt mechanism using **vxbIntConnect()**. If this fails, presumably because the BSP has not been adapted to support the vxBus interrupt mechanism, it attaches the ISR to the decrementer vector via a stub function. A global pointer to the **struct m85xxTimerInstance *** is maintained so that the stub can pass the structure's address to the vxBus-compliant ISR.

A method for methodId **VXB_METHOD_TIMER_FUNC_GET** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of the timer device. A pointer to **struct vxbTimerFunctionality** is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.

TARGET-SPECIFIC PARAMETERS

The parameters are retrieved from **hwconf.c** in the target BSP.

numTimers

no.of timer in this instance. The parameter type is **HCF_RES_INT**.

vectorDec

Decrementer Exception Vector The parameter type is **HCF_RES_INT**.

vectorFit

Fixed Interval Timer(FIT) Exception Vector The parameter type is **HCF_RES_INT**.

sysClockFreq

system clock frequency retrieving function address. The parameter type is **HCF_RES_ADDR**.

decMaxClkRate

Specifies the maximum clock rate(in ticks/sec) of Decrementer The parameter type is **HCF_RES_INT**.

decMinClkRate

Specifies the minimum clock rate(in ticks/sec) of Decrementer The parameter type is **HCF_RES_INT**.

fitMaxClkRate

Specifies the maximum clock rate(in ticks/sec) of FIT The parameter type is **HCF_RES_INT**.

fitMinClkRate

Specifies the minimum clock rate(in ticks/sec) of FIT The parameter type is **HCF_RES_INT**.

INCLUDE FILES	none
----------------------	------

vxbM85xxWDTimer

NAME	vxbM85xxWDTimer – driver for PowerPC Book E Watch Dog Timer(FIT)
-------------	---

ROUTINES	
-----------------	--

DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality specific to the Book E PowerPC Watch Dog Timer (WDT).
--------------------	---

The Book E PowerPc specific WDT gives time periods in a fixed range and it can reset the CPU after a selected period.

The WDT specific routines in this driver are appropriated from **m85xxTimer.c**.

A common timer specific data structure is declared in **vxbM85xxTimer.h**, which consists of members required by the timer like Count, timerState etc.,

The timer data structure is populated by **m85xxTimerInstInit()** during system boot up, as defined in **vxbM85xxTimer.c**.

WDT is initialized as one of three timers registered by the m85xxTimer instance.

This source file provides the WDT functionality specific routines to Timer abstraction layer.

Note: This source file is not an independent one which needs to build along with other m85xx timer driver files.

INCLUDE FILES	none
----------------------	------

vxbMc146818Rtc

NAME	vxbMc146818Rtc – driver for the MC146818 real time clock
-------------	---

ROUTINES	vxbMc146818RtcDrvRegister() – registers the driver for MC146818 RTC
DESCRIPTION	<p>This is the vxBus compliant timer driver which interacts with the MC146818 Real time clock.</p> <p>A timer specific data structure (struct mc146818RtcData) is maintained within this driver.</p> <p>The driver implements all the vxBus driver specific initialization routines like mc146818RtcInstInit (), mc146818RtcInstInit2 () and mc146818RtcInstConnect ().</p> <p>A variable of type struct mc146818RtcData is allocated for the instance and stored in pInst->pDrvCtrl in mc146818RtcInstInit ()</p> <p>mc146818RtcInstInit2 () hooks an ISR to be used.</p> <p>A method for methodId vxbTimerFuncGet_desc is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device.</p>
TARGET-SPECIFIC PARAMETERS	<p>The parameters are retrieved from hwconf.c in the target BSP.</p> <p><i>regBase</i> RTC base address. The parameter type is HCF_RES_INT.</p> <p><i>irq</i> Interrupt vector number for the timer interrupt source. The parameter type is HCF_RES_INT.</p> <p><i>irqLevel</i> Interrupt number for the timer interrupt source. The parameter type is HCF_RES_INT.</p> <p><i>clkFreq</i> clock frequency of RTC. The parameter type is HCF_RES_INT.</p> <p><i>clkRateMin</i> Specify the minimum clock rate(in ticks/sec) which can be supported by the RTC. The parameter type is HCF_RES_INT.</p> <p><i>clkRateMax</i> Specify the maximum clock rate(in ticks/sec) which can be supported by the RTC. The parameter type is HCF_RES_INT.</p>
INCLUDE FILES	none

vx~~b~~**MipsCavIntCtlr**

NAME	vxbMipsCavIntCtlr – interrupt controller for Cavium CN3xxx family
-------------	--

ROUTINES	vxbMipsCavIntCtlrRegister() – register the mipsIntCtlr driver mipsCavIntCtlrShow() – show pDrvCtrl for mipsIntCtlr
DESCRIPTION	This file supplies routines that handle the vxbus interrupt controller requirements for the Cavium CN3xxx family of targets. This driver handles interrupt connect, disconnect, enable and disable functions. Functions are published for interrupt connect, disconnect, enable, disable and acknowledge. The file hwconf.c in the BSP directory is used as the configuration source. It defines the device inputs into the interrupt controller and controller output pin crossconnects and destination cpu devices. This file is included by cmdLineBuild.c when DRV_INTCTLR_MIPS_CN3xxx is defined. The macro DRV_INTCTLR_MIPS also needs to be defined to include the registration of the generic MIPS interrupt controller and INCLUDE_INTCTLR_LIB to bring in the vxbus interrupt controller library.
INCLUDE FILES	vxbIntCtlrLib.h hwConf.h vxBus.h vxbPlbLib.h vxbAccess.h

vxbMipsIntCtlr

NAME	vxbMipsIntCtlr – generic interrupt controller for MIPS CPU
ROUTINES	vxbMipsIntCtlrRegister() – register the mips cpu driver mipsIntCtlrShow() – show pDrvCtrl for mipsIntCtlr
DESCRIPTION	This is the VxBus interrupt controller driver for MIPS cpu interrupts. It supplies the standard set of vxBus initialization routines to register the driver and to allow for configuration during the 3 phases of vxBus device init during startup. Note that this driver only uses the first phase of initialization for a vxBus device (vxbMipsIntCtlrInstInit) with the other phases left NULL (see mipsIntCtlrFuncs structure). Beside vxBus device initialization the driver supplies functions to handle interrupt connect, enabling, disabling, disconnecting and acknowledgement. All of these function are exported as methods using the mipsIntCtlr_methods structure which holds their function name and type. This structure is exported when a pointer to it is placed in the interrupt controller's device structure in the pMethods field. The driver's interrupt handler (vxbMipsIntCtlrISR) is registered as a callback function for interrupt handling by loading the address of the vxbMipsIntCtlrISR into _func_vxBusIntHdlr . The global _func_vxBusIntHdl is declared in the file exCALib.s in the MIPS architecture code and its examined by the architecture code during an interrupt to determine if a vxBus interrupt handler is available. If a vxBus ISR isn't configured into _func_vxBusIntHdl , the interrupt will be processed per the legacy interrupt handling rules.

Note that configuring vxbMipsIntCtlrISR as the interrupt handler (loading address of vxbMipsIntCtlrInstInit into _func_vxBusIntHdl) is done as the last step in vxbMipsIntCtlrInstInit (vxBus phase 1 init function).

INCLUDE FILES	<code>vxbIntCtlrLib.h hwConf.h vxBus.h vxbPlbLib.h vxbAccess.h</code>
---------------	---

vxbMipsR4KTimer

NAME	<code>vxbMipsR4KTimer</code> – MIPS R4000 on CPU Timer driver for VxBus
ROUTINES	<code>vxbR4KTimerDrvRegister()</code> – register mips R4K timer driver
DESCRIPTION	<p>This is the vxBus compliant timer driver which implements the functionality specific to MIPs R4000 CPU Based Timer</p> <p>A timer specific data structure (struct r4kTimerData) is maintained within this driver. This is given as <code>pCookie</code> in <code>r4kTimerAllocate</code> and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.</p> <p>The driver implements all the vxBus driver specific initialization routines like <code>r4KTimerInstInit()</code>, <code>r4KTimerInstInit2()</code> and <code>r4KTimerInstConnect()</code>.</p> <p>A variable of type struct r4KTimerData is allocated for a timer device and stored in <code>pInst->pDrvCtrl</code> in <code>r4KTimerInstInit()</code></p> <p><code>r4KTimerInstInit2()</code> hooks an ISR to be used for the timer device and also hooks the decrementeer exception handler.</p> <p>A method for methodId VXB_METHOD_TIMER_FUNC_GET is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.</p> <p>This driver requires the external routines <code>sysCompareSet()</code>, <code>sysCompareGet()</code>, <code>sysCountGet()</code>, and <code>sysCountSet()</code> to be defined.</p> <p>The driver is SMP-Aware. It uses intCpuLocks instead of inlocks for locking of interrupt. Hence it will work on SMP Environment.</p>
INCLUDE FILES	none

vxbMipsSbIntCtlr

NAME	vxbMipsSbIntCtlr – interrupt controller for BCM1480
ROUTINES	vxbMipsSbIntCtlrRegister() – register the mipsIntCtlr driver
DESCRIPTION	<p>This file supplies routines that handle the vxbus interrupt controller requirements for the Broadcom sibyte family of targets. This driver handles interrupt connect, disconnect, enable and disable functions.</p> <p>Functions are published for interrupt connect, disconnect, enable, disable and acknowledge. The file hwconf.c in the BSP directory is used as the configuration source. It defines the device inputs into the interrupt controller and controller output pin crossconnects and destination cpu devices.</p> <p>This file is included by cmdLineBuild.c when DRV_INTCTLR_MIPS_SBE is defined. The macro DRV_INTCTLR_MIPS also needs to be defined to include the registration of the generic MIPS interrupt controller and INCLUDE_INTCTLR_LIB to bring in the vxbus interrupt controller library.</p>
INCLUDE FILES	vxbIntCtlrLib.h hwConf.h vxBus.h vxbPlbLib.h vxbAccess.h

vxbMpApic

NAME	vxbMpApic – VxBus driver for the Intel MP APIC/xAPIC (Advanced PIC)
ROUTINES	vxbMpApicDrvRegister() – register mpApic driver vxbMpApicpDrvCtrlShow() – show pDrvCtrl for template controller vxbMpApicDataShow() – show MP Configuration Data acquired by vxBus vxbMpBiosShow() – show MP configuration table vxbMpBiosIoIntMapShow() – show MP IO interrupt mapping vxbMpBiosLocalIntMapShow() – show MP local interrupt mapping
DESCRIPTION	This is the VxBus driver for the Local APIC Interrupt Driver
INCLUDE FILES	none

vxbNs16550Sio

NAME	vxbNs16550Sio – National Semiconductor 16550 Serial Driver
ROUTINES	ns16550SioRegister() – register ns16550vxb driver
DESCRIPTION	This is the driver for the NS16550 (or compatible) DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a complete modem control capability. A NS16550VXB_CHAN structure is used to describe the serial channel. This data structure is defined in ns16550vxb.h . Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control.
USAGE	The BSP's sysHwInit() routine typically calls sysSerialHwInit() , which creates the NS16550VXB_CHAN structure and initializes all the values in the structure (except the SIO_DRV_FUNCS) before calling ns16550vxbDevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() to single interrupt `ns16550vxbInt'. This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshaking RTS/CTS. The function HUPCL(hang up on last close) is available. When hardware flow control is enabled, the signals RTS and DTR are set TRUE and remain set until a HUPCL is performed.
INCLUDE FILES	..h/sio/ns16550vxb.h

vxbOcteonSio

NAME	vxbOcteonSio – Octeon 16550ish Serial Driver
ROUTINES	octeonSioRegister() – register octeonVxb driver octeonVxbDevProbe() – probe for device presence at specific address octeonVxbDevInit() – initialize an OCTEON channel octeonVxbIntWr() – handle a transmitter interrupt octeonVxbIntRd() – handle a receiver interrupt octeonVxbIntEx() – miscellaneous interrupt processing octeonVxbInt() – interrupt level processing

DESCRIPTION	This is the driver for the OCTEON (or compatible) DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a complete modem control capability.
	A OCTEONVXB_CHAN structure is used to describe the serial channel. This data structure is defined in octeonVxb.h .
	Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control.

USAGE	The BSP's sysHwInit() routine typically calls sysSerialHwInit() , which creates the OCTEONVXB_CHAN structure and initializes all the values in the structure (except the SIO_DRV_FUNCS) before calling octeonVxbDevInit() . The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() , which connects the chips interrupts via intConnect() (either the single interrupt octeonVxbInt or the three interrupts octeonVxbIntWr , octeonVxbIntRd , and octeonVxbIntEx).
	This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshaking RTS/CTS. The function HUPCL(hang up on last close) is available. When hardware flow control is enabled, the signals RTS and DTR are set TRUE and remain set until a HUPCL is performed.
INCLUDE FILES	./h/sio/octeonVxb.h

vxbOpenPicTimer

NAME	vxbOpenPicTimer – Driver for OpenPIC timers
ROUTINES	openPicTimerDrvRegister() – register openPic timer driver
DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality of OpenPIC timers. A driver-specific data structure (struct openPicTimerInstance) is allocated for each timer instance and a pointer to it is stored in pInst->pDrvCtrl during openPicTimerInstInit() . openPicTimerInstance contains an array of (struct openPicTimerData), one for each timer in the instance; openPicTimerAllocate() returns a pointer to the corresponding (struct openPicTimerData) as pCookie for use by the Timer Abstraction Layer/application to communicate with this driver. The driver implements the vxBus driver specific initialization routines openPicTimerInstInit() and openPicTimerInstInit2() , but omits the Connect function because it is not needed.

openPicTimerInstInit2() attempts to connect the driver to the VxBus interrupt mechanism using **vxbIntConnect()**.

A method for methodId **VXB_METHOD_TIMER_FUNC_GET** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to **struct vxbTimerFunctionality** is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.

VxBus resources recognized by this driver are:

```
DRIVER_DATA {  
    DRIVER_NAME "openPicTimer";  
  
    RESOURCE(VXB_REG_BASE, HCF_RES_INT, required, "TFRR", {});  
    /* address of the TFRR */  
  
    RESOURCE("numTimers", HCF_RES_INT, 4, "", {[1,9]});  
    /* number of timers in the group */  
  
    RESOURCE("clkDivisor", HCF_RES_INT, 0, "MBX", {0,8,16,32,64});  
    /*  
     * Clock source and divisor. Zero selects the RTC input.  
     * Non-zero values are divisors applied to the MBX bus  
     * clock, selected from {8, 16, 32, 64}.  
     */  
  
    RESOURCE(VXB_CLK_FREQ, HCF_RES_ADDR, MHZ_33pt33, {}, {});  
    /*  
     * Function to return clock rate, based on the board's MBX  
     * and RTC clock rates, given the "clkDivisor" as a parameter.  
     */  
  
    RESOURCE("instanceName", HCF_RES_STRING, "", "", {});  
    /* instance name, used in constructing individual timer names */  
  
    RESOURCE("#vector#", HCF_RES_INT, none, "", {[0,65535]});  
    /*  
     * Value to put in the "vector" register, for coordination with  
     * the interrupt-controller driver. If not specified, this timer  
     * will not generate interrupts.  
     */
```

```

RESOURCE("#maxClkRate#", HCF_RES_INT, 50000, "", {});
/* maximum interrupt rate to be advertised */
}

```

INCLUDE FILES	none
----------------------	------

vxbParamSys

NAME	vxbParamSys – vxBus parameter system
ROUTINES	<p>vxbInstParamSet() – set driver parameter for specified instance</p> <p>vxbInstByNameFind() – retrieve the VXB_DEVICE_ID for an instance</p> <p>vxbInstParamByNameGet() – retrieve driver parameter value</p> <p>vxbInstParamByIndexGet() – retrieve driver parameter value</p>
DESCRIPTION	<p>This module provides the mechanism for drivers to obtain parameter values. Driver parameters are values which have a useful default value which can be overridden by BSPs and/or middleware.</p> <p>The driver provides a pointer to a table containing the driver default values of each parameter of interest. This is supplied during driver registration. Each parameter consists of name, type, and default value.</p> <p>At any time after the instance is created, middleware is able to specify parameters for the instance by calling the vxbInstParamSet() routine. If middleware tries to set a parameter before the instance is created, it will fail. If middleware tries to set a parameter after the instance is created, it will succeed, but the driver will probably ignore it.</p> <p>The BSP can provide instance override values for any parameter for any instance, using the table formats defined above. This is done at compile time.</p> <p>When the driver needs to fetch a parameter value, it makes a call to vxbInstParamByNameGet() or vxbInstParamByIndexGet().</p> <p>How do they all fit together?</p> <p>First of all, the first time a non-default parameter needs to be set, regardless of what is causing the change from the default value, "the system" will allocate memory and copy the default parameter table into it. It then overwrites the modified value. Subsequent modifications simply update the value fields of table entries. This means that memory is allocated at most once for each instance.</p> <p>If any middleware module calls vxbInstParamSet(), "the system" first checks whether this is the first changed parameter for the instance. If not, then the parameter table is already present, so it just modifies the table value in-place. However, if this is the first time a parameter is being changed for that instance, then it allocates the table as specified above,</p>

searches through the BSP-provided table, and finally sets the parameter as specified by middleware.

When the driver calls `vxbInstParamByNameGet()` or `vxbInstParamByIndexGet()` the first time, it first checks to see whether any parameters were overridden. If so, then it simply reads the value from the table and returns the value. However, if no modified parameters are known, then it reads (once) through the BSP-provided table, if any, to see if there are any BSP-provided parameters. If there are any, then it modifies all the parameters as specified by the BSP-supplied table.

This means that every instance reads through the BSP-supplied table exactly once.

The efficiency of modifying the value of a parameter is the same as the efficiency of reading the value of the parameter by name using the parameter system. That equals the number of driver parameters divided by two, since on average you search halfway through the table before finding a match. The time to read the value of a parameter by index is constant.

How does the system know whether a parameter has been modified?

The instance (current) parameter table is kept in the `VXB_DEVICE` structure. We can refer to this as `pInst->paramList`. The default parameter table will now be part of the driver `DRIVER_REGISTRATION` structure, and is supplied to VxBus when the driver registers. We can refer to this as `pDriver->paramList` or `pInst->pDriver->paramList`.

if `pInst->paramList` is non-NULL, then some parameter has been changed from the default, either by the BSP or by middleware. In this case, ALL parameters are obtained from `pInst->paramList`. If `pInst->paramList` is NULL, then neither the BSP nor middleware has specified any changes from the default value, so the value can be obtained from `pInst->pDriver->paramList`.

----- Another option -----

If the driver can handle modifications to parameters on-the-fly, then the driver should publish the `instParamModify` driver method. When middleware modifies a parameter with a call to `vxbInstParamSet()`, it checks for the driver method. If the driver method is present, `vxbInstParamSet()` first modifies `pInst->paramList`, and then calls the driver method with the `VXB_PARAMETERS` structure as the second argument. (Of course, the first argument is the `VXB_DEVICE_ID`.)

INCLUDE FILES

`vxbParamSys.h`

vx~~b~~Pci

NAME

`vxbPci` – PCI bus support module for vxBus

ROUTINES

pciRegister() – register PCI bus type
pciHcfRecordFind() – find device's HCF pciSlot record
pciDevMatch() – check whether device and driver go together
pciInit() – first-pass bus type initialization
pciInit2() – second-pass bus type initialization
pciConnect() – connect PCI bus type to bus subsystem
pciDeviceAnnounce() – notify the bus subsystem of a device on PCI
pciBusAnnounceDevices() – Notify the bus subsystem of all devices on PCI
pciDevShow() – show information about PCI device
pciConfigLibInit() – initialize the configuration access-method and addresses
vxbPciFindDevice() – find the nth device with the given device & vendor ID
vxbPciFindClass() – find the nth occurrence of a device by PCI class code.
vxbPciDevConfig() – configure a device on a PCI bus
vxbPciConfigBdfPack() – pack parameters for the Configuration Address Register
vxbPciConfigExtCapFind() – find extended capability in ECP linked list
vxbPciConfigInByte() – read one byte from the PCI configuration space
vxbPciConfigInWord() – read one word from the PCI configuration space
vxbPciConfigInLong() – read one longword from the PCI configuration space
vxbPciConfigOutByte() – write one byte to the PCI configuration space
vxbPciConfigOutWord() – write one 16-bit word to the PCI configuration space
vxbPciConfigOutLong() – write one longword to the PCI configuration space
vxbPciConfigModifyLong() – Perform a masked longword register update
vxbPciConfigModifyWord() – Perform a masked longword register update
vxbPciConfigModifyByte() – Perform a masked longword register update
pciSpecialCycle() – generate a special cycle with a message
vxbPciConfigForEachFunc() – check condition on specified bus
vxbPciConfigReset() – disable cards for warm boot
vxbGetPciDevice() – Locates a PCI device
vxbUpdateDeviceInfo() – Update vxBus device Information
vxbPciDeviceAnnounce() – Update vxBus to PCI bus orphans
vxbPciIntLibInit() – initialize the **vxbPciIntLib** module
vxbPciInt() – interrupt handler for shared PCI interrupt.
vxbPciIntConnect() – connect the interrupt handler to the PCI interrupt.
vxbPciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)
vxbPciIntDisconnect2() – disconnect an interrupt handler from the PCI interrupt.
vxbPciAutoConfig() – set standard parameters & initialize PCI
vxbPciAutoConfigLibInit() – initialize PCI autoconfig library
vxbPciAutoCfg() – Automatically configure all nonexcluded PCI headers
vxbPciAutoCfgCtl() – set or get **vxbPciAutoConfigLib** options
vxbPciAutoConfigListShow() – show the devices in the list
vxbPciAutoDevReset() – quiesce a PCI device and reset all writeable status bits
vxbPciAutoBusNumberSet() – set the primary, secondary, and subordinate bus number
vxbPciAutoFuncDisable() – disable a specific PCI function
vxbPciAutoFuncEnable() – perform final configuration and enable a function
vxbPciAutoGetNextClass() – find the next device of specific type from probe list
vxbPciAutoRegConfig() – assign PCI space to a single PCI base address register

vxbPciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge
vxbPciAutoAddrAlign() – align a PCI address and check boundary conditions
vxbPciDeviceShow() – print information about PCI devices
vxbPciHeaderShow() – print a header of the specified PCI device
vxbPciFindDeviceShow() – find a PCI device and display the information
vxbPciFindClassShow() – find a device by 24-bit class code
vxbPciConfigStatusWordShow() – show the decoded value of the status word
vxbPciConfigCmdWordShow() – show the decoded value of the command word
vxbPciConfigFuncShow() – show configuration details about a function
vxbPciConfigTopoShow() – show PCI topology
pciDeviceShow() – print information about PCI devices
pciHeaderShow() – print a header of the specified PCI device
pciFindDeviceShow() – find a PCI device and display the information
pciConfigStatusWordShow() – show the decoded value of the status word
pciConfigCmdWordShow() – show the decoded value of the command word
pciFindClassShow() – find a device by 24-bit class code
pciConfigFuncShow() – show configuration details about a function
pciConfigTopoShow() – show PCI topology
pciConfigEnable() – Set the globalBusCtrlID for the other LEGACY pci
pciConfigInByte() – read one byte from the PCI configuration space
pciConfigInWord() – read one word from the PCI configuration space
pciConfigInLong() – read one longword from the PCI configuration space
pciConfigOutByte() – write one byte to the PCI configuration space
pciConfigOutWord() – write one 16-bit word to the PCI configuration space
pciConfigOutLong() – write one longword to the PCI configuration space
pciConfigForEachFunc() – check condition on specified bus
pciIntConnect() – connect the interrupt handler to the PCI interrupt.
pciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)
pciIntDisconnect2() – disconnect the interrupt handler
pciFindDevice() – find the nth device with the given device & vendor ID
pciConfigExtCapFind() – find extended capability in ECP linked list
pciDevConfig() – configure a device on a PCI bus
pciFindClass() – find the nth occurrence of a device by PCI class code.
pciAutoCfgCtl() – set or get autoConfigLib options

DESCRIPTION	This library contains the support routines for PCI host controllers on the vxBus.
INCLUDE FILES	vxBus.h vxbPciBus.h pciConfigShow.h pciHeaderDefs.h , pciConfigLib.h pciAutoConfigLib.h pciIntLib.h

vxbPciAccess

NAME	vxbPciAccess – vxBus access routines for PCI bus type
ROUTINES	vxbPciBusTypeInit() – initialize the PCI bus type vxbPciAccessCopy() – copy access function pointers
DESCRIPTION	This library contains the support routines for PCI bus access on the vxBus.
INCLUDE FILES	vxBus.h vxbAccess.h

vxbPlb

NAME	vxbPlb – Processor Local Bus support module for vxBus
ROUTINES	plbRegister() – register PLB with bus subsystem plbInit1() – first-stage PLB bus initialization plbInit2() – second-stage PLB bus initialization plbConnect() – third-stage PLB bus initialization plbDevMatch() – check whether device and driver go together plbIntrSet() – set interrupt controller for device plbIntrGet() – Get interrupt controller for device plbIntrShow() – Show interrupt controllers for device's interrupts
DESCRIPTION	This library contains the support routines for PLB host controllers on the vxBus.
INCLUDE FILES	vxBus.h vxbPlbLib.h

vxbPlbAccess

NAME	vxbPlbAccess – vxBus access routines for PLB
ROUTINES	optimizeAccessFunction() – optimize a function based on flags vxbPlbAccessCopy() – copy the access data structure plbAccessInit() – initialize the plb access module
DESCRIPTION	This library contains the support routines for PLB bus access on the vxBus.

INCLUDE FILES vxBus.h vxbAccess.h vxbArchAccess.h

1

vxbPpcDecTimer

NAME vxbPpcDecTimer – timer driver for the PowerPC decrementer**ROUTINES** ppcDecTimerDrvRegister() – register ppcDec timer driver**DESCRIPTION** This is the vxBus compliant timer driver which implements the functionality specific to the PowerPC decrementer.

A timer specific data structure (struct ppcDecTimerData), maintained within this driver, is allocated for the timer device. A pointer to the allocated structure is stored in **pInst->pDrvCtrl** during **ppcDecTimerInstInit()** and is also returned as **pCookie** in **ppcDecTimerAllocate()** for use by the Timer Abstraction Layer/application to communicate with this driver.

The driver implements all the vxBus driver specific initialization routines like **ppcDecTimerInstInit()**, **ppcDecTimerInstInit2()** and **ppcDecTimerInstConnect()**.

ppcDecTimerInstInit2() attempts to connect the driver to the VxBus interrupt mechanism using **vxbIntConnect()**. If this fails, presumably because the BSP has not been adapted to support the VxBus interrupt mechanism, it attaches the ISR to the decrementer vector via a stub function. A global pointer to the **struct ppcDecTimerData *** is maintained so that the stub can pass the structure's address to the VxBus-compliant ISR.

A method for methodId **VXB_METHOD_TIMER_FUNC_GET** is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to **struct vxbTimerFunctionality** is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.

INCLUDE FILES

none

vxbPpcQuiccTimer

NAME vxbPpcQuiccTimer – timer driver for quiccTimer**ROUTINES** quiccTimerDrvRegister() – register coldFire timer driver

DESCRIPTION	This is the vxBus compliant timer driver which implements the functionality specific to quiccTimer (General purpose timers in PPC BSP). Not necessarily all the PPC BSP's will have these general purpose timers.
	A timer specific data structure (struct quiccTimerData) is maintained within this driver. This is given as pCookie in quiccTimerAllocate and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.
	The driver implements all the vxBus driver specific initialization routines like quiccTimerInstInit (), quiccTimerInstInit2 () and quiccTimerInstConnect ().
	A variable of type struct quiccTimerData is allocated for a timer device and stored in pInst->pDrvCtrl in quiccTimerInstInit ()
	quiccTimerInstInit2 () hooks an ISR to be used for the timer device.
	A method for methodId VXB_METHOD_TIMER_FUNC_GET is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.
INCLUDE FILES	none

vxbPrimeCellSio

NAME	vxbPrimeCellSio – ARM AMBA UART <i>tty</i> driver for VxBus
ROUTINES	vxbPrimeCellSioRegister() – register vxbAmbaSio driver
DESCRIPTION	<p>This is the device driver for the Advanced RISC Machines (ARM) AMBA UART. This is a generic design of UART used within a number of chips containing (or for use with) ARM CPUs</p> <p>This design contains a universal asynchronous receiver/transmitter, a baud-rate generator, and an InfraRed Data Association (IrDa) Serial InfraRed (SiR) protocol encoder. The SiR encoder is not supported by this driver. The UART contains two 16-entry deep FIFOs for receive and transmit: if a framing, overrun or parity error occurs during reception, the appropriate error bits are stored in the receive FIFO along with the received data. The FIFOs can be programmed to be one byte deep only, like a conventional UART with double buffering, but the only mode of operation supported is with the FIFOs enabled.</p> <p>The UART design does not support the modem control output signals: DTR, RI and RTS. Moreover, the implementation in the 21285 chip does not support the modem control inputs: DCD, CTS and DSR.</p>

The UART design can generate four interrupts: Rx, Tx, modem status change and a UART disabled interrupt (which is asserted when a start bit is detected on the receive line when the UART is disabled). The implementation in the 21285 chip has only two interrupts: Rx and Tx, but the Rx interrupt is a combination of the normal Rx interrupt status and the UART disabled interrupt status.

Only asynchronous serial operation is supported by the UART which supports 5 to 8 bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity.

The exact baud rates supported by this driver will depend on the crystal fitted (and consequently the input clock to the baud-rate generator), but in general, baud rates from about 300 to about 115200 are possible.

In theory, any number of UART channels could be implemented within a chip. This driver has been designed to cope with an arbitrary number of channels, but at the time of writing, has only ever been tested with one channel.

TARGET-SPECIFIC PARAMETERS

The parameters are provided through registration defined in **hwconf.c** in the target BSP.

regBase

ARM PrimeCellII UART register base address. Specify serial mode register address.
The parameter type is **HCF_RES_INT**.

irq

Interrupt vector number for the ARM PrimeCellII UART single interrupt source. The parameter type is **HCF_RES_INT**.

irqLevel

Interrupt number for the ARM PrimeCellII UART single interrupt source. The parameter type is **HCF_RES_INT**.

pclkFreq

Clock frequency for ARM PrimeCellII UART in Hz. The parameter type is **HCF_RES_INT**.

An **VXB_AMBA_CHAN** data structure is used to describe each channel, this structure is described in **h/drv/sio/vxbAmbsio.h**.

CALLBACKS

Servicing a "transmitter ready" interrupt involves making a callback to a higher level library in order to get a character to transmit. By default, this driver installs dummy callback routines which do nothing. A higher layer library that wants to use this driver (e.g. **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK** ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

MODES

This driver supports both polled and interrupt modes.

INCLUDE FILES **src/hwif/h/sio/vxbPrimeCellSio.h sioLib.h**

SEE ALSO *Advanced RISC Machines AMBA UART (AP13) Data Sheet, ARM PrimeCell UART PL011 Technical Reference manual*

vxbQeIntCtlr

NAME **vxbQeIntCtlr** – Quicc Engine interrupt controller driver

ROUTINES **vxbQeIntCtlrRegister()** – register qeIntCtlr driver
vxbQeIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller
qeAnd32() – Read then Write 32 bit register usign and mask
qeOr32() – Read then Write 32 bit register using or mask
vxbSysQeIntEnable() – enable the indicated interrupt
vxbSysQeIntDisable() – Disable one of the Level or IRQ interrupts into the SIU
qelVecToInum() – get the relevant interrupt number

DESCRIPTION The quiccEngine interrupt controller is a subordinate interrupt controller to the main CPU PIC. The CPU PIC receives interrupts from the quiccEngine and it calls this driver to service them. The interrupts are attached to the PIC and then interrupts from quicceengine devices are signalled to the quiccEngine and then over to the PIC. The knowledge of the PIC and the exact interrupt that is connected is in the hwconf file of the BSP. It is assumed that two outputs exist from the quicc engine that connect to the PIC hence the two calls to vxbIntConnect/Enable.

INCLUDE FILES **vxbQeIntCtlr.h**

vxbQuiccIntCtlr

NAME **vxbQuiccIntCtlr** – Motorola MPC 83XX interrupt controller driver

ROUTINES **vxbQuiccIntCtlrRegister()** – register quiccIntCtlr driver
vxbQuiccIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller
vxbSysQuiccInit() – initialize the interrupt manager for the PowerPC 83XX
quiccIntrInit() – initialize the interrupt manager for the PowerPC 83XX
quiccAnd32() – Read then Write 32 bit register usign and mask
quiccOr32() – Read then Write 32 bit register using or mask
vxbSysQuiccIntEnable() – enable the indicated interrupt
vxbSysQuiccIntDisable() – Disable one of the Level or IRQ interrupts into the SIU

quiccIvecToInum() – get the relevant interrupt number

DESCRIPTION	The PowerPC 83XX CPU is divided in three units: PowerPC core, System Interface Unit (SIU) and Quicc Engine (QE). The PowerPC core accepts only one external interrupt exception (vector 0x500). The SIU provides an interrupt controller which provides 128 interrupt sources. The Interrupt controller is connected to the PowerPC core external interrupt. This library provides the routines to manage this interrupt controllers.
	quiccIntrInit() connects the default demultiplexer, quiccIntrDeMux() , to the external interrupt vector and initializes a table containing a function pointer and an associated parameter for each interrupt source. quiccIntrInit() is called by via vxBus InstInit routine.
	The default demultiplexer, quiccIntrDeMux() detects which peripheral or controller has raised an interrupt and calls the associated routine with its parameter.

INCLUDE FILES **vxbQuiccIntCtr.h**

vxbRapidIO

NAME	vxbRapidIO -- RapidIO bus support module
ROUTINES	rapidIoRegister() – register RapidIO bus type rioDevMatch() – check whether device and driver go together rioInit1() – first-pass bus type initialization rioInit2() – second-pass bus type initialization rioConnect() – connect RapidIO bus type to bus subsystem vxbRapidIOBusTypeInit() – initialize RapidIO bus type
DESCRIPTION	This library contains the support routines for the RapidIO bus architecture.
INCLUDE FILES	vxBus.h vxbRapidIO.h

vxbRapidIOCfgTable

NAME	vxbRapidIOCfgTable – RapidIO table-based device enumeration
ROUTINES	vxbRapidIOTableOverride() – RapidIO Table-based override function vxbRapidIOCfg() – RapidIO Table-based device enumeration
DESCRIPTION	This library contains the RapidIO table-based device enumeration functionality.

INCLUDE FILES	vxWorks.h
----------------------	-----------

vxbSb1DuartSio

NAME	vxbSb1DuartSio – BCM1250/1480 serial communications driver
ROUTINES	vxbSb1DuartSioRegister() – register vxbSb1DuartSio driver
DESCRIPTION	<p>This is the driver for the Broadcom BCM1250/1480 DUART. This device includes two universal asynchronous receiver/transmitters, and baud rate generators. This device shares similarities with the m68681 Duart. However it is different enough to justify a separate device driver. This is because the BCM1250/1480 DUART has been enhanced to operate more like two independent UARTs instead of a single DUART. Where one channel may be used by first processor and the other channel used by the second processor. This is possible because, unlike the m68681, the BCM1250/1480 Duart asserts a separate interrupt line for each channel. Additional enhancements allow each channel is driven by a separate baud rate generator which is capable of run each channel at up to 1 Megabaud. In summary, this driver module provides control over each of the two serial channels and the baud-rate generators as two independent devices.</p>

A SB1_DUART_CHAN structure is used to describe the chip. The SB1_DUART_CHAN structure is defined in **vxbSb1DuartSio.h**.

Although, the BCM1250/1480 DUART is capable of running in synchronous mode (HDLC), only asynchronous serial operation is supported by this driver.

The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and no control. These default settings can be overridden on a channel-by-channel basis by setting the **options** and **baudRate** fields to the desired values in the BSP file **hwconf.c**. See **sioLib.h** for option values. The defaults for the module can be changed by redefining the macros **SB1_DUART_DEFAULT_OPTIONS** and **SB1_DUART_DEFAULT_BAUD** and recompiling this driver.

SPECIAL CONSIDERATIONS

The CLOCAL hardware option presumes that OP0 and OP1 output bits are wired to the CTS outputs for channel 0 and channel 1 respectively. If not wired correctly, then the user must not select the CLOCAL option.

This driver does not manipulate the output port or its configuration register in any way. If the user selects the CLOCAL option, then the output port bit must be wired correctly or the hardware flow control will not function correctly.

INCLUDE FILES	src/hwif/h/sio/vxbSb1DuartSio.h
----------------------	---------------------------------

vxbSb1Timer

NAME	vxbSb1Timer – Sb1Timer driver for VxBus
ROUTINES	vxbSb1TimerDrvRegister() – register sb1 timer driver
DESCRIPTION	<p>This is the vxBus compliant timer driver which implements the functionality specific to timers on BCM Core.</p> <p>The sb1 timer has 23-bit counter and clock upto a frequency of 1MHz. Hence it allows a maximum count of 8388608. It can operate in two modes:</p> <ul style="list-style-type: none">i) One shot mode - The timer counts down to 0 and stops.ii) Repeated mode - The timer counts done to 0 and gets automatically reloaded <p>A timer specific data structure (struct sb1TimerData) is maintained within this driver. This is given as pCookie in sb1TimerAllocate and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.</p> <p>The driver implements all the vxBus driver specific initialization routines like sb1TimerInstInit (), sb1TimerInstInit2 () and sb1TimerInstConnect ().</p> <p>A variable of type struct sb1TimerData is allocated for a timer device and stored in pInst->pDrvCtrl in sb1TimerInstInit ()</p> <p>sb1TimerInstInit2 () hooks an ISR to be used for the timer device and also hooks the decremener exception handler.</p> <p>A method for methodId VXB_METHOD_TIMER_FUNC_GET is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.</p> <p>The driver is SMP-Aware. It uses ISR-Task Callable spinlock instead of inlocks for locking of interrupt. Hence it will work on SMP Environment.</p>
INCLUDE FILES	none

vxbSh7700Timer

NAME	vxbSh7700Timer – SH77xx on-chip Timer driver for VxBus
ROUTINES	sh7700TimerDrvRegister() – register sh7700 timer driver

DESCRIPTION	<p>This is the vxBus compliant timer driver which implements the functionality specific to SH77xx Timer</p> <p>A timer specific data structure (struct sh7700TimerData) is maintained within this driver. This is given as pCookie in sh7700TimerAllocate and can be used from there-on by the Timer Abstraction Layer/application to communicate with this driver.</p> <p>The driver implements all the vxBus driver specific initialization routines like sh7700TimerInstInit (), sh7700TimerInstInit2 () and sh7700TimerInstConnect ().</p> <p>A variable of type struct sh7700TimerData is allocated for a timer device and stored in pInst->pDrvCtrl in sh7700TimerInstInit ()</p> <p>sh7700TimerInstInit2 () hooks an ISR to be used for the timer device and also hooks the decremener exception handler.</p> <p>A method for methodId vxbTimerFuncGet_desc is implemented in this driver and is used by the Timer Abstraction Layer/application to retrieve the characteristics of a timer device. A pointer to struct vxbTimerFunctionality is allocated by the Timer Abstraction Layer/application and pointer to this structure is given as the parameter to this method. The timer driver populates this data structure with the features of the timer device.</p>
TARGET-SPECIFIC PARAMETERS	
	<p>The parameters are provided through sh7700TimerDev registration defined in hwconf.c in the target BSP.</p>
<i>regBase</i>	SH77xx Timer register base address. Specify the TCOR (timer constant register) address. The parameter type is HCF_RES_INT .
<i>pclkFreq</i>	Peripheral clock frequency in Hz. The parameter type is HCF_RES_INT .
<i>irq</i>	Interrupt vector number for the timer interrupt source. The parameter type is HCF_RES_INT .
<i>irqLevel</i>	Interrupt number for the timer interrupt source. The parameter type is HCF_RES_INT .
<i>clkRateMin</i>	Specify the minimum period timer. Only required by the system and AUX timer which is the first and second sh7700TimerDev resources. The parameter type is HCF_RES_INT .
<i>clkRateMax</i>	Specify the maximum period timer. Only required by the system and AUX timer which is the first and second sh7700TimerDev resources. The parameter type is HCF_RES_INT .
INCLUDE FILES	none

vxbSh77xxPci

NAME	<i>vxbSh77xxPci</i> – sh77xx PCI bus controller VxBus driver
ROUTINES	<i>sh77xxPciRegister()</i> – register sh77xxPci driver <i>sh77xxPcipDrvCtrlShow()</i> – show pDrvCtrl for sh77xxPci bus controller
DESCRIPTION	<p>This is the VxBus driver for the PCI bus controller on the SH7751 / SH7780 CPUs.</p> <p>This driver works on PLB bus, and gets configuration information from the hwconf.c file entry. This driver supports only 1-1 address mappings, so it uses the PCI addresses and sizes specified in hwconf as the local addresses.</p>
TARGET-SPECIFIC PARAMETERS	
The parameters are provided through <i>sh77xxPci</i> registration defined in hwconf.c in the target BSP.	
<p><i>regBase</i> PCI controller configuration register base address. The parameter type is HCF_RES_INT.</p>	
<p><i>mem32Addr</i> Specifies the 32-bit prefetchable memory pool base address. Normally, this is given by the BSP constant PCI_MEM_ADRS. It can be set with the pciAutoCfgCtl() command PCI_MEM32_LOC_SET. The parameter type is HCF_RES_ADDR.</p>	
<p><i>mem32Size</i> Specifies the 32-bit prefetchable memory pool size. Normally, this is given by the BSP constant PCI_MEM_SIZE. It can be set with the pciAutoCfgCtl() command PCI_MEM32_SIZE_SET. The parameter type is HCF_RES_INT.</p>	
<p><i>memIo32Addr</i> Specifies the 32-bit non-prefetchable memory pool base address. Normally, this is given by the BSP constant PCI_MEMIO_ADRS. It can be set with the pciAutoCfgCtl() command PCI_MEMIO32_LOC_SET. The parameter type is HCF_RES_ADDR.</p>	
<p><i>memIo32Size</i> Specifies the 32-bit non-prefetchable memory pool size. Normally, this is given by the BSP constant PCI_MEMIO_SIZE. It can be set with the pciAutoCfgCtl() command PCI_MEMIO32_SIZE_SET. The parameter type is HCF_RES_INT.</p>	
<p><i>io32Addr</i> Specifies the 32-bit I/O pool base address. Normally, this is given by the BSP constant PCI_IO_ADRS. It can be set with the pciAutoCfgCtl() command PCI_IO32_LOC_SET. The parameter type is HCF_RES_ADDR.</p>	

io32Size

Specifies the 32-bit I/O pool size. Normally, this is given by the BSP constant **PCI_IO_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO32_SIZE_SET**. The parameter type is **HCF_RES_INT**.

io16Addr

Specifies the 16-bit I/O pool base address. Normally, this is given by the BSP constant **PCI_ISA_IO_ADDR**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO16_LOC_SET**. The parameter type is **HCF_RES_ADDR**.

io16Size

Specifies the 16-bit I/O pool size. Normally, this is given by the BSP constant **PCI_ISA_IO_SIZE**. It can be set with the **pciAutoCfgCtl()** command **PCI_IO16_SIZE_SET**. The parameter type is **HCF_RES_INT**.

maxBusSet

Specifies the highest sub-bus number. This parameter is optional, the default is zero. The parameter type is **HCF_RES_INT**.

cacheSize

Specifies the PCI cache line size. This parameter is optional, the default is zero. The parameter type is **HCF_RES_INT**.

maxLatAllSet

Specifies the maximum latency. This parameter is optional, the default is zero. The parameter type is **HCF_RES_INT**.

autoIntRouteSet

Can be set to **TRUE** to configure **pciAutoConfig()** only to call the BSP interrupt routing routine for devices on bus number 0. Setting **autoIntRoutine** to **FALSE** will configure **pciAutoConfig()** to call the BSP interrupt routing routine for every device regardless of the bus on which the device resides. This parameter is optional, the default is **FALSE**. The parameter type is **HCF_RES_INT**.

msgLogSet

The argument specifies a routine which is called to print warning or error messages from **pciAutoConfigLib** if **logMsg()** has not been initialized at the time **pciAutoConfigLib** is used. The specified routine must accept arguments in the same format as **logMsg()**, but it does not necessarily need to print the actual message. An example of this routine is presented below, which saves the message into a safe memory space and turns on an LED. This command is useful for BSPs which call **pciAutoCfg()** before message logging is enabled. Note that after **logMsg()** is configured, output goes to **logMsg()** even if this command has been called. Default = **NULL**.

```
/* sample PCI_MSG_LOG_SET function */
int pciLogMsg(char *fmt,int a1,int a2,int a3,int a4,int a5,int a6)
{
    int charsPrinted;
```

```
    sysLedOn(4);
    charsPrinted = sprintf (sysExcMsg, fmt, a1, a2, a3, a4, a5, a6);
    sysExcMsg += charsPrinted;
    return (charsPrinted);
}
```

This parameter is optional. The parameter type is **HCF_RES_ADDR**.

maxLatencyFuncSet

This routine is called for each function present on the bus when discovery takes place. The routine must accept four arguments, specifying bus, device, function, and a user-supplied argument of type **void ***. See *maxLatencyArgSet*. The routine should return a **UINT8** value, which will be put into the **MAX_LAT** field of the header structure. The user supplied routine must return a valid value each time it is called. There is no mechanism for any **ERROR** condition, but a default value can be returned in such a case. Default = **NULL**. This parameter is optional. The parameter type is **HCF_RES_ADDR**.

maxLatencyArgSet

When the routine specified in *maxLatencyFuncSet* is called, this will be passed to it as the fourth argument. This parameter is optional. The parameter type is **HCF_RES_ADDR**.

includeFuncSet

The device inclusion routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET** **pciAutoCfgCtl()** command:

```
pciAutoCfgCtl(pSystem, PCI_INCLUDE_FUNC_SET, sysPciAutoconfigInclude);
```

This optional user-supplied routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI vendorID and deviceID of the function. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigInclude
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This optional user-specified routine is called by PCI AutoConfig for each and every function encountered in the scan phase. The BSP developer may use any combination of the input data to ascertain whether a device is to be excluded from the autoconfig process. The exclusion routine then returns **ERROR** if a device is to be excluded, and **OK** if a device is to be included in the autoconfiguration process.

Note that PCI-to-PCI Bridges may not be excluded, regardless of the value returned by the BSP device inclusion routine. The return value is ignored for PCI-to-PCI bridges.

The Bridge device will be always be configured with proper primary, secondary, and subordinate bus numbers in the device scanning phase and proper I/O and Memory aperture settings in the configuration phase of autoconfig regardless of the value returned by the BSP device inclusion routine.

intAssignFuncSet

The interrupt assignment routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET** **pciAutoCfgCtl()** command:

```
pciAutoCfgCtl(pCookie, PCI_INT_ASSIGN_FUNC_SET, sysPciAutoconfigIntrAssign);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and an 8-bit quantity containing the contents of the interrupt Pin register from the PCI configuration header of the device under consideration. The interrupt pin register specifies which of the four PCI Interrupt request lines available are connected. The function prototype for this function is shown below:

```
UCHAR sysPciAutoconfigIntrAssign
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UCHAR pin
);
```

This routine may use any combination of these data to ascertain the interrupt level. This value is returned from the function, and is programmed into the interrupt line register of the function's PCI configuration header. In this manner, device drivers may subsequently read this register in order to calculate the appropriate interrupt vector which to attach an interrupt service routine. The parameter type is **HCF_RES_ADDR**.

bridgePreConfigFuncSet

The bridge pre-configuration pass initialization routine is provided so that the BSP Developer can initialize a bridge device prior to the configuration pass on the bus that the bridge implements. This routine is specified by calling **pciAutoCfgCtl()** with the **PCI_BRIDGE_PRE_CONFIG_FUNC_SET** command: The parameter type is **HCF_RES_ADDR**.

bridgePostConfigFuncSet

The bridge post-configuration pass initialization routine is provided so that the BSP Developer can initialize the bridge device after the bus that the bridge implements has been enumerated. This routine is specified by calling **pciAutoCfgCtl()** with the **PCI_BRIDGE_POST_CONFIG_FUNC_SET** command The parameter type is **HCF_RES_ADDR**.

rollcallFuncSet

The specified routine will be configured as a roll call routine.

If a roll call routine has been configured, before any configuration is actually done, the roll call routine is called repeatedly until it returns **TRUE**. A return value of **TRUE** indicates that either (1) the specified number and type of devices named in the roll call list have been found during PCI bus enumeration or (2) the timeout has expired without finding all of the specified number and type of devices. In either case, it is assumed that all of the PCI devices which are going to appear on the busses have appeared and we can proceed with PCI bus configuration. The parameter type is **HCF_RES_ADDR**.

fbbEnable

Enable and disable the functions which check Fast Back To Back functionality. **PCI_FBB_UPDATE** is for use with dynamic/HA applications. It first disables FBB on all functions, then enables FBB on all functions, if appropriate. In HA applications, it should be called any time a card is added or removed. The BOOL pointed to by pArg for **PCI_FBB_ENABLE** and **PCI_FBB_UPDATE** is set to TRUE if all cards allow FBB functionality and FALSE if either any card does not allow FBB functionality or if FBB is disabled. The BOOL pointed to by pArg for **PCI_FBB_STATUS_GET** is set to TRUE if **PCI_FBB_ENABLE** has been called and FBB is enabled, even if FBB is not activated on any card. It is set to FALSE otherwise. This parameter is optional, the default is FALSE.

NOTE: In the current implementation, FBB is enabled or disabled on the entire bus. If any device anywhere on the bus cannot support FBB, then it is not enabled, even if specific sub-busses could support it. The parameter type is **HCF_RES_INT**.

autoConfig

Enable and disable Auto Configuration. This parameter is optional, the default is FALSE. The parameter type is **HCF_RES_INT**.

pciConfigMechanism

Specify the PCI Configuration mechanisms. This parameter is optional, the default is **PCI_MECHANISM_0** (0). Only **PCI_MECHANISM_0** is supported by this driver, so far. The parameter type is **HCF_RES_INT**.

Non **vxbPciAutoConfig()** values:

lclMemAddr

Specify the local memory address on the BSP. The parameter type is **HCF_RES_ADDR**.

lclMemSize

Specify the local memory size on the BSP. The parameter type is **HCF_RES_INT**.

mstrMemIoLcl

Specify the local PCI master memory IO address. The parameter type is **HCF_RES_ADDR**.

mstrPcicBus

Specify the PCI master bus address. The parameter type is **HCF_RES_ADDR**.

mstrIoBus

Specify the PCI master bus IO address. The parameter type is **HCF_RES_ADDR**.

mstrIoLcl

Specify the PCI IO local address. The parameter type is **HCF_RES_ADDR**.

picIvecBase

Specify the PCI controller vector base address. The parameter type is **HCF_RES_INT**.

intr0

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr1

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr2

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr3

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr4

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr5

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr6

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr7

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr8

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr9

Interrupt vector number for the PCI controller. A temporary interrupt service routine is connected to the interrupt vector. This parameter is optional. The parameter type is **HCF_RES_INT**.

intr0Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr1Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr2Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr3Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr4Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr5Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr6Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr7Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr8Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

intr9Level

Interrupt number for the PCI controller. This parameter is optional. If the interrupt number is not set, the interrupt source will not be enabled through **vxbIntEnable()**. The parameter type is **HCF_RES_INT**.

bridgeInitFuncSet

Specify the BSP bridge initialization routine that needs to be done prior to the PCI Auto Configuration. The parameter type is **HCF_RES_ADDR**.

INCLUDE FILES	none
----------------------	------

vxbShScifSio

NAME	vxbShScifSio – Renesas SuperH SCIF driver for VxBus
-------------	--

ROUTINES	shScifSioRegister() – register shScif driver
-----------------	---

DESCRIPTION	This is the VxBus compliant driver for the Renesas SH series on-chip SCIF (Serial Communication Interface with FIFO). It uses the SCIF in asynchronous mode only.
--------------------	---

TARGET-SPECIFIC PARAMETERS

The parameters are provided through **scifSioDev** registration defined in **hwconf.c** in the target BSP.

regBase

SCIF register base address. Specify serial mode register address. The parameter type is **HCF_RES_INT**.

pclkFreq

Peripheral clock frequency in Hz. The parameter type is **HCF_RES_INT**.

intr0

Interrupt vector number for the SCIF Error. This parameter is ignored if *irq* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr1

Interrupt vector number for the SCIF Reception. This parameter is ignored if *irq* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr2

Interrupt vector number for the SCIF Transmit. This parameter is ignored if *irq* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr3

Interrupt vector number for the SCIF Break. This parameter is ignored if *irq* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr0Level

Interrupt number for the SCIF Error. This parameter is ignored if *irqLevel* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr1Level

Interrupt number for the SCIF Reception. This parameter is ignored if *irqLevel* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr2Level

Interrupt number for the SCIF Transmit. This parameter is ignored if *irqLevel* parameter is also specified. The parameter type is **HCF_RES_INT**.

intr3Level

Interrupt number for the SCIF Break. This parameter is ignored if *irqLevel* parameter is also specified. The parameter type is **HCF_RES_INT**.

irq

Interrupt vector number for the SCIF single interrupt source. This parameter must not be specified if *intr0*, *intr1*, *intr2* and *intr3* parameters are provided. The parameter type is **HCF_RES_INT**.

irqLevel

Interrupt number for the SCIF single interrupt source. This parameter must not be specified if *intr0Level*, *intr1Level*, *intr2Level* and *intr3Level* parameters are provided. The parameter type is **HCF_RES_INT**.

rfdr

Reception FIFO data count register offset. This parameter is optional, the default value is 0x1C. The parameter type is **HCF_RES_INT**.

fifoLen

Maximum FIFO length. This parameter is optional, the default value is 16. The parameter type is **HCF_RES_INT**.

fcrTrg

Bit [4-7] setting on the FIFO control register. This parameter is optional, the default value is 0xf0. The parameter type is **HCF_RES_INT**.

INCLUDE FILES `..//h/sio/vxbShScifSio.h sioLib.h`

vxbsShow

NAME **vxbsShow** – bus subsystem source file

ROUTINES **vxBusShow()** – show vxBus subsystem
vxvBusListPrint() – Show bus topology
vxvTopoShow() – Show bus topology
vxvPresStructShow() – Show bus information
vxvDevStructShow() – Show device information
vxvDevAccessShow() – Show bus access methods

vxbDevPathShow() – Show bus hierarchy

DESCRIPTION	This library provides the interfaces which can be used for displaying information about the elements in the vxBus subsystem.
INCLUDE FILES	vxBus.h

vxbSmEnd

NAME	vxbSmEnd – BSP configuration module for shared memory END driver
ROUTINES	smEndRegister() – register smEnd driver sysSmEndLoad() – load the shared memory END driver
DESCRIPTION	This is the WRS-supplied configuration module for the VxWorks shared memory END driver. It performs the dynamic parameterization of the smEnd driver. This technique of just-in-time parameterization allows driver parameter values to be declared as something other than static strings.
INCLUDE FILES	end.h smEnd.h

vxbSmSupport

NAME	vxbSmSupport – support for shared memory
ROUTINES	sysBusTasClear() – clear a reservation for a particular address sysBusTasClearHelper() – locate and clear a reservation for a particular address sysBusTas() – establish a reservation for a particular address sysBusTasHelper() – locate the correct device to do TAS sysBusIntGen() – call interrupt generator function hook sysBusIntGenHelp() – locate the correct call to generate an interrupt sysBusIntAck() – acknowledge an interrupt sysBusIntAckHelp() – locate the correct call to acknowledge an interrupt sysMailboxConnect() – TBD sysMailboxEnable() – TBD
DESCRIPTION	This library implements the hardware interface memory management,
INCLUDE FILES	vxbSmSupport.h

vxbsmcFdc37x

NAME	vxbsmcFdc37x – SMsC FDC37x Ultra I/O Configuration Driver for VxBus
ROUTINES	vxbsmcFdc37xRegister() – register smcFdc37x driver
DESCRIPTION	This module initializes and configures an SMsC FDC37x UltraIO chip which supports floppy disk controller, IDE controller, serial port, parallel port and so on.

TARGET-SPECIFIC PARAMETERS

The parameters are provided through **scifSioDev** registration defined in **hwconf.c** in the target BSP.

regBase

SMsC FDC37x register base address. The parameter type is **HCF_RES_INT**.

regInterval

Register interval size for the SMsC FDC37x controller. This parameter is optional, the default is 1. The parameter type is **HCF_RES_INT**.

cfgDevNum

Logical Device Number. The parameter type is **HCF_RES_INT**.

cfgBaseAdrs

Base I/O address to be configured for the device. This parameter is optional. Not all the device needs this parameter. The parameter type is **HCF_RES_INT**.

cfgIrq

IRQ select to be configured for the device. This parameter is optional. Not all the device needs this parameter. The parameter type is **HCF_RES_INT**.

cfgIrq2

IRQ2 select to be configured for the device. This parameter is optional. Not all the device needs this parameter. The parameter type is **HCF_RES_INT**.

cfgMode

Port mode to be configured for the device. This parameter is optional. Not all the device needs this parameter. The parameter type is **HCF_RES_INT**.

INCLUDE FILES

none

vxbsysClkLib

NAME	vxbsysClkLib – vxBus system clock library
-------------	---

ROUTINES	vxbSysClkLibInit() – initialize the system clock library sysClkHandleGet() – get the timer handle for system clock sysClkConnect() – connect a routine to the system clock interrupt sysClkDisable() – turn off system clock interrupts sysClkEnable() – turn on system clock interrupts sysClkRateGet() – get the system clock rate sysClkRateSet() – set the system clock rate vxbSysClkShow() – show the system clock information
DESCRIPTION	This module implements the system clock library interfaces.
INCLUDE FILES	none

vxbTimerLib

NAME	vxbTimerLib – vxBus timer interfaces module
ROUTINES	vxbTimerLibInit() – initialize the timer library vxbTimerFeaturesGet() – get the timer features vxbTimerAlloc() – allocate timer for the requested characteristics vxbTimerRelease() – release the allocated timer
DESCRIPTION	This library provides the timer specific interfaces
INCLUDE FILES	none

vxbTimerStub

NAME	vxbTimerStub – vxBus timer stub file
ROUTINES	vxbTimerStubInit() – initialization function for the timer stub
DESCRIPTION	This file implements the stub functions for the vxbus delay functions. This file is included when the vxbus timer support is not available for the BSP.
INCLUDE FILES	none

vxbTimestampLib

NAME	vxbTimestampLib – vxBus Timestamp library
ROUTINES	<code>vxbTimestampLibInit()</code> – initialize the timestamp library <code>sysTimestampHandleGet()</code> – get the timer handle for timestamp timer <code>sysTimestampConnect()</code> – connect a user routine to the timestamp timer interrupt <code>sysTimestampEnable()</code> – initialize and enable the timestamp timer <code>sysTimestampDisable()</code> – disable the timestamp timer <code>sysTimestampPeriod()</code> – get the timestamp timer period <code>sysTimestampFreq()</code> – get the timestamp timer clock frequency <code>sysTimestamp()</code> – get the timestamp timer tick count <code>sysTimestampLock()</code> – get the timestamp timer tick count <code>vxbTimestampShow()</code> – show the timestamp information <code>vxbTimestampCookieGet()</code> – to get the cookie information
DESCRIPTION	This module implements the timestamp timer specific interfaces.
INCLUDE FILES	none

vxbVxSimIpi

NAME	vxbVxSimIpi – VxBus driver for Linux simulator IPI management
ROUTINES	<code>vxbVxSimIntCtlrRegister()</code> – register vxbVxSimIntCtlr driver
DESCRIPTION	This driver provides inter-processor interrupts for the SMP version of VxSim.
INCLUDE FILES	none

wancomEnd

NAME	wancomEnd – END style Marvell/Galileo GT642xx Ethernet network interface driver
ROUTINES	<code>wancomEndLoad()</code> – initialize the driver and device <code>wancomEndDbg()</code> – Print pDrvCtrl information regarding Tx ring and Rx queue desc.

DESCRIPTION	This module implements an Galileo Ethernet network interface driver. This is a fast Ethernet IEEE 802.3 10Base-T and 100Base-T compatible.
	The Galileo establishes a shared memory communication system with the CPU, which is divided into two parts: the Transmit Frame Area (TFA) and the Receive Frame Area (RFA).
	The TFA consists of a linked list of frame descriptors through which packet are transmitted. The linked list is in a form of a ring.
	The RFA is a linked list of receive frame descriptors through which packet receive is performed. The linked list is in a form of queue. The RFA also contains two Receive Buffers Area. One area is used for clusters (See netBuflib) and the other one is used for Galileo device receive buffers. This is done as we must keep receive buffers at 64bit alignment !
BOARD LAYOUT	This device is on-board. No jumpering diagram is necessary.
EXTERNAL INTERFACE	<p>The driver provides the standard external interface, wancomEndLoad(), which takes a string of colon separated parameters. The parameters should be specified in hexadecimal, optionally preceded by "0x" or a minus sign "-".</p> <p>The parameter string is parsed using strtok_r() and each parameter is converted from a string representation to binary by a call to strtoul(parameter, NULL, 16).</p> <p>The format of the parameter string is: <i>"memBase:memSize:nCFDs:nRFDs:flags"</i></p>
TARGET-SPECIFIC PARAMETERS	
<i>memBase</i>	This parameter is passed to the driver via wancomEndLoad() . This parameter can be used to specify an explicit memory region for use by the Galileo device. This should be done on targets that restrict the Galileo device memory to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case the driver will allocate cache safe memory for its use using cacheDmaMalloc() .
<i>memSize</i>	The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of Command Frame Descriptor, Receive Frame Descriptor and reception buffers.
<i>nTfds</i>	This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is less than 32, a default of 32 is used.

nRfds

This parameter specifies the number of receive descriptor/buffers to be allocated. If this parameter is less than 32, a default of 32 is used.

flags

User flags control the run-time characteristics of the Ethernet chip. The bit 0 specifies the copy send capability which is used when CFDs are short of multiple fragmented data sent through multiple CFDs and at least one CFD is available which can be used to transfer the packet with copying the fragmented data to one buffer. Setting the bit 1 enables this capability and requires 1536 bytes (depends on **WANCOM_BUF_DEF_SIZE** and **_CACHE_ALIGN_SIZE**) per CFD. Otherwise it disables the copy send.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

sysWancomInit()

```
STATUS sysWancomInit (int unit, WANCOM_PORT_INFO *pPort)
```

This routine performs any target-specific initialization required before the GT642xx ethernet ports are initialized by the driver. The driver calls this routine every time it wants to [re]initialize the device. This routine returns **OK**, or **ERROR** if it fails.

sysWancomMdioWrite()

```
STATUS sysWancomMdioWrite (int unit, int reg, UINT16 data)
```

Write the data parameter to the specified Phy selected by the unit parameter.

SYSTEM RESOURCE USAGE

The driver uses **cacheDmaMalloc()** to allocate memory to share with the Galileo Ethernet port if **NONE** is passed to memBase parameter through **wancomLoad**. This driver requires the allocated memory in cache safe area, thus the board specific memory allocation feature is required through **_func_wancomEndMallocMemBase** function binding. (For the hash table memory, it is also required through **_func_wancomEndMallocHash** function binding in cache snoop mode.) The size of this area is affected by the configuration parameters specified in the **wancomEndLoad()** call.

TUNING HINTS

The only adjustable parameters are the number of TFDs and RFDs that will be created at run-time. These parameters are given to the driver when **wancomEndLoad()** is called. There is one TFD and one RFD associated with each transmitted frame and each received frame respectively. For memory-limited applications, decreasing the number of TFDs and RFDs may be desirable. Increasing the number of TFDs will provide no performance benefit after a certain point. Increasing the number of RFDs will provide more buffering before packets are dropped. This can be useful if there are tasks running at a higher priority than the net task.

ALIGNMENT

Some architectures do not support unaligned access to 32-bit data items. On these architectures (eg MIPS), it will be necessary to adjust the offset parameter in the port

information to realign the packet. Failure to do so will result in received packets being absorbed by the network stack, although transmit functions should work **OK**.

INCLUDE FILES	none
----------------------	------

SEE ALSO	ifLib , Marvell GT64240 Data Sheet, Marvell GT64260 Data Sheet, Marvell GT64260 Errata
-----------------	---

wdbEndPktDrv

NAME	wdbEndPktDrv – END based packet driver for lightweight UDP/IP
ROUTINES	wdbEndPktDevInit() – initialize an END packet device
DESCRIPTION	This is an END based driver for the WDB system. It uses the MUX and END based drivers to allow for interaction between the target and target server.
USAGE	The driver is typically only called only from the configlette wdbEnd.c . The only directly callable routine in this module is wdbEndPktDevInit() . To use this driver, just select the component INCLUDE_WDB_COMM_END in the folder SELECT_WDB_COMM_TYPE . This is the default selection. To modify the MTU, change the value of parameter WDB_END_MTU in component INCLUDE_WDB_COMM_END .
DATA BUFFERING	The drivers only need to handle one input packet at a time because the WDB protocol only supports one outstanding host-request at a time. If multiple input packets arrive, the driver can simply drop them. The driver then loans the input buffer to the WDB agent, and the agent invokes a driver callback when it is done with the buffer. For output, the agent will pass the driver a chain of mbufs, which the driver must send as a packet. When it is done with the mbufs, it calls wdbMbufChainFree() to free them. The header file wdbMbufLib.h provides the calls for allocating, freeing, and initializing mbufs for use with the lightweight UDP/IP interpreter. It ultimately makes calls to the routines wdbMbufAlloc and wdbMbufFree , which are provided in source code in the configlette usrWdbCore.c .
INCLUDE FILES	drv/wdb/wdbEndPktDrv.h

wdbNetromPktDrv

NAME	wdbNetromPktDrv – NETROM packet driver for the WDB agent
-------------	---

ROUTINES	wdbNetromPktDevInit() – initialize a NETROM packet device for the WDB agent
DESCRIPTION	This is a lightweight NETROM driver that interfaces with the WDB agent's UDP/IP interpreter. It allows the WDB agent to communicate with the host using the NETROM ROM emulator. It uses the emulator's read-only protocol for bi-directional communication. It requires that NetROM's udpsrcmode option is on.
INCLUDE FILES	none

wdbPipePktDrv

NAME	wdbPipePktDrv – pipe packet driver for lightweight UDP/IP
ROUTINES	wdbPipePktDevInit() – initialize a pipe packet device
DESCRIPTION	
OVERVIEW	<p>This module is a pipe for drivers interfacing with the WDB agent's lightweight UDP/IP interpreter. It can be used as a starting point when writing new drivers. Such drivers are the lightweight equivalent of a network interface driver.</p> <p>These drivers, along with the lightweight UDP-IP interpreter, have two benefits over the stand combination of a netif driver + the full VxWorks networking stack; First, they can run in a much smaller amount of target memory because the lightweight UDP-IP interpreter is much smaller than the VxWorks network stack (about 800 bytes total). Second, they provide a communication path which is independent of the OS, and thus can be used to support an external mode (e.g., monitor style) debug agent.</p> <p>Throughout this file the word "pipe" is used in place of a real driver name. For example, if you were writing a lightweight driver for the lance ethernet chip, you would want to substitute "pipe" with "ln" throughout this file.</p>
PACKET READY CALLBACK	
	<p>When the driver detects that a packet has arrived (either in its receiver ISR or in its poll input routine), it invokes a callback to pass the data to the debug agent. Right now the callback routine is called "udpRcv", however other callbacks may be added in the future. The driver's wdbPipeDevInit() routine should be passed the callback as a parameter and place it in the device data structure. That way the driver will continue to work if new callbacks are added later.</p>
MODES	Ideally the driver should support both polled and interrupt mode, and be capable of switching modes dynamically. However this is not required. When the agent is not running, the driver will be placed in "interrupt mode" so that the agent can be activated as soon as a

packet arrives. If your driver does not support an interrupt mode, you can simulate this mode by spawning a VxWorks task to poll the device at periodic intervals and simulate a receiver ISR when a packet arrives.

For dynamically mode switchable drivers, be aware that the driver may be asked to switch modes in the middle of its input ISR. A driver's input ISR will look something like this:

```
doSomeStuff();
pPktDev->wdbDrvIf.stackRcv (pMbuf); /* invoke the callback */
doMoreStuff();
```

If this channel is used as a communication path to an external mode debug agent, then the agent's callback will lock interrupts, switch the device to polled mode, and use the device in polled mode for awhile. Later on the agent will unlock interrupts, switch the device back to interrupt mode, and return to the ISR. In particular, the callback can cause two mode switches, first to polled mode and then back to interrupt mode, before it returns. This may require careful ordering of the callback within the interrupt handler. For example, you may need to acknowledge the interrupt within the `doSomeStuff()` processing rather than the `doMoreStuff()` processing.

USAGE

The driver is typically only called only from `usrWdb.c`. The only directly callable routine in this module is `wdbPipePktDevInit()`. You will need to modify `usrWdb.c` to allow your driver to be initialized by the debug agent. You will want to modify `usrWdb.c` to include your driver's header file, which should contain a definition of `WDB_PIPE_PKT_MTU`. There is a default user-selectable macro called `WDB_MTU`, which must be no larger than `WDB_PIPE_PKT_MTU`. Modify the begining of `usrWdb.c` to insure that this is the case by copying the way it is done for the other drivers. The routine `wdbCommIfInit()` also needs to be modified so that if your driver is selected as the `WDB_COMM_TYPE`, then your drivers init routine will be called. Search `usrWdb.c` for the macro "`WDB_COMM_CUSTOM`" and mimic that style of initialization for your driver.

DATA BUFFERING

The drivers only need to handle one input packet at a time because the WDB protocol only supports one outstanding host-request at a time. If multiple input packets arrive, the driver can simply drop them. The driver then loans the input buffer to the WDB agent, and the agent invokes a driver callback when it is done with the buffer.

For output, the agent will pass the driver a chain of mbufs, which the driver must send as a packet. When it is done with the mbufs, it calls `wdbMbufChainFree()` to free them. The header file `wdbMbuflib.h` provides the calls for allocating, freeing, and initializing mbufs for use with the lightweight UDP/IP interpreter. It ultimately makes calls to the routines `wdbMbufAlloc` and `wdbMbufFree`, which are provided in source code in `usrWdb.c`. This module is a pipe for drivers interfacing with the WDB agent's lightweight UDP/IP interpreter. Such a driver are the lightweight equivalent of a network interface driver.

INCLUDE FILES

`drv/wdb/wdbPipePktDrv.h`

wdbSlipPktDrv

NAME	wdbSlipPktDrv – a serial line packetizer for the WDB agent
ROUTINES	wdbSlipPktDevInit() – initialize a SLIP packet device for a WDB agent
DESCRIPTION	This is a lightweight SLIP driver that interfaces with the WDB agents UDP/IP interpreter. It is the lightweight equivalent of the VxWorks SLIP netif driver, and uses the same protocol to assemble serial characters into IP datagrams (namely the SLIP protocol). SLIP is a simple protocol that uses four token characters to delimit each packet: <ul style="list-style-type: none">- FRAME_END (0300)- FRAME_ESC (0333)- FRAME_TRANS_END (0334)- FRAME_TRANS_ESC (0335) The END character denotes the end of an IP packet. The ESC character is used with TRANS_END and TRANS_ESC to circumvent potential occurrences of END or ESC within a packet. If the END character is to be embedded, SLIP sends "ESC TRANS_END" to avoid confusion between a SLIP-specific END and actual data whose value is END. If the ESC character is to be embedded, then SLIP sends "ESC TRANS_ESC" to avoid confusion. (Note that the SLIP ESC is not the same as the ASCII ESC.)
	On the receiving side of the connection, SLIP uses the opposite actions to decode the SLIP packets. Whenever an END character is received, SLIP assumes a full packet has been received and sends on.
	This driver has an MTU of 1006 bytes. If the host is using a real SLIP driver with a smaller MTU, then you will need to lower the definition of WDB_MTU in configAll.h so that the host and target MTU match. If you are not using a SLIP driver on the host, but instead are using the target server's wdbserial backend to connect to the agent, then you do not need to worry about incompatibilities between the host and target MTUs.
INCLUDE FILES	none

wdbTsfsDrv

NAME	wdbTsfsDrv – virtual generic file I/O driver for the WDB agent
ROUTINES	wdbTsfsDrv() – initialize the TSFS device driver for a WDB agent

DESCRIPTION

This library provides a virtual file I/O driver for use with the WDB agent. I/O is performed on this virtual I/O device exactly as it would be on any device referencing a VxWorks file system. File operations, such as **read()** and **write()**, move data over a virtual I/O channel created between the WDB agent and the Workbench target server. The operations are then executed on the host file system. Because file operations are actually performed on the host file system by the target server, the file system presented by this virtual I/O device is known as the target-server file system, or TSFS.

The driver is installed with **wdbTsfsDrv()**, creating a device typically called **/tgttsvr**. See the manual page for **wdbTsfsDrv()** for more information about using this function. To use this driver, just select the component **INCLUDE_WDB_TSFS** in the folder **FOLDER_WDB_OPTIONS**. The initialization is done automatically, enabling access to TSFS, when **INCLUDE_WDB_TSFS** is defined. The target server also must have TSFS enabled in order to use TSFS. See the *System Viewer User's Guide: Upload Method* and *Wind River Workbench User's Guide: Target Server*.

TSFS SOCKETS

TSFS provides all of the functionality of other VxWorks file systems. For details, see the VxWorks programmer guides. In addition to normal files, however, TSFS also provides basic access to TCP sockets. This includes opening the client side of a TCP socket, reading, writing, and closing the socket. Basic **setsockopt()** commands are also supported.

To open a TCP socket using TSFS, use a filename of the form:

To open and connect a TCP socket to a server socket located on a server named **mongoose**, listening on port 2010, use the following:

```
fd = open ("/tgttsvr/TCP:mongoose:2010", 0, 0)
```

The open flags and permission arguments to the open call are ignored when opening a socket through TSFS. If the server **mongoose** has an IP number of **144.12.44.12**, you can use the following equivalent form of the command:

```
fd = open ("/tgttsvr/TCP:144.12.44.12:2010", 0, 0)
```

DIRECTORIES

All directory functions, such as **mkdir()**, **rmdir()**, **opendir()**, **readdir()**, **closedir()**, and **rewinddir()** are supported by TSFS, regardless of whether the target server providing TSFS is being run on a UNIX or Windows host.

While it is possible to open and close directories using **open()** and **close()**, it is not possible to read from a directory using **read()**. Instead, **readdir()** must be used. It is also not possible to write to an open directory, and opening a directory for anything other than read-only results in an error, with **errno** set to **EISDIR**. Calling **read()** on a directory returns **ERROR** with **errno** set to **EISDIR**.

OPEN FLAGS

When the target server that is providing the TSFS is running on a Windows host, the default file-translation mode is binary translation. If text translation is required, then **WDB_TSFS_O_TEXT** can be included in the mode argument to **open()**. For example:

```
fd = open ("/tgttsvr/foo", O_CREAT | O_RDWR | WDB_TSFS_O_TEXT, 0777)
```

If the target server providing TSFS services is running on a UNIX host, **WDB_TSFS_O_TEXT** is ignored.

TARGET SERVER For general information on the target server, see the reference entry for **tgtsvr**. In order to use this library, the target server must support and be configured with the following options:

-R root

Specify the root of the host's file system that is visible to target processes using TSFS. This flag is required to use TSFS. Files under this root are by default read only. To allow read/write access, specify **-RW**.

-RW

Allow read and write access to host files by target processes using TSFS. When this option is specified, access to the target server is restricted as if **-L** were also specified.

IOCTL SUPPORT TSFS supports the following **ioctl()** functions for controlling files and sockets. Details about each function can be found in the documentation listed below.

FIOSEEK

FIOWHERE

FIOMKDIR

Create a directory. The path, in this case **/tgtsvr/tmp**, must be an absolute path prefixed with the device name. To create the directory **/tmp** on the root of the TSFS file system use the following:

```
status = ioctl (fd, FIOMKDIR, "/tgtsvr/tmp");
```

FIORMDIR

Remove a directory. The path, in this case **/tgtsvr/foo**, must be an absolute path prefixed with the device name. To remove the directory **/foo** from the root of the TSFS file system, use the following:

```
status = ioctl (fd, FIORMDIR, "/tgtsvr/foo");
```

FIORENAME

Rename the file or directory represented by **fd** to the name in the string pointed to by **arg**. The path indicated by **arg** may be prefixed with the device name or not. Using this **ioctl()** function with the path **/foo/goo** produces the same outcome as the path **/tgtsvr/foo/goo**. The path is not modified to account for the current working directory, and therefore must be an absolute path.

```
char *arg = "/tgtsvr/foo/goo";
status = ioctl (fd, FIORENAME, arg);
```

FIOREADDIR

FIONREAD

Return the number of bytes ready to read on a TSFS socket file descriptor.

FIOFSTATGET**FIOGETFL****FIOUNLINK**

Remove a file. The path, in this case **/tgtsvr/foo**, must be an absolute path prefixed with the device name. To remove the file **/foo** from the root of the TSFS file system, use the following:

```
fd = open ("/tgtsvr/foo", O_RDWR, 0);
status = ioctl (fd, FIOUNLINK, 0);
close (fd);
```

The following **ioctl()** functions can be used only on socket file descriptors. Using these functions with **ioctl()** provides similar behavior to the **setsockopt()** and **getsockopt()** functions usually used with socket descriptors. Each command's name is derived from a **getsockopt()**/**setsockopt()** command and works in exactly the same way as the respective **getsockopt()**/**setsockopt()** command. The functions **setsockopt()** and **getsockopt()** can not be used with TSFS socket file descriptors.

For example, to enable recording of debugging information on the TSFS socket file descriptor, call:

```
int arg = 1;
status = ioctl (fd, SO_SETDEBUG, arg);
```

To determine whether recording of debugging information for the TSFS-socket file descriptor is enabled or disabled, call:

```
int arg;
status = ioctl (fd, SO_GETDEBUG, & arg);
```

After the call to **ioctl()**, **arg** contains the state of the debugging attribute.

The **ioctl()** functions supported for TSFS sockets are:

SO_SETDEBUG

Equivalent to **setsockopt()** with the **SO_DEBUG** command.

SO_GETDEBUG

Equivalent to **getsockopt()** with the **SO_DEBUG** command.

SO_SETSNDBUF

This command changes the size of the send buffer of the host socket. The configuration of the WDB channel between the host and target also affects the number of bytes that can be written to the TSFS file descriptor in a single attempt.

SO_SETRCVBUF

This command changes the size of the receive buffer of the host socket. The configuration of the WDB channel between the host and target also affects the number of bytes that can be read from the TSFS file descriptor in a single attempt.

SO_SETDONTROUTE

Equivalent to **setsockopt()** with the **SO_DONTRROUTE** command.

SO_GETDONTROUTE

Equivalent to `getsockopt()` with the `SO_DONTROUTE` command.

SO_SETOOBINLINE

Equivalent to `setsockopt()` with the `SO_OOBINLINE` command.

SO_GETOOBINLINE

Equivalent to `getsockopt()` with the `SO_OOBINLINE` command.

SO SNDURGB

The `SO SNDURGB` command sends one out-of-band byte (pointed to by `arg`) through the socket.

The routines in this library return the VxWorks error codes that most closely match the `errno`s generated by the corresponding host function. If an error is encountered that is due to a WDB failure, a WDB error is returned instead of the standard VxWorks `errno`. If an `errno` generated on the host has no reasonable VxWorks counterpart, the host `errno` is passed to the target calling routine unchanged.

INCLUDE FILES `wdb/wdbVioLib.h`

SEE ALSO the VxWorks programmer guides, *Wind River Workbench User's Guide*.

wdbVioDrv

NAME `wdbVioDrv` – virtual `tty` I/O driver for the WDB agent

ROUTINES `wdbVioDrv()` – initialize the `tty` driver for a WDB agent

DESCRIPTION This library provides a pseudo-tty driver for use with the WDB debug agent. I/O is performed on a virtual I/O device just like it is on a VxWorks serial device. The difference is that the data is not moved over a physical serial channel, but rather over a virtual channel created between the WDB debug agent and the Tornado host tools.

The driver is installed with `wdbVioDrv()`. Individual virtual I/O channels are created by opening the device (see `wdbVioDrv()` for details). The virtual I/O channels are defined as follows:

Channel	Usage
0	Virtual console
1-0xfffffff	Dynamically created on the host
$\geq 0x1000000$	User defined

Once data is written to a virtual I/O channel on the target, it is sent to the host-based target server. The target server allows this data to be sent to another host tool, redirected to the

"virtual console," or redirected to a file. For details see the *Wind River Workbench User's Guide*.

USAGE	To use this driver, just select the component INCLUDE_WDB_VIO_DRV at configuration time.
INCLUDE FILES	drv/wdb/wdbVioDrv.h

xbd

NAME	xbd – Extended Block Device Library
ROUTINES	xbdInit() – initialize the XBD library xbdAttach() – attach an XBD device xbdDetach() – detach an XBD device xbdIoctl() – XBD device ioctl routine xbdStrategy() – XBD strategy routine xbdDump() – XBD dump routine xbdSize() – retrieve the total number of bytes xbdNBlocks() – retrieve the total number of blocks xbdBlockSize() – retrieve the block size
DESCRIPTION	This module implements the extended block device.
INCLUDE FILES	drv/xbd/xbd.h

z8530Sio

NAME	z8530Sio – Z8530 SCC Serial Communications Controller driver
ROUTINES	z8530DevInit() – intialize a Z8530_DUSART z8530IntWr() – handle a transmitter interrupt z8530IntRd() – handle a reciever interrupt z8530IntEx() – handle error interrupts z8530Int() – handle all interrupts in one vector
DESCRIPTION	This is the driver for the Z8530 SCC (Serial Communications Controller). It uses the SCCs in asynchronous mode only.

USAGE

A Z8530_DUSART structure is used to describe the chip. This data structure contains two Z8530_CHAN structures which describe the chip's two serial channels. Supported baud rates range from 50 to 38400. The default baud rate is Z8530_DEFAULT_BAUD (9600). The BSP may redefine this.

The BSP's **sysHwInit()** routine typically calls **sysSerialHwInit()** which initializes all the values in the Z8530_DUSART structure (except the SIO_DRV_FUNCS) before calling **z8530DevInit()**.

The BSP's **sysHwInit2()** routine typically calls **sysSerialHwInit2()** which connects the chips interrupts via **intConnect()** (either the single interrupt z8530Int or the three interrupts z8530IntWr, z8530IntRd, and z8530IntEx).

This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the signals CTS on transmit and DSR on read. Refer to the target documentation for the RS232 port configuration. The function HUPCL(hang up on last close) is supported. Default hardware options are defined by Z8530_DEFAULT_OPTIONS. The BSP may redefine them.

All device registers are accessed via BSP-defined macros so that memory- mapped as well as I/O space accesses can be supported. The BSP may re- define the REG_8530_READ and REG_8530_WRITE macros as needed. By default, they are defined as simple memory-mapped accesses.

The BSP may define DATA_REG_8530_DIRECT to cause direct access to the Z8530 data register, where hardware permits it. By default, it is not defined.

The BSP may redefine the macro for the channel reset delay Z8530_RESET_DELAY as well as the channel reset delay counter value Z8530_RESET_DELAY_COUNT as required. The delay is defined as the minimum time between successive chip accesses (6 PCLKs + 200 nSec for a Z8530, 4 PCLKs for a Z85C30 or Z85230) plus an additional 4 PCLKs. At a typical PCLK frequency of 10 MHz, each PCLK is 100 nSec, giving a minimum reset delay of:

Z8530

10 PCLKs + 200 nSec = 1200 nSec = 1.2 uSec

Z85x30: 8 PCLKs = 800 nSec = 0.8 uSec

INCLUDE FILES

drv/sio/z8530Sio.h

2

Routines

<u>_archOptRegRd16_00()</u> – read 16 bits from mem space register	229
<u>_archOptRegRd16_01()</u> – read 16 bits from mem space register and swap data	229
<u>_archOptRegRd16_20()</u> – read 16 bits from IO space register	230
<u>_archOptRegRd16_21()</u> – read 16 bits from IO space register and swap data	230
<u>_archOptRegRd32_00()</u> – read 32 bits from mem space register	231
<u>_archOptRegRd32_03()</u> – read 32 bits from mem space register and swap data	231
<u>_archOptRegRd32_20()</u> – read 32 bits from IO space register	232
<u>_archOptRegRd32_23()</u> – read 32 bits from IO space register and swap data	232
<u>_archOptRegRd64_00()</u> – read 64 bits from mem space register	233
<u>_archOptRegRd64_07()</u> – read 64 bits from mem space register and swap data	233
<u>_archOptRegRd64_20()</u> – read 64 bits from IO space register	234
<u>_archOptRegRd64_27()</u> – read 64 bits from IO space register and swap data	234
<u>_archOptRegRd8_00()</u> – read 8 bits from mem space register	235
<u>_archOptRegRd8_20()</u> – read 8 bits from IO space register	235
<u>_archOptRegWr16_00()</u> – write 16 bits to a mem space register	236
<u>_archOptRegWr16_01()</u> – swap and write 16 bits to a mem space register	236
<u>_archOptRegWr16_10()</u> – write 16 bits to a mem space register and flush data	237
<u>_archOptRegWr16_11()</u> – swap, write 16 bits to mem space register & flush data	237
<u>_archOptRegWr16_20()</u> – write 16 bits to an IO space register	238
<u>_archOptRegWr16_21()</u> – swap and write 16 bits to an IO space register	238
<u>_archOptRegWr16_30()</u> – write 16 bits to IO space register & flush data	239
<u>_archOptRegWr16_31()</u> – swap & write 16 bits to IO space register & flush data	239
<u>_archOptRegWr32_00()</u> – write 32 bits to mem space register	240
<u>_archOptRegWr32_03()</u> – swap the data and write 32 bits to mem space register	240
<u>_archOptRegWr32_10()</u> – write 32 bits to an mem space register and flush	241
<u>_archOptRegWr32_10()</u> – swap & write 32 bits to an mem space register and flush	241
<u>_archOptRegWr32_20()</u> – write 32 bits to an IO space register	242
<u>_archOptRegWr32_23()</u> – swap and write 32 bits to an IO space register	242
<u>_archOptRegWr32_30()</u> – write 32 bits to an IO space register and flush	243
<u>_archOptRegWr32_33()</u> – swap and write 32 bits to an IO space register & flush	243

<u>_archOptRegWr64_00()</u> – write 64 bits to a mem space register	244
<u>_archOptRegWr64_07()</u> – swap 64 bit data and write to a mem space register	244
<u>_archOptRegWr64_10()</u> – write 64 bits to a mem space register and flush data	245
<u>_archOptRegWr64_17()</u> – swap, write 64 bits to a mem space register & flush data	245
<u>_archOptRegWr64_20()</u> – write 64 bits to an IO space register	246
<u>_archOptRegWr64_27()</u> – swap and write 64 bits to an IO space register	246
<u>_archOptRegWr64_30()</u> – write 64 bits to a IO space register and flush data	247
<u>_archOptRegWr64_37()</u> – swap, write 64 bits to a IO space register & flush data	247
<u>_archOptRegWr8_00()</u> – write 8 bits to mem space register	248
<u>_archOptRegWr8_10()</u> – write 8 bits to mem space register and flush data	248
<u>_archOptRegWr8_20()</u> – write 8 bits to IO space register	249
<u>_archOptRegWr8_30()</u> – write 8 bits to IO space register and flush data	249
<u>_archOptRegWrRd16_00()</u> – write 16 bits to a mem space register & read back	250
<u>_archOptRegWrRd16_01()</u> – swap, write 16 bits to a mem space register & read	250
<u>_archOptRegWrRd16_10()</u> – write 16 bits to memspace, flush data & read	251
<u>_archOptRegWrRd16_11()</u> – swap, write 16 bits to memspace, flush & read data	251
<u>_archOptRegWrRd16_20()</u> – write 16 bits to an IO space register & read	252
<u>_archOptRegWrRd16_21()</u> – swap, write 16 bits to an IO space register & read	252
<u>_archOptRegWrRd16_30()</u> – write 16 bits to IO space, flush & read back data	253
<u>_archOptRegWrRd16_31()</u> – swap, write 16 bits to IO space, flush data & read	253
<u>_archOptRegWrRd32_00()</u> – write 32 bits to mem space register & read back	254
<u>_archOptRegWrRd32_03()</u> – swap, write 32 bits to mem space register & read	254
<u>_archOptRegWrRd32_10()</u> – write 32 bits to memspace register, flush & read	255
<u>_archOptRegWrRd32_13()</u> – swap, write 32 bits to memspace, flush & read	255
<u>_archOptRegWrRd32_20()</u> – write 32 bits to an IO space register & read back	256
<u>_archOptRegWrRd32_23()</u> – swap, write 32 bits to an IO space register & read	256
<u>_archOptRegWrRd32_30()</u> – write 32 bits to an IO space register, flush & read	257
<u>_archOptRegWrRd32_33()</u> – swap, write 32 bits to IO space, flush & read	257
<u>_archOptRegWrRd64_00()</u> – write 64 bits to a mem space register and read back	258
<u>_archOptRegWrRd64_07()</u> – swap 64 bit data and write & read from mem space	258
<u>_archOptRegWrRd64_10()</u> – write 64 bits to mem space register, flush data & read	259
<u>_archOptRegWrRd64_17()</u> – swap, write 64 bits to memspace, flush data & read	259
<u>_archOptRegWrRd64_20()</u> – write 64 bits to an IO space register and read back	260
<u>_archOptRegWrRd64_27()</u> – swap, write 64 bits & read from an IO space register	260
<u>_archOptRegWrRd64_30()</u> – write 64 bits to IO space register, flush and read	261
<u>_archOptRegWrRd64_37()</u> – swap, write 64 bits to IO space, flush & read data	261
<u>_archOptRegWrRd8_00()</u> – write 8 bits to mem space register and read back data	262
<u>_archOptRegWrRd8_10()</u> – write 8 bits to mem space register, flush data & read	262
<u>_archOptRegWrRd8_20()</u> – write 8 bits to IO space register and read back	263
<u>_archOptRegWrRd8_30()</u> – write 8 bits to IO space register, flush data & read	263
<u>_archRegProbe()</u> – probe a register on the device	264
<u>_archRegisterRead16()</u> – read 16-bit value from a register	264
<u>_archRegisterRead32()</u> – read 32-bit value from a register	265
<u>_archRegisterRead64()</u> – read 64-bit value from a register	265
<u>_archRegisterRead8()</u> – read 8-bit value from a register	266

<u>_archRegisterWrite16()</u> – write 16-bits to a register	266
<u>_archRegisterWrite32()</u> – write 32-bits to a register	267
<u>_archRegisterWrite64()</u> – write 64-bits to a register	267
<u>_archRegisterWrite8()</u> – write 8-bits to a register	268
<u>_archVolatileRegisterRead()</u> – read from a volatile register	268
<u>_archVolatileRegisterWrite()</u> – write to a volatile register	269
<u>ambaDevInit()</u> – initialise an AMBA channel	269
<u>ambaIntRx()</u> – handle a receiver interrupt	270
<u>ambaIntTx()</u> – handle a transmitter interrupt	270
<u>amd8111LanDumpPrint()</u> – Display statistical counters	270
<u>amd8111LanEndLoad()</u> – initialize the driver and device	271
<u>amd8111LanErrCounterDump()</u> – dump statistical counters	272
<u>ataBlkRW()</u> – read or write sectors to a ATA/IDE disk.	272
<u>ataCmd()</u> – issue a RegisterFile command to ATA/ATAPI device.	273
<u>ataCtrlReset()</u> – reset the specified ATA/IDE disk controller	277
<u>ataDevCreate()</u> – create a device for a ATA/IDE disk	277
<u>ataDevIdentify()</u> – identify device	278
<u>ataDmaRW()</u> – read/write a number of sectors on the current track in DMA mode	278
<u>ataDmaToggle()</u> – turn on or off an individual controllers dma support	279
<u>ataDrv()</u> – Initialize the ATA driver	279
<u>ataDumpPartTable()</u> – dump the partition table from sector 0	280
<u>ataDumpTest()</u> – a quick test of the dump functionality for ATA driver	281
<u>ataInit()</u> – initialize ATA device.	281
<u>ataParamRead()</u> – Read drive parameters	282
<u>ataPiInit()</u> – init a ATAPI CD-ROM disk controller	282
<u>ataRW()</u> – read/write a data from/to required sector.	283
<u>ataRawio()</u> – do raw I/O access	283
<u>ataShow()</u> – show the ATA/IDE disk parameters	284
<u>ataShowInit()</u> – initialize the ATA/IDE disk driver show routine	284
<u>ataStatusChk()</u> – Check status of drive and compare to requested status.	285
<u>ataXbdDevCreate()</u> – create an XBD device for a ATA/IDE disk	285
<u>ataXbdRawio()</u> – do raw I/O access	286
<u>atapiBytesPerSectorGet()</u> – get the number of Bytes per sector.	287
<u>atapiBytesPerTrackGet()</u> – get the number of Bytes per track.	287
<u>atapiCtrlMediumRemoval()</u> – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command	288
<u>atapiCurrentCylinderCountGet()</u> – get logical number of cylinders in the drive.	288
<u>atapiCurrentHeadCountGet()</u> – get the number of read/write heads in the drive.	289
<u>atapiCurrentMDmaModeGet()</u> – get the enabled Multi word DMA mode.	289
<u>atapiCurrentPioModeGet()</u> – get the enabled PIO mode.	290
<u>atapiCurrentRwModeGet()</u> – get the current Data transfer mode.	290
<u>atapiCurrentSDmaModeGet()</u> – get the enabled Single word DMA mode.	291
<u>atapiCurrentUDmaModeGet()</u> – get the enabled Ultra DMA mode.	291
<u>atapiCylinderCountGet()</u> – get the number of cylinders in the drive.	292
<u>atapiDriveSerialNumberGet()</u> – get the drive serial number.	292
<u>atapiDriveTypeGet()</u> – get the drive type.	293

atapiFeatureEnabledGet() – get the enabled features.	293
atapiFeatureSupportedGet() – get the features supported by the drive.	294
atapiFirmwareRevisionGet() – get the firm ware revision of the drive.	295
atapiHeadCountGet() – get the number heads in the drive.	295
atapiInit() – init ATAPI CD-ROM disk controller	296
atapiIoctl() – Control the drive.	296
atapiMaxMDmaModeGet() – get the Maximum Multi word DMA mode the drive supports.	301
atapiMaxPioModeGet() – get the Maximum PIO mode that drive can support.	301
atapiMaxSDmaModeGet() – get the Maximum Single word DMA mode the drive supports	302
atapiMaxUDmaModeGet() – get the Maximum Ultra DMA mode the drive can support.	302
atapiModelNumberGet() – get the model number of the drive.	303
atapiParamsPrint() – Print the drive parameters.	303
atapiPktCmd() – execute an ATAPI command with error processing	304
atapiPktCmdSend() – Issue a Packet command.	304
atapiRead10() – read one or more blocks from an ATAPI Device.	305
atapiReadCapacity() – issue a READ CD-ROM CAPACITY command to a ATAPI device	306
atapiReadTocPmaAtip() – issue a READ TOC command to a ATAPI device	306
atapiRemovMediaStatusNotifyVerGet() – get the Media Stat Notification Version.	307
atapiScan() – issue SCAN packet command to ATAPI drive.	307
atapiSeek() – issues a SEEK packet command to drive.	308
atapiSetCDSpeed() – issue SET CD SPEED packet command to ATAPI drive.	308
atapiStartStopUnit() – Issues START STOP UNIT packet command	309
atapiStopPlayScan() – issue STOP PLAY/SCAN packet command to ATAPI drive.	309
atapiTestUnitRdy() – issue a TEST UNIT READY command to a ATAPI drive	310
atapiVersionNumberGet() – get the ATA/ATAPI version number of the drive.	310
auDump() – display device status	313
auEndLoad() – initialize the driver and device	314
auInitParse() – parse the initialization string	314
bcm1250MacEndLoad() – initialize the driver and device	315
bcm1250MacPhyShow() – display the physical register values	316
bcm1250MacRxDmaShow() – display RX DMA register values	316
bcm1250MacShow() – display the MAC register values	317
bcm1250MacTxDmaShow() – display TX DMA register values	317
bioInit() – initialize the bio library	318
bio_alloc() – allocate memory blocks	318
bio_done() – terminates a bio operation	319
bio_free() – free the bio memory	319
cisConfigregGet() – get the PCMCIA configuration register	319
cisConfigregSet() – set the PCMCIA configuration register	320
cisFree() – free tuples from the linked list	320
cisGet() – get information from a PC card's CIS	321
cisShow() – show CIS information	321
coldFireSioRegister() – register coldFire sio driver	322
ctB69000VgaInit() – initializes the B69000 chip and loads font in memory.	322
dec21140SromWordRead() – read two bytes from the serial ROM	323

dec21145SPIReadBack() – Read all PHY registers out	323
dec21x40EndLoad() – initialize the driver and device	324
dec21x40PhyFind() – Find the first PHY connected to DEC MII port.	324
devAttach() – attach a device	325
devDetach() – detach a device	325
devInit() – initialize the device manager	326
devMap() – map a device	326
devMapUnsafe() – map a device unconditionally	326
devName() – name a device	327
devResourceGet() – find vxBus resource	327
devResourceIntrGet() – find vxBus interrupt resources	328
devResourceIntrShow() – show interrupt values for specified device	328
devUnmap() – unmap a device	329
devUnmapUnsafe() – unmap a device unconditionally	329
displayRapidIOCfgRegs() – displays RIO registers given a base address	330
drvUnloadAll() – unload all drivers which provide download/unload support	330
el3c90xEndLoad() – initialize the driver and device	331
el3c90xInitParse() – parse the initialization string	331
elt3c509Load() – initialize the driver and device	332
elt3c509Parse() – parse the init string	333
emacEndLoad() – initialize the driver and device	334
emacTimerDebugDump() – Enable debugging output in timer handler	335
endEtherAddressForm() – form an Ethernet address into a packet	335
endEtherCrc32BeGet() – calculate a big-endian CRC-32 checksum	336
endEtherCrc32LeGet() – calculate a little-endian CRC-32 checksum	336
endEtherPacketAddrGet() – locate the addresses in a packet	337
endEtherPacketDataGet() – return the beginning of the packet data	337
endMcacheFlush() – flush all tuples from the mBlk recycle cache	338
endMcacheGet() – allocate an mBlk tuple from the mBlk recycle cache	338
endMcachePutt() – store an mBlk tuple in the mBlk recycle cache	339
endObjFlagSet() – set the flags member of an END_OBJ structure	339
endObjInit() – initialize an END_OBJ structure	340
endPollStatsInit() – initialize polling statistics updates	341
endPoolCreate() – create a buffer pool for an END driver	341
endPoolCreate() – create a jumbo buffer pool for an END driver	342
endPoolDestroy() – destroy a pool created with endPoolCreate()	342
endPoolTupleFree() – free an mBlk tuple for an END driver	343
endPoolTupleGet() – allocate an mBlk tuple for an END driver	343
endTok_r() – get a token string (modified version)	344
erfCategoriesAvailable() – Get the number of unallocated User Categories.	345
erfCategoriesAvailable() – Get the maximum number of Categories.	345
erfCategoriesAvailable() – Get the maximum number of Types.	346
erfCategoryAllocate() – Allocates a User Defined Event Category.	346
erfCategoryQueueCreate() – Creates a Category Event Processing Queue.	347
erfDefaultQueueSizeGet() – Get the size of the default queue.	347

erfEventRaise() – Raises an event.	348
erfHandlerRegister() – Registers an event handler for a particular event.	348
erfHandlerUnregister() – Registers an event handler for a particular event.	349
erfLibInit() – Initialize the Event Reporting Framework library	350
erfShow() – Shows debug info for this library.	350
erfTypeAllocate() – Allocates a User Defined Type for this Category.	351
erfTypesAvailable() – Get the number of unallocated User Types for a category.	351
evbNs16550HrdInit() – initialize the NS 16550 chip	352
evbNs16550Int() – handle a receiver/transmitter interrupt for the NS 16550 chip	352
fdDevCreate() – create a device for a floppy disk	352
fdDrv() – initialize the floppy disk driver	354
fdRawio() – provide raw I/O access	354
fei82557DumpPrint() – Display statistical counters	355
fei82557EndLoad() – initialize the driver and device	355
fei82557ErrCounterDump() – dump statistical counters	357
fei82557GetRUStatus() – Return the current RU status and int mask	358
fei82557ShowRxRing() – Show the Receive ring	358
g64120aMfRegister() – register g64120aMf driver	358
g64120aMfpDrvCtrlShow() – show pDrvCtrl for g64120a system controller	359
g64120aPciRegister() – register g64120aPci driver	359
gei82543EndLoad() – initialize the driver and device	360
gei82543LedOff() – turn off LED	360
gei82543LedOn() – turn on LED	361
gei82543PhyRegGet() – get the register value in PHY	361
gei82543PhyRegSet() – set the register value in PHY	362
gei82543RegGet() – get the specified register value in 82543 chip	362
gei82543RegSet() – set the specified register value	363
gei82543TbiCompWr() – enable/disable the TBI compatibility workaround	363
gei82543Unit() – return a pointer to the END_DEVICE for a gei unit	364
hardWareInterFaceInit() – Hardware Interface Pre-Kernel Initialization	364
hcfDeviceGet() – get the HCF_DEVICE pointer	364
hcfResourceAllShow() – show all devices and resource values	365
hcfResourceDevShow() – show the device and resource values	365
hcfResourceShow() – show values for specified resource	366
hwMemAlloc() – allocate a buffer from the hardware memory pool	366
hwMemFree() – return buffer to the hardware memory pool	367
hwMemLibInit() – initialize hardware memory allocation library	367
hwMemPoolCreate() – create or add a memory pool for driver memory allocation	368
hwMemShow() –	368
hwifDevCfgRead() – read from device configuration space	368
hwifDevCfgWrite() – write device configuration space	369
hwifDevCfgWrite64() – write device register	370
hwifDevRegRead() – read device register	371
hwifDevRegWrite() – write device register	371
hwifDevRegWrite64() – write device register	372

i8250HrdInit() – initialize the chip 373
i8250Int() – handle a receiver/transmitter interrupt 373
iOlicomEndLoad() – initialize the driver and device 374
iOlicomIntHandle() – interrupt service for card interrupts 374
iPIIX4AtaInit() – low level initialization of ATA device 375
iPIIX4FdInit() – initializes the floppy disk device 375
iPIIX4GetIntr() – give device an interrupt level to use 375
iPIIX4Init() – initialize PIIX4 376
iPIIX4IntrRoute() – Route PIRQ[A:D] 376
iPIIX4KbdInit() – initializes the PCI-ISA/IDE bridge 377
ibmPciConfigRead() – reads one PCI configuration space register 377
ibmPciConfigWrite() – write to one PCI configuration register location 378
ibmPciSpecialCycle() – generates a PCI special cycle 378
intCtrlrChainISR() – process interrupt chain 379
intCtrlrHwConfGet() – get interrupt input pin configuration from BSP 379
intCtrlrHwConfShow() – print interrupt input pin configuration 380
intCtrlrISRAdd() – Install ISR at specified pin 380
intCtrlrISRDisable() – Disable specified ISR 381
intCtrlrISREnable() – Enable specified ISR 381
intCtrlrISRRemove() – Remove ISR from specified pin 382
intCtrlrPinFind() – Retrieve input pin associated with device 382
intCtrlrStrayISR() – flag stray interrupt 383
intCtrlrTableArgGet() – Fetch ISR argument from ISR table entry 383
intCtrlrTableCreate() – Create table structure 384
intCtrlrTableFlagsGet() – Fetch ISR flags from ISR table entry 384
intCtrlrTableFlagsSet() – Set ISR flags in ISR table entry 385
intCtrlrTableIsrGet() – Fetch ISR function pointer from ISR table entry 385
intCtrlrTableUserSet() – Fill in ISR table entry: driver name, unit, index 386
isrDeferIsrReroute() – Reroute deferral work to a new CPU in the system. 386
isrDeferJobAdd() – Add a job to the deferral queue 387
isrDeferLibInit() – Initialize the ISR deferral library. 387
isrDeferQueueGet() – Get a deferral queue 388
isrRerouteNotify() – Notify driver that ISR has been rerouted 388
ixp400I2CAckReceive (Control Signal)() – get an acknowledgement 389
ixp400I2CAckSend (Control signal)() – send an acknowledgement 389
ixp400I2CByteReceive() – receive a byte on the I2C bus 389
ixp400I2CByteTransmit() – transmit a byte on the I2C bus 390
ixp400I2CReadTransfer() – read from a slave device 390
ixp400I2CStart (Control signal)() – initiate an I2C bus transfer 391
ixp400I2CStop (Control signal)() – terminate an I2C bus transfer 391
ixp400I2CWriteTransfer() – write to a slave device 392
ixp400SioDevInit() – initialise a UART channel 392
ixp400SioInt() – interrupt level processing 393
ixp400SioIntErr() – handle an error interrupt 393
ixp400SioIntWr() – handle a transmitter interrupt 394

ixp400SioRegister() – register ixp400Sio driver	394
ixp400SioStatsShow() – display UART error statistics.	395
ixp400TimerDrvRegister() – register ixp400 timer driver	395
In97xEndLoad() – initialize the driver and device	395
In97xInitParse() – parse the initialization string	396
lptDevCreate() – create a device for an LPT port	397
lptDrv() – initialize the LPT driver	398
lptShow() – show LPT statistics	398
m548xDmaDrvRegister() – register coldFire DMA driver	399
m54x5TimerDrvRegister() – register coldFire timer driver	399
m8260SccEndLoad() – initialize the driver and device	400
m85xxCpuRegister() – register m85xxCpu driver	401
m85xxRioRegDbgRead() – read rapidIO registers	401
m85xxRioRegister() – register PowerPC 85xx rapidIO with bus subsystem	401
m85xxRioStatusShow() – show state of RapidIO interface	402
m85xxTimerDrvRegister() – register m85xx timer driver	402
mcf5475PciRegister() – register Coldfire mcf5475 host controller	403
mib2ErrorAdd() – change a MIB-II error count	403
mib2Init() – initialize a MIB-II structure	404
miiAnCheck() – check the auto-negotiation process result	404
miiLibInit() – initialize the MII library	405
miiLibUnInit() – uninitialized the MII library	405
miiPhyInit() – initialize and configure the PHY devices	406
miiPhyOptFuncMultiSet() – set pointers to MII optional registers handlers	408
miiPhyOptFuncSet() – set the pointer to the MII optional registers handler	409
miiPhyUnInit() – uninitialized a PHY	409
miiRegsGet() – get the contents of MII registers	410
miiShow() – show routine for MII library	410
mipsCavIntCtlrShow() – show pDrvCtrl for mipsIntCtlr	411
mipsIntCtlrShow() – show pDrvCtrl for mipsIntCtlr	411
motFccDrvShow() – debug function to show FCC parameter ram addresses, initial BD and cluster settings	412
motFccDumpRxRing() – Show the Receive Ring details	412
motFccDumpTxRing() – Show the Transmit Ring details	413
motFccEndLoad() – initialize the driver and device	413
motFccEramShow() – Debug Function to show FCC CP ethernet parameter ram.	414
motFccIramShow() – Debug Function to show FCC CP internal ram parameters.	415
motFccMibShow() – Debug Function to show MIB statistics.	415
motFccMiiShow() – debug function to show the Mii settings in the Phy Info structure	415
motFccPramShow() – Debug Function to show FCC CP parameter ram.	416
motFccShow() – Debug Function to show driver specific control data.	416
motFecEndLoad() – initialize the driver and device	417
ncr810CtrlCreate() – create a control structure for the NCR 53C8xx SIOP	418
ncr810CtrlInit() – initialize a control structure for the NCR 53C8xx SIOP	419
ncr810SetHwRegister() – set hardware-dependent registers for the NCR 53C8xx SIOP	420

ncr810Show() – display values of all readable NCR 53C8xx SIOP registers	421
ne2000EndLoad() – initialize the driver and device	422
noDev() – optional driver functionality not present	422
ns16550DevInit() – intialize an NS16550 channel	422
ns16550Int() – interrupt level processing	423
ns16550IntEx() – miscellaneous interrupt processing	423
ns16550IntRd() – handle a receiver interrupt	424
ns16550IntWr() – handle a transmitter interrupt	424
ns16550SioRegister() – register ns16550vxb driver	425
ns83902EndLoad() – initialize the driver and device	425
ns83902RegShow() – prints the current value of the NIC registers	426
nullDrv() – optional driver functionality not present	426
oceanSioRegister() – register oceanVxb driver	427
oceanVxbDevInit() – initialize an OCTEON channel	427
oceanVxbDevProbe() – probe for device presence at specific address	428
oceanVxbInt() – interrupt level processing	428
oceanVxbIntEx() – miscellaneous interrupt processing	429
oceanVxbIntRd() – handle a receiver interrupt	429
oceanVxbIntWr() – handle a transmitter interrupt	430
openPicTimerDrvRegister() – register openPic timer driver	430
optimizeAccessFunction() – optimize a function based on flags	431
pccardAtaEnabler() – enable the PCMCIA-ATA device	431
pccardEltEnabler() – enable the PCMCIA Etherlink III card	432
pccardMkfs() – initialize a device and mount a DOS file system	432
pccardMount() – mount a DOS file system	433
pccardSramEnabler() – enable the PCMCIA-SRAM driver	433
pccardTffsEnabler() – enable the PCMCIA-TFFS driver	434
pciAllHeaderShow() – show PCI header for all devices	434
pciAutoAddrAlign() – align a PCI address and check boundary conditions	434
pciAutoBusNumberSet() – set the primary, secondary, and subordinate bus number	435
pciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge	436
pciAutoCfg() – Automatically configure all nonexcluded PCI headers	436
pciAutoCfgCtl() – set or get pciAutoConfigLib options	437
pciAutoCfgCtl() – set or get autoConfigLib options	444
pciAutoConfig() – automatically configure all nonexcluded PCI headers (obsolete)	445
pciAutoConfigLibInit() – initialize PCI autoconfig library	446
pciAutoDevReset() – quiesce a PCI device and reset all writeable status bits	446
pciAutoFuncDisable() – disable a specific PCI function	447
pciAutoFuncEnable() – perform final configuration and enable a function	447
pciAutoGetNextClass() – find the next device of specific type from probe list	448
pciAutoRegConfig() – assign PCI space to a single PCI base address register	448
pciBusAnnounceDevices() – Notify the bus subsystem of all devices on PCI	449
pciConfigBdfPack() – pack parameters for the Configuration Address Register	449
pciConfigCmdWordShow() – show the decoded value of the command word	450
pciConfigCmdWordShow() – show the decoded value of the command word	450

pciConfigEnable() – Set the globalBusCtrlID for the other LEGACY pci	451
pciConfigExtCapFind() – find extended capability in ECP linked list	451
pciConfigExtCapFind() – find extended capability in ECP linked list	452
pciConfigForEachFunc() – check condition on specified bus	452
pciConfigForEachFunc() – check condition on specified bus	453
pciConfigFuncShow() – show configuration details about a function	453
pciConfigFuncShow() – show configuration details about a function	454
pciConfigInByte() – read one byte from the PCI configuration space	454
pciConfigInByte() – read one byte from the PCI configuration space	455
pciConfigInLong() – read one longword from the PCI configuration space	455
pciConfigInLong() – read one longword from the PCI configuration space	456
pciConfigInWord() – read one word from the PCI configuration space	456
pciConfigInWord() – read one word from the PCI configuration space	457
pciConfigLibInit() – initialize the configuration access-method and addresses	457
pciConfigLibInit() – initialize the configuration access-method and addresses	459
pciConfigModifyByte() – Perform a masked longword register update	460
pciConfigModifyLong() – Perform a masked longword register update	461
pciConfigModifyWord() – Perform a masked longword register update	461
pciConfigOutByte() – write one byte to the PCI configuration space	462
pciConfigOutByte() – write one byte to the PCI configuration space	463
pciConfigOutLong() – write one longword to the PCI configuration space	463
pciConfigOutLong() – write one longword to the PCI configuration space	464
pciConfigOutWord() – write one 16-bit word to the PCI configuration space	464
pciConfigOutWord() – write one 16-bit word to the PCI configuration space	465
pciConfigReset() – disable cards for warm boot	465
pciConfigStatusWordShow() – show the decoded value of the status word	466
pciConfigStatusWordShow() – show the decoded value of the status word	466
pciConfigTopoShow() – show PCI topology	467
pciConfigTopoShow() – show PCI topology	467
pciConnect() – connect PCI bus type to bus subsystem	468
pciDevConfig() – configure a device on a PCI bus	468
pciDevConfig() – configure a device on a PCI bus	469
pciDevMatch() – check whether device and driver go together	469
pciDevShow() – show information about PCI device	470
pciDeviceAnnounce() – notify the bus subsystem of a device on PCI	470
pciDeviceShow() – print information about PCI devices	471
pciDeviceShow() – print information about PCI devices	471
pciFindClass() – find the nth occurrence of a device by PCI class code.	471
pciFindClass() – find the nth occurrence of a device by PCI class code.	472
pciFindClassShow() – find a device by 24-bit class code	473
pciFindClassShow() – find a device by 24-bit class code	473
pciFindDevice() – find the nth device with the given device & vendor ID	473
pciFindDevice() – find the nth device with the given device & vendor ID	474
pciFindDeviceShow() – find a PCI device and display the information	475
pciFindDeviceShow() – find a PCI device and display the information	475

pciHcfRecordFind() – find device's HCF pciSlot record	476
pciHeaderShow() – print a header of the specified PCI device	476
pciHeaderShow() – print a header of the specified PCI device	476
pciInit() – first-pass bus type initialization	477
pciInit2() – second-pass bus type initialization	477
pciInt() – interrupt handler for shared PCI interrupt	478
pciIntConnect() – connect the interrupt handler to the PCI interrupt.	478
pciIntConnect() – connect the interrupt handler to the PCI interrupt.	479
pciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)	479
pciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)	480
pciIntDisconnect2() – disconnect an interrupt handler from the PCI interrupt.	481
pciIntDisconnect2() – disconnect the interrupt handler	481
pciIntLibInit() – initialize the pciIntLib module	482
pciRegister() – register PCI bus type	482
pciSpecialCycle() – generate a special cycle with a message	482
pciSpecialCycle() – generate a special cycle with a message	483
pcicInit() – initialize the PCIC chip	483
pcicShow() – show all configurations of the PCIC chip	484
pcmciaInit() – initialize the PCMCIA event-handling package	484
pcmciaShow() – show all configurations of the PCMCIA chip	485
pcmciaShowInit() – initialize all show routines for PCMCIA drivers	485
pcmciad() – handle task-level PCMCIA events	485
pentiumPciAllHeaderShow() – show PCI header for specified device	486
pentiumPciBusDevGet() – find bus controller	486
pentiumPciMmuMapAdd() – memory map sysPhysMemDesc	487
pentiumPciPhysMemHandle() – configure PCI memory for a device	487
pentiumPciPhysMemShow() – display sysPhysMemDesc entries	488
pentiumPciRegAddrShow() – debug routine to print register addresses	488
pentiumPciRegister() – register Pentium PCI host bridge device driver	489
plbAccessInit() – initialize the plb access module	489
plbConnect() – third-stage PLB bus initialization	489
plbDevMatch() – check whether device and driver go together	490
plbInit1() – first-stage PLB bus initialization	490
plbInit2() – second-stage PLB bus initialization	491
plbIntrGet() – Get interrupt controller for device	491
plbIntrSet() – set interrupt controller for device	491
plbIntrShow() – Show interrupt controllers for device's interrupts	492
plbRegister() – register PLB with bus subsystem	492
ppc403DevInit() – initialize the serial port unit	493
ppc403DummyCallback() – dummy callback routine	493
ppc403IntEx() – handle error interrupts	494
ppc403IntRd() – handle a receiver interrupt	494
ppc403IntWr() – handle a transmitter interrupt	495
ppc440gpPciHostBridgeInit() – Initialize the PCI-X Host Bridge	495
ppc440gpPciRegAddrShow() – debug routine to print register addresses	496

ppc440gpPciRegister() – register PowerPC 440GP host bridge	496
ppc860DevInit() – initialize the SMC	497
ppc860Int() – handle an SMC interrupt	497
ppcDecTimerDrvRegister() – register ppcDec timer driver	498
qeAnd32() – Read then Write 32 bit register usign and mask	498
qeIvecToInum() – get the relevant interrupt number	498
qeOr32() – Read then Write 32 bit register using or mask	499
quiccAnd32() – Read then Write 32 bit register usign and mask	499
quiccIntrInit() – initialize the interrupt manager for the PowerPC 83XX	500
quiccIvecToInum() – get the relevant interrupt number	500
quiccOr32() – Read then Write 32 bit register using or mask	501
quiccTimerDrvRegister() – register coldFire timer driver	501
rapidIoRegister() – register RapidIO bus type	502
rioConnect() – connect RapidIO bus type to bus subsystem	502
rioDevMatch() – check whether device and driver go together	502
rioInit1() – first-pass bus type initialization	503
rioInit2() – second-pass bus type initialization	503
rm9000x2gDevInit() – intialize an NS16550 channel	504
rm9000x2gIInt() – interrupt level processing	504
rm9000x2gIIntEx() – miscellaneous interrupt processing	505
rm9000x2gIIntMod() – interrupt level processing	505
rm9000x2gIIntRd() – handle a receiver interrupt	506
rm9000x2gIIntWr() – handle a transmitter interrupt	506
sh7700TimerDrvRegister() – register sh7700 timer driver	507
sh77xxPciRegister() – register sh77xxPci driver	507
sh77xxPcipDrvCtrlShow() – show pDrvCtrl for sh77xxPci bus controller	507
shSciDevInit() – initialize a on-chip serial communication interface	508
shSciIntErr() – handle a channel's error interrupt.	508
shSciIntRcv() – handle a channel's receive-character interrupt.	509
shSciIntTx() – handle a channels transmitter-ready interrupt.	509
shScifDevInit() – initialize a on-chip serial communication interface	510
shScifIntErr() – handle a channel's error interrupt.	510
shScifIntRcv() – handle a channel's receive-character interrupt.	510
shScifIntTx() – handle a channels transmitter-ready interrupt.	511
shScifSioRegister() – register shScif driver	511
sioNextChannelNumberAssign() – assign a new serial channel number	512
smEndLoad() – attach the SM interface to the MUX, initialize driver and device	512
smEndRegister() – register smEnd driver	514
smNetShow() – show information about a shared memory network	515
smcFdc37b78xDevCreate() – set correct IO port addresses for Super I/O chip	516
smcFdc37b78xInit() – initializes Super I/O chip Library	516
smcFdc37b78xKbdInit() – initializes the keyboard controller	517
sramDevCreate() – create a PCMCIA memory disk device	517
sramDrv() – install a PCMCIA SRAM memory driver	518
sramMap() – map PCMCIA memory onto a specified ISA address space	518

sym895CtrlCreate() – create a structure for a SYM895 device	519
sym895CtrlInit() – initialize a SCSI Controller Structure	520
sym895GPIOConfig() – configures general purpose pins GPIO 0-4	520
sym895GPIOCtrl() – controls general purpose pins GPIO 0-4	521
sym895Intr() – interrupt service routine for the SCSI Controller	522
sym895Loopback() – this routine performs loopback diagnostics on 895 chip	523
sym895SetHwOptions() – sets the Sym895 chip Options	523
sym895Show() – display values of all readable SYM 53C8xx SIOP registers	524
sysAuxClkConnect() – connect a routine to the auxiliary clock interrupt	525
sysAuxClkDisable() – turn off auxiliary clock interrupts	526
sysAuxClkEnable() – turn on auxiliary clock interrupts	526
sysAuxClkHandleGet() – get the timer handle for auxiliary clock	527
sysAuxClkRateGet() – get the auxiliary clock rate	527
sysAuxClkRateSet() – set the auxiliary clock rate	527
sysBusIntAck() – acknowledge an interrupt	528
sysBusIntAckHelp() – locate the correct call to acknowledge an interrupt	528
sysBusIntGen() – call interrupt generator function hook	529
sysBusIntGenHelp() – locate the correct call to generate an interrupt	529
sysBusTas() – establish a reservation for a particular address	530
sysBusTasClear() – clear a reservation for a particular address	530
sysBusTasClearHelper() – locate and clear a reservation for a particular address	531
sysBusTasHelper() – locate the correct device to do TAS	531
sysClkConnect() – connect a routine to the system clock interrupt	531
sysClkDisable() – turn off system clock interrupts	532
sysClkEnable() – turn on system clock interrupts	532
sysClkHandleGet() – get the timer handle for system clock	533
sysClkRateGet() – get the system clock rate	533
sysClkRateSet() – set the system clock rate	534
sysMailboxConnect() – TBD	534
sysMailboxEnable() – TBD	534
sysPciHostBridgeInit() – Initialize the PCI Host Bridge	535
sysPciHostBridgeInit() – initialize the PCI Host Bridge	535
sysPciHostBridgeInit() – Initialize the PCI-X Host Bridge	536
sysSerialChanConnect() – connect the SIO_CHAN device	536
sysSerialChanGet() – get the SIO_CHAN device associated with a serial channel	537
sysSerialConnectAll() – connect all SIO_CHAN devices	537
sysSmEndLoad() – load the shared memory END driver	538
sysTimestamp() – get the timestamp timer tick count	538
sysTimestampConnect() – connect a user routine to the timestamp timer interrupt	539
sysTimestampDisable() – disable the timestamp timer	539
sysTimestampEnable() – initialize and enable the timestamp timer	540
sysTimestampFreq() – get the timestamp timer clock frequency	540
sysTimestampHandleGet() – get the timer handle for timestamp timer	540
sysTimestampLock() – get the timestamp timer tick count	541
sysTimestampPeriod() – get the timestamp timer period	541

tcicInit() – initialize the TCIC chip 542
tcicShow() – show all configurations of the TCIC chip 542
tffsBootImagePut() – write to the boot-image region of the flash device 543
tffsShow() – show device information on a specific socket interface 544
tffsShowAll() – show device information on all socket interfaces 544
vgaInit() – initializes the VGA chip and loads font in memory. 545
vxBusShow() – show vxBus subsystem 545
vxbAccessMethodGet() – find specific method for accessing device 546
vxbAuxClkLibInit() – initialize the auxiliary clock library 546
vxbAuxClkShow() – show the auxiliary clock information 546
vxbBusAnnounce() – announce bus discovery to bus subsystem 547
vxbBusListPrint() – Show bus topology 547
vxbBusTypeRegister() – register a bus type 548
vxbBusTypeString() – retrieve bus type string 548
vxbBusTypeUnregister() – unregister a bus type 549
vxbCn3xxxTimerDrvRegister() – register cn3xxx timer driver 549
vxbDelay() – delay for a moment 549
vxbDelayLibInit() – initialize the delay library 550
vxbDelayTimersShow() – show the delay timers information 550
vxbDevAccessShow() – Show bus access methods 551
vxbDevConnect() – HWIF Post-Kernel Connection 551
vxbDevConnectInternal() – third-pass initialization of devices 551
vxbDevError() – driver does not support specified functionality 552
vxbDevInit() – HWIF Post-Kernel Init 552
vxbDevInitInternal() – second-pass initialization of devices 553
vxbDevIterate() – perform specified action for each device 553
vxbDevMethodGet() – find entry point of method 553
vxbDevMethodRun() – run method on devices 554
vxbDevParent() – find parent device 554
vxbDevPath() – trace from device to nexus 555
vxbDevPathShow() – Show bus hierarchy 555
vxbDevRegister() – register a device driver 556
vxbDevRemovalAnnounce() – announce device removal to bus subsystem 556
vxbDevStructAlloc() – allocate VXB_DEVICE structure 557
vxbDevStructFree() – free VXB_DEVICE structure 557
vxbDevStructShow() – Show device information 557
vxbDevTblEnumerate() – enumerate VxBus devices from hwconf device table 558
vxbDeviceAnnounce() – announce device discovery to bus subsystem 559
vxbDeviceDriverRelease() – turn an instance into an orphan 559
vxbDeviceMethodRun() – run method on device 560
vxbDmaBufFlush() – partial cache flush for DMA map 560
vxbDmaBufInit() – initialize buffer and DMA system 561
vxbDmaBufInvalidate() – partial cache invalidate for DMA map 561
vxbDmaBufMapCreate() – create/allocate a DMA map 562
vxbDmaBufMapDestroy() – release a DMA map 562

vxbDmaBufMapFlush() – flush DMA Map cache	563
vxbDmaBufMapInvalidate() – invalidate DMA Map cache	563
vxbDmaBufMapIoVecLoad() – map a virtual buffer with scatter/gather	564
vxbDmaBufMapLoad() – map a virtual buffer	564
vxbDmaBufMapMblkLoad() – map a virtual buffer with mBlk	565
vxbDmaBufMapUnload() – unmap/destroy a previous virtual buffer mapping	565
vxbDmaBufMemAlloc() – allocate DMA-able memory	566
vxbDmaBufMemFree() – release DMA-able memory	567
vxbDmaBufSync() – do cache flushes or invalidates	567
vxbDmaBufTagCreate() – create a DMA tag	568
vxbDmaBufTagDestroy() – destroy a DMA tag	568
vxbDmaBufTagParentGet() – retrieve parent DMA tag	569
vxbDmaChanAlloc() – allocate and initialize a DMA channel	569
vxbDmaChanFree() – free a DMA channel	570
vxbDmaLibInit() – initialize the VxBus slave DMA library	570
vxbDriverUnregister() – remove a device driver from the bus subsystem	571
vxbDrvRescan() – rescan all orphans to match against driver	571
vxbEndQctrlInit() – initialize the vxbEndQctrl library	572
vxbEndQnumSet() – set the network job queue for a VxBus END device	572
vxbEpicIntCtrrRegister() – register epicIntCtrr driver	573
vxbEpicIntCtrrpDrvCtrlShow() – show pDrvCtrl for template controller	573
vxbEpicSharedMsgHandler() – Handles the interrupt and calls the relevant ISRs	574
vxbGetPciDevice() – Locates a PCI device	574
vxbI8253TimerDrvRegister() – registers the driver for i8253 timer	575
vxbI8259IntCtrrRegister() – registers I8259 driver with vxBus	575
vxbI8259IntDisablePIC() – disables a PIC interrupt level	575
vxbI8259IntEOI() – to send an EOI signal	576
vxbI8259IntEnablePIC() – enable a PIC interrupt level	576
vxbIaTimestampDrvRegister() – register intel timestamp driver	577
vxbInit() – initialize vxBus	577
vxbInstByNameFind() – retrieve the VXB_DEVICE_ID for an instance	577
vxbInstParamByIndexGet() – retrieve driver parameter value	578
vxbInstParamByNameGet() – retrieve driver parameter value	579
vxbInstParamSet() – set driver parameter for specified instance	579
vxbInstUnitGet() – get the unit number	580
vxbInstUnitSet() – set the unit number	580
vxbIntAcknowledge() – Acknowledge device's interrupt	581
vxbIntConnect() – connect device's interrupt	581
vxbIntDisable() – disable device's interrupt	582
vxbIntDisconnect() – disconnect device's interrupt	582
vxbIntDynaConnect() – Initializes/connects the dynamic interrupt	583
vxbIntDynaCtrrInit() – Intializes function pointers to enable library use.	583
vxbIntDynaVecAlloc() – Finds the the next vector for dynamic interrupt	584
vxbIntDynaVecDevMultiProgram() – program multiple vectors	584
vxbIntDynaVecMultiConnect() – connect to multiple vectors	585

vxbIntDynaVecOwnerFind() – Finds the owner of dynnamic vector alloc	585
vxbIntEnable() – Enable device's interrupt	586
vxbIntReroute() – Route specified interrupt to destination CPU	586
vxbIntToCpuRoute() – reroute all interrupts for a specified CPU	587
vxbIntVectorGet() – get device's interrupt vector	587
vxbIoApicIntrDataShow() – show IO APIC register data acquired by vxBus	588
vxbIoApicIntrDrvRegister() – register ioApic driver	588
vxbIoApicIntrShowAll() – show All IO APIC registers	589
vxbLibError() – handle error conditions	589
vxbLibInit() – initialize vxBus library	589
vxbLoApicIntrDrvRegister() – register loApic driver	590
vxbLoApicIntrShow() – show Local APIC registers	590
vxbLoApicIpiConnect() – Connect ISR to specified IPI	591
vxbLoApicIpiDisable() – Disable specified IPI	591
vxbLoApicIpiDisconnect() – Disconnect ISR from specified IPI	592
vxbLoApicIpiEnable() – Enable ISR for specified IPI	592
vxbLoApicIpiGen() – Generate specified IPI	593
vxbLoApicIpiPrioGet() – Find priority of specified IPI	593
vxbLoApicIpiPrioSet() – Set priority of specified IPI	594
vxbLoApicStatusShow() – show Local APIC TMR, IRR, ISR registers	594
vxbLoApicTimerDrvRegister() – register loApic timer driver	595
vxbLockGive() – release a VxBus lock	595
vxbLockInit() – initialize a VxBus lock	596
vxbLockTake() – take a VxBus lock	596
vxbMalRegister() – register the vxbMalDma driver	597
vxbMc146818RtcDrvRegister() – registers the driver for MC146818 RTC	597
vxbMipsCavIntCtlrRegister() – register the mipsIntCtlr driver	597
vxbMipsIntCtlrRegister() – register the mips cpu driver	598
vxbMipsSbIntCtlrRegister() – register the mipsIntCtlr driver	598
vxbMpApicDataShow() – show MP Configuration Data acquired by vxBus	599
vxbMpApicDrvRegister() – register mpApic driver	599
vxbMpApicpDrvCtrlShow() – show pDrvCtrl for template controller	599
vxbMpBiosIoIntMapShow() – show MP IO interrupt mapping	600
vxbMpBiosLocalIntMapShow() – show MP local interrupt mapping	600
vxbMpBiosShow() – show MP configuration table	601
vxbMsDelay() – delay for delayTime milliseconds	601
vxbMsiConnect() – calls vxbIntDynaConnect if available	601
vxbNextUnitGet() – get the next available unit number for a driver	602
vxbPciAccessCopy() – copy access function pointers	602
vxbPciAutoAddrAlign() – align a PCI address and check boundary conditions	603
vxbPciAutoBusNumberSet() – set the primary, secondary, and subordinate bus number	603
vxbPciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge	604
vxbPciAutoCfg() – Automatically configure all nonexcluded PCI headers	605
vxbPciAutoCfgCtl() – set or get vxbPciAutoConfigLib options	606
vxbPciAutoConfig() – set standard parameters & initialize PCI	612

vxbPciAutoConfigLibInit() – initialize PCI autoconfig library	613
vxbPciAutoConfigListShow() – show the devices in the list	613
vxbPciAutoDevReset() – quiesce a PCI device and reset all writeable status bits	614
vxbPciAutoFuncDisable() – disable a specific PCI function	614
vxbPciAutoFuncEnable() – perform final configuration and enable a function	615
vxbPciAutoGetNextClass() – find the next device of specific type from probe list	615
vxbPciAutoRegConfig() – assign PCI space to a single PCI base address register	616
vxbPciBusTypeInit() – initialize the PCI bus type	617
vxbPciConfigBdfPack() – pack parameters for the Configuration Address Register	617
vxbPciConfigCmdWordShow() – show the decoded value of the command word	617
vxbPciConfigExtCapFind() – find extended capability in ECP linked list	618
vxbPciConfigForEachFunc() – check condition on specified bus	619
vxbPciConfigFuncShow() – show configuration details about a function	619
vxbPciConfigInByte() – read one byte from the PCI configuration space	620
vxbPciConfigInLong() – read one longword from the PCI configuration space	620
vxbPciConfigInWord() – read one word from the PCI configuration space	621
vxbPciConfigModifyByte() – Perform a masked longword register update	621
vxbPciConfigModifyLong() – Perform a masked longword register update	622
vxbPciConfigModifyWord() – Perform a masked longword register update	623
vxbPciConfigOutByte() – write one byte to the PCI configuration space	624
vxbPciConfigOutLong() – write one longword to the PCI configuration space	624
vxbPciConfigOutWord() – write one 16-bit word to the PCI configuration space	625
vxbPciConfigReset() – disable cards for warm boot	625
vxbPciConfigStatusWordShow() – show the decoded value of the status word	626
vxbPciConfigTopoShow() – show PCI topology	626
vxbPciDevConfig() – configure a device on a PCI bus	627
vxbPciDeviceAnnounce() – Update vxBus to PCI bus orphans	628
vxbPciDeviceShow() – print information about PCI devices	628
vxbPciFindClass() – find the nth occurrence of a device by PCI class code.	628
vxbPciFindClassShow() – find a device by 24-bit class code	629
vxbPciFindDevice() – find the nth device with the given device & vendor ID	630
vxbPciFindDeviceShow() – find a PCI device and display the information	630
vxbPciHeaderShow() – print a header of the specified PCI device	631
vxbPciInt() – interrupt handler for shared PCI interrupt.	631
vxbPciIntConnect() – connect the interrupt handler to the PCI interrupt.	632
vxbPciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)	632
vxbPciIntDisconnect2() – disconnect an interrupt handler from the PCI interrupt.	633
vxbPciIntLibInit() – initialize the vxbPciIntLib module	633
vxbPlbAccessCopy() – copy the access data structure	634
vxbPresStructShow() – Show bus information	634
vxbPrimeCellSioRegister() – register vxbAmbaSio driver	635
vxbQeIntCtlrRegister() – register qeIntCtlr driver	635
vxbQeIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller	635
vxbQuiccIntCtlrRegister() – register quiccIntCtlr driver	636
vxbQuiccIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller	636

vxbR4KTimerDrvRegister() – register mips R4K timer driver	637
vxbRapidIOBusTypeInit() – initialize RapidIO bus type	637
vxbRapidIOCfg() – RapidIO Table-based device enumeration	638
vxbRapidIOTableOverride() – RapidIO Table-based override function	638
vxbRegMap() – obtain an access handle for a given register mapping	639
vxbResourceFind() – find and allocate a vxBus resource	640
vxbSb1DuartSioRegister() – register vxbSb1DuartSio driver	640
vxbSb1TimerDrvRegister() – register sb1 timer driver	641
vxbSmcFdc37xRegister() – register smcFdc37x driver	641
vxbSubDevAction() – perform an action on all devs on bus controller	642
vxbSysClkLibInit() – initialize the system clock library	642
vxbSysClkShow() – show the system clock information	642
vxbSysQeIntDisable() – Disable one of the Level or IRQ interrupts into the SIU	643
vxbSysQeIntEnable() – enable the indicated interrupt	643
vxbSysQuiccInit() – initialize the interrupt manager for the PowerPC 83XX	644
vxbSysQuiccIntDisable() – Disable one of the Level or IRQ interrupts into the SIU	645
vxbSysQuiccIntEnable() – enable the indicated interrupt	645
vxbTimerAlloc() – allocate timer for the requested characteristics	646
vxbTimerFeaturesGet() – get the timer features	646
vxbTimerLibInit() – initialize the timer library	647
vxbTimerRelease() – release the allocated timer	647
vxbTimerStubInit() – initialization function for the timer stub	648
vxbTimestampCookieGet() – to get the cookie information	648
vxbTimestampLibInit() – initialize the timestamp library	649
vxbTimestampShow() – show the timestamp information	649
vxbTopoShow() – Show bus topology	649
vxbUpdateDeviceInfo() – Update vxBus device Information	650
vxbUsDelay() – delay for delayTime microseconds	650
vxbVolRegWrite() – volatile register writes	651
vxbVxSimIntCtlrRegister() – register vxbVxSimIntCtlr driver	651
wancomEndDbg() – Print pDrvCtrl information regarding Tx ring and Rx queue desc.	652
wancomEndLoad() – initialize the driver and device	652
wdbEndPktDevInit() – initialize an END packet device	653
wdbNetromPktDevInit() – initialize a NETROM packet device for the WDB agent	654
wdbPipePktDevInit() – initialize a pipe packet device	654
wdbSlipPktDevInit() – initialize a SLIP packet device for a WDB agent	655
wdbTsfsDrv() – initialize the TSFS device driver for a WDB agent	655
wdbVioDrv() – initialize the <i>tty</i> driver for a WDB agent	656
xbdAttach() – attach an XBD device	657
xbdBlockSize() – retrieve the block size	657
xbdDetach() – detach an XBD device	658
xbdDump() – XBD dump routine	658
xbdInit() – initialize the XBD library	659
xbdIoctl() – XBD device ioctl routine	659
xbdNBlocks() – retrieve the total number of blocks	659

xbdSize() – retrieve the total number of bytes	660
xbdStrategy() – XBD strategy routine	660
z8530DevInit() – initialize a Z8530_DUSART	661
z8530Int() – handle all interrupts in one vector	661
z8530IntEx() – handle error interrupts	662
z8530IntRd() – handle a receiver interrupt	662
z8530IntWr() – handle a transmitter interrupt	663

_archOptRegRd16_00()

NAME **_archOptRegRd16_00()** – read 16 bits from mem space register

SYNOPSIS

```
STATUS _archOptRegRd16_00
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT16 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 16 bits from a mem space register.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegRd16_01()

NAME **_archOptRegRd16_01()** – read 16 bits from mem space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd16_01
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT16 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 16 bits from a mem space register and swap the data read.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegRd16_20\(\)](#)

NAME [_archOptRegRd16_20\(\)](#) – read 16 bits from IO space register

SYNOPSIS

```
STATUS _archOptRegRd16_20
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT16 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 16 bits from an IO space register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegRd16_21\(\)](#)

NAME [_archOptRegRd16_21\(\)](#) – read 16 bits from IO space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd16_21
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT16 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 16 bits from an IO space register and swap the data read.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegRd32_00()

NAME **_archOptRegRd32_00()** – read 32 bits from mem space register

SYNOPSIS

```
STATUS _archOptRegRd32_00
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT32 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 32 bits from a mem space register.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegRd32_03()

NAME **_archOptRegRd32_03()** – read 32 bits from mem space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd32_03
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT32 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 32 bits from a mem space register and swap the data read.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegRd32_20\(\)](#)

NAME [_archOptRegRd32_20\(\)](#) – read 32 bits from IO space register

SYNOPSIS

```
STATUS _archOptRegRd32_20
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT32 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 32 bits from an IO space register.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegRd32_23\(\)](#)

NAME [_archOptRegRd32_23\(\)](#) – read 32 bits from IO space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd32_23
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT32 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 32 bits from an IO space register and swap the data read.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegRd64_00()

NAME **_archOptRegRd64_00()** – read 64 bits from mem space register

SYNOPSIS

```
STATUS _archOptRegRd64_00
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT64 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 64 bits from a mem space register.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegRd64_07()

NAME **_archOptRegRd64_07()** – read 64 bits from mem space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd64_07
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT64 *       pDataBuf,    /* buffer to put data in */
    UINT32 *       pFlags       /* flags */
)
```

DESCRIPTION This is an optimized function which reads 64 bits from a mem space register and swap the data read.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegRd64_20\(\)](#)

NAME [_archOptRegRd64_20\(\)](#) – read 64 bits from IO space register

SYNOPSIS

```
STATUS _archOptRegRd64_20
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT64 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 64 bits from a IO space register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **[vxbArchAccess](#)**

[_archOptRegRd64_27\(\)](#)

NAME [_archOptRegRd64_27\(\)](#) – read 64 bits from IO space register and swap data

SYNOPSIS

```
STATUS _archOptRegRd64_27
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT64 *       pDataBuf,       /* buffer to put data in */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which reads 64 bits from a IO space register and swap the data read.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **[vxbArchAccess](#)**

_archOptRegRd8_00()

NAME	_archOptRegRd8_00() – read 8 bits from mem space register
SYNOPSIS	<pre>STATUS _archOptRegRd8_00 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT8 * pDataBuf, /* buffer to put data in */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which reads 8 bits from a mem space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegRd8_20()

NAME	_archOptRegRd8_20() – read 8 bits from IO space register
SYNOPSIS	<pre>STATUS _archOptRegRd8_20 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT8 * pDataBuf, /* buffer to put data in */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which reads 8 bits from an IO space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWr16_00\(\)](#)

NAME	<u>_archOptRegWr16_00()</u> – write 16 bits to a mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr16_00 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 16 bit data to the mem space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<u>vxbArchAccess</u>

[_archOptRegWr16_01\(\)](#)

NAME	<u>_archOptRegWr16_01()</u> – swap and write 16 bits to a mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr16_01 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 16 bit data and writes to the mem space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<u>vxbArchAccess</u>

_archOptRegWr16_10()

NAME **_archOptRegWr16_10()** – write 16 bits to a mem space register and flush data

SYNOPSIS

```
STATUS _archOptRegWr16_10
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE::pRegBase[] entry */
    UINT32          byteOffset,     /* offset, in bytes, of register */
    UINT16 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which writes the 16 bit data to the mem space register and flushes the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archOptRegWr16_11()

NAME **_archOptRegWr16_11()** – swap, write 16 bits to mem space register & flush data

SYNOPSIS

```
STATUS _archOptRegWr16_11
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE::pRegBase[] entry */
    UINT32          byteOffset,     /* offset, in bytes, of register */
    UINT16 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which swaps the 16 bit data, writes to the mem space register and flushes the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

[_archOptRegWr16_20\(\)](#)

NAME	<code>_archOptRegWr16_20()</code> – write 16 bits to an IO space register
SYNOPSIS	<pre>STATUS _archOptRegWr16_20 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes the 16 bit data to the IO space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWr16_21\(\)](#)

NAME	<code>_archOptRegWr16_21()</code> – swap and write 16 bits to an IO space register
SYNOPSIS	<pre>STATUS _archOptRegWr16_21 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps and writes the 16 bit data to the IO space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr16_30()

NAME	<code>_archOptRegWr16_30()</code> – write 16 bits to IO space register & flush data
SYNOPSIS	<pre>STATUS _archOptRegWr16_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 16 bits to the IO space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr16_31()

NAME	<code>_archOptRegWr16_31()</code> – swap & write 16 bits to IO space register & flush data
SYNOPSIS	<pre>STATUS _archOptRegWr16_31 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 16 bit data and writes the data to the IO space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWr32_00\(\)](#)

NAME	<u>_archOptRegWr32_00()</u> – write 32 bits to mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr32_00 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 32 bits to the memory space register, without any modifications to the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWr32_03\(\)](#)

NAME	<u>_archOptRegWr32_03()</u> – swap the data and write 32 bits to mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr32_03 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data and then writes to the memory space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegWr32_10()

NAME	<code>_archOptRegWr32_10()</code> – write 32 bits to an mem space register and flush
SYNOPSIS	<pre>STATUS _archOptRegWr32_10 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes to the mem space register and then flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr32_10()

NAME	<code>_archOptRegWr32_10()</code> – swap & write 32 bits to an mem space register and flush
SYNOPSIS	<pre>STATUS _archOptRegWr32_13 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data, writes to the mem space register and then flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWr32_20\(\)](#)

NAME	<u>_archOptRegWr32_20()</u> – write 32 bits to an IO space register
SYNOPSIS	<pre>STATUS _archOptRegWr32_20 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 32 bits to the IO space register, without any modifications to the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWr32_23\(\)](#)

NAME	<u>_archOptRegWr32_23()</u> – swap and write 32 bits to an IO space register
SYNOPSIS	<pre>STATUS _archOptRegWr32_23 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data and then writes to the IO space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegWr32_30()

NAME	<code>_archOptRegWr32_30()</code> – write 32 bits to an IO space register and flush
SYNOPSIS	<pre>STATUS _archOptRegWr32_30 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 32 bit data and then flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr32_33()

NAME	<code>_archOptRegWr32_33()</code> – swap and write 32 bits to an IO space register & flush
SYNOPSIS	<pre>STATUS _archOptRegWr32_33 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data, writes to the IO space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWr64_00\(\)](#)

NAME	<u>_archOptRegWr64_00()</u> – write 64 bits to a mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr64_00 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 64 bit data to the mem space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWr64_07\(\)](#)

NAME	<u>_archOptRegWr64_07()</u> – swap 64 bit data and write to a mem space register
SYNOPSIS	<pre>STATUS _archOptRegWr64_07 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 64 bit data and writes to the mem space register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegWr64_10()

NAME	<code>_archOptRegWr64_10()</code> – write 64 bits to a mem space register and flush data
SYNOPSIS	<pre>STATUS _archOptRegWr64_10 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 64 bit data to the mem space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr64_17()

NAME	<code>_archOptRegWr64_17()</code> – swap, write 64 bits to a mem space register & flush data
SYNOPSIS	<pre>STATUS _archOptRegWr64_17 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 64 bit data to be written, writes to the mem space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWr64_20\(\)](#)

NAME [_archOptRegWr64_20\(\)](#) – write 64 bits to an IO space register

SYNOPSIS

```
STATUS _archOptRegWr64_20
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT64 *       pDataBuf,       /* buffer to copy data from */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which writes 64 bits to the IO space register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **[vxbArchAccess](#)**

[_archOptRegWr64_27\(\)](#)

NAME [_archOptRegWr64_27\(\)](#) – swap and write 64 bits to an IO space register

SYNOPSIS

```
STATUS _archOptRegWr64_27
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT64 *       pDataBuf,       /* buffer to copy data from */
    UINT32 *       pFlags          /* flags */
)
```

DESCRIPTION This is an optimized function which swaps the 64 bit data and writes to the IO space register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **[vxbArchAccess](#)**

_archOptRegWr64_30()

NAME	<code>_archOptRegWr64_30()</code> – write 64 bits to a IO space register and flush data
SYNOPSIS	<pre>STATUS _archOptRegWr64_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 64 bit data to the IO space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr64_37()

NAME	<code>_archOptRegWr64_37()</code> – swap, write 64 bits to a IO space register & flush data
SYNOPSIS	<pre>STATUS _archOptRegWr64_37 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 64 bit data to be written, writes to the IO space register and flushes the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWr8_00()

NAME **_archOptRegWr8_00()** – write 8 bits to mem space register

SYNOPSIS

```
STATUS _archOptRegWr8_00
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT8 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which writes 8 bits to the memory space register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archOptRegWr8_10()

NAME **_archOptRegWr8_10()** – write 8 bits to mem space register and flush data

SYNOPSIS

```
STATUS _archOptRegWr8_10
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT8 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which writes 8 bits to the mem space register and flush the data.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archOptRegWr8_20()

NAME **_archOptRegWr8_20()** – write 8 bits to IO space register

SYNOPSIS

```
STATUS _archOptRegWr8_20
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT8 *        pDataBuf,    /* buffer to copy data from */
    UINT32 *        pFlags      /* flags */
)
```

DESCRIPTION This is an optimized function which writes 8 bits to the IO space register.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegWr8_30()

NAME **_archOptRegWr8_30()** – write 8 bits to IO space register and flush data

SYNOPSIS

```
STATUS _archOptRegWr8_30
(
    VXB_DEVICE_ID pDevInfo,      /* device info */
    void *         pRegBase,     /* VXB_DEVICE::pRegBase[] entry */
    UINT32         byteOffset,   /* offset, in bytes, of register */
    UINT8 *        pDataBuf,    /* buffer to copy data from */
    UINT32 *        pFlags      /* flags */
)
```

DESCRIPTION This is an optimized function which writes 8 bits to the IO space register and flush the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegWrRd16_00\(\)](#)

NAME	<code>_archOptRegWrRd16_00()</code> – write 16 bits to a mem space register & read back
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_00 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 16 bit data to the mem space register and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd16_01\(\)](#)

NAME	<code>_archOptRegWrRd16_01()</code> – swap, write 16 bits to a mem space register & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_01 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 16 bit data, writes to the mem space register and read back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd16_10()

NAME	<code>_archOptRegWrRd16_10()</code> – write 16 bits to memspace, flush data & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_10 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes the 16 bit data to the mem space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd16_11()

NAME	<code>_archOptRegWrRd16_11()</code> – swap, write 16 bits to memspace, flush & read data
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_11 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 16 bit data, writes to the mem space register, flushes the data & reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd16_20\(\)](#)

NAME	<u>_archOptRegWrRd16_20()</u> – write 16 bits to an IO space register & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_20 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes the 16 bit data to the IO space register and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWrRd16_21\(\)](#)

NAME	<u>_archOptRegWrRd16_21()</u> – swap, write 16 bits to an IO space register & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_21 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the data, writes the 16 bit data to the IO space register and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegWrRd16_30()

NAME	<code>_archOptRegWrRd16_30()</code> – write 16 bits to IO space, flush & read back data
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 16 bits to the IO space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd16_31()

NAME	<code>_archOptRegWrRd16_31()</code> – swap, write 16 bits to IO space, flush data & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd16_31 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 16 bit data, writes the data to the IO space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd32_00\(\)](#)

NAME [_archOptRegWrRd32_00\(\)](#) – write 32 bits to mem space register & read back

SYNOPSIS STATUS _archOptRegWrRd32_00
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT32 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which writes 32 bits to the memory space register, without any modifications to the data and reads back the contents.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegWrRd32_03\(\)](#)

NAME [_archOptRegWrRd32_03\(\)](#) – swap, write 32 bits to mem space register & read

SYNOPSIS STATUS _archOptRegWrRd32_03
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT32 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which swaps the 32 bit data, writes to the memory space register and reads back the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegWrRd32_10()

NAME	<code>_archOptRegWrRd32_10()</code> – write 32 bits to memspace register, flush & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd32_10 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes to the mem space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd32_13()

NAME	<code>_archOptRegWrRd32_13()</code> – swap, write 32 bits to memspace, flush & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd32_13 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data, writes to the mem space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd32_20\(\)](#)

NAME [_archOptRegWrRd32_20\(\)](#) – write 32 bits to an IO space register & read back

SYNOPSIS STATUS _archOptRegWrRd32_20
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT32 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which writes 32 bits to the IO space register, without any modifications to the data and reads back the contents.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegWrRd32_23\(\)](#)

NAME [_archOptRegWrRd32_23\(\)](#) – swap, write 32 bits to an IO space register & read

SYNOPSIS STATUS _archOptRegWrRd32_23
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT32 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which swaps the 32 bit data, writes to the IO space register and reads back the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegWrRd32_30()

NAME	<code>_archOptRegWrRd32_30()</code> – write 32 bits to an IO space register, flush & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd32_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 32 bit data, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd32_33()

NAME	<code>_archOptRegWrRd32_33()</code> – swap, write 32 bits to IO space, flush & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd32_33 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 32 bit data, writes to the IO space register, flushes the data and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd64_00\(\)](#)

NAME [_archOptRegWrRd64_00\(\)](#) – write 64 bits to a mem space register and read back

SYNOPSIS STATUS _archOptRegWrRd64_00(
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT64 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which writes 64 bit data to the mem space register and read back the contents.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegWrRd64_07\(\)](#)

NAME [_archOptRegWrRd64_07\(\)](#) – swap 64 bit data and write & read from mem space

SYNOPSIS STATUS _archOptRegWrRd64_07(
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT64 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which swaps the 64 bit data and writes to the mem space register and read back the contents.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegWrRd64_10()

NAME **_archOptRegWrRd64_10()** – write 64 bits to mem space register, flush data & read

SYNOPSIS

```
STATUS _archOptRegWrRd64_10
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE::pRegBase[] entry */
    UINT32          byteOffset,     /* offset, in bytes, of register */
    UINT64 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which writes 64 bit data to the mem space register, flushes the data and reads back the contents.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archOptRegWrRd64_17()

NAME **_archOptRegWrRd64_17()** – swap, write 64 bits to memspace, flush data & read

SYNOPSIS

```
STATUS _archOptRegWrRd64_17
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE::pRegBase[] entry */
    UINT32          byteOffset,     /* offset, in bytes, of register */
    UINT64 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This is an optimized function which swaps the 64 bit data to be written, writes to the mem space register, flushes the data and reads back the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

[_archOptRegWrRd64_20\(\)](#)

NAME	<u>_archOptRegWrRd64_20()</u> – write 64 bits to an IO space register and read back
SYNOPSIS	<pre>STATUS _archOptRegWrRd64_20 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 64 bits to the IO space register and read back the contents.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archOptRegWrRd64_27\(\)](#)

NAME	<u>_archOptRegWrRd64_27()</u> – swap, write 64 bits & read from an IO space register
SYNOPSIS	<pre>STATUS _archOptRegWrRd64_27 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 64 bit data and writes to the IO space register and also reads back the contents.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archOptRegWrRd64_30()

NAME	<code>_archOptRegWrRd64_30()</code> – write 64 bits to IO space register, flush and read
SYNOPSIS	<pre>STATUS _archOptRegWrRd64_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 64 bit data to the IO space register, flushes the data and reads back the contents.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd64_37()

NAME	<code>_archOptRegWrRd64_37()</code> – swap, write 64 bits to IO space, flush & read data
SYNOPSIS	<pre>STATUS _archOptRegWrRd64_37 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which swaps the 64 bit data to be written, writes to the IO space register, flushes the data and reads back the contents.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archOptRegWrRd8_00\(\)](#)

NAME [_archOptRegWrRd8_00\(\)](#) – write 8 bits to mem space register and read back data

SYNOPSIS STATUS _archOptRegWrRd8_00(
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT8 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which writes 8 bits to the memory space register and reads back the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

[_archOptRegWrRd8_10\(\)](#)

NAME [_archOptRegWrRd8_10\(\)](#) – write 8 bits to mem space register, flush data & read

SYNOPSIS STATUS _archOptRegWrRd8_10(
(
 VXB_DEVICE_ID pDeviceInfo, /* device info */
 void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */
 UINT32 byteOffset, /* offset, in bytes, of register */
 UINT8 * pDataBuf, /* buffer to copy data from */
 UINT32 * pFlags /* flags */
)

DESCRIPTION This is an optimized function which writes 8 bits to the mem space register, flushes the data and reads back the data.

RETURNS OK on success, or ERROR otherwise

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

_archOptRegWrRd8_20()

NAME	<code>_archOptRegWrRd8_20()</code> – write 8 bits to IO space register and read back
SYNOPSIS	<pre>STATUS _archOptRegWrRd8_20 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT8 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 8 bits to the IO space register and reads back the data.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archOptRegWrRd8_30()

NAME	<code>_archOptRegWrRd8_30()</code> – write 8 bits to IO space register, flush data & read
SYNOPSIS	<pre>STATUS _archOptRegWrRd8_30 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT8 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This is an optimized function which writes 8 bits to the IO space register, flush the data and read back the contents.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

[_archRegProbe\(\)](#)

NAME	<u>_archRegProbe()</u> – probe a register on the device
SYNOPSIS	<pre>STATUS _archRegProbe (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 transactionSize, /* register size */ char * pProbeDatum, /* value to write */ char * pRetVal, /* value read back */ UINT32 * pFlags /* Flags used */)</pre>
DESCRIPTION	This routine is used by the driver to identify whether the device register is present and is available.
RETURNS	OK on success, or ERROR if register is not present or unavailable
ERRNO	N/A
SEE ALSO	vxbArchAccess

[_archRegisterRead16\(\)](#)

NAME	<u>_archRegisterRead16()</u> – read 16-bit value from a register
SYNOPSIS	<pre>STATUS _archRegisterRead16 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT16 * pDataBuf, /* buffer to put data in */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to read 16-bits from a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archRegisterRead32()

NAME	_archRegisterRead32() – read 32-bit value from a register
SYNOPSIS	<pre>STATUS _archRegisterRead32 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to put data in */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to read 32-bits from a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archRegisterRead64()

NAME	_archRegisterRead64() – read 64-bit value from a register
SYNOPSIS	<pre>STATUS _archRegisterRead64 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to put data in */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to read 64-bits from a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archRegisterRead8()

NAME **_archRegisterRead8()** – read 8-bit value from a register

SYNOPSIS

```
STATUS _archRegisterRead8
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT8 *        pDataBuf,       /* buffer to put data in */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This routine is used by the driver to read 8-bits from a device register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archRegisterWrite16()

NAME **_archRegisterWrite16()** – write 16-bits to a register

SYNOPSIS

```
STATUS _archRegisterWrite16
(
    VXB_DEVICE_ID pDeviceInfo,      /* device info */
    void *         pRegBase,        /* VXB_DEVICE:::pRegBase[] entry */
    UINT32         byteOffset,     /* offset, in bytes, of register */
    UINT16 *        pDataBuf,       /* buffer to copy data from */
    UINT32 *        pFlags         /* flags */
)
```

DESCRIPTION This routine is used by the driver to write 16-bits to a device register.

RETURNS OK on success, or **ERROR** otherwise

ERRNO N/A

SEE ALSO **vxbArchAccess**

_archRegisterWrite32()

NAME	<code>_archRegisterWrite32()</code> – write 32-bits to a register
SYNOPSIS	<pre>STATUS _archRegisterWrite32 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to write 32-bits to a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archRegisterWrite64()

NAME	<code>_archRegisterWrite64()</code> – write 64-bits to a register
SYNOPSIS	<pre>STATUS _archRegisterWrite64 (VXB_DEVICE_ID pDevInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT64 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to write 64-bits to a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	<code>vxbArchAccess</code>

_archRegisterWrite8()

NAME	_archRegisterWrite8() – write 8-bits to a register
SYNOPSIS	<pre>STATUS _archRegisterWrite8 (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT8 * pDataBuf, /* buffer to copy data from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used by the driver to write 8-bits to a device register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archVolatileRegisterRead()

NAME	_archVolatileRegisterRead() – read from a volatile register
SYNOPSIS	<pre>STATUS _archVolatileRegisterRead (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE:::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 transactionSize, /* transaction size, in bytes */ char * pDataBuf, /* buffer to read-from */ UINT32 * pFlags /* flags */)</pre>
DESCRIPTION	This routine is used to read from a volatile register. This function does not split the read into multiple transactions.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

_archVolatileRegisterWrite()

NAME	_archVolatileRegisterWrite() – write to a volatile register
SYNOPSIS	STATUS _archVolatileRegisterWrite (VXB_DEVICE_ID pDeviceInfo, /* device info */ void * pRegBase, /* VXB_DEVICE::pRegBase[] entry */ UINT32 byteOffset, /* offset, in bytes, of register */ UINT32 transactionSize, /* transaction size, in bytes */ char * pDataBuf, /* buffer to read-from/write-to */ UINT32 * pFlags /* flags */)
DESCRIPTION	This routine is used to write to a volatile register and read back the data from the volatile register.
RETURNS	OK on success, or ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbArchAccess

ambaDevInit()

NAME	ambaDevInit() – initialise an AMBA channel
SYNOPSIS	void ambaDevInit (AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */)
DESCRIPTION	This routine initialises some SIO_CHAN function pointers and then resets the chip to a quiescent state. Before this routine is called, the BSP must already have initialised all the device addresses, etc. in the AMBA_CHAN structure.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ambaSio

ambaIntRx()

NAME	ambaIntRx() – handle a receiver interrupt
SYNOPSIS	<pre>void ambaIntRx (AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */)</pre>
DESCRIPTION	This routine handles read interrupts from the UART.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ambaSio

ambaIntTx()

NAME	ambaIntTx() – handle a transmitter interrupt
SYNOPSIS	<pre>void ambaIntTx (AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */)</pre>
DESCRIPTION	This routine handles write interrupts from the UART.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ambaSio

amd8111LanDumpPrint()

NAME	amd8111LanDumpPrint() – Display statistical counters
SYNOPSIS	<pre>STATUS amd8111LanDumpPrint</pre>

```
(  
int unit /* pointer to DRV_CTRL structure */  
)
```

DESCRIPTION This routine displays i82557 statistical counters

RETURNS OK, or ERROR if the DUMP command failed.

ERRNO Not Available

SEE ALSO [amd8111LanEnd](#)

amd8111LanEndLoad()

NAME [amd8111LanEndLoad\(\)](#) – initialize the driver and device

SYNOPSIS END_OBJ * amd8111LanEndLoad
(
char * initString /* string to be parse by the driver */
)

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

```
<unit:devMemAddr:devIoAddr:pciMemBase:vecnum:intLvl:memAdrs  
:memSize:memWidth:csr3b:offset:flags>
```

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "lnPci") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

ERRNO Not Available

SEE ALSO [amd8111LanEnd](#)

amd8111LanErrCounterDump()

NAME	amd8111LanErrCounterDump() – dump statistical counters
SYNOPSIS	<pre>STATUS amd8111LanErrCounterDump (AMD8111_LAN_DRV_CTRL * pDrvCtrl, /* pointer to DRV_CTRL structure */ UINT32 * memAddr /* pointer to receive stat data */)</pre>
DESCRIPTION	<p>This routine dumps statistical counters for the purpose of debugging and tuning the 82557. The <i>memAddr</i> parameter is the pointer to an array of 68 bytes in the local memory. This memory region must be allocated before this routine is called. The memory space must also be DWORD (4 bytes) aligned. When the last DWORD (4 bytes) is written to a value, 0xa007, it indicates the dump command has completed. To determine the meaning of each statistical counter, see the Intel 82557 manual.</p>
RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	amd8111LanEnd

ataBlkRW()

NAME	ataBlkRW() – read or write sectors to a ATA/IDE disk.
SYNOPSIS	<pre>STATUS ataBlkRW (ATA_DEV *pDev, sector_t startBlk, UINT32 nBlks, char *pBuf, int direction)</pre>
DESCRIPTION	<p>Read or write sectors to a ATA/IDE disk. <i>startBlk</i> is the start Block, <i>nBlks</i> is the number of blocks, <i>pBuf</i> is data buffer pointer and <i>direction</i> is the direction either to read or write. It should be O_WRONLY for data write to drive or O_RDONLY for read data from drive.</p>
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available

SEE ALSO**ataDrv**

ataCmd()

NAME**ataCmd()** – issue a RegisterFile command to ATA/ATAPI device.**SYNOPSIS**

```
STATUS ataCmd
(
    int ctrl,    /* Controller number. 0 or 1 */
    int drive,   /* Drive number.      0 or 1 */
    int cmd,     /* Command Register */
    int arg0,    /* argument0 */
    int arg1,    /* argument1 */
    int arg2,    /* argument2 */
    int arg3,    /* argument3 */
    int arg4,    /* argument4 */
    int arg5    /* argument5 */
)
```

DESCRIPTION

This function executes ATA command to ATA/ATAPI devices specified by arguments *ctrl* and *drive*. *cmd* is command to be executed and other arguments *arg0* to *arg5* are interpreted differently in each case depending on the *cmd* command. Some commands (like **ATA_CMD_SET_FEATURE**) have sub commands the case in which *arg0* is interpreted as subcommand and *arg1* is subcommand specific.

In general these arguments *arg0* to *arg5* are interpreted as command registers of the device as mentioned below.

- arg0** - Feature Register
- arg1** - Sector count
- arg2** - Sector number
- arg3** - CylLo
- arg4** - CylHi
- arg5** - sdh Register

As these registers are interpreted for different purpose for each command, arguments are not named after registers.

The following commands are valid in this function and the validity of each argument for different commands. Each command is tabulated in the form

COMMAND	ARG0		ARG1		ARG2		ARG3		ARG4		ARG5
---------	------	--	------	--	------	--	------	--	------	--	------

ataCmd()

ATA_CMD_INITP	0	0	0	0	0
ATA_CMD_RECALIB	0	0	0	0	0
ATA_PI_CMD_SRST	0	0	0	0	0
ATA_CMD_EXECUTE_DEVICE_DIAGNOSTIC	0	0	0	0	0
ATA_CMD_SEEK	cylinder or LBA high	head LBA low	0	0	0
ATA_CMD_SET_FEATURE	FR (SUBCOMMAND)	SC (SubCommand Specific Value)	0	0	0
ATA_CMD_SET_MULTI	sectors per block	0	0	0	0
ATA_CMD_IDLE	SC (Timer Period)	0	0	0	0
ATA_CMD_STANDBY	SC (Timer Period)	0	0	0	0
ATA_CMD_STANDBY_IMMEDIATE	0	0	0	0	0
ATA_CMD_SLEEP	0	0	0	0	0
ATA_CMD_CHECK_POWER_MODE	0	0	0	0	0
ATA_CMD_IDLE_IMMEDIATE	0	0	0	0	0
ATA_CMD_SECURITY_DISABLE_PASSWORD	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_SECURITY_ERASE_PREPARE	0	0	0	0	0
ATA_CMD_SECURITY_ERASE_UNIT	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_SECURITY_FREEZE_LOCK	0	0	0	0	0
ATA_CMD_SECURITY_SET_PASSWORD	0	0	0	0	0
ATA_CMD_SECURITY_UNLOCK	0	0	0	0	0
ATA_CMD_SMART (not implemented)	FR (SUBCOMMAND)	SC (SubCommand Specific Value)	SN (SubCommand Specific Value)	ATA_ZERO	ATA_ZERO
ATA_CMD_GET_MEDIA_STATUS	0	0	0	0	0
ATA_CMD_MEDIA_EJECT					

0	0	0	0	0	0
ATA_CMD_MEDIA_LOCK	0	0	0	0	0
ATA_CMD_MEDIA_UNLOCK	0	0	0	0	0
ATA_CMD_CFA_ERASE_SECTORS	0	0	0	0	0
ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE	ATA_ZERO	SC	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE	ATA_ZERO	SC	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_CFA_TRANSLATE_SECTOR	ATA_ZERO	ATA_ZERO	SN	cylLo	cylHi
ATA_CMD_CFA_REQUEST_EXTENDED_ERROR_CODE	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_SET_MAX	FR (SUBCOMMAND)	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO

The following are the subcommands valid for ATA_CMD_SET_MAX and are tabulated as below

SUBCOMMAND (in ARG0)					
ARG1	ARG2	ARG3	ARG4	ARG5	
ATA_SUB_SET_MAX_ADDRESS	SC (SET_MAX_VOLATILE or SET_MAX_NON_VOLATILE)	sector no	cylLo	cylHi	head + modebit
ATA_SUB_SET_MAX_SET_PASS	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_SUB_SET_MAX_LOCK	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_SUB_SET_MAX_UNLOCK	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_SUB_SET_MAX_FREEZE_LOCK	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO

In ATA_CMD_SET_FEATURE subcommand only arg0 and arg1 are valid, all other are ATA_ZERO.

SUBCOMMAND (ARG0)		ARG1
ATA_SUB_ENABLE_8BIT		ATA_ZERO
ATA_SUB_ENABLE_WCACHE		ATA_ZERO
ATA_SUB_SET_RWMODE		mode (see page no 168 table 28 in atapi Spec5) 0x90
ATA_SUB_ENB_ADV_POW_MNGMNT		
ATA_SUB_ENB_POW_UP_STDBY		ATA_ZERO

ATA_SUB_POW_UP_STDBY_SPIN	ATA_ZERO
ATA_SUB_BOOTMETHOD	ATA_ZERO
ATA_SUB_ENA_CFA_POW_MOD1	ATA_ZERO
ATA_SUB_DISABLE_NOTIFY	ATA_ZERO
ATA_SUB_DISABLE_RETRY	ATA_ZERO
ATA_SUB_SET_LENGTH	ATA_ZERO
ATA_SUB_SET_CACHE	ATA_ZERO
ATA_SUB_DISABLE_LOOK	ATA_ZERO
ATA_SUB_ENA_INTR_RELEASE	ATA_ZERO
ATA_SUB_ENA_SERV_INTR	ATA_ZERO
ATA_SUB_DISABLE_REV	ATA_ZERO
ATA_SUB_DISABLE_ECC	ATA_ZERO
ATA_SUB_DISABLE_8BIT	ATA_ZERO
ATA_SUB_DISABLE_WCACHE	ATA_ZERO
ATA_SUB_DIS_ADV_POW_MNGMT	ATA_ZERO
ATA_SUB_DISB_POW_UP_STDBY	ATA_ZERO
ATA_SUB_ENABLE_ECC	ATA_ZERO
ATA_SUB_BOOTMETHOD_REPORT	ATA_ZERO
ATA_SUB_DIS_CFA_POW_MOD1	ATA_ZERO
ATA_SUB_ENABLE_NOTIFY	ATA_ZERO
ATA_SUB_ENABLE_RETRY	ATA_ZERO
ATA_SUB_ENABLE_LOOK	ATA_ZERO
ATA_SUB_SET_PREFETCH	ATA_ZERO
ATA_SUB_SET_4BYTES	ATA_ZERO
ATA_SUB_ENABLE_REV	ATA_ZERO
ATA_SUB_DIS_INTR_RELEASE	ATA_ZERO
ATA_SUB_DIS_SERV_INTR	ATA_ZERO

RETURNS OK, ERROR if the command didn't succeed.

ERRNO Not Available

SEE ALSO [ataDrv](#)

ataCtrlReset()

NAME	ataCtrlReset() – reset the specified ATA/IDE disk controller
SYNOPSIS	STATUS ataCtrlReset (int ctrl)
DESCRIPTION	This routine resets the ATA controller specified by <i>ctrl</i> . The device control register is written with SRST=1
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataDevCreate()

NAME	ataDevCreate() – create a device for a ATA/IDE disk
SYNOPSIS	BLK_DEV * ataDevCreate (int ctrl, /* ATA controller number, 0 is the primary controller */ int drive, /* ATA drive number, 0 is the master drive */ UINT32 nBlocks, /* number of blocks on device, 0 = use entire disc */ UINT32 blkOffset /* offset BLK_DEV nBlocks from the start of the drive */)
DESCRIPTION	This routine creates a device for a specified ATA/IDE or ATAPI CDROM disk. <i>ctrl</i> is a controller number for the ATA controller; the primary controller is 0. The maximum is specified via ATA_MAX_CTRLS . <i>drive</i> is the drive number for the ATA hard drive; the master drive is 0. The maximum is specified via ATA_MAX_DRIVES . The <i>nBlocks</i> parameter specifies the size of the device in blocks. If <i>nBlocks</i> is zero, the whole disk is used. The <i>blkOffset</i> parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the hard disk. This offset is added to the block numbers passed by

the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.)

RETURNS	A pointer to a block device structure (BLK_DEV) or NULL if memory cannot be allocated for the device structure.
ERRNO	Not Available
SEE ALSO	ataDrv , dosFsMkfs() , dosFsDevInit() , rawFsDevInit()

ataDevIdentify()

NAME	ataDevIdentify() – identify device
SYNOPSIS	<pre>STATUS ataDevIdentify (int ctrl, int dev)</pre>
DESCRIPTION	This routine checks whether the device is connected to the controller, if it is, this routine determines drive type. The routine set type field in the corresponding ATA_DRIVE structure. If device identification failed, the routine set state field in the corresponding ATA_DRIVE structure to ATA_DEV_NONE .
RETURNS	TRUE if a device present, FALSE otherwise
ERRNO	Not Available
SEE ALSO	ataDrv

ataDmaRW()

NAME	ataDmaRW() – read/write a number of sectors on the current track in DMA mode
SYNOPSIS	<pre>STATUS ataDmaRW (int ctrl, int drive, UINT32 cylinder,</pre>

```
    UINT32    head,
    UINT32    sector,
    void *    buffer,
    UINT32    nSecs,
    int       direction,
    sector_t  startBlk
)
```

DESCRIPTION	Read/write a number of sectors on the current track in DMA mode
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataDmaToggle()

NAME	ataDmaToggle() – turn on or off an individual controllers dma support
SYNOPSIS	void ataDmaToggle (int ctrl)
DESCRIPTION	This routine lets you toggle the DMA setting for an individual controller. The controller number is passed in as a parameter, and the current value is toggled.
RETURNS	OK, or ERROR if the parameters are invalid.
ERRNO	Not Available
SEE ALSO	ataShow

ataDrv()

NAME	ataDrv() – Initialize the ATA driver
SYNOPSIS	STATUS ataDrv (int ctrl, /* controller no. 0,1 */

```

int drives,      /* number of drives 1,2 */
int vector,      /* interrupt vector      */
int level,       /* interrupt level      */
int configType, /* configuration type   */
int semTimeout, /* timeout seconds for sync semaphore */
int wdgTimeout  /* timeout seconds for watch dog      */
)

```

DESCRIPTION

This routine initializes the ATA/ATAPI device driver, initializes IDE host controller and sets up interrupt vectors for requested controller. This function must be called once for each controller, before any access to drive on the controller, usually which is called by **usrRoot()** in **usrConfig.c**.

If it is called more than once for the same controller, it returns **OK** with a message display **Host controller already initialized**, and does nothing as already required initialization is done.

Additionally it identifies devices available on the controller and initializes depending on the type of the device (ATA or ATAPI). Initialization of device includes reading parameters of the device and configuring to the defaults.

RETURNS

OK, or **ERROR** if initialization fails.

ERRNO

Not Available

SEE ALSO

ataDrv, **ataDevCreate()**

ataDumpPartTable()

NAME

ataDumpPartTable() – dump the partition table from sector 0

SYNOPSIS

```

STATUS ataDumpPartTable
(
    int ctrl,
    int port
)

```

DESCRIPTION

of the specified hard disk

This routine will display on stdout, the partition table of the specified hard drive.

INPUTS

ctrl - controller number of the ata driver 0 or 1
 port - disk number of the drive 0 or 1

RETURNS OK or ERROR

ERRNO Not Available

SEE ALSO **ataShow**

ataDumpTest()

NAME **ataDumpTest()** – a quick test of the dump functionality for ATA driver

SYNOPSIS

```
void ataDumpTest
(
    device_t d,
    sector_t sector,
    UINT32   blocks,
    char     *data
)
```

DESCRIPTION *device_t* device id of the device to dump to. This can be any XBD device. Could be the XBD of the disk device itself, or could be the xbd of a partition overlayed on the drive.

sector sector offset to begin dump relative to start of xbd. *blocks* number of blocks to dump to device **data* buffer that contains data to dump

RETURNS N/A

ERRNO Not Available

SEE ALSO **ataShow**

ataInit()

NAME **ataInit()** – initialize ATA device.

SYNOPSIS

```
STATUS ataInit
(
    int ctrl,
```

```
    int drive
)
```

DESCRIPTION	This routine issues a soft reset command to ATA device for initialization.
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataParamRead()

NAME	ataParamRead() – Read drive parameters
SYNOPSIS	<pre>STATUS ataParamRead (int ctrl, int drive, void *buffer, int command)</pre>
DESCRIPTION	Read drive parameters.
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataPiInit()

NAME	ataPiInit() – init a ATAPI CD-ROM disk controller
SYNOPSIS	<pre>STATUS ataPiInit (int ctrl, int drive)</pre>

DESCRIPTION	This routine resets a ATAPI CD-ROM disk controller.
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataRW()

NAME	ataRW() – read/write a data from/to required sector.
SYNOPSIS	<pre>STATUS ataRW (int ctrl, int drive, UINT32 cylinder, UINT32 head, UINT32 sector, void * buffer, UINT32 nSecs, int direction, sector_t startBlk)</pre>
DESCRIPTION	Read/write a number of sectors on the current track
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

ataRawio()

NAME	ataRawio() – do raw I/O access
SYNOPSIS	<pre>STATUS ataRawio (int ctrl, int drive,</pre>

```
    ATA_RAW *pAtaRaw  
)
```

DESCRIPTION	This routine is called to perform raw I/O access. <i>drive</i> is a drive number for the hard drive: it must be 0 or 1. The <i>pAtaRaw</i> is a pointer to the structure <code>ATA_RAW</code> which is defined in <code>ataDrv.h</code> .
RETURNS	<code>OK</code> , or <code>ERROR</code> if the parameters are not valid.
ERRNO	Not Available
SEE ALSO	<code>ataDrv</code>

ataShow()

NAME	<code>ataShow()</code> – show the ATA/IDE disk parameters
SYNOPSIS	<pre>STATUS ataShow (int ctrl, int drive)</pre>
DESCRIPTION	This routine shows the ATA/IDE disk parameters. Its first argument is a controller number, 0 or 1; the second argument is a drive number, 0 or 1.
RETURNS	<code>OK</code> , or <code>ERROR</code> if the parameters are invalid.
ERRNO	Not Available
SEE ALSO	<code>ataShow</code>

ataShowInit()

NAME	<code>ataShowInit()</code> – initialize the ATA/IDE disk driver show routine
SYNOPSIS	<code>STATUS ataShowInit (void)</code>

DESCRIPTION	This routine links the ATA/IDE disk driver show routine into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:
	<ul style="list-style-type: none">- If you use the configuration header files, define INCLUDE_SHOW_ROUTINES in config.h.- If you use the Tornado project facility, select INCLUDE_ATA_SHOW.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ataShow

ataStatusChk()

NAME	ataStatusChk() – Check status of drive and compare to requested status.
SYNOPSIS	<pre>STATUS ataStatusChk (ATA_CTRL * pCtrl, UINT8 mask, UINT8 status)</pre>
DESCRIPTION	Wait until the drive is ready.
RETURNS	OK, ERROR if the drive status check times out.
ERRNO	Not Available
SEE ALSO	ataDrv

ataXbdDevCreate()

NAME	ataXbdDevCreate() – create an XBD device for a ATA/IDE disk
SYNOPSIS	<code>device_t ataXbdDevCreate</code>

```
(  
    int      ctrl,        /* ATA controller number, 0 is the primary  
controller */  
    int      drive,       /* ATA drive number, 0 is the master drive */  
    uint32   nBlocks,     /* number of blocks on device, 0 = use entire  
disc */  
    uint32   blkOffset,   /* offset BLK_DEV nBlocks from the start of the  
drive */  
    const char * name    /* name of xbd device to create */  
)
```

DESCRIPTION Use the existing code to create a standard block dev device, then create an XBD device associated with the BLKDEV.

RETURNS a device identifier upon success, or NULLDEV otherwise

ERRNO

SEE ALSO [ataDrv](#)

ataXbdRawio()

NAME [ataXbdRawio\(\)](#) – do raw I/O access

SYNOPSIS STATUS ataXbdRawio
(
 device_t device,
 sector_t sector,
 uint32 numSecs,
 char *data,
 int direction
)

DESCRIPTION This routine is called to perform raw I/O access.

device is the XBD device identifier for the drive *sector* starting sector for I/O operation
numSecs number of sectors to read/write *data* pointer to data buffer *dir* read or write

The *pAtaRaw* is a pointer to the structure **ATA_RAW** which is defined in [ataDrv.h](#).

RETURNS OK, or ERROR if the parameters are not valid.

ERRNO Not Available

SEE ALSO [ataDrv](#)

atapiBytesPerSectorGet()

NAME **atapiBytesPerSectorGet()** – get the number of Bytes per sector.

SYNOPSIS

```
UINT16 atapiBytesPerSectorGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function will return the number of Bytes per sector. This function will return correct values for drives of ATA/ATAPI-4 or less as this field is retired for the drives compliant to ATA/ATAPI-5 or higher.

RETURNS Bytes per sector.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiBytesPerTrackGet()

NAME **atapiBytesPerTrackGet()** – get the number of Bytes per track.

SYNOPSIS

```
UINT16 atapiBytesPerTrackGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function will return the number of Bytes per track. This function will return correct values for drives of ATA/ATAPI-4 or less as this feild is retired for the drives compliant to ATA/ATAPI-5 or higher.

RETURNS Bytes per track.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiCtrlMediumRemoval()

NAME	atapiCtrlMediumRemoval() – Issues PREVENT/ALLOW MEDIUM REMOVAL packet command
SYNOPSIS	<pre>STATUS atapiCtrlMediumRemoval (ATA_DEV * pAtapiDev, int arg0)</pre>
DESCRIPTION	This function issues a command to drive to PREVENT or ALLOW MEDIA removal. Argument <i>arg0</i> selects to LOCK_EJECT or UNLOCK_EJECT . To lock media eject <i>arg0</i> should be LOCK_EJECT To unload media eject <i>arg0</i> should be UNLOCK_EJECT
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiCurrentCylinderCountGet()

NAME	atapiCurrentCylinderCountGet() – get logical number of cylinders in the drive.
SYNOPSIS	<pre>UINT16 atapiCurrentCylinderCountGet (int ctrl, int drive)</pre>
DESCRIPTION	This function will return the number of logical cylinders in the drive. This value represents the no of cylinders that can be addressed.
RETURNS	Cylinder count.
ERRNO	Not Available
SEE ALSO	ataShow

atapiCurrentHeadCountGet()

NAME	atapiCurrentHeadCountGet() – get the number of read/write heads in the drive.
SYNOPSIS	<pre>UINT8 atapiCurrentHeadCountGet (int ctrl, int drive)</pre>
DESCRIPTION	This function will return the number of heads in the drive from device structure.
RETURNS	Number of heads.
ERRNO	Not Available
SEE ALSO	ataShow

atapiCurrentMDmaModeGet()

NAME	atapiCurrentMDmaModeGet() – get the enabled Multi word DMA mode.
SYNOPSIS	<pre>UINT8 atapiCurrentMDmaModeGet (int ctrl, int drive)</pre>
DESCRIPTION	This function is used to get drive MDMA mode enable in the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure. The following bit is set for corresponding mode selected.
	<ul style="list-style-type: none">- Bit2 Multi DMA mode 2 is Selected- Bit1 Multi DMA mode 1 is Selected- Bit0 Multi DMA mode 0 is Selected
RETURNS	Enabled Multi word DMA mode.
ERRNO	Not Available
SEE ALSO	ataShow

atapiCurrentPioModeGet()

NAME **atapiCurrentPioModeGet()** – get the enabled PIO mode.

SYNOPSIS

```
UINT8 atapiCurrentPioModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive current PIO mode enabled in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure.

RETURNS Enabled PIO mode.

ERRNO Not Available

SEE ALSO **ataShow**

atapiCurrentRwModeGet()

NAME **atapiCurrentRwModeGet()** – get the current Data transfer mode.

SYNOPSIS

```
UINT8 atapiCurrentRwModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function will return the current Data transfer mode if it is PIO 0,1,2,3,4 mode, SDMA 0,1,2 mode, MDMA 0,1,2 mode or UDMA 0,1,2,3,4,5 mode.

RETURNS current PIO mode.

ERRNO Not Available

SEE ALSO **ataShow**

atapiCurrentSDmaModeGet()

NAME	atapiCurrentSDmaModeGet() – get the enabled Single word DMA mode.
SYNOPSIS	<pre>UINT8 atapiCurrentSDmaModeGet (int ctrl, int drive)</pre>
DESCRIPTION	This function is used to get drive SDMA mode enable in the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure
RETURNS	Enabled Single word DMA mode.
ERRNO	Not Available
SEE ALSO	ataShow

atapiCurrentUDmaModeGet()

NAME	atapiCurrentUDmaModeGet() – get the enabled Ultra DMA mode.
SYNOPSIS	<pre>UINT8 atapiCurrentUDmaModeGet (int ctrl, int drive)</pre>
DESCRIPTION	This function is used to get drive UDMA mode enable in the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure The following bit is set for corresponding mode selected. <ul style="list-style-type: none">- Bit4 Ultra DMA mode 4 is Selected- Bit3 Ultra DMA mode 3 is Selected- Bit2 Ultra DMA mode 2 is Selected- Bit1 Ultra DMA mode 1 is Selected- Bit0 Ultra DMA mode 0 is Selected
RETURNS	Enabled Ultra DMA mode.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiCylinderCountGet()

NAME **atapiCylinderCountGet()** – get the number of cylinders in the drive.

SYNOPSIS

```
UINT16 atapiCylinderCountGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get cylinder count of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure.

RETURNS Cylinder count.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiDriveSerialNumberGet()

NAME **atapiDriveSerialNumberGet()** – get the drive serial number.

SYNOPSIS

```
char * atapiDriveSerialNumberGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive serial number of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 20 bytes length which contains serial number in ascii.

RETURNS Drive serial number.

ERRNO Not Available

SEE ALSO

[ataShow](#)

2

atapiDriveTypeGet()

NAME [atapiDriveTypeGet\(\)](#) – get the drive type.

SYNOPSIS

```
UINT8 atapiDriveTypeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function routine will return the type of the drive if it is CD-ROM or Printer etc. The following table indicates the type depending on the return value.

0x00h	Direct-access device
0x01h	Sequential-access device
0x02h	Printer Device
0x03h	Processor device
0x04h	Write-once device
0x05h	CD-ROM device
0x06h	Scanner device
0x07h	Optical memory device
0x08h	Medium Change Device
0x09h	Communications device
0x0Ch	Array Controller Device
0x0Dh	Encloser Services Device
0x0Eh	Reduced Block Command Devices
0x0Fh	Optical Card Reader/Writer Device
0x1Fh	Unknown or no device type

RETURNS drive type.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiFeatureEnabledGet()

NAME [atapiFeatureEnabledGet\(\)](#) – get the enabled features.

SYNOPSIS	<pre>UINT32 atapiFeatureEnabledGet (int ctrl, int drive)</pre>
DESCRIPTION	This function is used to get drive Features Enabled by the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure. It returns a 32 bit value whose bits represents the features Enabled. The following table gives the cross reference for the bits.
Bit 21	Power-up in Standby Feature
Bit 20	Removable Media Status Notification Feature
Bit 19	Adavanced Power Management Feature
Bit 18	CFA Feature
Bit 10	Host protected Area Feature
Bit 4	Packet Command Feature
Bit 3	Power Management Feature
Bit 2	Removable Media Feature
Bit 1	Security Mode Feature
Bit 0	SMART Feature
RETURNS	enabled features.
ERRNO	Not Available
SEE ALSO	ataShow

atapiFeatureSupportedGet()

NAME	atapiFeatureSupportedGet() – get the features supported by the drive.
SYNOPSIS	<pre>UINT32 atapiFeatureSupportedGet (int ctrl, int drive)</pre>
DESCRIPTION	This function is used to get drive Feature supported by the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure. It returns a 32 bit value whose bits represents the features supported. The following table gives the cross reference for the bits.
Bit 21	Power-up in Standby Feature
Bit 20	Removable Media Status Notification Feature
Bit 19	Adavanced Power Management Feature
Bit 18	CFA Feature

Bit 10	Host protected Area Feature
Bit 4	Packet Command Feature
Bit 3	Power Management Feature
Bit 2	Removable Media Feature
Bit 1	Security Mode Feature
Bit 0	SMART Feature

RETURNS Supported features.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiFirmwareRevisionGet()

NAME [atapiFirmwareRevisionGet\(\)](#) – get the firm ware revision of the drive.

SYNOPSIS

```
char * atapiFirmwareRevisionGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive Firmware revision of the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 8 bytes length which contains serial number in ascii.

RETURNS firmware revision.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiHeadCountGet()

NAME [atapiHeadCountGet\(\)](#) – get the number heads in the drive.

SYNOPSIS

```
UINT8 atapiHeadCountGet
(
    int ctrl,
```

```
    int drive
)
```

DESCRIPTION	This function is used to get head count of the ATA/ATAPI drive specified by <i>ctrl</i> and <i>drive</i> from drive structure.
RETURNS	Number of heads in the drive.
ERRNO	Not Available
SEE ALSO	ataShow

atapiInit()

NAME	atapiInit() – init ATAPI CD-ROM disk controller
SYNOPSIS	<pre>STATUS atapiInit (int ctrl, int drive)</pre>
DESCRIPTION	This routine resets the ATAPI CD-ROM disk controller.
RETURNS	OK, ERROR if the command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

atapiIocctl()

NAME	atapiIocctl() – Control the drive.
SYNOPSIS	<pre>STATUS atapiIocctl (int function, /* The IO operation to do */ int ctrl, /* Controller number of the drive */ int drive, /* Drive number */ int password [16], /* Password to set. NULL if not applicable */</pre>

```
int arg0,      /* 1st arg to pass. NULL if not
applicable */
UINT32 * arg1,      /* Ptr to 2nd arg. NULL if not
applicable */
UINT8 ** ppBuf      /* The data buffer */
)
```

DESCRIPTION This routine is used to control the drive like setting the password, putting in power save mode, locking/unlocking the drive, ejecting the medium etc. The argument *function* defines the ioctl command, *password*, and integer array is the password required or set password value for some commands. Arguments *arg0*, pointer *arg1*, pointer to pointer buffer *ppBuf* are command specific.

The following commands are supported for various functionality.

IOCTL_DIS_MASTER_PWD

Disable the master password. where 4th parameter is the master password.

IOCTL_DIS_USER_PWD

Disable the user password.

IOCTL_ERASE_PREPARE

Prepare the drive for erase incase the user password lost, and it is in max security mode.

IOCTL_ENH_ERASE_UNIT_USR

Erase in enhanced mode supplying the user password.

IOCTL_ENH_ERASE_UNIT_MSTR

Erase in enhanced mode supplying the master password.

IOCTL_NORMAL_ERASE_UNIT_MSTR

Erase the drive in normal mode supplying the master password.

IOCTL_NORMAL_ERASE_UNIT_USR

Erase the drive in normal mode supplying the user password.

IOCTL_FREEZE_LOCK

Freeze lock the drive.

IOCTL_SET_PASS_MSTR

Set the master password.

IOCTL_SET_PASS_USR_MAX

Set the user password in Maximum security mode.

IOCTL_SET_PASS_USR_HIGH

Set the user password in High security mode.

IOCTL_UNLOCK_MSTR

Unlock the master password.

IOCTL_UNLOCK_USR

Unlock the user password.

IOCTL_CHECK_POWER_MODE

Find the drive power saving mode.

IOCTL_IDLE_IMMEDIATE

Idle the drive immediatly. this will get the drive from the standby or active mode to idle mode immediatly.

IOCTL_SLEEP

Set the drive in sleep mode. this is the highest power saving mode. to return to the normal active or IDLE mode, drive need an hardware reset or power on reset or device reset command.

IOCTL_STANDBY_IMMEDIATE

Standby the drive immediatly.

IOCTL_EJECT_DISK

Eject the media of an ATA drive. Use IOsystem ioctl function for ATAPI drive.

IOCTL_GET_MEDIA_STATUS

Find the media status.

IOCTL_ENA_REMOVE_NOTIFY

Enable the drive's removable media notification feature set.

The following table describes these arguments validity. These are tabulated in the following form

FUNCTION	password [16]	arg0	*arg1	**ppBuf
IOCTL_DIS_MASTER_PWD	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_DIS_USER_PWD	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_ERASE_PREPARE	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_ENH_ERASE_UNIT_USR	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_ENH_ERASE_UNIT_MSTR	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_NORMAL_ERASE_UNIT_MSTR	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_NORMAL_ERASE_UNIT_USR	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_FREEZE_LOCK	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SET_PASS_MSTR	password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SET_PASS_USR_MAX	password	ATA_ZERO	ATA_ZERO	ATA_ZERO

IOCTL_SET_PASS_USR_HIGH password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_UNLOCK_MSTR password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_UNLOCK_USR password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_READ_NATIVE_MAX_ADDRESS ATA_ZERO	(ATA_SDH_IBM or ATA_SDH_LBA)	- it returns address in <arg1> LBA/CHS add (LBA 27:24 / Head LBA 23:16 / cylHi LBA 15:8 / cylLow LBA 7:0 / sector no)	ATA_ZERO
IOCTL_SET_MAX_ADDRESS ATA_ZERO	SET_MAX_VOLATILE or SET_MAX_NON_VOLATILE	- <arg1> is pointer to LBA address LBA address	ATA_ZERO
IOCTL_SET_MAX_SET_PASS password	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SET_MAX_LOCK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SET_MAX_UNLOCK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SET_MAX_FREEZE_LOCK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_CHECK_POWER_MODE ATA_ZERO	-	returns power mode in <arg1> power modes : 1) 0x00 Device in standby mode 2) 0x80 Device in Idle mode 3) 0xff Device in Active or Idle mode	ATA_ZERO
IOCTL_IDLE_IMMEDIATE ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SLEEP ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_STANDBY_IMMEDIATE ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_ENB_POW_UP_STDBY ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_ENB_SET_ADV_POW_MNGMNT ATA_ZERO	arg0	ATA_ZERO	ATA_ZERO
NOTE:- arg0 value - 1). for minimum power consumption with standby 0x01h 2). for minimum power consumption without standby 0x01h 3). for maximum performance 0xFEh			
IOCTL_DISABLE_ADV_POW_MNGMNT ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_EJECT_DISK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_LOAD_DISK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_MEDIA_LOCK ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_MEDIA_UNLOCK			

atapiioctl()

ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_GET_MEDIA_STATUS ATA_ZERO	- returns status in <arg1> ATA_ZERO	status	ATA_ZERO
NOTE: value in <arg1> is			
0x04	-Command aborted		
0x02	-No media in drive		
0x08	-Media change is requested		
0x20	-Media changed		
0x40	-Write Protected		
IOCTL_ENA_REMOVE_NOTIFY ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_DISABLE_REMOVE_NOTIFY ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_DISABLE_OPER ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_ENABLE_ATTRIB_AUTO ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_DISABLE_ATTRIB_AUTO ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_ENABLE_OPER ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_OFFLINE_IMMED ATA_ZERO	SubCommand (refer to ref1 page no 190)	ATA_ZERO	ATA_ZERO
IOCTL_SMART_READ_DATA ATA_ZERO	- returns pointer to pointer <ppBuf> of read data ATA_ZERO	read data	
IOCTL_SMART_READ_LOG_SECTOR ATA_ZERO	no of sector to be read	log Address	read data
IOCTL_SMART_RETURN_STATUS ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_SAVE_ATTRIB ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_SMART_WRITE_LOG_SECTOR ATA_ZERO	no of to be written	Log Sector address	write data
NOTE: - <ppBuf> contains pointer to pointer data buffer to be written			
IOCTL_CFA_ERASE_SECTORS ATA_ZERO	sector count	PackedCHS/LBA	ATA_ZERO
IOCTL_CFA_REQUEST_EXTENDED_ERROR_CODE ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
IOCTL_CFA_TRANSLATE_SECTOR ATA_ZERO	- <ppbuf>	returns pointer to data pointer. PackedLBA/CHS	read data
IOCTL_CFA_WRITE_MULTIPLE_WITHOUT_ERASE ATA_ZERO	sector count	PackedCHS/LBA	write data
NOTE: -<pbuf> contains pointer to data pointer.			
IOCTL_CFA_WRITE_SECTORS_WITHOUT_ERASE ATA_ZERO	sector count	PackedCHS/LBA	write data

RETURNS

OK or ERROR

ERRNO Not Available

SEE ALSO [ataDrv](#)

atapiMaxMDmaModeGet()

NAME [atapiMaxMDmaModeGet\(\)](#) – get the Maximum Multi word DMA mode the drive supports.

SYNOPSIS

```
UINT8 atapiMaxMDmaModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive maximum MDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. The following bits are set for corresponding modes supported.

Bit2	Multi DMA mode 2 and below are supported
Bit1	Multi DMA mode 1 and below are supported
Bit0	Multi DMA mode 0 is supported

RETURNS Maximum Multi word DMA mode.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiMaxPioModeGet()

NAME [atapiMaxPioModeGet\(\)](#) – get the Maximum PIO mode that drive can support.

SYNOPSIS

```
UINT8 atapiMaxPioModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive maximum PIO mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure

RETURNS maximum PIO mode.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiMaxSDmaModeGet()

NAME **atapiMaxSDmaModeGet()** – get the Maximum Single word DMA mode the drive supports

SYNOPSIS

```
UINT8 atapiMaxSDmaModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive maximum SDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure

RETURNS Maximum Single word DMA mode.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiMaxUDmaModeGet()

NAME **atapiMaxUDmaModeGet()** – get the Maximum Ultra DMA mode the drive can support.

SYNOPSIS

```
UINT8 atapiMaxUDmaModeGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive maximum UDMA mode supported by the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure. The following bits are set for corresponding modes supported.

Bit4 Ultra DMA mode 4 and below are supported
Bit3 Ultra DMA mode 3 and below are supported
Bit2 Ultra DMA mode 2 and below are supported
Bit1 Ultra DMA mode 1 and below are supported
Bit0 Ultra DMA mode 0 is supported

RETURNS Maximum Ultra DMA mode.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiModelNumberGet()

NAME [atapiModelNumberGet\(\)](#) – get the model number of the drive.

SYNOPSIS

```
char * atapiModelNumberGet
(
    int ctrl,
    int drive
)
```

DESCRIPTION This function is used to get drive Model Number of the ATA / ATAPI drive specified by *ctrl* and *drive* from drive structure. It returns a pointer to character array of 40 bytes length which contains serial number in ascii.

RETURNS pointer to the model number.

ERRNO Not Available

SEE ALSO [ataShow](#)

atapiParamsPrint()

NAME [atapiParamsPrint\(\)](#) – Print the drive parameters.

SYNOPSIS

```
void atapiParamsPrint
```

```
(  
    int ctrl,  
    int drive  
)
```

DESCRIPTION	This user callable routine will read the current parameters from the corresponding drive and will print the specified range of parameters on the console.
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	ataDrv

atapiPktCmd()

NAME	atapiPktCmd() – execute an ATAPI command with error processing
SYNOPSIS	<pre>UINT8 atapiPktCmd (ATA_DEV * pAtapiDev, ATAPI_CMD * pComPack)</pre>
DESCRIPTION	This routine executes a single ATAPI command, checks the command completion status and tries to recover if an error encountered during command execution at any stage.
RETURN	SENSE_NO_SENSE if success, or ERROR if not successful for any reason.
RETURNS	Not Available
ERRNO	S_ioLib_DEVICE_ERROR
SEE ALSO	ataDrv

atapiPktCmdSend()

NAME	atapiPktCmdSend() – Issue a Packet command.
SYNOPSIS	<pre>UINT8 atapiPktCmdSend</pre>

```
(  
ATA_DEV    * pAtapiDev,  
ATAPI_CMD * pComPack  
)
```

DESCRIPTION	This function issues a packet command to specified drive. See library file description for more details.
RETURN	SENSE_NO_SENSE if success, or ERROR if not successful for any reason
RETURNS	Not Available
ERRNO	S_ioLib_DEVICE_ERROR
SEE ALSO	ataDrv

atapiRead10()

NAME	atapiRead10() – read one or more blocks from an ATAPI Device.
SYNOPSIS	<pre>STATUS atapiRead10 (ATA_DEV * pAtapiDev, UINT32 startBlk, UINT32 nBlks, UINT32 transferLength, char * pBuf)</pre>
DESCRIPTION	This routine reads one or more blocks from the specified device, starting with the specified block number. The name of this routine relates to the SFF-8090i (Mt. Fuji), used for DVD-ROM, and indicates that the entire packet command uses 10 bytes, rather than the normal 12.
RETURNS	OK, ERROR if the read command didn't succeed.
ERRNO	Not Available
SEE ALSO	ataDrv

atapiReadCapacity()

NAME	atapiReadCapacity() – issue a READ CD-ROM CAPACITY command to a ATAPI device
SYNOPSIS	<pre>STATUS atapiReadCapacity (ATA_DEV * pAtapiDev)</pre>
DESCRIPTION	This routine issues a READ CD-ROM CAPACITY command to a specified ATAPI device.
RETURN	OK, or ERROR if the command fails.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiReadTocPmaAtip()

NAME	atapiReadTocPmaAtip() – issue a READ TOC command to a ATAPI device
SYNOPSIS	<pre>STATUS atapiReadTocPmaAtip (ATA_DEV * pAtapiDev, UINT32 transferLength, char * resultBuf)</pre>
DESCRIPTION	This routine issues a READ TOC command to a specified ATAPI device.
RETURN	OK, or ERROR if the command fails.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiRemovMediaStatusNotifyVerGet()

NAME	atapiRemovMediaStatusNotifyVerGet() – get the Media Stat Notification Version.
SYNOPSIS	<pre>UINT16 atapiRemovMediaStatusNotifyVerGet (int ctrl, int drive)</pre>
DESCRIPTION	This function will return the removable media status notification version of the drive.
RETURNS	Version Number.
ERRNO	Not Available
SEE ALSO	ataShow

atapiScan()

NAME	atapiScan() – issue SCAN packet command to ATAPI drive.
SYNOPSIS	<pre>STATUS atapiScan (ATA_DEV * pAtapiDev, UINT32 startAddressField, int function)</pre>
DESCRIPTION	This function issues SCAN packet command to ATAPI drive. The <i>function</i> argument should be 0x00 for fast forward and 0x10 for fast reversed operation.
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiSeek()

NAME	atapiSeek() – issues a SEEK packet command to drive.
SYNOPSIS	<pre>STATUS atapiSeek (ATA_DEV * pAtapiDev, UINT32 addressLBA)</pre>
DESCRIPTION	This function issues a SEEK packet command (not ATA SEEK command) to the specified drive.
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiSetCDSpeed()

NAME	atapiSetCDSpeed() – issue SET CD SPEED packet command to ATAPI drive.
SYNOPSIS	<pre>STATUS atapiSetCDSpeed (ATA_DEV * pAtapiDev, int readDriveSpeed, int writeDriveSpeed)</pre>
DESCRIPTION	This function issues SET CD SPEED packet command to ATAPI drive while reading and writing of ATAPI drive(CD-ROM) data. The arguments <i>readDriveSpeed</i> and <i>writeDriveSpeed</i> are in Kbytes/Second.
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiStartStopUnit()

NAME	atapiStartStopUnit() – Issues START STOP UNIT packet command
SYNOPSIS	STATUS atapiStartStopUnit (ATA_DEV * pAtapiDev, int arg0)
DESCRIPTION	This function issues a command to drive to MEDIA EJECT and MEDIA LOAD. Argument <i>arg0</i> selects to EJECT or LOAD. To eject media <i>arg0</i> should be EJECT_DISK To load media <i>arg0</i> should be LOAD_DISK
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiStopPlayScan()

NAME	atapiStopPlayScan() – issue STOP PLAY/SCAN packet command to ATAPI drive.
SYNOPSIS	STATUS atapiStopPlayScan (ATA_DEV * pAtapiDev)
RETURN	OK or ERROR
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	ataDrv

atapiTestUnitRdy()

NAME	atapiTestUnitRdy() – issue a TEST UNIT READY command to a ATAPI drive
SYNOPSIS	<pre>STATUS atapiTestUnitRdy (ATA_DEV * pAtapiDev)</pre>
DESCRIPTION	This routine issues a TEST UNIT READY command to a specified ATAPI drive.
RETURNS	OK, or ERROR if the command fails.
ERRNO	Not Available
SEE ALSO	ataDrv

atapiVersionNumberGet()

NAME	atapiVersionNumberGet() – get the ATA/ATAPI version number of the drive.																
SYNOPSIS	<pre>UINT32 atapiVersionNumberGet (int ctrl, int drive)</pre>																
DESCRIPTION	This function will return the ATA/ATAPI version number of the drive. Most significant 16 bits represent the Major Version Number and the Least significant 16 bits represents the minor Version Number. Major Version Number <table><tr><td>Bit 22</td><td>ATA/ATAPI-6</td></tr><tr><td>Bit 21</td><td>ATA/ATAPI-5</td></tr><tr><td>Bit 20</td><td>ATA/ATAPI-4</td></tr><tr><td>Bit 19</td><td>ATA-3</td></tr><tr><td>Bit 18</td><td>ATA-2</td></tr></table> Minor version Number (bit 15 through bit 0) <table><tr><td>0001h</td><td>Obsolete</td></tr><tr><td>0002h</td><td>Obsolete</td></tr><tr><td>0003h</td><td>Obsolete</td></tr></table>	Bit 22	ATA/ATAPI-6	Bit 21	ATA/ATAPI-5	Bit 20	ATA/ATAPI-4	Bit 19	ATA-3	Bit 18	ATA-2	0001h	Obsolete	0002h	Obsolete	0003h	Obsolete
Bit 22	ATA/ATAPI-6																
Bit 21	ATA/ATAPI-5																
Bit 20	ATA/ATAPI-4																
Bit 19	ATA-3																
Bit 18	ATA-2																
0001h	Obsolete																
0002h	Obsolete																
0003h	Obsolete																

0004h	ATA-2 published, ANSI X3.279-1996
0005h	ATA-2 X3T10 948D prior to revision 2k
0006h	ATA-3 X3T10 2008D revision 1
0007h	ATA-2 X3T10 948D revision 2k
0008h	ATA-3 X3T10 2008D revision 0
0009h	ATA-2 X3T10 948D revision 3
000Ah	ATA-3 published, ANSI X3.298-199x
000Bh	ATA-3 X3T10 2008D revision 6
000Ch	ATA-3 X3T13 2008D revision 7 and 7a
000Dh	ATA/ATAPI-4 X3T13 1153D revision 6
000Eh	ATA/ATAPI-4 T13 1153D revision 13
000Fh	ATA/ATAPI-4 X3T13 1153D revision 7
0010h	ATA/ATAPI-4 T13 1153D revision 18
0011h	ATA/ATAPI-4 T13 1153D revision 15
0012h	ATA/ATAPI-4 published, ANSI NCITS 317-1998
0013h	Reserved
0014h	ATA/ATAPI-4 T13 1153D revision 14

0004h	ATA-2 published, ANSI X3.279-1996
0005h	ATA-2 X3T10 948D prior to revision 2k
0006h	ATA-3 X3T10 2008D revision 1
0007h	ATA-2 X3T10 948D revision 2k
0008h	ATA-3 X3T10 2008D revision 0
0009h	ATA-2 X3T10 948D revision 3
000Ah	ATA-3 published, ANSI X3.298-199x
000Bh	ATA-3 X3T10 2008D revision 6
000Ch	ATA-3 X3T13 2008D revision 7 and 7a
000Dh	ATA/ATAPI-4 X3T13 1153D revision 6
000Eh	ATA/ATAPI-4 T13 1153D revision 13
000Fh	ATA/ATAPI-4 X3T13 1153D revision 7
0010h	ATA/ATAPI-4 T13 1153D revision 18
0011h	ATA/ATAPI-4 T13 1153D revision 15
0012h	ATA/ATAPI-4 published, ANSI NCITS 317-1998
0013h	Reserved
0014h	ATA/ATAPI-4 T13 1153D revision 14

000Fh	ATA/ATAPI-4 X3T13 1153D revision 7
0010h	ATA/ATAPI-4 T13 1153D revision 18
0011h	ATA/ATAPI-4 T13 1153D revision 15
0012h	ATA/ATAPI-4 published, ANSI NCITS 317-1998
0013h	Reserved
0014h	ATA/ATAPI-4 T13 1153D revision 14

RETURNS ATA/ATAPI version number.

ERRNO Not Available

SEE ALSO **ataShow**

auDump()

NAME **auDump()** – display device status

SYNOPSIS void auDump
(
 int unit
)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **auEnd**

auEndLoad()

NAME **auEndLoad()** – initialize the driver and device

SYNOPSIS

```
END_OBJ * auEndLoad
(
    char * initString /* string to be parse by the driver */
)
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:devMemAddr:devIoAddr:enableAddr:vecNum:intLvl:offset :qtyCluster:flags

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "au") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

ERRNO Not Available

SEE ALSO **auEnd**

auInitParse()

NAME **auInitParse()** – parse the initialization string

SYNOPSIS

```
STATUS auInitParse
(
    AU_DRV_CTRL * pDrvCtrl, /* pointer to the control structure */
    char *        initString /* initialization string */
)
```

DESCRIPTION	Parse the input string. This routine is called from auEndLoad() which initializes some values in the driver control structure with the values passed in the initialization string. The initialization string format is: <i>unit:devMemAddr:devIoAddr:vecNum:intLvl:offset;flags</i>
	<i>unit</i> Device unit number, a small integer.
	<i>devMemAddr</i> Device register base memory address
	<i>devIoAddr</i> I/O register base memory address
	<i>enableAddr</i> Address of MAC enable register
	<i>vecNum</i> Interrupt vector number.
	<i>intLvl</i> Interrupt level.
	<i>offset</i> Offset of starting of data in the device buffers.
	<i>qtyCluster</i> Number of clusters to allocate
	<i>flags</i> Device specific flags, for future use.
RETURNS	OK, or ERROR if any arguments are invalid.
ERRNO	Not Available
SEE ALSO	auEnd

bcm1250MacEndLoad()

NAME	bcm1250MacEndLoad() – initialize the driver and device
SYNOPSIS	<pre>END_OBJ * bcm1250MacEndLoad (char * initString /* String to be parsed by the driver. */)</pre>

DESCRIPTION	This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in <i>initString</i> , which expects a string of the following format: The initialization string format is: <i>"unit:hwunit:vecnum:flags:numRds0:numTds0:numRds1:numTds1"</i> The hwunit field is not used, but must be present to parse properly. This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "sbe0", "sbe1", or "sbe2") into the <i>initString</i> and returns NULL. If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.
RETURNS	An END object pointer or NULL on error.
ERRNO	Not Available
SEE ALSO	bcm1250MacEnd

bcm1250MacPhyShow()

NAME	bcm1250MacPhyShow() – display the physical register values
SYNOPSIS	<pre>void bcm1250MacPhyShow (int inst /* driver instance */)</pre>
DESCRIPTION	This routine prints the enet PHY registers to stdout.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	bcm1250MacEnd

bcm1250MacRxDmaShow()

NAME	bcm1250MacRxDmaShow() – display RX DMA register values
-------------	--

SYNOPSIS	void bcm1250MacRxDmaShow (int inst /* driver instance */)
DESCRIPTION	This routine prints the enet RX DMA registers to stdout.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	bcm1250MacEnd

bcm1250MacShow()

NAME	bcm1250MacShow() – display the MAC register values
SYNOPSIS	void bcm1250MacShow (int inst /* driver instance */)
DESCRIPTION	This routine prints the enet MAC registers to stdout.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	bcm1250MacEnd

bcm1250MacTxDmaShow()

NAME	bcm1250MacTxDmaShow() – display TX DMA register values
SYNOPSIS	void bcm1250MacTxDmaShow (int inst /* driver instance */)
DESCRIPTION	This routine prints the enet TX DMA registers to stdout.

RETURNS	N/A
ERRNO	Not Available
SEE ALSO	bcm1250MacEnd

bioInit()

NAME	bioInit() – initialize the bio library
SYNOPSIS	STATUS bioInit (void)
DESCRIPTION	none
RETURNS	OK
ERRNO	Not Available
SEE ALSO	bio

bio_alloc()

NAME	bio_alloc() – allocate memory blocks
SYNOPSIS	<pre>void * bio_alloc (device_t xbd, /* XBD for which to allocate */ int numBlocks /* number of blocks to allocate */)</pre>
DESCRIPTION	This routine allocates <i>numBlocks</i> of memory. The size of the block is extracted from the <i>xbd</i> .
RETURNS	pointer to the allocated memory, NULL on error
ERRNO	Not Available
SEE ALSO	bio

bio_done()

NAME **bio_done()** – terminates a bio operation

SYNOPSIS

```
void bio_done
(
    struct bio * pBio, /* pointer to bio structure */
    int         error /* error code */
)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **bio**

bio_free()

NAME **bio_free()** – free the bio memory

SYNOPSIS

```
void bio_free
(
    void * pBioData /* pointer to data to free */
)
```

DESCRIPTION This routine frees the memory in a bio structure. Note that *bio_data* is NOT a pointer to the the bio structure, but rather the pointer to the memory to free.

RETURNS N/A

ERRNO Not Available

SEE ALSO **bio**

cisConfigregGet()

NAME **cisConfigregGet()** – get the PCMCIA configuration register

SYNOPSIS	<pre>STATUS cisConfigregGet (int sock, /* socket no. */ int reg, /* configuration register no. */ int *pValue /* content of the register */)</pre>
DESCRIPTION	This routine gets that PCMCIA configuration register.
RETURNS	OK, or ERROR if it cannot set a value on the PCMCIA chip.
ERRNO	Not Available
SEE ALSO	cisLib

cisConfigregSet()

NAME	cisConfigregSet() – set the PCMCIA configuration register
SYNOPSIS	<pre>STATUS cisConfigregSet (int sock, /* socket no. */ int reg, /* register no. */ int value /* content of the register */)</pre>
DESCRIPTION	This routine sets the PCMCIA configuration register.
RETURNS	OK, or ERROR if it cannot set a value on the PCMCIA chip.
ERRNO	Not Available
SEE ALSO	cisLib

cisFree()

NAME	cisFree() – free tuples from the linked list
SYNOPSIS	<pre>void cisFree</pre>

```
(  
int sock /* socket no. */  
)
```

DESCRIPTION	This routine free tuples from the linked list.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	cisLib

cisGet()

NAME	cisGet() – get information from a PC card's CIS
SYNOPSIS	STATUS cisGet (int sock /* socket no. */)
DESCRIPTION	This routine gets information from a PC card's CIS, configures the PC card, and allocates resources for the PC card.
RETURNS	OK, or ERROR if it cannot get the CIS information, configure the PC card, or allocate resources.
ERRNO	Not Available
SEE ALSO	cisLib

cisShow()

NAME	cisShow() – show CIS information
SYNOPSIS	void cisShow (int sock /* socket no. */)

DESCRIPTION	This routine shows CIS information.
NOTE	This routine uses floating point calculations. The calling task needs to be spawned with the VX_FP_TASK flag. If this is not done, the data printed by cisShow may be corrupted and unreliable.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	cisShow

coldFireSioRegister()

NAME	coldFireSioRegister() – register coldFire sio driver
SYNOPSIS	<code>void coldFireSioRegister(void)</code>
DESCRIPTION	This routine registers the coldFire Sio driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbColdFireSio

ctB69000VgaInit()

NAME	ctB69000VgaInit() – initializes the B69000 chip and loads font in memory.
SYNOPSIS	<code>STATUS ctB69000VgaInit (void)</code>
DESCRIPTION	This routine will initialize the VGA card if present in PCI connector, sets up register set in VGA 3+ mode and loads the font in plane 2.
RETURNS	OK/ERROR

ERRNO Not Available

SEE ALSO ctB69000Vga

dec21140SromWordRead()

NAME **dec21140SromWordRead()** – read two bytes from the serial ROM

SYNOPSIS USHORT dec21140SromWordRead
(
 DRV_CTRL * pDrvCtrl,
 UCHAR lineCnt /* Serial ROM line Number */
)

DESCRIPTION This routine returns the two bytes of information that is associated with it the specified ROM line number. This will later be used by the **dec21140GetEthernetAdr** function. It can also be used to review the ROM contents itself. The function must first send some initial bit patterns to the CSR9 that contains the Serial ROM Control bits. Then the line index into the ROM is evaluated bit-by-bit to program the ROM. The 2 bytes of data are extracted and processed into a normal pair of bytes.

RETURNS Value from ROM or **ERROR**.

ERRNO Not Available

SEE ALSO **dec21x40End**

dec21145SPIReadBack()

NAME **dec21145SPIReadBack()** – Read all PHY registers out

SYNOPSIS void dec21145SPIReadBack
(
 DRV_CTRL * pDrvCtrl /* pointer to DRV_CTRL structure */
)

DESCRIPTION none

RETURNS nothing

ERRNO Not Available

SEE ALSO **dec21x40End**

dec21x40EndLoad()

NAME **dec21x40EndLoad()** – initialize the driver and device

SYNOPSIS

```
END_OBJ* dec21x40EndLoad
(
    char* initStr /* String to be parse by the driver. */
)
```

DESCRIPTION This routine initializes the driver and the device to an operational state. All of the device-specific parameters are passed in the *initStr*. If this routine is called with an empty but allocated string, it puts the name of this device (that is, "dc") into the *initStr* and returns 0. If the string is allocated but not empty, this routine tries to load the device.

RETURNS An END object pointer or NULL on error.

ERRNO Not Available

SEE ALSO **dec21x40End**

dec21x40PhyFind()

NAME **dec21x40PhyFind()** – Find the first PHY connected to DEC MII port.

SYNOPSIS

```
UINT8 dec21x40PhyFind
(
    DRV_CTRL *pDrvCtrl
)
```

DESCRIPTION none

RETURNS Address of PHY or 0xFF if not found.

ERRNO Not Available

SEE ALSO **dec21x40End**

devAttach()

NAME **devAttach()** – attach a device

SYNOPSIS

```
int devAttach
(
    struct device * dev,
    const char * name,
    int type,
    device_t * result
)
```

DESCRIPTION none

RETURNS 0 upon success, non-zero otherwise

ERRNO Not Available

SEE ALSO **device**

devDetach()

NAME **devDetach()** – detach a device

SYNOPSIS

```
STATUS devDetach
(
    struct device * dev /* pointer to device to detach */
)
```

DESCRIPTION none

RETURNS 0 upon success, non-zero otherwise

ERRNO Not Available

SEE ALSO **device**

devInit()

NAME	devInit() – initialize the device manager
SYNOPSIS	STATUS devInit (uint16_t ndevs /* number of devices */)
DESCRIPTION	This routine initializes the device manager for <i>ndevs</i> devices.
RETURNS	OK upon success, ERROR otherwise
ERRNO	EINVAL
SEE ALSO	device

devMap()

NAME	devMap() – map a device
SYNOPSIS	struct device *devMap (device_t dev /* device to map */)
DESCRIPTION	none
RETURNS	pointer to device upon success, NULL otherwise
ERRNO	Not Available
SEE ALSO	device

devMapUnsafe()

NAME	devMapUnsafe() – map a device unconditionally
SYNOPSIS	struct device *devMapUnsafe

```
(  
device_t dev /* device to map */  
)
```

DESCRIPTION	none
RETURNS	pointer to device upon success, NULL otherwise
ERRNO	Not Available
SEE ALSO	device

devName()

NAME	devName() – name a device
SYNOPSIS	STATUS devName (device_t dev, /* device to name */ devname_t name /* name to assign */)
DESCRIPTION	none
RETURNS	OK upon success, ERROR otherwise
ERRNO	Not Available
SEE ALSO	device

devResourceGet()

NAME	devResourceGet() – find vxBus resource
SYNOPSIS	STATUS devResourceGet (const struct hcfDevice * pDevice, /* hcf device */ char * name, /* resource name */ int type, /* resource type */ void ** pDest /* resource value */)

DESCRIPTION Search the HCF records for a match.

RETURNS OK, or ERROR

ERRNO

SEE ALSO [hwConfig](#)

devResourceIntrGet()

NAME **devResourceIntrGet()** – find vxBus interrupt resources

SYNOPSIS

```
STATUS devResourceIntrGet
(
    VXB_DEVICE_ID          pDev,    /* device information */
    const struct hcfDevice * pHcf   /* HCF record */
)
```

DESCRIPTION This routine extracts interrupt information from the HCF record provided, creates an interrupt structure, and updates the pDev structure.

The updates to pDev are as follows:

- Insert the interrupt structure into pDev-pIntrInfo field.

- Modify pDev->pAccess with any updates to the interrupt related access methods.

RETURNS OK, or ERROR

ERRNO

SEE ALSO [hwConfig](#)

devResourceIntrShow()

NAME **devResourceIntrShow()** – show interrupt values for specified device

SYNOPSIS

```
int devResourceIntrShow
```

```
(  
VXB_DEVICE_ID pDev  
)
```

DESCRIPTION This routine displays the interrupt values of the specified device.

RETURNS Not Available

ERRNO

SEE ALSO [hwConfig](#)

devUnmap()

NAME **devUnmap()** – unmap a device

SYNOPSIS

```
int devUnmap
(
    struct device * dev /* pointer to device to unmap */
)
```

DESCRIPTION none

RETURNS

ERRNO Not Available

SEE ALSO [device](#)

devUnmapUnsafe()

NAME **devUnmapUnsafe()** – unmap a device unconditionally

SYNOPSIS

```
int devUnmapUnsafe
(
    struct device * dev /* pointer to device to unmap */
)
```

DESCRIPTION none

RETURNS

ERRNO Not Available

SEE ALSO **device**

displayRapidIOCfgRegs()

NAME **displayRapidIOCfgRegs()** – displays RIO registers given a base address

SYNOPSIS `void displayRapidIOCfgRegs
 (
 RAPIDIOCAR* rioRegs
)`

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **m85xxRio**

drvUnloadAll()

NAME **drvUnloadAll()** – unload all drivers which provide download/unload support

SYNOPSIS `STATUS drvUnloadAll(void)`

DESCRIPTION function description

RETURNS: **OK**, always

RETURNS Not Available

ERRNO Not Available

SEE ALSO **drvDownLoad**

el3c90xEndLoad()

NAME	el3c90xEndLoad() – initialize the driver and device
SYNOPSIS	<pre>END_OBJ * el3c90xEndLoad (char * initString /* String to be parsed by the driver. */)</pre>
DESCRIPTION	This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in <i>initString</i> , which expects a string of the following format: <i>unit:devMemAddr:devIoAddr:pciMemBase:<vecnum:intLvl:memAdrs :memSize:memWidth:flags:buffMultiplier</i>
	This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "elPci") into the <i>initString</i> and returns 0.
	If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.
RETURNS	An END object pointer, or NULL on error, or 0 and the name of the device if the <i>initString</i> was NULL .
ERRNO	Not Available
SEE ALSO	el3c90xEnd

el3c90xInitParse()

NAME	el3c90xInitParse() – parse the initialization string
SYNOPSIS	<pre>STATUS el3c90xInitParse (EL3C90X_DEVICE * pDrvCtrl, /* pointer to the control structure */ char * initString /* initialization string */)</pre>
DESCRIPTION	Parse the input string. This routine is called from el3c90xEndLoad() which initializes some values in the driver control structure with the values passed in the initialization string.

The initialization string format is:

*unit:devMemAddr:devIoAddr:pciMemBase:<vecNum:intLvl:memAdrs:
:memSize:memWidth:flags:buffMultiplier*

unit

Device unit number, a small integer.

devMemAddr

Device register base memory address

devIoAddr

Device register base IO address

pciMemBase

Base address of PCI memory space

vecNum

Interrupt vector number.

intLvl

Interrupt level.

memAdrs

Memory pool address or NONE.

memSize

Memory pool size or zero.

memWidth

Memory system size, 1, 2, or 4 bytes (optional).

flags

Device specific flags, for future use.

buffMultiplier

Buffer Multiplier or NONE. If NONE is specified, it defaults to 2

RETURNS OK, or ERROR if any arguments are invalid.

ERRNO Not Available

SEE ALSO **el3c90xEnd**

elt3c509Load()

NAME **elt3c509Load()** – initialize the driver and device

SYNOPSIS `END_OBJ * elt3c509Load`

```
(  
char * initString /* String to be parsed by the driver. */  
)
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:port:intVector:intLevel:attachementType:noRxFrames

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "elt") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or **NULL** on error, or 0 and the name of the device if the *initString* was **NULL**.

ERRNO Not Available

SEE ALSO **elt3c509End**

elt3c509Parse()

NAME **elt3c509Parse()** – parse the init string

SYNOPSIS STATUS elt3c509Parse
(
ELT3C509_DEVICE * pDrvCtrl, /* device pointer */
char * initString /* initialization info string */
)

DESCRIPTION Parse the input string. Fill in values in the driver control structure.

The initialization string format is:

```
<unit>:<port>:<intVector>:<intLevel>:<attachementType>:<noRxFrames>  
unit  
Device unit number, a small integer.  
port  
base I/O address  
intVector  
Interrupt vector number (used with sysIntConnect)
```

intLevel

Interrupt level

attachmentType

type of Ethernet connector

nRxFrames

no. of Rx Frames in integer format

RETURNS OK or ERROR for invalid arguments.

ERRNO Not Available

SEE ALSO [elt3c509End](#)

emacEndLoad()

NAME **emacEndLoad()** – initialize the driver and device

SYNOPSIS

```
END_OBJ * emacEndLoad
{
    char * initString /* String to be parsed by the driver */
}
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in the initString.

See [ibmEmacInitParse\(\)](#) for the specific format of the string.

This function is meant to be called two different times during the driver load process. If this routine is called with the first character of the initialization string equal to **NULL**, this routine will return with the name of the device "emac" copied into initString. If this routine is called with the actual driver parameters in initString, it will use the params to initialize the device and prepare the rest of the driver for operation.

RETURNS An END object pointer, **NULL** if there is an error, or 0 and the name of the device if the first character of initString is **NULL**.

ERRNO

SEE ALSO [emacEnd](#)

emacTimerDebugDump()

NAME	emacTimerDebugDump() – Enable debugging output in timer handler
SYNOPSIS	<pre>void emacTimerDebugDump (int unit, BOOL enable /* 0 for disable */)</pre>
DESCRIPTION	This function will enable/disable statistic data output in timer handler
RETURN	N/A
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	emacEnd

endEtherAddressForm()

NAME	endEtherAddressForm() – form an Ethernet address into a packet
SYNOPSIS	<pre>M_BLK_ID endEtherAddressForm (M_BLK_ID pMblk, /* pointer to packet mBlk */ M_BLK_ID pSrcAddr, /* pointer to source address */ M_BLK_ID pDstAddr, /* pointer to destination address */ BOOL bcastFlag /* use link-level broadcast? */)</pre>
DESCRIPTION	This routine accepts the source and destination addressing information through <i>pSrcAddr</i> and <i>pDstAddr</i> and returns an M_BLK_ID that points to the assembled link-level header. To do this, this routine prepends the link-level header into the cluster associated with <i>pMblk</i> if there is enough space available in the cluster. It then returns a pointer to the pointer referenced in <i>pMblk</i> . However, if there is not enough space in the cluster associated with <i>pMblk</i> , this routine reserves a new mBlk -‘clBlk’-cluster construct for the header information. It then prepends the new mBlk to the mBlk passed in <i>pMblk</i> . As the function value, this routine then returns a pointer to the new mBlk , which is the head of a chain of mBlk structures. The second element of this chain is the mBlk referenced in <i>pMblk</i> .
RETURNS	M_BLK_ID or NULL .

ERRNO

SEE ALSO [endLib](#)

endEtherCrc32BeGet()

NAME [**endEtherCrc32BeGet\(\)**](#) – calculate a big-endian CRC-32 checksum

SYNOPSIS

```
UINT32 endEtherCrc32BeGet
(
    const UINT8 *pBuf,
    size_t       len
)
```

DESCRIPTION This routine performs a big endian CRC 32 calculation over an arbitrary block of data. Many ethernet controllers implement multicast filtering using a hash table where a CRC-32 checksum of the multicast group address is used as the table index. This function is provided as a convenience so that each driver doesn't need its own private copy.

RETURNS the 32-bit checksum of the supplied buffer

ERRNO N/A

SEE ALSO [endLib](#)

endEtherCrc32LeGet()

NAME [**endEtherCrc32LeGet\(\)**](#) – calculate a little-endian CRC-32 checksum

SYNOPSIS

```
UINT32 endEtherCrc32LeGet
(
    const UINT8 *pBuf,
    size_t       len
)
```

DESCRIPTION This routine performs a little endian CRC 32 calculation over an arbitrary block of data. Many ethernet controllers implement multicast filtering using a hash table where a CRC-32 checksum of the multicast group address is used as the table index. This function is provided as a convenience so that each driver doesn't need its own private copy.

RETURNS the 32-bit checksum of the supplied buffer

ERRNO N/A

SEE ALSO **endLib**

endEtherPacketAddrGet()

NAME **endEtherPacketAddrGet()** – locate the addresses in a packet

SYNOPSIS STATUS endEtherPacketAddrGet
(
 M_BLK_ID pMblk, /* pointer to packet */
 M_BLK_ID pSrc, /* pointer to local source address */
 M_BLK_ID pDst, /* pointer to local destination address */
 M_BLK_ID pESrc, /* pointer to remote source address (if any) */
 M_BLK_ID pEDst /* pointer to remote destination address (if any) */
)

DESCRIPTION This routine takes a **M_BLK_ID**, locates the address information, and adjusts the **M_BLK_ID** structures referenced in *pSrc*, *pDst*, *pESrc*, and *pEDst* so that their *pData* members point to the addressing information in the packet. The addressing information is not copied. All **mBlk** structures share the same cluster.

RETURNS OK

ERRNO

SEE ALSO **endLib**

endEtherPacketDataGet()

NAME **endEtherPacketDataGet()** – return the beginning of the packet data

SYNOPSIS STATUS endEtherPacketDataGet
(
 M_BLK_ID pMblk,
 LL_HDR_INFO * pLinkHdrInfo
)

DESCRIPTION This routine fills the given *pLinkHdrInfo* with the appropriate offsets.

RETURNS OK or ERROR.

ERRNO

SEE ALSO [endLib](#)

endMcacheFlush()

NAME **endMcacheFlush()** – flush all tuples from the mBlk recycle cache

SYNOPSIS

```
void endMcacheFlush
(
    void
)
```

DESCRIPTION This function removes all mBlk tuple pointers from the mBlk recycle cache and frees them with **netMblkClChainFree()**. This function must be called whenever any interface that uses the recycle cache is stopped so that cached buffers can be properly reclaimed.

RETURNS N/A

ERRNO N/A

SEE ALSO [endLib](#)

endMcacheGet()

NAME **endMcacheGet()** – allocate an mBlk tuple from the mBlk recycle cache

SYNOPSIS

```
M_BLK_ID endMcacheGet
(
    void
)
```

DESCRIPTION This function obtains an mBlk tuple stored in a fast mBlk recycle cache.

This cache is used to provide a performance boost for drivers used in devices designed for traffic forwarding or bridging. In these devices, buffers from the RX side of an inbound interface are often directly retransmitted via the TX side of an outbound interface, after which they are released back to the RX interface's mBlk pool. This may consume a lot of overhead depending on the pool implementation.

Using the mBlk recycle cache may reduce this overhead. The cache is implemented as a simple 128-entry pointer array. When a tuple is to be released by the TX completion handler in a network driver, it can be put in the recycle cache instead of released with **netMblkClChainFree()**. Whenever another interface needs a tuple to replenish its RX resources, it can allocate it from the pool instead of doing a **netTupleGet()**. The cache only holds 128 tuple pointers, the cache get and put operations use a minimum of processing and are usually faster than the **netTupleGet()** and **netMblkClChainFree()** paths, which shaves some cycles off the time needed to process a given packet.

RETURNS	a pointer to an mBlk tuple, or NULL if the cache is empty
ERRNO	N/A
SEE ALSO	endLib

endMcachePut()

NAME	endMcachePut() – store an mBlk tuple in the mBlk recycle cache
SYNOPSIS	STATUS endMcachePut (M_BLK_ID pMblk)
DESCRIPTION	This function places a pointer to an mBlk tuple in the mBlk recycle cache, as long as there is room to store it. The cache only holds 128 pointers, so storing the pointer will fail if the cache is full.
RETURNS	OK if the tuple was successfully cached, otherwise ERROR
ERRNO	N/A
SEE ALSO	endLib

endObjFlagSet()

NAME	endObjFlagSet() – set the flags member of an END_OBJ structure
SYNOPSIS	STATUS endObjFlagSet

```
(  
    END_OBJ * pEnd,  
    UINT      flags  
)
```

DESCRIPTION	As input, this routine expects a pointer to an END_OBJ structure (the <i>pEnd</i> parameter) and a flags value (the <i>flags</i> parameter). This routine sets the flags member of the END_OBJ structure to the value of the <i>flags</i> parameter. Because this routine assumes that the driver interface is now up, this routine also sets the attached member of the referenced END_OBJ structure to TRUE.
RETURNS	OK
ERRNO	
SEE ALSO	endLib

endObjInit()

NAME	endObjInit() – initialize an END_OBJ structure
SYNOPSIS	<pre>STATUS endObjInit (END_OBJ * pEndObj, /* object to be initialized */ DEV_OBJ* pDevice, /* ptr to device struct */ char * pBaseName, /* device base name, for example, "ln" */ int unit, /* unit number */ NET_FUNCS * pFuncTable, /* END device functions */ char* pDescription)</pre>
DESCRIPTION	This routine initializes an END_OBJ structure and fills it with data from the argument list. It also creates and initializes semaphores and protocol list.
RETURNS	OK or ERROR .
ERRNO	
SEE ALSO	endLib

endPollStatsInit()

NAME	endPollStatsInit() – initialize polling statistics updates
SYNOPSIS	STATUS endPollStatsInit (void * pCookie, FUNCPTR pIfPollRtn)
DESCRIPTION	This routine is used to begin polling of the interface specified by pCookie and will periodically call out to the pIfPollRtn function, which will collect the interface statistics. If the driver supports polling updates, this routine will start a watchdog that will invoke the pIfPollRtn routine periodically. The watchdog will automatically re-arm itself. The pIfPollRtn will be passed a pointer to the driver's END_IFDRVCONF structure as an argument.
RETURNS	ERROR if the driver doesn't support polling, otherwise OK.
ERRNO	
SEE ALSO	endLib

endPoolCreate()

NAME	endPoolCreate() – create a buffer pool for an END driver
SYNOPSIS	STATUS endPoolCreate (int tupleCnt, NET_POOL_ID * ppNetPool)
DESCRIPTION	This function is a helper routine for END drivers to simplify the creation of a driver-private buffer pool. The caller need only specify the number of tuples in the pool. The resulting pool will have an equal number of mBlks, cluster blocks and clusters (this is the typical requirement for driver usage). The clusters in the pool will always be the same size, governed by the endPoolClSize global variable. The default size is 1536 bytes. Note that clusters will be allocated from cached storage. This may be unsuitable for some drivers, however this is not an issue for drivers that are VxBus-compliant: cache coherence issues should be handled internally by the vxbDmaBuf API.

RETURNS	OK if pool is successfully created, otherwise ERROR
ERRNO	N/A
SEE ALSO	endLib

endPoolCreate()

NAME	endPoolCreate() – create a jumbo buffer pool for an END driver
SYNOPSIS	<pre>STATUS endPoolJumboCreate (int tupleCnt, NET_POOL_ID * ppNetPool)</pre>
DESCRIPTION	This function is a helper routine for END drivers to simplify the creation of a driver-private buffer pool. It's similar to the endPoolCreate() function, except that it creates a pool of 9K jumbo buffers for drivers that support jumbo frames.
RETURNS	OK if pool is successfully created, otherwise ERROR
ERRNO	N/A
SEE ALSO	endLib

endPoolDestroy()

NAME	endPoolDestroy() – destroy a pool created with endPoolCreate()
SYNOPSIS	<pre>STATUS endPoolDestroy (NET_POOL_ID pNetPool)</pre>
DESCRIPTION	This function is used to release a buffer pool created with the endPoolCreate() function. All memory allocated for use by the pool will be returned.
RETURNS	OK if pool is successfully destroyed, otherwise ERROR

ERRNO N/A

SEE ALSO [endLib](#)

endPoolTupleFree()

NAME [endPoolTupleFree\(\)](#) – free an mBlk tuple for an END driver

SYNOPSIS

```
void endPoolTupleFree
(
    M_BLK_ID pMblk
)
```

DESCRIPTION This function releases an mBlk tuple from a pool created with the [endPoolCreate\(\)](#) routine. It works in conjunction with the mBlk recycle cache to optimize performance when allocating buffers in the transmit and receive paths in END drivers. If the buffer was allocated using [endPoolTupleAlloc\(\)](#), it will be put back into the mBlk recycle cache, assuming there's room for it. If not, it will be released back to its original pool.

For buffers not allocated using [endPoolTupleFree\(\)](#), this routine is equivalent to [netMblkClChainFree\(\)](#).

RETURNS N/A

ERRNO N/A

SEE ALSO [endLib](#)

endPoolTupleGet()

NAME [endPoolTupleGet\(\)](#) – allocate an mBlk tuple for an END driver

SYNOPSIS

```
M_BLK_ID endPoolTupleGet
(
    NET_POOL_ID pNetPool
)
```

DESCRIPTION This function allocate an mBlk tuple from a pool created with the [endPoolCreate\(\)](#) routine. It works in conjunction with the mBlk recycle cache to optimize performance when allocating buffers in the transmit and receive paths in END drivers. If an mBlk tuple is

available in the cache, the buffer will be allocated from it directly. Otherwise, a tuple will be allocated from the pool.

Buffers allocated with **endPoolTupleAlloc()** should always be released with **endPoolTupleFree()**.

RETURNS a pointer to an mBlk, otherwise **NULL** if none are available

ERRNO N/A

SEE ALSO **endLib**

endTok_r()

NAME **endTok_r()** – get a token string (modified version)

SYNOPSIS

```
char * endTok_r
(
    char *      string,      /* string to break into tokens */
    const char * separators, /* the separators */
    char **     ppLast       /* pointer to serve as string index */
)
```

DESCRIPTION

This modified version can be used with optional parameters. If the parameter is not specified, this version returns **NULL**. It does not signify the end of the original string, but that the parameter is null.

```
/* required parameters */

string = endTok_r (initString, ":", &pLast);
if (string == NULL)
    return ERROR;
reqParam1 = strtoul (string);

string = endTok_r (NULL, ":", &pLast);
if (string == NULL)
    return ERROR;
reqParam2 = strtoul (string);

/* optional parameters */

string = endTok_r (NULL, ":", &pLast);
if (string != NULL)
    optParam1 = strtoul (string);

string = endTok_r (NULL, ":", &pLast);
if (string != NULL)
    optParam2 = strtoul (string);
```

RETURNS Not Available

ERRNO Not Available

SEE ALSO **dec21x40End**

erfCategoriesAvailable()

NAME **erfCategoriesAvailable()** – Get the number of unallocated User Categories.

SYNOPSIS `UINT16 erfCategoriesAvailable(void)`

DESCRIPTION none

RETURNS Number of categories

ERRNO

SEE ALSO **erfLib**

erfCategoriesAvailable()

NAME **erfCategoriesAvailable()** – Get the maximum number of Categories.

SYNOPSIS `UINT16 erfMaxCategoriesGet(void)`

DESCRIPTION none

RETURNS Number of categories

ERRNO

SEE ALSO **erfShow**

erfCategoriesAvailable()

NAME	erfCategoriesAvailable() – Get the maximum number of Types.
SYNOPSIS	<code>UINT16 erfMaxTypesGet(void)</code>
DESCRIPTION	none
RETURNS	Number of types
ERRNO	
SEE ALSO	erfShow

erfCategoryAllocate()

NAME	erfCategoryAllocate() – Allocates a User Defined Event Category.
SYNOPSIS	<code>STATUS erfCategoryAllocate (UINT16 * pEventCat /* variable to return Event Category */)</code>
DESCRIPTION	This routine allocates an Event Category for a new User Defined Event Category. Once defined, User defined Categories cannot be deleted.
RETURNS	OK, or ERROR if unable to define this Event Category
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_TOO_MANY_USER_CATS The number of User defined Event Categories exceeds the maximum number allowed.
SEE ALSO	erfLib

erfCategoryQueueCreate()

NAME	erfCategoryQueueCreate() – Creates a Category Event Processing Queue.
SYNOPSIS	<pre>STATUS erfCategoryQueueCreate (UINT16 eventCat, /* variable to return Event Category */ int queueSize, /* max number of events for queue */ int priority /* priority of event processing task */)</pre>
DESCRIPTION	This routine creates an Event Processing Queue for a Category. A new task will be created to process events from this queue. Once defined, all events for this category will be stored on this queue.
RETURNS	OK, or ERROR if unable to create the queue.
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_QUEUE_ALREADY_CREATED The event processing queue has already been setup for this Event Category. S_erfLib_MEMORY_ERROR A memory allocation failed.
SEE ALSO	erfLib

erfDefaultQueueSizeGet()

NAME	erfDefaultQueueSizeGet() – Get the size of the default queue.
SYNOPSIS	<code>UINT16 erfDefaultQueueSizeGet(void)</code>
DESCRIPTION	none
RETURNS	Number of elements that can be stored on the default queue.
ERRNO	
SEE ALSO	erfShow

erfEventRaise()

NAME	erfEventRaise() – Raises an event.
SYNOPSIS	<pre>STATUS erfEventRaise (UINT16 eventCat, /* Event Category */ UINT16 eventType, /* Event Type */ int procType, /* Processing Type */ void * pEventData, /* Pointer to Event Data */ erfFreePrototype * pFreeFunc /* Function to free Event Data when done */))</pre>
DESCRIPTION	This function will directly call all the handlers that are registered for this Event Category and either this Event Type or T_erfLib_ALL_TYPES. This routine should not be called from an Exception or Interrupt handler. If the Event Processing routine flag has the ERF_FLAG_AUTO_UNREG option set, the event processing routine will be unregistered after the routine is called.
RETURNS	OK, or ERROR if an error occurs
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_AUTO_UNREG_ERROR An error occurred during Auto Unregistration.
SEE ALSO	erfLib

erfHandlerRegister()

NAME	erfHandlerRegister() – Registers an event handler for a particular event.
SYNOPSIS	<pre>STATUS erfHandlerRegister (UINT16 eventCat, /* Event Category */ UINT16 eventType, /* Event Type */ erfHandlerPrototype * pEventHandler, /* Pointer to Event Handler */ void * pUserData, /* User data to be sent to Handler */ UINT16 flags /* Event Processing flags */)</pre>

DESCRIPTION	This routine registers an event handler for an event. The handler pointer will be saved and called when the event is raised. The userData will be sent to handler.
	The handler must be ready to handle the event prior to calling this function.
	Do NOT call this routine from an interrupt or exception handler.
RETURNS	OK, or ERROR if unable to register this handler
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_MEMORY_ERROR A memory allocation failed.
SEE ALSO	erfLib

erfHandlerUnregister()

NAME	erfHandlerUnregister() – Registers an event handler for a particular event.
SYNOPSIS	<pre>STATUS erfHandlerUnregister (UINT16 eventCat, /* Event Category */ UINT16 eventType, /* Event Type */ erfHandlerPrototype * pEventHandler, /* Pointer to Event Handler */ void * pUserData /* pointer to User data structure))</pre>
DESCRIPTION	This routine unregisters an event handler that was previously registered for an event. The handler pointer will be deleted. The userData will be returned to caller for destruction.
	The handler must be able to handle the event until this routine returns successfully.
	If multiple instances of this handler are registered for the same event, only the first instance matched in the database will be deleted.
RETURNS	OK, or ERROR if unable to unregister this handler
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_HANDLER_NOT_FOUND An event handler was not found.
SEE ALSO	erfLib

erfLibInit()

NAME	erfLibInit() – Initialize the Event Reporting Framework library
SYNOPSIS	<pre>STATUS erfLibInit (UINT16 maxUserCat, /* Maximum number of User Categories */ UINT16 maxUserType /* Maximum number of User Types */)</pre>
DESCRIPTION	This routine initializes the library to handle registrations of event handlers. The number of User defined Categories and Types is passed in by the caller.
RETURNS	OK or ERROR if an error occurred during initialization
ERRNO	S_erfLib_INIT_ERROR A general Initialization Error. S_erfLib_MEMORY_ERROR A memory allocation failed.
SEE ALSO	erfLib

erfShow()

NAME	erfShow() – Shows debug info for this library.
SYNOPSIS	<pre>void erfShow ()</pre>
DESCRIPTION	none
RETURNS	Nothing
ERRNO	
SEE ALSO	erfShow

erfTypeAllocate()

NAME	erfTypeAllocate() – Allocates a User Defined Type for this Category.
SYNOPSIS	<pre>STATUS erfTypeAllocate (UINT16 eventCat, /* Event Category */ UINT16 * pEventType /* pointer to returned Event Type */)</pre>
DESCRIPTION	This routine allocates a User Defined Event Type for a Category. Once defined, User defined Type cannot be deleted.
RETURNS	OK, or ERROR if unable to define this Event Type
ERRNO	S_erfLib_INVALID_PARAMETER A parameter was out of range. S_erfLib_TOO_MANY_USER_TYPES The number of User defined Event Categories exceeds the maximum number allowed for this category.
SEE ALSO	erfLib

erfTypesAvailable()

NAME	erfTypesAvailable() – Get the number of unallocated User Types for a category.
SYNOPSIS	<pre>UINT16 erfTypesAvailable (UINT16 eventCat /* Event Category */)</pre>
DESCRIPTION	none
RETURNS	Number of types or 0 for a bad category
ERRNO	
SEE ALSO	erfLib

evbNs16550HrdInit()

NAME **evbNs16550HrdInit()** – initialize the NS 16550 chip

SYNOPSIS

```
void evbNs16550HrdInit
(
    EVBNS16550_CHAN *pChan
)
```

DESCRIPTION This routine is called to reset the NS 16550 chip to a quiescent state.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **evbNs16550Sio**

evbNs16550Int()

NAME **evbNs16550Int()** – handle a receiver/transmitter interrupt for the NS 16550 chip

SYNOPSIS

```
void evbNs16550Int
(
    EVBNS16550_CHAN *pChan
)
```

DESCRIPTION This routine is called to handle interrupts. If there is another character to be transmitted, it sends it. If the interrupt handler is called erroneously (for example, if a device has never been created for the channel), it disables the interrupt.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **evbNs16550Sio**

fdDevCreate()

NAME **fdDevCreate()** – create a device for a floppy disk

SYNOPSIS

```
BLK_DEV *fdDevCreate
(
    int drive,      /* driver number of floppy disk (0 - 3) */
    int fdType,     /* type of floppy disk */
    int nBlocks,    /* device size in blocks (0 = whole disk) */
    int blkOffset   /* offset from start of device */
)
```

DESCRIPTION

This routine creates a device for a specified floppy disk.

The *drive* parameter is the drive number of the floppy disk; valid values are 0 to 3.

The *fdType* parameter specifies the type of diskette, which is described in the structure table **fdTypes[]** in **sysLib.c**. *fdType* is an index to the table. Currently the table contains two diskette types:

- An *fdType* of 0 indicates the first entry in the table (3.5" 2HD, 1.44MB);
- An *fdType* of 1 indicates the second entry in the table (5.25" 2HD, 1.2MB).

Members of the **fdTypes[]** structure are:

```
int sectors;          /* no of sectors */
int sectorsTrack;    /* sectors per track */
int heads;           /* no of heads */
int cylinders;       /* no of cylinders */
int secSize;          /* bytes per sector, 128 << secSize */
char gap1;            /* gap1 size for read, write */
char gap2;            /* gap2 size for format */
char dataRate;        /* data transfer rate */
char stepRate;        /* stepping rate */
char headUnload;     /* head unload time */
char headLoad;        /* head load time */
char mfm;             /* MFM bit for read, write, format */
char sk;               /* SK bit for read */
char *name;            /* name */
```

The *nBlocks* parameter specifies the size of the device, in blocks. If *nBlocks* is zero, the whole disk is used.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the floppy disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.) Normally, *blkOffset* is 0.

RETURNS

A pointer to a block device structure (**BLK_DEV**) or **NULL** if memory cannot be allocated for the device structure.

ERRNO

Not Available

SEE ALSO

nec765Fd, **fdDrv()**, **fdRawio()**, **dosFsMkfs()**, **dosFsDevInit()**, **rawFsDevInit()**

fdDrv()

NAME	fdDrv() – initialize the floppy disk driver
SYNOPSIS	STATUS fdDrv (int vector, /* interrupt vector */ int level /* interrupt level */)
DESCRIPTION	This routine initializes the floppy driver, sets up interrupt vectors, and performs hardware initialization of the floppy chip. This routine should be called exactly once, before any reads, writes, or calls to fdDevCreate() . Normally, it is called by usrRoot() in usrConfig.c .
RETURNS	OK.
ERRNO	Not Available
SEE ALSO	nec765Fd , fdDevCreate() , fdRawio()

fdRawio()

NAME	fdRawio() – provide raw I/O access
SYNOPSIS	STATUS fdRawio (int drive, /* drive number of floppy disk (0 - 3) */ int fdType, /* type of floppy disk */ FD_RAW *pFdRaw /* pointer to FD_RAW structure */)
DESCRIPTION	This routine is called when the raw I/O access is necessary. The <i>drive</i> parameter is the drive number of the floppy disk; valid values are 0 to 3. The <i>fdType</i> parameter specifies the type of diskette, which is described in the structure table fdTypes[] in sysLib.c . <i>fdType</i> is an index to the table. Currently the table contains two diskette types: <ul style="list-style-type: none">- An <i>fdType</i> of 0 indicates the first entry in the table (3.5" 2HD, 1.44MB);- An <i>fdType</i> of 1 indicates the second entry in the table (5.25" 2HD, 1.2MB). The <i>pFdRaw</i> is a pointer to the structure FD_RAW , defined in nec765Fd.h

RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	nec765Fd , fdDrv() , fdDevCreate()

fei82557DumpPrint()

NAME	fei82557DumpPrint() – Display statistical counters
SYNOPSIS	STATUS fei82557DumpPrint (int unit /* pointer to DRV_CTRL structure */)
DESCRIPTION	This routine displays i82557 statistical counters
RETURNS	OK, or ERROR if the DUMP command failed.
ERRNO	Not Available
SEE ALSO	fei82557End

fei82557EndLoad()

NAME	fei82557EndLoad() – initialize the driver and device
SYNOPSIS	END_OBJ* fei82557EndLoad (char *initString /* parameter string */)
DESCRIPTION	This routine initializes both, driver and device to an operational state using device specific parameters specified by <i>initString</i> . The parameter string, <i>initString</i> , is an ordered list of parameters each separated by a colon. The format of <i>initString</i> is, "unit:memBase:memSize:nCFDs:nRFDs:flags:offset:deviceId: maxRxFrames:c1ToRfdRatio:nClusters"

The 82557 shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.

A default number of transmit/receive frames of 32 and 128 respectively and can be selected by passing zero in the parameters *nTfds* and *nRfds*. In other cases, the number of frames selected should be greater than two.

All optional parameters can be set to their default value by specifying NONE (-1) as their value.

The *memBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The *memSize* parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive frames.

If the caller provides the shared memory region, then the driver assumes that this region is non-cached.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use **memalign()** to allocate some cache aligned memory.

The *memSize* parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of RFDs, RBDs, CFDs, and clusters specified. The number of clusters required will be at least equal to (*nRFDs* * 2) + *nCFDs*. Otherwise the End Load routine will return **ERROR**. The number of clusters can be specified by either passing a value in the *nCluster* parameter, in which case the *nCluster* value must be at least *nRFDs* * 2, or by setting the cluster to RFD ratio (*clToRfdRatio*) to a number equal or greater than 2.

The *nTfds* parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is less than two, a default of 64 is used.

The *nRfds* parameter specifies the number of receive descriptors to be allocated. If this parameter is less than two or NONE (-1) a default of 128 is used.

The *flags* parameter specifies the user flags may control the run-time characteristics of the Ethernet chip. Not implemented.

The *offset* parameter is used to align IP header on word boundary for CPUs that need long word aligned access to the IP packet (this will normally be zero or two). This parameter is optional, the default value is zero.

The *deviceId* parameter is used to indicate the specific type of device being used, the 82557 or subsequent. This is used to determine if features which were introduced after the 82557 can be used. The default is the 82557. If this is set to any value other than ZERO (0), NONE (-1), or FEI82557_DEVICE_ID (0x1229) it is assumed that the device will support features not in the 82557.

The *maxRxFrames* parameter limits the number of frames the receive handler will service in one pass. It is intended to prevent the tNetTask from monopolizing the CPU and starving applications. This parameter is optional, the default value is nRFDs * 2.

The *clToRfdRatio* parameter sets the number of clusters as a ratio of nRFDs. The minimum setting for this parameter is 2. This parameter is optional, the default value is 5.

The *nClusters* parameter sets the number of clusters to allocate. This value must be at least nRFD * 2. If this value is set then the *clToRfdRatio* is ignored. This parameter is optional, the default is nRFDs * clToRfdRatio.

RETURNS an END object pointer, or NULL on error.

ERRNO Not Available

SEE ALSO [fei82557End](#), [ifLib](#), Intel 82557 User's Manual

fei82557ErrCounterDump()

NAME [fei82557ErrCounterDump\(\)](#) – dump statistical counters

SYNOPSIS

```
STATUS fei82557ErrCounterDump
(
    DRV_CTRL * pDrvCtrl, /* pointer to DRV_CTRL structure */
    UINT32 *   memAddr
)
```

DESCRIPTION This routine dumps statistical counters for the purpose of debugging and tuning the 82557.

The *memAddr* parameter is the pointer to an array of 68 bytes in the local memory. This memory region must be allocated before this routine is called. The memory space must also be DWORD (4 bytes) aligned. When the last DWORD (4 bytes) is written to a value, 0xa007, it indicates the dump command has completed. To determine the meaning of each statistical counter, see the Intel 82557 manual.

RETURNS OK or ERROR.

ERRNO Not Available

SEE ALSO [fei82557End](#)

fei82557GetRUStatus()

NAME	fei82557GetRUStatus() – Return the current RU status and int mask
SYNOPSIS	void fei82557GetRUStatus (int unit)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	fei82557End

fei82557ShowRxRing()

NAME	fei82557ShowRxRing() – Show the Receive ring
SYNOPSIS	void fei82557ShowRxRing (int unit)
DESCRIPTION	This routine dumps the contents of the RFDs and RBDs in the Rx ring.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	fei82557End

g64120aMfRegister()

NAME	g64120aMfRegister() – register g64120aMf driver
SYNOPSIS	void g64120aMfRegister(void)

DESCRIPTION	This routine registers the g64120aMf driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	g64120aMf

g64120aMfpDrvCtrlShow()

NAME	g64120aMfpDrvCtrlShow() – show pDrvCtrl for g64120a system controller
SYNOPSIS	<pre>int g64120aMfpDrvCtrlShow (VXB_DEVICE_ID pInst)</pre>
DESCRIPTION	none
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	g64120aMf

g64120aPciRegister()

NAME	g64120aPciRegister() – register g64120aPci driver
SYNOPSIS	<pre>void g64120aPciRegister(void)</pre>
DESCRIPTION	This routine registers the g64120aPci driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO g64120aPci

gei82543EndLoad()

NAME gei82543EndLoad() – initialize the driver and device

SYNOPSIS

```
END_OBJ* gei82543EndLoad
(
    char *initString /* String to be parsed by the driver. */
)
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

The string contains the target specific parameters like this:
"unitnum:shmem_addr:shmem_size:rxDescNum:txDescNum:usrFlags:offset:mtu"

RETURNS an END object pointer, NULL if error, or zero

ERRNO Not Available

SEE ALSO gei82543End

gei82543LedOff()

NAME gei82543LedOff() – turn off LED

SYNOPSIS

```
void gei82543LedOff
(
    int unit /* device unit */
)
```

DESCRIPTION This routine turns LED off

RETURNS N/A

ERRNO Not Available

SEE ALSO [gei82543End](#)

gei82543LedOn()

NAME [gei82543LedOn\(\)](#) – turn on LED

SYNOPSIS

```
void gei82543LedOn
(
    int unit /* device unit */
)
```

DESCRIPTION This routine turns LED on

RETURNS N/A

ERRNO Not Available

SEE ALSO [gei82543End](#)

gei82543PhyRegGet()

NAME [gei82543PhyRegGet\(\)](#) – get the register value in PHY

SYNOPSIS

```
int gei82543PhyRegGet
(
    int unit, /* device unit */
    int reg   /* PHY's register */
)
```

DESCRIPTION This routine returns the PHY's register value, or -1 if an error occurs.

RETURNS PHY's register value

ERRNO Not Available

SEE ALSO [gei82543End](#)

gei82543PhyRegSet()

NAME gei82543PhyRegSet() – set the register value in PHY

SYNOPSIS

```
int gei82543PhyRegSet
(
    int      unit,    /* device unit */
    int      reg,     /* PHY's register */
    UINT16  tmp
)
```

DESCRIPTION This routine returns the PHY's register value, or -1 if an error occurs.

RETURNS PHY's register value

ERRNO Not Available

SEE ALSO gei82543End

gei82543RegGet()

NAME gei82543RegGet() – get the specified register value in 82543 chip

SYNOPSIS

```
UINT32 gei82543RegGet
(
    int      unit,    /* device unit */
    UINT32  offset   /* register offset */
)
```

DESCRIPTION This routine gets and shows the specified register value in 82543 chip

RETURNS Register value

ERRNO Not Available

SEE ALSO gei82543End

gei82543RegSet()

NAME	gei82543RegSet() – set the specified register value
SYNOPSIS	<pre>void gei82543RegSet (int unit, /* device unit */ UINT32 offset, /* register offset */ UINT32 regVal /* value to write */)</pre>
DESCRIPTION	This routine sets the specified register value
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	gei82543End

gei82543TbiCompWr()

NAME	gei82543TbiCompWr() – enable/disable the TBI compatibility workaroud
SYNOPSIS	<pre>void gei82543TbiCompWr (int unit, /* device unit */ int flag /* 0 - off, and others on */)</pre>
DESCRIPTION	This routine enables/disables TBI compatibility workaroud if needed Input: unit - unit number of the gei device flag - 0 to turn off TBI compatibility, others to turn on
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	gei82543End

gei82543Unit()

NAME	gei82543Unit() – return a pointer to the END_DEVICE for a gei unit
SYNOPSIS	<pre>END_DEVICE * gei82543Unit (int unit)</pre>
DESCRIPTION	none
RETURNS	A pointer the END_DEVICE for the unit, or NULL.
ERRNO	Not Available
SEE ALSO	gei82543End

hardWareInterFaceInit()

NAME	hardWareInterFaceInit() – Hardware Interface Pre-Kernel Initialization
SYNOPSIS	<pre>void hardWareInterFaceInit (void)</pre>
DESCRIPTION	This function performs the pre-kernel harware interface initialization
RETURNS	None
ERRNO	None
SEE ALSO	cmdLineBuild

hcfDeviceGet()

NAME	hcfDeviceGet() – get the HCF_DEVICE pointer
SYNOPSIS	<pre>HCF_DEVICE * hcfDeviceGet (VXB_DEVICE_ID pInst)</pre>

DESCRIPTION	Search the device entry table for a match
RETURNS	pointer to HCF_DEVICE or NULL if device is not located
ERRNO	
SEE ALSO	hwConfig

hcfResourceAllShow()

NAME	hcfResourceAllShow() – show all devices and resource values
SYNOPSIS	void hcfResourceAllShow(void)
DESCRIPTION	This routines displays the device and resource values for all HCF records.
RETURNS	N/A
ERRNO	
SEE ALSO	hwConfig

hcfResourceDevShow()

NAME	hcfResourceDevShow() – show the device and resource values
SYNOPSIS	void hcfResourceDevShow (const struct hcfDevice * pHcf /* HCF record */)
DESCRIPTION	This routine displays the device and the resource values associated with the specified resource.
RETURNS	N/A
ERRNO	
SEE ALSO	hwConfig

hcfResourceShow()

NAME **hcfResourceShow()** – show values for specified resource

SYNOPSIS

```
void hcfResourceShow
(
    const struct hcfResource * pRes /* HCF record */
)
```

DESCRIPTION This routine displays the values of the specified resource.

RETURNS N/A

ERRNO

SEE ALSO [hwConfig](#)

hwMemAlloc()

NAME **hwMemAlloc()** – allocate a buffer from the hardware memory pool

SYNOPSIS

```
char * hwMemAlloc
(
    int reqSize
)
```

DESCRIPTION This function allocates a buffer from the hardware memory pool. Buffers from this pool have one of the following sizes:

48 bytes

112 bytes

240 bytes

496 bytes

1008 bytes

2032 bytes

These sizes are power-of-two minus 12. Each buffer is at least 4-byte aligned. Immediately before and after each buffer is a 4-byte sentinel. Immediately before the initial sentinel is a pointer to the next buffer in the chain.

This results in an 8-byte sentinel between each buffer, in order to detect buffer overrun or underrun.

RETURNS NULL, or pointer to allocated memory

ERRNO

SEE ALSO **hwMemLib**

hwMemFree()

NAME **hwMemFree()** – return buffer to the hardware memory pool

SYNOPSIS STATUS hwMemFree
 (
 char * pMem
)

DESCRIPTION This routine returns a buffer to the hardware memory pool.

First, this routine checks each pool, to see which one the buffer was allocated from. If the buffer did not come from any hardware memory pool, this routine returns **ERROR**.

Next, this routine finds the size of the buffer by searching through each **pAlloc[i]** list. When the buffer is found, this routine locks interrupts, removes the buffer from this list, and adds it to the **pHead[i]** list.

RETURNS OK, or **ERROR**

ERRNO

SEE ALSO **hwMemLib**

hwMemLibInit()

NAME **hwMemLibInit()** – initialize hardware memory allocation library

SYNOPSIS void hwMemLibInit(void)

DESCRIPTION This function initializes the hardware memory allocation library.

RETURNS N/A

ERRNO

SEE ALSO **hwMemLib**

hwMemPoolCreate()

NAME **hwMemPoolCreate()** – create or add a memory pool for driver memory allocation

SYNOPSIS

```
STATUS hwMemPoolCreate
(
    char * pMem,
    int    size
)
```

DESCRIPTION This function creates a memory pool for use by **hwMemLib** if one does not already exist. If one does exist, this function creates an additional pool which can be used if/when the previous pool(s) have insufficient space available.

RETURNS OK

ERRNO

SEE ALSO **hwMemLib**

hwMemShow()

NAME **hwMemShow()** –

SYNOPSIS

```
void hwMemShow(void)
```

DESCRIPTION none

RETURNS N/A

ERRNO

SEE ALSO **hwMemLib**

hwifDevCfgRead()

NAME **hwifDevCfgRead()** – read from device configuration space

SYNOPSIS

```
int hwifDevCfgRead
```

```

(
VXB_DEVICE_ID pInst,    /* DEVICE ID */
UINT32        size,     /* size of transaction */
UINT32        offset,   /* offset into register set */
UINT32        flagVal  /* flags */
)

```

DESCRIPTION This routine reads the specified size value from the specified offset into device configuration space, and prints the value on stdout.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbHwifDebug](#)

hwifDevCfgWrite()

NAME **hwifDevCfgWrite()** – write device configuration space

SYNOPSIS

```

int hwifDevCfgWrite
(
VXB_DEVICE_ID pInst,    /* DEVICE ID */
UINT32        size,     /* size of transaction */
UINT32        offset,   /* offset into register set */
UINT32        val,      /* value to write */
UINT32        flagVal  /* flags */
)

```

DESCRIPTION This routine writes the specified size value into device configuration space.

This routine supports only 8-bit, 16-bit, and 32-bit operations. Use **hwifDevCfgWrite64()** for 64-bit write operations.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxHwifDebug**

hwifDevCfgWrite64()

NAME **hwifDevCfgWrite64()** – write device register

SYNOPSIS

```
int hwifDevCfgWrite64(
    (
        VXB_DEVICE_ID pInst,      /* DEVICE ID */
        UINT32          offset,   /* offset into register set */
        UINT32          valLow,   /* value to write */
        UINT32          valHigh,  /* value to write */
        UINT32          flagVal   /* flags */
    )
)
```

DESCRIPTION This routine writes a 64-bit value into a device register, using the 64-bit vxBus register write routine.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxHwifDebug**

hwifDevRegRead()

NAME **hwifDevRegRead()** – read device register

SYNOPSIS

```
int hwifDevRegRead
(
    VXB_DEVICE_ID pInst,          /* DEVICE ID */
    UINT32 size,                 /* size of transaction */
    UINT32 regBaseIndex,         /* INDEX! of register set */
    UINT32 offset,               /* offset into register set */
    UINT32 flagVal              /* flags */
)
```

DESCRIPTION This routine reads the specified size value from a device register, using the appropriate vxBus register read routine.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbHwifDebug](#)

hwifDevRegWrite()

NAME **hwifDevRegWrite()** – write device register

SYNOPSIS

```
int hwifDevRegWrite
(
    VXB_DEVICE_ID pInst,          /* DEVICE ID */
    UINT32 size,                 /* size of transaction */
    UINT32 regBaseIndex,         /* INDEX! of register set */
    UINT32 offset,               /* offset into register set */
    UINT32 val,                  /* value to write */
    UINT32 flagVal              /* flags */
)
```

DESCRIPTION This routine writes the specified size value into a device register, using the appropriate vxBus register read routine.

This routine supports only 8-bit, 16-bit, and 32-bit operations. Use **hwifDevRegWrite64()** for 64-bit write operations.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxHwifDebug](#)

hwifDevRegWrite64()

NAME **hwifDevRegWrite64()** – write device register

SYNOPSIS

```
int hwifDevRegWrite64(
    (
        VXB_DEVICE_ID pInst,          /* DEVICE ID */
        UINT32 regBaseIndex,         /* INDEX! of register set */
        UINT32 offset,              /* offset into register set */
        UINT32 valLow,              /* value to write */
        UINT32 valHigh,              /* value to write */
        UINT32 flagVal              /* flags */
    )
)
```

DESCRIPTION This routine writes a 64-bit value into a device register, using the 64-bit vxBus register write routine.

RETURNS: 0, always

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbHwifDebug**

i8250HrdInit()

NAME **i8250HrdInit()** – initialize the chip

SYNOPSIS

```
void i8250HrdInit
(
    I8250_CHAN * pChan /* pointer to device */
)
```

DESCRIPTION This routine is called to reset the chip in a quiescent state.

RETURNS N/A

ERRNO Not Available

SEE ALSO **i8250Sio**

i8250Int()

NAME **i8250Int()** – handle a receiver/transmitter interrupt

SYNOPSIS

```
void i8250Int
(
    I8250_CHAN * pChan
)
```

DESCRIPTION This routine handles four sources of interrupts from the UART. If there is another character to be transmitted, the character is sent. When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

ERRNO Not Available

SEE ALSO **i8250Sio**

iOlicomEndLoad()

NAME	iOlicomEndLoad() – initialize the driver and device
SYNOPSIS	<pre>END_OBJ * iOlicomEndLoad (char * initString /* String to be parsed by the driver. */)</pre>
DESCRIPTION	<p>This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.</p> <p>This routine can be called in two modes. If it is called with an empty, but allocated string then it places the name of this device (i.e. oli) into the initString and returns 0.</p> <p>If the string is allocated then the routine attempts to perform its load functionality.</p>
RETURNS	An END object pointer or NULL on error or 0 and the name of the device if the initString was NULL.
ERRNO	Not Available
SEE ALSO	iOlicomEnd

iOlicomIntHandle()

NAME	iOlicomIntHandle() – interrupt service for card interrupts
SYNOPSIS	<pre>void iOlicomIntHandle (END_DEVICE * pDrvCtrl /* pointer to END_DEVICE structure */)</pre>
DESCRIPTION	This routine is called when an interrupt has been detected from the Olicom card.
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	iOlicomEnd

iPIIX4AtaInit()

NAME	iPIIX4AtaInit() – low level initialization of ATA device
SYNOPSIS	STATUS iPIIX4AtaInit ()
DESCRIPTION	This routine will initialize PIIX4 - PCI-ISA/IDE bridge for proper working of ATA device.
RETURNS	OK or ERROR.
ERRNO	
SEE ALSO	iPIIX4

iPIIX4FdInit()

NAME	iPIIX4FdInit() – initializes the floppy disk device
SYNOPSIS	STATUS iPIIX4FdInit ()
DESCRIPTION	This routine will initialize PIIX4 - PCI-ISA/IDE bridge and DMA for proper working of floppy disk device
RETURNS	OK or ERROR.
ERRNO	
SEE ALSO	iPIIX4

iPIIX4GetIntr()

NAME	iPIIX4GetIntr() – give device an interrupt level to use
SYNOPSIS	char iPIIX4GetIntr

```
(  
    int pintx  
)
```

DESCRIPTION	This routine will give device an interrupt level to use based on PCI INT A through D, valid values for pintx are 0, 1, 2 and 3. An autoroute in disguise.
RETURNS	char - interrupt level
ERRNO	
SEE ALSO	iPIIX4

iPIIX4Init()

NAME	iPIIX4Init() – initialize PIIX4
SYNOPSIS	<pre>STATUS iPIIX4Init ()</pre>
DESCRIPTION	initialize PIIX4
RETURNS	OK or ERROR.
ERRNO	
SEE ALSO	iPIIX4

iPIIX4IntrRoute()

NAME	iPIIX4IntrRoute() – Route PIRQ[A:D]
SYNOPSIS	<pre>STATUS iPIIX4IntrRoute (int pintx, char irq)</pre>
DESCRIPTION	This routine will connect an irq to a pci interrupt.

RETURNS OK or ERROR.

ERRNO

SEE ALSO iPIIX4

iPIIX4KbdInit()

NAME iPIIX4KbdInit() – initializes the PCI-ISA/IDE bridge

SYNOPSIS STATUS iPIIX4KbdInit
(
)

DESCRIPTION This routine will initialize PIIX4 - PCI-ISA/IDE bridge to enable keyboard device and IRQ routing

RETURNS OK or ERROR.

ERRNO

SEE ALSO iPIIX4

ibmPciConfigRead()

NAME ibmPciConfigRead() – reads one PCI configuration space register

SYNOPSIS STATUS ibmPciConfigRead
(
int bus,
int device,
int function,
int offset, /* offset into the configuration space */
int width, /* data width */
void *pResult
)

DESCRIPTION This routine reads one PCI configuration register of the specified size (1, 2 or 4 bytes) using a Type 0 or 1 configuration transaction.

RETURNS OK, always

ERRNO Not Available

SEE ALSO [ppc440gpPci](#)

ibmPciConfigWrite()

NAME **ibmPciConfigWrite()** – write to one PCI configuration register location

SYNOPSIS

```
STATUS ibmPciConfigWrite
(
    int bus,
    int device,
    int function,
    int offset,    /* offset into the configuration space */
    int width,    /* data width */
    UINT data     /* data to be written */
)
```

DESCRIPTION This routine writes one PCI configuration space location of the specified size (1, 2 or 4 bytes) using a Type 0 or 1 configuration transaction.

RETURNS OK, always

ERRNO Not Available

SEE ALSO [ppc440gpPci](#)

ibmPciSpecialCycle()

NAME **ibmPciSpecialCycle()** – generates a PCI special cycle

SYNOPSIS

```
STATUS ibmPciSpecialCycle
(
    int bus,
    UINT32 data
)
```

DESCRIPTION This routine generates a PCI special cycle on the PCI bus.

RETURNS OK, always

ERRNO Not Available

SEE ALSO [ppc440gpPci](#)

intCtlrChainISR()

NAME [intCtlrChainISR\(\)](#) – process interrupt chain

SYNOPSIS

```
void intCtlrChainISR
(
    void * pArg,
    int   inputPin /* unused */
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbIntCtlrLib](#)

intCtlrHwConfGet()

NAME [intCtlrHwConfGet\(\)](#) – get interrupt input pin configuration from BSP

SYNOPSIS

```
STATUS intCtlrHwConfGet
(
    VXB_DEVICE_ID      pInst,
    HCF_DEVICE *       pHcf,
    struct intCtlrHwConf * pEntries
)
```

DESCRIPTION This routine reads entries from the hcf record for the intCtlr device specified by pInst, and fills them in to a table.

RETURNS OK, or ERROR if any component table cannot be allocated

ERRNO Not Available

SEE ALSO [vxblntCtlrLib](#)

intCtlrHwConfShow()

NAME [**intCtlrHwConfShow\(\)**](#) – print interrupt input pin configuration

SYNOPSIS

```
STATUS intCtlrHwConfShow
(
    struct intCtlrHwConf * pEntries
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxblntCtlrLib](#)

intCtlrISRAdd()

NAME [**intCtlrISRAdd\(\)**](#) – Install ISR at specified pin

SYNOPSIS

```
STATUS intCtlrISRAdd
(
    struct intCtlrHwConf * pEntries,
    int                  inputPin,
    void                (*isr),
    void *              pArg
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxblntCtlrLib](#)

intCtrlISRDisable()

NAME	intCtrlISRDisable() – Disable specified ISR
SYNOPSIS	<pre>BOOL intCtrlISRDisable (struct intCtrlrHwConf * pEntries, int inputPin, void (*isr), void * pArg)</pre>
DESCRIPTION	This routine disables the specified ISR. If all ISRs on the chain are disabled, then this routine returns TRUE. The interrupt controller driver calling this routine should check the return value, and disable the input pin in hardware if no ISRs on this chain are enabled.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbIntCtrlLib

intCtrlISREnable()

NAME	intCtrlISREnable() – Enable specified ISR
SYNOPSIS	<pre>STATUS intCtrlISREnable (struct intCtrlrHwConf * pEntries, int inputPin, void (*isr), void * pArg)</pre>
DESCRIPTION	This routine marks the specified ISR as enabled and callable when an interrupt occurs on this input. If the input is chained, then the specified ISR is enabled as well as the top-level chain handler.
RETURNS	Not Available

ERRNO Not Available

SEE ALSO **vxblntCtlrLib**

intCtlrISRRemove()

NAME **intCtlrISRRemove()** – Remove ISR from specified pin

SYNOPSIS

```
STATUS intCtlrISRRemove
(
    struct intCtlrHwConf * pEntries,
    int                  inputPin,
    void                (*isr),
    void *              pArg
)
```

DESCRIPTION This routine removes the specified ISR and argument from the specified input pin. If this is the last ISR, then the stray handler is re-installed. Note that once a chain has been created, the only way to remove it is to remove the last ISR. We do not replace a single-element chain with a direct call.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxblntCtlrLib**

intCtlrPinFind()

NAME **intCtlrPinFind()** – Retrieve input pin associated with device

SYNOPSIS

```
int intCtlrPinFind
(
    VXB_DEVICE_ID      pDev,
    int                pinIndex,
    VXB_DEVICE_ID      pIntCtlrCaller,
    struct intCtlrHwConf * pConf
)
```

DESCRIPTION This routine returns the input pin on the controller device, to which the specified device is connected. If no device is connected, this routine returns -1.

Note that the pDev passed to this function is the **VXB_DEVICE_ID** of the attached device, not the **VXB_DEVICE_ID** of the interrupt controller instance.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntCtlrLib**

intCtlrStrayISR()

NAME **intCtlrStrayISR()** – flag stray interrupt

SYNOPSIS

```
void intCtlrStrayISR
(
    void * pArg,
    int    inputPin
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbIntCtlrLib**

intCtlrTableArgGet()

NAME **intCtlrTableArgGet()** – Fetch ISR argument from ISR table entry

SYNOPSIS

```
void * intCtlrTableArgGet
```

```
(  
    struct intCtlrHwConf * pInputs,  
    int                  inputPin  
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxBI ntCtlrLib](#)

intCtlrTableCreate()

NAME [**intCtlrTableCreate\(\)**](#) – Create table structure

SYNOPSIS STATUS intCtlrTableCreate
(
 struct intCtlrHwConf * pInputs,
 int inputPin
)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxBI ntCtlrLib](#)

intCtlrTableFlagsGet()

NAME [**intCtlrTableFlagsGet\(\)**](#) – Fetch ISR flags from ISR table entry

SYNOPSIS UINT32 intCtlrTableFlagsGet
(
 struct intCtlrHwConf * pInputs,
 int inputPin
)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO vxbIntCtrlrLib

intCtrlrTableFlagsSet()

NAME intCtrlrTableFlagsSet() – Set ISR flags in ISR table entry

SYNOPSIS

```
void intCtrlrTableFlagsSet
(
    struct intCtrlrHwConf * pInputs,
    int                  inputPin,
    UINT32               flagValue
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO vxbIntCtrlrLib

intCtrlrTableIsrGet()

NAME intCtrlrTableIsrGet() – Fetch ISR function pointer from ISR table entry

SYNOPSIS

```
void * intCtrlrTableIsrGet
(
    struct intCtrlrHwConf * pInputs,
    int                  inputPin
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxblntCtlrLib](#)

intCtlrTableUserSet()

NAME [**intCtlrTableUserSet\(\)**](#) – Fill in ISR table entry: driver name, unit, index

SYNOPSIS

```
void intCtlrTableUserSet
(
    struct intCtlrHwConf * pInputs,
    int                  inputPin,
    char *               drvName,
    int                  drvUnit,
    int                  drvIndex
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxblntCtlrLib](#)

isrDeferIsrReroute()

NAME [**isrDeferIsrReroute\(\)**](#) – Reroute deferral work to a new CPU in the system.

SYNOPSIS

```
ISR_DEFER_QUEUE_ID isrDeferIsrReroute
(
    ISR_DEFER_QUEUE_ID queueId, /* queue allocated by isrDeferQueueGet */
    int                cpuldx   /* CPU to handle this queue's interrrupts */
)
```

DESCRIPTION This routine is called by a driver when the driver's interrupts have been rerouted to a different CPU within the SMP system. The driver provides the previous queue ID it has been using for this interrupt channel, and receives a new queue in exchange.

RETURNS **ISR_DEFER_QUEUE_ID**, or **NULL** if no queue available after reroute.

ERRNO Not Available

SEE ALSO [isrDeferLib](#)

isrDeferJobAdd()

NAME [isrDeferJobAdd\(\)](#) – Add a job to the deferral queue

SYNOPSIS

```
void isrDeferJobAdd
(
    ISR_DEFER_QUEUE_ID queueId, /* queue allocated by isrDeferQueueGet */
    ISR_DEFER_JOB *      pJob     /* job to enqueue */
)
```

DESCRIPTION This routine adds a job to a deferral queue. The caller is responsible for allocating the **ISR_DEFER_JOB** storage, and for ensuring that the deferred work has completed before attempting to reuse the storage. It is an error to enqueue an **ISR_DEFER_JOB** structure more than once.

RETURNS N/A

ERRNO Not Available

SEE ALSO [isrDeferLib](#)

isrDeferLibInit()

NAME [isrDeferLibInit\(\)](#) – Initialize the ISR deferral library.

SYNOPSIS

```
STATUS isrDeferLibInit
(
    int mode /* global deferral queue mode */
)
```

DESCRIPTION This routine initializes the ISR deferral library. Queues are allocated on an as-needed basis, so no queues are created during initialization.

RETURNS OK, if initialization succeeds, otherwise ERROR.

ERRNO Not Available

SEE ALSO

isrDeferLib

isrDeferQueueGet()

NAME **isrDeferQueueGet()** – Get a deferral queue

SYNOPSIS

```
ISR_DEFER_QUEUE_ID isrDeferQueueGet
(
    VXB_DEVICE_ID pInst, /* VxBus device ID of requester */
    int         intIdx, /* interrupt source index */
    int         cpuIdx, /* CPU index for deferral task */
    int         mode     /* deferral queue mode(for future use) */
)
```

DESCRIPTION none

RETURNS A deferral queue, or NULL if a queue could not be created.

ERRNO Not Available

SEE ALSO **isrDeferLib**

isrRerouteNotify()

NAME **isrRerouteNotify()** – Notify driver that ISR has been rerouted

SYNOPSIS

```
STATUS isrRerouteNotify
(
    struct intCtlrHwConf * pInputs,
    int                 inputPin,
    int                 cpuNo
)
```

DESCRIPTION This routine walks the chain of ISRs connected to a specific interrupt input in the isrHandle. For each instance with a connected ISR, the routine calls the {isrRerouteNotify}() driver method, if one has been published.

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbIntCtlrLib](#)

ixp400I2CAckReceive (Control Signal)()

NAME [**ixp400I2CAckReceive \(Control Signal\)\(\)**](#) – get an acknowledgement

SYNOPSIS STATUS `ixp400I2CAckReceive (void)`

DESCRIPTION This routine gets an acknowledgement from the I2C bus.

RETURNS OK if an acknowledgement is received, else ERROR.

ERRNO Not Available

SEE ALSO [ixp400I2c](#)

ixp400I2CAckSend (Control signal)()

NAME [**ixp400I2CAckSend \(Control signal\)\(\)**](#) – send an acknowledgement

SYNOPSIS void `ixp400I2CAckSend (void)`

DESCRIPTION This routine sends an acknowledgement on the I2C bus.

RETURNS N/A

ERRNO Not Available

SEE ALSO [ixp400I2c](#)

ixp400I2CByteReceive()

NAME [**ixp400I2CByteReceive\(\)**](#) – receive a byte on the I2C bus

SYNOPSIS	<pre>void ixp400I2CByteReceive (unsigned char * dataByte)</pre>
DESCRIPTION	This routine receives a byte of data from the I2C bus and stores it to the location specified by the <i>dataByte</i> parameter. All byte transfers are Most Significant Bit (MSB) first.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ixp400I2c

ixp400I2CByteTransmit()

NAME	ixp400I2CByteTransmit() – transmit a byte on the I2C bus
SYNOPSIS	<pre>void ixp400I2CByteTransmit (unsigned char dataByte)</pre>
DESCRIPTION	This routine transmits a specified <i>dataByte</i> on the I2C bus. All byte transfers are Most Significant Bit (MSB) first.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ixp400I2c

ixp400I2CReadTransfer()

NAME	ixp400I2CReadTransfer() – read from a slave device
SYNOPSIS	<pre>int ixp400I2CReadTransfer (UINT8 devAddr, UINT8 * buffer, UINT32 num,</pre>

```
    UINT8    offset  
)
```

DESCRIPTION	This routine reads <i>num</i> bytes into <i>buffer</i> from <i>offset</i> in a slave device with address <i>devAddr</i> .
RETURNS	The number of bytes actually read, which may be zero or less than <i>num</i> in the event of an error.
ERRNO	Not Available
SEE ALSO	ixp400I2c

ixp400I2CStart (Control signal)()

NAME	ixp400I2CStart (Control signal)() – initiate an I2C bus transfer
SYNOPSIS	<code>STATUS ixp400I2CStart (void)</code>
DESCRIPTION	This routine initiates a transfer on the I2C bus. This should only be called from a master device (i.e. GPIO controller on IXP400).
RETURNS	OK when the bus is free, else ERROR if the bus is already in use by another task.
ERRNO	Not Available
SEE ALSO	ixp400I2c

ixp400I2CStop (Control signal)()

NAME	ixp400I2CStop (Control signal)() – terminate an I2C bus transfer
SYNOPSIS	<code>void ixp400I2CStop (void)</code>
DESCRIPTION	This routine terminates a transfer on the I2C bus. The I2C bus will be left in a free state; namely, SCL HIGH and SDA HIGH. This should only be called from a master device (i.e. GPIO controller on IXP400)
RETURNS	N/A

ERRNO Not Available

SEE ALSO [ixp400I2c](#)

ixp400I2CWriteTransfer()

NAME [**ixp400I2CWriteTransfer\(\)**](#) – write to a slave device

SYNOPSIS

```
int ixp400I2CWriteTransfer
(
    UINT8    devAddr,
    UINT8 * buffer,
    UINT32   num,
    UINT8    offset
)
```

DESCRIPTION This routine writes *num* bytes from *buffer* to *offset* in a slave device with address *devAddr*.

RETURNS The number of bytes actually written, which may be zero or less than *num* in the event of an error.

ERRNO Not Available

SEE ALSO [ixp400I2c](#)

ixp400SioDevInit()

NAME [**ixp400SioDevInit\(\)**](#) – initialise a UART channel

SYNOPSIS

```
void ixp400SioDevInit
(
    IXP400_SIO_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine initialises some **SIO_CHAN** function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialised all the device addresses, etc. in the **IXP400_SIO_CHAN** structure.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxbIxp400Sio](#)

ixp400SioInt()

NAME `ixp400SioInt()` – interrupt level processing

SYNOPSIS

```
void ixp400SioInt
(
    IXP400_SIO_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This function handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register **Receiver Line Status**, **Received Data Ready**, 'Transmit Holding Register Empty' and **Modem Status**.

When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxbIxp400Sio](#)

ixp400SioIntErr()

NAME `ixp400SioIntErr()` – handle an error interrupt

SYNOPSIS

```
void ixp400SioIntErr
(
    IXP400_SIO_CHAN * pChan, /* pointer to channel */
    char           intErr
)
```

DESCRIPTION This routine handles error interrupts from the UART.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxblxp400Sio](#)

ixp400SioIntWr()

NAME **ixp400SioIntWr()** – handle a transmitter interrupt

SYNOPSIS

```
void ixp400SioIntWr
(
    IXP400_SIO_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles write interrupts from the UART. It reads a character and puts it in the transmit holding register of the device for transfer. If there are no more characters to transmit, transmission is disabled by clearing the transmit interrupt enable bit in the IER(int enable register). Any pending receive data is also serviced.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxblxp400Sio](#)

ixp400SioRegister()

NAME **ixp400SioRegister()** – register **ixp400Sio** driver

SYNOPSIS

```
void ixp400SioRegister (void)
```

DESCRIPTION This routine registers the **ixp400Sio** driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO [vxblxp400Sio](#)

ixp400SioStatsShow()

NAME	ixp400SioStatsShow() – display UART error statistics.
SYNOPSIS	void ixp400SioStatsShow (IXP400_SIO_CHAN * pChan)
DESCRIPTION	PRE: This routine requires that printf() is available.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbIxp400Sio

ixp400TimerDrvRegister()

NAME	ixp400TimerDrvRegister() – register ixp400 timer driver
SYNOPSIS	void ixp400TimerDrvRegister(void)
DESCRIPTION	This routine registers the ixp400 timer driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbIxp400Timer

In97xEndLoad()

NAME	In97xEndLoad() – initialize the driver and device
SYNOPSIS	END_OBJ * In97xEndLoad (char * initString /* string to be parse by the driver */)

DESCRIPTION	This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in <i>initString</i> , which expects a string of the following format: <code><unit:devMemAddr:devIoAddr:pciMemBase:vecnum:intLvl:memAdrs :memSize:memWidth:csr3b:offset:flags></code>
	This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "InPci") into the <i>initString</i> and returns 0.
	If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.
RETURNS	An END object pointer, or NULL on error, or 0 and the name of the device if the <i>initString</i> was NULL.
ERRNO	Not Available
SEE ALSO	In97xEnd

In97xInitParse()

NAME	In97xInitParse() – parse the initialization string
SYNOPSIS	<pre>STATUS In97xInitParse (LN_97X_DRV_CTRL * pDrvCtrl, /* pointer to the control structure */ char * initString /* initialization string */)</pre>
DESCRIPTION	Parse the input string. This routine is called from In97xEndLoad() which initializes some values in the driver control structure with the values passed in the initialization string. The initialization string format is: <code><unit:devMemAddr:devIoAddr:pciMemBase:vecNum:intLvl:memAdrs :memSize:memWidth:csr3b:offset:flags></code>
	<i>unit</i> The device unit number. Unit numbers are integers starting at zero and increasing for each device controlled by the driver.
	<i>devMemAddr</i> The device memory mapped I/O register base address. Device registers must be mapped into the host processor address space in order for the driver to be functional. Thus, this is a required parameter.

<i>devIoAddr</i>	Device register base I/O address (obsolete).
<i>pciMemBase</i>	Base address of PCI memory space.
<i>vecNum</i>	Interrupt vector number.
<i>intLvl</i>	Interrupt level. Generally, this value specifies an interrupt level defined for an external interrupt controller.
<i>memAdrs</i>	Memory pool address or NONE.
<i>memSize</i>	Memory pool size or zero.
<i>memWidth</i>	Memory system size, 1, 2, or 4 bytes (optional).
<i>CSR3</i>	Control and Status Register 3 (CSR3) options.
<i>offset</i>	Memory alignment offset.
<i>flags</i>	Device specific flags reserved for future use.
RETURNS	OK, or ERROR if any arguments are invalid.
ERRNO	Not Available
SEE ALSO	ln97xEnd

IptDevCreate()

NAME	<i>IptDevCreate()</i> – create a device for an LPT port
SYNOPSIS	<pre>STATUS lptDevCreate (char *name, /* name to use for this device */ int channel /* physical channel for this device (0 - 2) */)</pre>

DESCRIPTION	This routine creates a device for a specified LPT port. Each port to be used should have exactly one device associated with it by calling this routine.
	For instance, to create the device /lpt/0 , the proper call would be:
	<pre>lptDevCreate (" /lpt/0 ", 0);</pre>
RETURNS	OK, or ERROR if the driver is not installed, the channel is invalid, or the device already exists.
ERRNO	Not Available
SEE ALSO	lptDrv, lptDrv()

lptDrv()

NAME	lptDrv() – initialize the LPT driver
SYNOPSIS	<pre>STATUS lptDrv (int channels, /* LPT channels */ LPT_RESOURCE *pResource /* LPT resources */)</pre>
DESCRIPTION	This routine initializes the LPT driver, sets up interrupt vectors, and performs hardware initialization of the LPT ports. This routine should be called exactly once, before any reads, writes, or calls to lptDevCreate() . Normally, it is called by usrRoot() in usrConfig.c .
RETURNS	OK, or ERROR if the driver cannot be installed.
ERRNO	Not Available
SEE ALSO	lptDrv, lptDevCreate()

lptShow()

NAME	lptShow() – show LPT statistics
SYNOPSIS	<pre>void lptShow</pre>

```
(  
    UINT channel /* channel (0 - 2) */  
)
```

DESCRIPTION	This routine shows statistics for a specified LPT port.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	lptDrv

m548xDmaDrvRegister()

NAME	m548xDmaDrvRegister() – register coldFire DMA driver
SYNOPSIS	<code>void m548xDmaDrvRegister (void)</code>
DESCRIPTION	This routine registers the coldFire DMA driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbM548xDma

m54x5TimerDrvRegister()

NAME	m54x5TimerDrvRegister() – register coldFire timer driver
SYNOPSIS	<code>void m54x5TimerDrvRegister(void)</code>
DESCRIPTION	This routine registers the coldFire timer driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbM54x5SliceTimer

m8260SccEndLoad()

NAME	m8260SccEndLoad() – initialize the driver and device
SYNOPSIS	<pre>END_OBJ * m8260SccEndLoad (char * initString)</pre>
DESCRIPTION	<p>This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the <i>initString</i>, which is of the following format:</p> <p><i>unit:motCpmAddr:ivec:sccNum:txBdNum:rxBdNum:txBdBase:rxBdBase:bufBase</i></p> <p>The parameters of this string are individually described in the motCpmEnd man page.</p> <p>The SCC shares a region of memory with the driver. The caller of this routine can specify the address of a non-cacheable memory region with <i>bufBase</i>. Or, if this parameter is "NONE", the driver obtains this memory region by making calls to cacheDmaMalloc(). Non-cacheable memory space is important whenever the host processor uses cache memory. This is also the case when the MC68EN360 is operating in companion mode and is attached to a processor with cache memory.</p> <p>After non-cacheable memory is obtained, this routine divides up the memory between the various buffer descriptors (BDs). The number of BDs can be specified by <i>txBdNum</i> and <i>rxBdNum</i>, or if "NULL", a default value of 32 BDs will be used. An additional number of buffers are reserved as receive loaner buffers. The number of loaner buffers is a default number of 16.</p> <p>The user must specify the location of the transmit and receive BDs in the processor's dual ported RAM. <i>txBdBase</i> and <i>rxBdBase</i> give the offsets from <i>motCpmAddr</i> for the base of the BD rings. Each BD uses 8 bytes. Care must be taken so that the specified locations for Ethernet BDs do not conflict with other dual ported RAM structures.</p> <p>Multiple individual device units are supported by this driver. Device units can reside on different chips, or could be on different SCCs within a single processor. The <i>sccNum</i> parameter is used to explicitly state which SCC is being used. SCC1 is most commonly used, thus this parameter most often equals "1".</p> <p>Before this routine returns, it connects up the interrupt vector <i>ivec</i>.</p>
RETURNS	An END object pointer or NULL on error.
ERRNO	Not Available
SEE ALSO	m8260SccEnd , <i>Motorola MPC8260 User's Manual</i>

m85xxCpuRegister()

NAME	m85xxCpuRegister() – register m85xxCpu driver
SYNOPSIS	<code>void m85xxCpuRegister(void)</code>
DESCRIPTION	This routine registers the m85xxCpu driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	m85xxCpu

m85xxRioRegDbgRead()

NAME	m85xxRioRegDbgRead() – read rapidIO registers
SYNOPSIS	<code>void m85xxRioRegDbgRead (VXB_DEVICE_ID pInst >)</code>
DESCRIPTION	none
RETURNS	NONE
ERRNO	Not Available
SEE ALSO	m85xxRio

m85xxRioRegister()

NAME	m85xxRioRegister() – register PowerPC 85xx rapidIO with bus subsystem
-------------	---

SYNOPSIS	void m85xxRioRegister(void)
DESCRIPTION	This passes the registration structure for this device to the vxBus registration function. Only when this is called can the driver be used.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	m85xxRio

m85xxRioStatusShow()

NAME	<i>m85xxRioStatusShow()</i> – show state of RapidIO interface
SYNOPSIS	int m85xxRioStatusShow (VXB_DEVICE_ID pInst)
DESCRIPTION	none
RETURNS	NONE
ERRNO	Not Available
SEE ALSO	m85xxRio

m85xxTimerDrvRegister()

NAME	<i>m85xxTimerDrvRegister()</i> – register m85xx timer driver
SYNOPSIS	void m85xxTimerDrvRegister(void)
DESCRIPTION	This routine registers the m85xx timer driver with the vxBus subsystem.
RETURNS	N/A

ERRNO

SEE ALSO [vxM85xxTimer](#)

2

mcf5475PciRegister()

NAME [**mcf5475PciRegister\(\)**](#) – register Coldfire mcf5475 host controller

SYNOPSIS `void mcf5475PciRegister (void)`

DESCRIPTION This routine registers the Coldfire md5475 host controller with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO [**mcf5475Pci**](#)

mib2ErrorAdd()

NAME [**mib2ErrorAdd\(\)**](#) – change a MIB-II error count

SYNOPSIS `STATUS mib2ErrorAdd
(
 M2_INTERFACE * pMib,
 int errCode,
 int value
)`

DESCRIPTION This function adds a specified value to one of the MIB-II error counters in a MIB-II interface table. The counter to be altered is specified by the `errCode` argument. `errCode` can be `MIB2_IN_ERRS`, `MIB2_IN_UCAST`, `MIB2_OUT_ERRS` or `MIB2_OUT_UCAST`. Specifying a negative value reduces the error count, a positive value increases the error count.

RETURNS OK

ERRNO

SEE ALSO [**endLib**](#)

mib2Init()

NAME	mib2Init() – initialize a MIB-II structure
SYNOPSIS	<pre>STATUS mib2Init (M2_INTERFACE_TBL *pMib, /* struct to be initialized */ long ifType, /* ifType from m2Lib.h */ UCHAR * phyAddr, /* MAC/PHY address */ int addrLength, /* MAC/PHY address length */ int mtuSize, /* MTU size */ int speed /* interface speed */)</pre>
DESCRIPTION	Initialize a MIB-II structure. Set all error counts to zero. Assume a 10Mbps Ethernet device.
RETURNS	OK or ERROR.
ERRNO	
SEE ALSO	endLib

miiAnCheck()

NAME	miiAnCheck() – check the auto-negotiation process result
SYNOPSIS	<pre>STATUS miiAnCheck (PHY_INFO * pPhyInfo, /* pointer to PHY_INFO structure */ UINT8 phyAddr /* address of a PHY */)</pre>
DESCRIPTION	This routine checks the auto-negotiation process has completed successfully and no faults have been detected by any of the PHYs engaged in the process.
NOTE	In case the cable is pulled out and reconnect to the same/different hub/switch again. PHY probably starts a new auto-negotiation process and get different negotiation results. User should call this routine to check link status and update phyFlags. pPhyInfo should include a valid PHY bus number (phyAddr), and include the phyFlags that was used last time to configure auto-negotiation process.
RETURNS	OK or ERROR.

ERRNO Not Available

SEE ALSO **miiLib**

miiLibInit()

NAME **miiLibInit()** – initialize the MII library

SYNOPSIS STATUS miiLibInit (void)

DESCRIPTION This routine initializes the MII library.

PROTECTION DOMAINS

(VxAE) This function can only be called from within the kernel protection domain.

RETURNS OK or ERROR.

ERRNO Not Available

SEE ALSO **miiLib**

miiLibUnInit()

NAME **miiLibUnInit()** – uninitialized the MII library

SYNOPSIS STATUS miiLibUnInit
(
)

DESCRIPTION This routine uninitializeds the MII library. Previously allocated resources are reclaimed back to the system.

RETURNS OK or ERROR.

ERRNO Not Available

SEE ALSO **miiLib**

miiPhyInit()

NAME miiPhyInit() – initialize and configure the PHY devices

SYNOPSIS

```
STATUS miiPhyInit
(
    PHY_INFO * pPhyInfo /* pointer to PHY_INFO structure */
)
```

DESCRIPTION This routine scans, initializes and configures the PHY device described in *phyInfo*. Space for *phyInfo* is to be provided by the calling task.

This routine is called from the driver's Start routine to perform media initialization and configuration. To access the PHY device through the MII-management interface, it uses the read and write routines which are provided by the driver itself in the fields **phyReadRtn()**, **phyWriteRtn()** of the *phyInfo* structure. Before it attempts to use this routine, the driver has to properly initialize some of the fields in the *phyInfo* structure, and optionally fill in others, as below:

```
/* fill in mandatory fields in phyInfo */

pDrvCtrl->phyInfo->pDrvCtrl = (void *) pDrvCtrl;
pDrvCtrl->phyInfo->phyWriteRtn = (FUNCPTR) xxxMiiWrite;
pDrvCtrl->phyInfo->phyReadRtn = (FUNCPTR) xxxMiiRead;

/* fill in some optional fields in phyInfo */

pDrvCtrl->phyInfo->phyFlags = 0;
pDrvCtrl->phyInfo->phyAddr = (UINT8) MII_PHY_DEF_ADDR;
pDrvCtrl->phyInfo->phyDefMode = (UINT8) PHY_10BASE_T;
pDrvCtrl->phyInfo->phyAnOrderTbl = (MII_AN_ORDER_TBL *)
    &xxxPhyAnOrderTbl;

/*
 * fill in some more optional fields in phyInfo: the delay stuff
 * we want this routine to use our xxxDelay () routine, with
 * the constant one as an argument, and the max delay we may
 * tolerate is the constant MII_PHY_DEF_DELAY, in xxxDelay units
 */

pDrvCtrl->phyInfo->phyDelayRtn = (FUNCPTR) xxxDelay;
pDrvCtrl->phyInfo->phyMaxDelay = MII_PHY_DEF_DELAY;
pDrvCtrl->phyInfo->phyDelayParm = 1;

/*
 * fill in some more optional fields in phyInfo: the PHY's callback
 * to handle "link down" events. This routine is invoked whenever
 * the link status in the PHY being used is detected to be low.
 */

pDrvCtrl->phyInfo->phyStatChngRtn = (FUNCPTR) xxxRestart;
```

Some of the above fields may be overwritten by this routine, since for instance, the logical address of the PHY actually used may differ from the user's initial setting. Likewise, the specific PHY being initialized, may not support all the technology abilities the user has allowed for its operations.

This routine first scans for all possible PHY addresses in the range 0-31, checking for an MII-compliant PHY, and attempts at running some diagnostics on it. If none is found, **ERROR** is returned.

Typically PHYs are scanned from address 0, but if the user specifies an alternative start PHY address via the parameter *phyAddr* in the *phyInfo* structure, PHYs are scanned in order starting with the specified PHY address. In addition, if the flag *MII_ALL_BUS_SCAN* is set, this routine will scan the whole bus even if a valid PHY has already been found, and stores bus topology information. If the flags *MII_PHY_ISO*, *MII_PHY_PWR_DOWN* are set, all of the PHYs found but the first will be respectively electrically isolated from the MII interface and/or put in low-power mode. These two flags are meaningless in a configuration where only one PHY is present.

The *phyAddr* parameter is very important from a performance point of view. Since the MII management interface, through which the PHY is configured, is a very slow one, providing an incorrect or invalid address in this field may result in a particularly long boot process.

If the flag *MII_ALL_BUS_SCAN* is not set, this routine will assume that the first PHY found is the only one.

This routine then attempts to bring the link up. This routine offers two strategies to select a PHY and establish a valid link. The default strategy is to use the standard 802.3 style auto-negotiation, where both link partners negotiate all their technology abilities at the same time, and the highest common denominator ability is chosen. Before the auto-negotiation is started, the next-page exchange mechanism is disabled.

If GMII interface is used, users can specify it through *userFlags* -- *MII_PHY_GMII_TYPE*.

The user can prevent the PHY from negotiating certain abilities via *userFlags* -- *MII_PHY_FD*, *MII_PHY_100*, *MII_PHY_HD*, and *MII_PHY_10*. as well as *MII_PHY_1000T_HD* and *MII_PHY_1000T_FD* if GMII is used. When *MII_PHY_FD* is not specified, full duplex will not be negotiated; when *MII_PHY_HD* is not specified half duplex will not be negotiated, when *MII_PHY_100* is not specified, 100Mbps ability will not be negotiated; when *MII_PHY_10* is not specified, 10Mbps ability will not be negotiated. Also, if GMII is used, when *MII_PHY_1000T_HD* is not specified, 1000T with half duplex mode will not be negotiated. Same thing applied to 1000T with full duplex mode via *MII_PHY_1000T_FD*.

Flow control ability can also be negotiated via user flags -- *MII_PHY_TX_FLOW_CTRL* and *MII_PHY_RX_FLOW_CTRL*. For symmetric PAUSE ability (MII), user can set/clean both flags together. For asymmetric PAUSE ability (GMII), user can separate transmit and receive flow control ability. However, user should be aware that flow control ability is meaningful only if full duplex mode is used.

When *MII_PHY_TBL* is set in the user flags, the BSP specific table whose address may be provided in the *phyAnOrderTbl* field of the *phyInfo* structure, is used to obtain the list, and the order of technology abilities to be negotiated. The entries in this table are ordered such that entry 0 is the highest priority, entry 1 in next and so on. Entries in this table may be repeated, and multiple technology abilities can be OR'd to create a single entry. If a PHY cannot support a ability in an entry, that entry is ignored.

If no PHY provides a valid link, and if *MII_PHY_DEF_SET* is set in the *phyFlags* field of the *PHY_INFO* structure, the first PHY that supports the default abilities defined in the *phyDefMode* of the *phyInfo* structure will be selected, regardless of the link status.

In addition, this routine adds an entry in a linked list of PHY devices for each active PHY it found. If the flag *MII_PHY_MONITOR* is set, the link status for the relevant PHY is continually monitored for a link down event. If such event is detected, and if the *phyLinkDownRtn* in the *PHY_INFO* * structure is a valid function pointer, then the routine it points at is executed in the context of the netTask(). The parameter *MII_MONITOR_DELAY* may be used to define the period in seconds with which the link status is checked. Its default value is 5.

RETURNS OK or ERROR if the PHY could not be initialized,

ERRNO Not Available

SEE ALSO **miiLib**

miiPhyOptFuncMultiSet()

NAME **miiPhyOptFuncMultiSet()** – set pointers to MII optional registers handlers

SYNOPSIS

```
void miiPhyOptFuncMultiSet
{
    PHY_INFO * pPhyInfo, /* device specific pPhyInfo pointer */
    FUNCPTR    optRegsFunc /* function pointer */
}
```

DESCRIPTION This routine sets the function pointers in *pPhyInfo-optRegsFunc* to the MII optional, PHY-specific registers handler. The handler will be executed before the PHY's technology abilities are negotiated. If a system employees more than one type of network device requiring a PHY-specific registers handler use this routine instead of **miiPhyOptFuncSet()** to ensure device specific handlers and to avoid overwriting one's with the other's.

PROTECTION DOMAINS

(VxAE) This function can only be called from within the kernel protection domain. The argument *optRegsFunc* MUST be a pointer to function in the kernel protection domain.

RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	miiLib

miiPhyOptFuncSet()

NAME	miiPhyOptFuncSet() – set the pointer to the MII optional registers handler
SYNOPSIS	<pre>void miiPhyOptFuncSet (FUNCPTR optRegsFunc /* function pointer */)</pre>
DESCRIPTION	This routine sets the function pointer in <i>optRegsFunc</i> to the MII optional, PHY-specific registers handler. The handler will be executed before the PHY's technology abilities are negotiated.
PROTECTION DOMAINS	

(VxAE) This function can only be called from within the kernel protection domain. The argument *optRegsFunc* MUST be a pointer to function in the kernel protection domain.

RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	miiLib

miiPhyUnInit()

NAME	miiPhyUnInit() – uninitialized a PHY
SYNOPSIS	<pre>STATUS miiPhyUnInit (PHY_INFO * pPhyInfo /* pointer to PHY_INFO structure */)</pre>

DESCRIPTION	This routine uninitializeds the PHY specified in <i>pPhyInfo</i> . It brings it in low-power mode, and electrically isolate it from the MII management interface to which it is attached. In addition, it frees resources previously allocated.
RETURNS	OK, ERROR in case of fatal errors.
ERRNO	Not Available
SEE ALSO	miiLib

miiRegsGet()

NAME	miiRegsGet() – get the contents of MII registers
SYNOPSIS	<pre>STATUS miiRegsGet (PHY_INFO * pPhyInfo, /* pointer to PHY_INFO structure */ UINT regNum, /* number of registers to display */ UCHAR * buff /* where to read registers to */)</pre>
DESCRIPTION	This routine gets the contents of the first <i>regNum</i> MII registers, and, if <i>buff</i> is not NULL, copies them to the space pointed to by <i>buff</i> .
RETURNS	OK, or ERROR if could not perform the read.
ERRNO	Not Available
SEE ALSO	miiLib

miiShow()

NAME	miiShow() – show routine for MII library
SYNOPSIS	<pre>void miiShow (PHY_INFO * pPhyInfo /* pointer to PHY_INFO structure */)</pre>
DESCRIPTION	This is a show routine for the MII library

RETURNS OK, always.

ERRNO Not Available

SEE ALSO miiLib

mipsCavIntCtlrShow()

NAME mipsCavIntCtlrShow() – show pDrvCtrl for mipsIntCtlr

SYNOPSIS

```
void mipsCavIntCtlrShow
(
    VXB_DEVICE_ID pInst,
    int            verboseLevel
)
```

DESCRIPTION Based on verboseLevel value, display interrupt controller connection and methods information

RETURNS N/A

ERRNO

SEE ALSO vxbMipsCavIntCtlr

mipsIntCtlrShow()

NAME mipsIntCtlrShow() – show pDrvCtrl for mipsIntCtlr

SYNOPSIS

```
void mipsIntCtlrShow
(
    VXB_DEVICE_ID pInst,
    int            verboseLevel
)
```

DESCRIPTION Based on verboseLevel value, display interrupt controller connection and methods information

RETURNS N/A

ERRNO

SEE ALSO [vxmMipsIntCtlr](#)

motFccDrvShow()

NAME	motFccDrvShow() – debug function to show FCC parameter ram addresses, initial BD and cluster settings
SYNOPSIS	<pre>void motFccDrvShow (DRV_CTRL * pDrvCtrl)</pre>
DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFccDumpRxRing()

NAME	motFccDumpRxRing() – Show the Receive Ring details
SYNOPSIS	<pre>void motFccDumpRxRing (int fccNum)</pre>
DESCRIPTION	This routine displays the receive ring descriptors.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End , motFccDumpTxRing()

motFccDumpTxRing()

NAME	<i>motFccDumpTxRing()</i> – Show the Transmit Ring details
SYNOPSIS	void motFccDumpTxRing (int fccNum)
DESCRIPTION	This routine displays the transmit ring descriptors.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	<i>motFcc2End, motFccDumpRxRing()</i>

motFccEndLoad()

NAME	<i>motFccEndLoad()</i> – initialize the driver and device
SYNOPSIS	END_OBJ* motFcc2EndLoad (char *initString)
DESCRIPTION	This routine initializes both driver and device to an operational state using device specific parameters specified by <i>initString</i> . The parameter string, <i>initString</i> , is an ordered list of parameters each separated by a colon. The format of <i>initString</i> is: <i>"unitImmrVal:fccNum:bdBase:bdSize:bufBase:bufSize:fifoTxBase: fifoRxBase:tbdNum:rbdNum:phyAddr:phyDefMode:phyAnOrderTbl: userFlags:function table(:maxRxFrames)"</i>
	The FCC shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.
	A default number of transmit/receive buffer descriptors of 32 can be selected by passing zero in the parameters <i>tbdNum</i> and <i>rbdNum</i> . In other cases, the number of buffers selected should be greater than two.

The *bufBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine attempts to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The *bufSize* parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive buffer descriptors.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine uses **cacheDmaMalloc()** to obtain some cache-safe memory. The attributes of this memory is checked, and if the memory is not write coherent, this routine aborts and returns NULL.

RETURNS an END object pointer, or NULL on error.

ERRNO Not Available

SEE ALSO **motFcc2End**, **ifLib**, *MPC8260 PowerQUICC II User's Manual*

motFccEramShow()

NAME **motFccEramShow()** – Debug Function to show FCC CP ethernet parameter ram.

SYNOPSIS

```
void motFccEramShow
(
    DRV_CTRL * pDrvCtrl
)
```

DESCRIPTION This function is only available when **MOT_FCC_DBG** is defined. It should be used for debugging purposes only.

RETURNS N/A

ERRNO Not Available

SEE ALSO **motFcc2End**

motFccIramShow()

NAME	motFccIramShow() – Debug Function to show FCC CP internal ram parameters.
SYNOPSIS	<pre>void motFccIramShow (DRV_CTRL * pDrvCtrl)</pre>
DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFccMibShow()

NAME	motFccMibShow() – Debug Function to show MIB statistics.
SYNOPSIS	<pre>void motFccMibShow (DRV_CTRL * pDrvCtrl)</pre>
DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFccMiiShow()

NAME	motFccMiiShow() – debug function to show the Mii settings in the Phy Info structure
-------------	--

SYNOPSIS	<pre>void motFccMiiShow (DRV_CTRL * pDrvCtrl)</pre>
DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFccPramShow()

NAME	motFccPramShow() – Debug Function to show FCC CP parameter ram.
SYNOPSIS	<pre>void motFccPramShow (DRV_CTRL * pDrvCtrl)</pre>
DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFccShow()

NAME	motFccShow() – Debug Function to show driver specific control data.
SYNOPSIS	<pre>void motFccShow (DRV_CTRL * pDrvCtrl)</pre>

DESCRIPTION	This function is only available when MOT_FCC_DBG is defined. It should be used for debugging purposes only.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	motFcc2End

motFecEndLoad()

NAME ***motFecEndLoad()*** – initialize the driver and device

SYNOPSIS `END_OBJ* motFecEndLoad
(
 char *initString /* parameter string */
)`

DESCRIPTION This routine initializes both driver and device to an operational state using device specific parameters specified by *initString*.

The parameter string, *initString*, is an ordered list of parameters each separated by a colon. The format of *initString* is:

```
"motCpmAddr:ivec:bufBase:bufSize:fifoTxBase:fifoRxBase  
:tbdNum:rbdNum:phyAddr:isoPhyAddr:phyDefMode:userFlags :clockSpeed"
```

The FEC shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.

A default number of transmit/receive buffer descriptors of 32 can be selected by passing zero in the parameters *tbdNum* and *rbdNum*. In other cases, the number of buffers selected should be greater than two.

The *bufBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The *bufSize* parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive buffer descriptors.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use **cacheDmaMalloc()** to obtain some cache-safe memory. The attributes of this memory will be checked, and if the memory is not write coherent, this routine will abort and return NULL.

RETURNS an END object pointer, or NULL on error.

ERRNO Not Available

SEE ALSO **motFecEnd**, **ifLib**, *MPC860T Fast Ethernet Controller (Supplement to MPC860 User's Manual)*

ncr810CtrlCreate()

NAME **ncr810CtrlCreate()** – create a control structure for the NCR 53C8xx SIOP

SYNOPSIS

```
NCR_810_SCSI_CTRL *ncr810CtrlCreate
(
    UINT8  *baseAdrs, /* base address of the SIOP */
    UINT    clkPeriod, /* clock controller period (nsec*100) */
    UINT16 devType    /* NCR8XX SCSI device type */
)
```

DESCRIPTION This routine creates an SIOP data structure and must be called before using an SIOP chip. It must be called exactly once for a specified SIOP controller. Since it allocates memory for a structure needed by all routines in **ncr810Lib**, it must be called before any other routines in the library. After calling this routine, **ncr810CtrlInit()** must be called at least once before any SCSI transactions are initiated using the SIOP.

A detailed description of the input parameters follows:

baseAdrs

the address at which the CPU accesses the lowest (SCNTL0/SIEN) register of the SIOP.

clkPeriod

the period of the SIOP SCSI clock input, in nanoseconds, multiplied by 100. This is used to determine the clock period for the SCSI core of the chip and affects the timing of both asynchronous and synchronous transfers. Several commonly-used values are defined in **ncr810.h** as follows:

```
NCR810_1667MHZ  6000  /* 16.67Mhz chip */
NCR810_20MHZ    5000  /* 20Mhz chip */
NCR810_25MHZ    4000  /* 25Mhz chip */
NCR810_3750MHZ   2667  /* 37.50Mhz chip */
NCR810_40MHZ    2500  /* 40Mhz chip */
NCR810_50MHZ    2000  /* 50Mhz chip */
NCR810_66MHZ    1515  /* 66Mhz chip */
```

```
NCR810_6666MHZ 1500 /* 66.66Mhz chip */  
devType  
the specific NCR 8xx device type. Current device types are defined in the header file  
ncr810.h.
```

RETURNS	A pointer to the NCR_810_SCSI_CTRL structure, or NULL if memory is unavailable or there are invalid parameters.
ERRNO	Not Available
SEE ALSO	ncr810Lib

ncr810CtrlInit()

NAME	ncr810CtrlInit() – initialize a control structure for the NCR 53C8xx SIOP
SYNOPSIS	<pre>STATUS ncr810CtrlInit (FAST NCR_810_SCSI_CTRL *pSiop, /* ptr to SIOP struct */ int scsiCtrlBusId /* SCSI bus ID of this SIOP */)</pre>
DESCRIPTION	This routine initializes an SIOP structure, after the structure is created with ncr810CtrlCreate() . This structure must be initialized before the SIOP can be used. It may be called more than once if needed; however, it must only be called while there is no activity on the SCSI interface.
	A detailed description of the input parameters follows:
<i>pSiop</i>	a pointer to the NCR_810_SCSI_CTRL structure created with ncr810CtrlCreate() .
<i>scsiCtrlBusId</i>	the SCSI bus ID of the SIOP. Its value is somewhat arbitrary: seven (7), or highest priority, is conventional. The value must be in the range 0 - 7.
RETURNS	OK, or ERROR if parameters are out of range.
ERRNO	Not Available
SEE ALSO	ncr810Lib

ncr810SetHwRegister()

NAME **ncr810SetHwRegister()** – set hardware-dependent registers for the NCR 53C8xx SIOP

SYNOPSIS

```
STATUS ncr810SetHwRegister
(
    FAST_SIOP      *pSiop,    /* pointer to SIOP info */
    NCR810_HW_REGS *pHwRegs /* pointer to a NCR810_HW_REGS info */
)
```

DESCRIPTION This routine sets up the registers used in the hardware implementation of the chip. Typically, this routine is called by the **sysScsiInit()** routine from the BSP.

The input parameters are as follows:

pSiop

a pointer to the **NCR_810_SCSI_CTRL** structure created with **ncr810CtrlCreate()**.

pHwRegs

a pointer to a **NCR810_HW_REGS** structure that is filled with the logical values 0 or 1 for each bit of each register described below.

This routine includes only the bit registers that can be used to modify the behavior of the chip. The default configuration used during **ncr810CtrlCreate()** and **ncr810CtrlInit()** is {0,0,0,0,0,1,0,0,0,0}.

```
typedef struct
{
    int stest1Bit7;           /* Disable external SCSI clock */
    int stest2Bit7;           /* SCSI control enable */
    int stest2Bit5;           /* Enable differential SCSI bus */
    int stest2Bit2;           /* Always WIDE SCSI */
    int stest2Bit1;           /* Extend SREQ/SACK filtering */
    int stest3Bit7;           /* TolerANT enable */
    int dmodeBit7;            /* Burst Length transfer bit 1 */
    int dmodeBit6;            /* Burst Length transfer bit 0 */
    int dmodeBit5;            /* Source I/O memory enable */
    int dmodeBit4;            /* Destination I/O memory enable */
    int scnt11Bit7;           /* Slow cable mode */
} NCR810_HW_REGS;
```

For a more detailed explanation of the register bits, see the appropriate NCR 53C8xx data manuals.

NOTE Because this routine writes to the NCR 53C8xx chip registers, it cannot be used when there is any SCSI bus activity.

RETURNS OK, or ERROR if any input parameter is NULL

ERRNO Not Available

SEE ALSO**ncr810Lib, ncr810.h, ncr810CtrlCreate()**

ncr810Show()**NAME****ncr810Show()** – display values of all readable NCR 53C8xx SIOP registers**SYNOPSIS**

```
STATUS ncr810Show
(
    FAST SCSI_CTRL *pScsiCtrl /* ptr to SCSI controller info */
)
```

DESCRIPTION

This routine displays the state of the SIOP registers in a user-friendly way. It is useful primarily for debugging. The input parameter is the pointer to the SIOP information structure returned by the **ncr810CtrlCreate()** call.

NOTE

The only readable register during a script execution is the Istat register. If you use this routine during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE

```
-> ncr810Show
NCR810 Registers
-----
0xffff47000: Sien      = 0xa5 Sdid      = 0x00 Scntl11  = 0x00 Scntl10  = 0x04
0xffff47004: Socl      = 0x00 Sodl      = 0x00 Sxfer     = 0x80 Scid      = 0x80
0xffff47008: Sbcl      = 0x00 Sbdl      = 0x00 Sidl      = 0x00 Sfbr      = 0x00
0xffff4700c: Sstat2    = 0x00 Sstat1    = 0x00 Sstat0    = 0x00 Dstat     = 0x80
0xffff47010: Dsa       = 0x00000000
0xffff47014: Ctest3    = ???? Ctest2    = 0x21 Ctest1    = 0xf0 Ctest0    = 0x00
0xffff47018: Ctest7    = 0x32 Ctest6    = ???? Ctest5    = 0x00 Ctest4    = 0x00
0xffff4701c: Temp      = 0x00000000
0xffff47020: Lcrc      = 0x00 Ctest8    = 0x00 Istat     = 0x00 Dfifo     = 0x00
0xffff47024: Dcmd/Ddc= 0x50000000
0xffff47028: Dnad      = 0x00066144
0xffff4702c: Dsp       = 0x00066144
0xffff47030: Dpsp      = 0x00066174
0xffff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
0xffff47038: Dcntl     = 0x21 Dwt       = 0x00 Dien      = 0x37 Dmode     = 0x01
0xffff4703c: Adder     = 0x000cc2b8
value = 0 = 0x0
```

RETURNSOK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.**ERRNO**

Not Available

SEE ALSO**ncr810Lib, ncr810CtrlCreate()**

ne2000EndLoad()

NAME	<i>ne2000EndLoad()</i> – initialize the driver and device
SYNOPSIS	<pre>END_OBJ* ne2000EndLoad (char* initString, /* String to be parsed by the driver. */ void* pBSP /* for BSP group */)</pre>
DESCRIPTION	This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString. The string contains the target specific parameters like this: "unit:register addr:int vector:int level:shmem addr:shmem size:shmem width"
RETURNS	An END object pointer or NULL on error.
ERRNO	Not Available
SEE ALSO	<i>ne2000End</i>

noDev()

NAME	<i>noDev()</i> – optional driver functionality not present
SYNOPSIS	BOOL noDev(void)
DESCRIPTION	This routine indicates that there is no optional device driver functionality present.
RETURNS	FALSE, always
ERRNO	
SEE ALSO	<i>vxBus</i>

ns16550DevInit()

NAME	<i>ns16550DevInit()</i> – initialize an NS16550 channel
-------------	---

SYNOPSIS	<pre>void ns16550DevInit (NS16550_CHAN * pChan /* pointer to channel */)</pre>
DESCRIPTION	This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialized all the device addresses, etc. in the NS16550_CHAN structure.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ns16550Sio

ns16550Int()

NAME	ns16550Int() – interrupt level processing
SYNOPSIS	<pre>void ns16550Int (NS16550_CHAN * pChan /* pointer to channel */)</pre>
DESCRIPTION	This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status. When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ns16550Sio

ns16550IntEx()

NAME	ns16550IntEx() – miscellaneous interrupt processing
-------------	---

SYNOPSIS	<pre>void ns16550IntEx (NS16550_CHAN *pChan /* pointer to channel */)</pre>
DESCRIPTION	This routine handles miscellaneous interrupts on the UART. Not implemented yet.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ns16550Sio

ns16550IntRd()

NAME	ns16550IntRd() – handle a receiver interrupt
SYNOPSIS	<pre>void ns16550IntRd (NS16550_CHAN * pChan /* pointer to channel */)</pre>
DESCRIPTION	This routine handles read interrupts from the UART.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ns16550Sio

ns16550IntWr()

NAME	ns16550IntWr() – handle a transmitter interrupt
SYNOPSIS	<pre>void ns16550IntWr (NS16550_CHAN * pChan /* pointer to channel */)</pre>
DESCRIPTION	This routine handles write interrupts from the UART. It reads a character and puts it in the transmit holding register of the device for transfer.

If there are no more characters to transmit, transmission is disabled by clearing the transmit interrupt enable bit in the IER(int enable register).

RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ns16550Sio

ns16550SioRegister()

NAME	ns16550SioRegister() – register ns16550vxb driver
SYNOPSIS	<code>void ns16550SioRegister(void)</code>
DESCRIPTION	This routine registers the ns16550vxb driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbNs16550Sio

ns83902EndLoad()

NAME	ns83902EndLoad() – initialize the driver and device
SYNOPSIS	<code>END_OBJ* ns83902EndLoad (char* initString /* string to be parsed */)</code>
DESCRIPTION	This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in <i>initString</i> . This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "In") into the <i>initString</i> and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS	An END object pointer, or NULL on error, or 0 and the name of the device if the <i>initString</i> was NULL.
ERRNO	Not Available
SEE ALSO	ns83902End

ns83902RegShow()

NAME	ns83902RegShow() – prints the current value of the NIC registers
SYNOPSIS	void ns83902RegShow (NS83902_END_DEVICE* pDrvCtrl)
DESCRIPTION	This routine reads and displays the register values of the NIC registers
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	ns83902End

nullDrv()

NAME	nullDrv() – optional driver functionality not present
SYNOPSIS	STATUS nullDrv(void)
DESCRIPTION	This routine indicates that there is no optional device driver functionality present.
RETURNS	OK, always

ERRNO

SEE ALSO [vxBus](#)

octeonSioRegister()

NAME	octeonSioRegister() – register octeonVxb driver
SYNOPSIS	<code>void octeonSioRegister(void)</code>
DESCRIPTION	This routine registers the octeonVxb driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbOcteonSio

octeonVxbDevInit()

NAME	octeonVxbDevInit() – initialize an OCTEON channel
SYNOPSIS	<code>void octeonVxbDevInit (OCTEONVXB_CHAN * pChan /* pointer to channel */)</code>
DESCRIPTION	This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialized all the device addresses, etc. in the OCTEONVXB_CHAN structure.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbOcteonSio

octeonVxbDevProbe()

NAME	octeonVxbDevProbe() – probe for device presence at specific address
SYNOPSIS	<pre>BOOL octeonVxbDevProbe (struct vxbDev * pDev, /* Device information */ int regBaseIndex /* Index of base address */)</pre>
DESCRIPTION	<p>Check for octeonVxb (or compatible) device at the specified base address. We assume one is present at that address, but we need to verify.</p> <p>Probe the device by the following:</p> <ul style="list-style-type: none">- set the device to loopback mode - mask interrupts - attempt to generate an interrupt - check interrupt status <p>If the interrupt status register shows that an interrupt did occur, then we presume it's our device.</p>
NOTE	This routine is called early during system initialization, and *MUST* *NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	TRUE if probe passes and assumed a valid octeonVxb (or compatible) device. FALSE otherwise.
ERRNO	Not Available
SEE ALSO	vxbOcteonSio

octeonVxbInt()

NAME	octeonVxbInt() – interrupt level processing
SYNOPSIS	<pre>void octeonVxbInt (VXB_DEVICE_ID pDev)</pre>
DESCRIPTION	This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status.

When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

ERRNO

SEE ALSO vxbOcteonSio

octeonVxbIntEx()

NAME **octeonVxbIntEx()** – miscellaneous interrupt processing

SYNOPSIS

```
void octeonVxbIntEx
(
    OCTEONVXB_CHAN *pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles miscellaneous interrupts on the UART. Not implemented yet.

RETURNS N/A

ERRNO

SEE ALSO vxbOcteonSio

octeonVxbIntRd()

NAME **octeonVxbIntRd()** – handle a receiver interrupt

SYNOPSIS

```
void octeonVxbIntRd
(
    OCTEONVXB_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles read interrupts from the UART.

RETURNS N/A

ERRNO

SEE ALSO [vxbOcteonSio](#)

octeonVxbIntWr()

NAME **octeonVxbIntWr()** – handle a transmitter interrupt

SYNOPSIS

```
void octeonVxbIntWr
(
    OCTEONVXB_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles write interrupts from the UART.

RETURNS N/A

ERRNO

SEE ALSO [vxbOcteonSio](#)

openPicTimerDrvRegister()

NAME **openPicTimerDrvRegister()** – register openPic timer driver

SYNOPSIS

```
void openPicTimerDrvRegister(void)
```

DESCRIPTION This routine registers the openPic timer driver with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO [vxbOpenPicTimer](#)

optimizeAccessFunction()

NAME	optimizeAccessFunction() – optimize a function based on flags
SYNOPSIS	<pre>void optimizeAccessFunction (UINT32 flags, /* flags */ struct vxbAccessList * pAccess, /* bus access structure ptr */ UINT32 operationId /* operation indicator */)</pre>
DESCRIPTION	This routine is used to determine an optimized function based on the flags for the operation and update the access function pointer with the pointer to the optimized function
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPlbAccess

pccardAtaEnabler()

NAME	pccardAtaEnabler() – enable the PCMCIA-ATA device
SYNOPSIS	<pre>STATUS pccardAtaEnabler (int sock, /* socket no. */ ATA_RESOURCE *pAtaResource, /* pointer to ATA resources */ int numEnt, /* number of ATA resource entries */ FUNCPTR showRtn /* ATA show routine */)</pre>
DESCRIPTION	This routine enables the PCMCIA-ATA device.
RETURNS	OK , ERROR_FIND if there is no ATA card, or ERROR if another error occurs.
ERRNO	Not Available
SEE ALSO	pccardLib

pccardEltEnabler()

NAME **pccardEltEnabler()** – enable the PCMCIA Etherlink III card

SYNOPSIS

```
STATUS pccardEltEnabler
(
    int          sock,           /* socket no. */
    ELT_RESOURCE *pEltResource, /* pointer to ELT resources */
    int          numEnt,        /* number of ELT resource entries */
    FUNCPTR      showRtn       /* show routine */
)
```

DESCRIPTION This routine enables the PCMCIA Etherlink III (ELT) card.

RETURNS OK, ERROR_FIND if there is no ELT card, or ERROR if another error occurs.

ERRNO Not Available

SEE ALSO **pccardLib**

pccardMkfs()

NAME **pccardMkfs()** – initialize a device and mount a DOS file system

SYNOPSIS

```
STATUS pccardMkfs
(
    int  sock,   /* socket number */
    char *pName  /* name of a device */
)
```

DESCRIPTION This routine initializes a device and mounts a DOS file system.

RETURNS OK or ERROR.

ERRNO Not Available

SEE ALSO **pccardLib**

pccardMount()

NAME	pccardMount() – mount a DOS file system
SYNOPSIS	STATUS pccardMount (int sock, /* socket number */ char *pName /* name of a device */)
DESCRIPTION	This routine mounts a DOS file system.
RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	pccardLib

pccardSramEnabler()

NAME	pccardSramEnabler() – enable the PCMCIA-SRAM driver
SYNOPSIS	STATUS pccardSramEnabler (int sock, /* socket no. */ SRAM_RESOURCE *pSramResource, /* pointer to SRAM resources */ int numEnt, /* number of SRAM resource entries */ FUNCPTR showRtn /* SRAM show routine */)
DESCRIPTION	This routine enables the PCMCIA-SRAM driver.
RETURNS	OK, ERROR_FIND if there is no SRAM card, or ERROR if another error occurs.
ERRNO	Not Available
SEE ALSO	pccardLib

pccardTffsEnabler()

NAME	pccardTffsEnabler() – enable the PCMCIA-TFFS driver
SYNOPSIS	<pre>STATUS pccardTffsEnabler (int sock, /* socket no. */ TFFS_RESOURCE *pTffsResource, /* pointer to TFFS resources */ int numEnt, /* number of SRAM resource entries */ FUNCPTR showRtn /* TFFS show routine */)</pre>
DESCRIPTION	This routine enables the PCMCIA-TFFS driver.
RETURNS	OK, ERROR_FIND if there is no TFFS(Flash) card, or ERROR if another error occurs.
ERRNO	Not Available
SEE ALSO	pccardLib

pciAllHeaderShow()

NAME	pciAllHeaderShow() – show PCI header for all devices
SYNOPSIS	<pre>void pciAllHeaderShow (void)</pre>
DESCRIPTION	This routine displays the PCI header information of all PCI devices in the system.
RETURNS	N/A
ERRNO	
SEE ALSO	pentiumPci

pciAutoAddrAlign()

NAME	pciAutoAddrAlign() – align a PCI address and check boundary conditions
SYNOPSIS	<pre>STATUS pciAutoAddrAlign</pre>

```
(  
    UINT32 base,           /* base of available memory */  
    UINT32 limit,          /* last addr of available memory */  
    UINT32 reqSize,         /* required size */  
    UINT32 *pAlignedBase   /* output: aligned address put here */  
)
```

DESCRIPTION This routine handles address alignment/checking.

RETURNS OK, or ERROR if available memory has been exceeded.

ERRNO

SEE ALSO [pciAutoConfigLib](#)

pciAutoBusNumberSet()

NAME [pciAutoBusNumberSet\(\)](#) – set the primary, secondary, and subordinate bus number

SYNOPSIS STATUS pciAutoBusNumberSet
(
 PCI_LOC * pPciLoc, /* device affected */
 UINT primary, /* primary bus specification */
 UINT secondary, /* secondary bus specification */
 UINT subordinate /* subordinate bus specification */
)

DESCRIPTION This routine sets the primary, secondary, and subordinate bus numbers for a device that implements the Type 1 PCI Configuration Space Header.

This routine has external visibility to enable it to be used by BSP Developers for initialization of PCI Host Bridges that may implement registers similar to those found in the Type 1 Header.

RETURNS OK, always

ERRNO

SEE ALSO [pciAutoConfigLib](#)

pciAutoCardBusConfig()

NAME	pciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge
SYNOPSIS	<pre>LOCAL void pciAutoCardBusConfig (PCI_AUTO_CONFIG_OPTS * pSystem, /* PCI system info */ PCI_LOC * pPciLoc, /* PCI address of this bridge */ PCI_LOC ** ppPciList, /* Pointer to function list pointer */ UINT * nSize /* Number of remaining functions */)</pre>
DESCRIPTION	<p>This routine sets up memory and I/O base/limit registers for an individual PCI-Cardbus bridge.</p> <p>Cardbus bridges have four windows - 2 memory windows and 2 IO windows. The 2 memory windows can be setup individually for either prefetchable or non-prefetchable memory accesses.</p> <p>Since PC Cards can be inserted at any time, and are not necessarily present when this code is run, the code does not probe any further after encountering a Cardbus bridge. Instead, the code allocates default window sizes for the Cardbus bridge. Three windows are used:</p> <p>Warning: do not sort the include function list before this routine is called. This routine requires each function in the list to be in the same order as the probe occurred.</p>
RETURNS	N/A
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoCfg()

NAME	pciAutoCfg() – Automatically configure all nonexcluded PCI headers
SYNOPSIS	<pre>STATUS pciAutoCfg (void *pCookie /* cookie returned by pciAutoConfigLibInit() */)</pre>
DESCRIPTION	Top level function in the PCI configuration process.

CALLING SEQUENCE

```
pCookie = pciAutoConfigLibInit(NULL);  
pciAutoCfgCtl(pCookie, COMMAND, VALUE);  
...  
pciAutoCfgCtl(pCookie, COMMAND, VALUE);  
pciAutoCfg(pCookie);
```

For ease in converting from the old interface to the new one, a **pciAutoCfgCtl()** command **PCI_PSYSTEM_STRUCT_COPY** has been implemented. This can be used just like any other **pciAutoCfgCtl()** command, and it initializes all the values in pSystem. If used, it should be the first call to **pciAutoCfgCtl()**.

For a description of the COMMANDs and VALUEs to **pciAutoCfgCtl()**, see the **pciAutoCfgCtl()** documentation.

For all nonexcluded PCI functions on all PCI bridges, this routine automatically configures the PCI configuration headers for PCI devices and subbridges. The fields that are programmed are:

1. Status register.
2. Command Register.
3. Latency timer.
4. Cache Line size.
5. Memory and/or I/O base address and limit registers.
6. Primary, secondary, subordinate bus number (for PCI-PCI bridges).
7. Expansion ROM disable.
8. Interrupt Line.

ALGORITHM

Probe PCI config space and create a list of available PCI functions. Call device exclusion function, if registered, to exclude/include device. Disable all devices before we initialize any. Allocate and assign PCI space to each device. Calculate and set interrupt line value. Initialize and enable each device.

RETURNS

N/A

ERRNO**SEE ALSO**

pciAutoConfigLib

pciAutoCfgCtl()

NAME

pciAutoCfgCtl() – set or get **pciAutoConfigLib** options

SYNOPSIS

```
STATUS pciAutoCfgCtl
(
    void * pCookie, /* system configuration information */
    int cmd,        /* command word */
    void * pArg     /* argument for the cmd */
)
```

DESCRIPTION

pciAutoCfgCtl() can be considered analogous to **ioctl()** calls: the call takes arguments of (1) a pCookie, returned by **pciAutoConfigLibInit()**. (2) A command, macros for which are defined in **pciAutoConfigLib.h**. And, (3) an argument, the type of which depends on the specific command, but will always fit in a pointer variable. Currently, only globally effective commands are implemented.

The commands available are:

PCI_FBB_ENABLE - BOOL * pArg

PCI_FBB_DISABLE - void

PCI_FBB_UPDATE - BOOL * pArg

PCI_FBB_STATUS_GET - BOOL * pArg

Enable and disable the functions which check Fast Back To Back functionality.

PCI_FBB_UPDATE is for use with dynamic/HA applications. It first disables FBB on all functions, then enables FBB on all functions, if appropriate. In HA applications, it should be called any time a card is added or removed. The BOOL pointed to by pArg for **PCI_FBB_ENABLE** and **PCI_FBB_UPDATE** is set to **TRUE** if all cards allow FBB functionality and **FALSE** if either any card does not allow FBB functionality or if FBB is disabled. The BOOL pointed to by pArg for **PCI_FBB_STATUS_GET** is set to **TRUE** if **PCI_FBB_ENABLE** has been called and FBB is enabled, even if FBB is not activated on any card. It is set to **FALSE** otherwise.

NOTE: In the current implementation, FBB is enabled or disabled on the entire bus. If any device anywhere on the bus cannot support FBB, then it is not enabled, even if specific sub-busses could support it.

PCI_MAX_LATENCY_FUNC_SET - FUNCPTR * pArg

This routine is called for each function present on the bus when discovery takes place. The routine must accept four arguments, specifying bus, device, function, and a user-supplied argument of type **void ***. See **PCI_MAX_LATENCY_ARG_SET**. The routine should return a **UINT8** value, which will be put into the **MAX_LAT** field of the header structure. The user supplied routine must return a valid value each time it is called. There is no mechanism for any **ERROR** condition, but a default value can be returned in such a case. Default = **NULL**.

PCI_MAX_LATENCY_ARG_SET - void * pArg

When the routine specified in **PCI_MAX_LATENCY_FUNC_SET** is called, this will be passed to it as the fourth argument.

PCI_MAX_LAT_ALL_SET - int pArg

Specifies a constant max latency value for all cards, if no function has been specified with PCI_MAX_LATENCY_FUNC_SET..

PCI_MAX_LAT_ALL_GET - UINT * pArg

Retrieves the value of max latency for all cards, if no function has been specified with PCI_MAX_LATENCY_FUNC_SET. Otherwise, the integer pointed to by pArg is set to the value 0xffffffff.

PCI_MSG_LOG_SET - FUNCPTR * pArg

The argument specifies a routine which is called to print warning or error messages from **pciAutoConfigLib** if **logMsg()** has not been initialized at the time **pciAutoConfigLib** is used. The specified routine must accept arguments in the same format as **logMsg()**, but it does not necessarily need to print the actual message. An example of this routine is presented below, which saves the message into a safe memory space and turns on an LED. This command is useful for BSPs which call **pciAutoCfg()** before message logging is enabled. Note that after **logMsg()** is configured, output goes to **logMsg()** even if this command has been called. Default = NULL.

```
/* sample PCI_MSG_LOG_SET function */
int pciLogMsg(char *fmt, int a1, int a2, int a3, int a4, int a5, int a6)
{
    int charsPrinted;

    sysLedOn(4);
    charsPrinted = sprintf (sysExcMsg, fmt, a1, a2, a3, a4, a5, a6);
    sysExcMsg += charsPrinted;
    return (charsPrinted);
}
```

PCI_MAX_BUS_GET - int * pArg

During autoconfiguration, the library maintains a counter with the highest numbered bus. This can be retrieved by

```
pciAutoCfgCtl(pCookie, PCI_MAX_BUS_GET, &maxBus)
```

PCI_CACHE_SIZE_SET - int pArg

Sets the pci cache line size to the specified value. See CONFIGURATION SPACE PARAMETERS in the **pciAutoConfigLib** documentation for more details.

PCI_CACHE_SIZE_GET - int * pArg

Retrieves the value of the pci cache line size.

PCI_AUTO_INT_ROUTE_SET - BOOL pArg

Enables or disables automatic interrupt routing across bridges during the autoconfig process. See "INTERRUPT ROUTING ACROSS PCI-TO-PCI BRIDGES" in the **pciAutoConfigLib** documentation for more details.

PCI_AUTO_INT_ROUTE_GET - BOOL * pArg

Retrieves the status of automatic interrupt routing.

PCI_MEM32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit memory space. Normally, this is given by the BSP constant **PCI_MEM_ADRS**.

PCI_MEM32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit memory space. Normally, this is given by the BSP constant **PCI_MEM_SIZE**.

PCI_MEM32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit memory space.

PCI_MEMIO32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit non-prefetch memory space. Normally, this is given by the BSP constant **PCI_MEMIO_ADRS**.

PCI_MEMIO32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit non-prefetch memory space. Normally, this is given by the BSP constant **PCI_MEMIO_SIZE**.

PCI_MEMIO32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit non-prefetch memory space.

PCI_IO32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit I/O space. Normally, this is given by the BSP constant **PCI_IO_ADRS**.

PCI_IO32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit I/O space. Normally, this is given by the BSP constant **PCI_IO_SIZE**.

PCI_IO32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit I/O space.

PCI_IO16_LOC_SET - UINT32 pArg

Sets the base address of the PCI 16-bit I/O space. Normally, this is given by the BSP constant **PCI_ISA_IO_ADRS**.

PCI_IO16_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 16-bit I/O space. Normally, this is given by the BSP constant **PCI_ISA_IO_SIZE**.

PCI_IO16_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 16-bit I/O space.

PCI_INCLUDE_FUNC_SET - FUNCPTR * pArg

The device inclusion routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET** **pciAutoCfgCtl()** command:

```
pciAutoCfgCtl(pSystem, PCI_INCLUDE_FUNC_SET, sysPciAutoconfigInclude);
```

This optional user-supplied routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI vendorID and deviceID of the function.

The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigInclude
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This optional user-specified routine is called by PCI AutoConfig for each and every function encountered in the scan phase. The BSP developer may use any combination of the input data to ascertain whether a device is to be excluded from the autoconfig process. The exclusion routine then returns **ERROR** if a device is to be excluded, and **OK** if a device is to be included in the autoconfiguration process.

Note that PCI-to-PCI Bridges may not be excluded, regardless of the value returned by the BSP device inclusion routine. The return value is ignored for PCI-to-PCI bridges.

The Bridge device will be always be configured with proper primary, secondary, and subordinate bus numbers in the device scanning phase and proper I/O and Memory aperture settings in the configuration phase of autoconfig regardless of the value returned by the BSP device inclusion routine.

PCI_INT_ASSIGN_FUNC_SET - FUNCPTR * pArg

The interrupt assignment routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET** **pciAutoCfgCtl()** command:

```
pciAutoCfgCtl(pCookie, PCI_INT_ASSIGN_FUNC_SET, sysPciAutoconfigIntrAssign);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and an 8-bit quantity containing the contents of the interrupt Pin register from the PCI configuration header of the device under consideration. The interrupt pin register specifies which of the four PCI Interrupt request lines available are connected. The function prototype for this function is shown below:

```
UCHAR sysPciAutoconfigIntrAssign
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UCHAR pin
);
```

This routine may use any combination of these data to ascertain the interrupt level. This value is returned from the function, and is programmed into the interrupt line register of the function's PCI configuration header. In this manner, device drivers may subsequently read this register in order to calculate the appropriate interrupt vector which to attach an interrupt service routine.

PCI_BRIDGE_PRE_CONFIG_FUNC_SET - FUNCPTR * pArg

The bridge pre-configuration pass initialization routine is provided so that the BSP Developer can initialize a bridge device prior to the configuration pass on the bus that the bridge implements. This routine is specified by calling **pciAutoCfgCtl()** with the **PCI_BRIDGE_PRE_CONFIG_FUNC_SET** command:

```
pciAutoCfgCtl(pCookie, PCI_BRIDGE_PRE_CONFIG_FUNC_SET,
               sysPciAutoconfigPreEnumBridgeInit);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI deviceID and vendorID of the device. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigPreEnumBridgeInit
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This routine may use any combination of these input data to ascertain any special initialization requirements of a particular type of bridge at a specified geographic location.

PCI_BRIDGE_POST_CONFIG_FUNC_SET - FUNCPTR * pArg

The bridge post-configuration pass initialization routine is provided so that the BSP Developer can initialize the bridge device after the bus that the bridge implements has been enumerated. This routine is specified by calling **pciAutoCfgCtl()** with the **PCI_BRIDGE_POST_CONFIG_FUNC_SET** command

```
pciAutoCfgCtl(pCookie, PCI_BRIDGE_POST_CONFIG_FUNC_SET,
               sysPciAutoconfigPostEnumBridgeInit);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI deviceID and vendorID of the device. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigPostEnumBridgeInit
(
    PCI_SYSTEM *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This routine may use any combination of these input data to ascertain any special initialization requirements of a particular type of bridge at a specified geographic location.

PCI_ROLLCALL_FUNC_SET - FUNCPTR * pArg

The specified routine will be configured as a roll call routine.

If a roll call routine has been configured, before any configuration is actually done, the roll call routine is called repeatedly until it returns TRUE. A return value of TRUE indicates that either (1) the specified number and type of devices named in the roll call

list have been found during PCI bus enumeration or (2) the timeout has expired without finding all of the specified number and type of devices. In either case, it is assumed that all of the PCI devices which are going to appear on the busses have appeared and we can proceed with PCI bus configuration.

PCI_TEMP_SPACE_SET - char * pArg

This command is not currently implemented. It allows the user to set aside memory for use during **pciAutoConfigLib** execution, e.g. memory set aside using **USER_RESERVED_MEM**. After PCI configuration has been completed, the memory can be added to the system memory pool using **memAddToPool()**.

PCI_MINIMIZE_RESOURCES

This command is not currently implemented. It specifies that **pciAutoConfigLib** minimize requirements for memory and I/O space.

PCI_PSYSTEM_STRUCT_COPY - PCI_SYSTEM * pArg

This command has been added for ease of converting from the old interface to the new one. This will set each value as specified in the pSystem structure. If the **PCI_SYSTEM** structure has already been filled, the **pciAutoConfig(pSystem)** call can be changed to:

```
void *pCookie;  
pCookie = pciAutoConfigLibInit(NULL);  
pciAutoCfgCtl(pCookie, PCI_PSYSTEM_STRUCT_COPY, (void *)pSystem);  
pciAutoCfgFunc(pCookie);
```

The fields of the **PCI_SYSTEM** structure are defined below. For more information about each one, see the paragraphs above and the documentation for **pciAutoConfigLib**.

pciMem32

Specifies the 32-bit prefetchable memory pool base address.

pciMem32Size

Specifies the 32-bit prefetchable memory pool size.

pciMemIo32

Specifies the 32-bit non-prefetchable memory pool base address.

pciMemIo32Size

Specifies the 32-bit non-prefetchable memory pool size

pcIo32

Specifies the 32-bit I/O pool base address.

pcIo32Size

Specifies the 32-bit I/O pool size.

pcIo16

Specifies the 16-bit I/O pool base address.

pcIo16Size

Specifies the 16-bit I/O pool size.

includeRtn

Specifies the device inclusion routine.

intAssignRtn

Specifies the interrupt assignment routine.

autoIntRouting

Can be set to TRUE to configure **pciAutoConfig()** only to call the BSP interrupt routing routine for devices on bus number 0. Setting autoIntRoutine to FALSE will configure **pciAutoConfig()** to call the BSP interrupt routing routine for every device regardless of the bus on which the device resides.

bridgePreInit

Specifies the bridge initialization routine to call before initializing devices on the bus that the bridge implements.

bridgePostInit

Specifies the bridge initialization routine to call after initializing devices on the bus that the bridge implements.

RETURNS OK, or ERROR if the command or argument is invalid.

ERRNO EINVAL
if pCookie is not NULL or cmd is not recognized

SEE ALSO **pciAutoConfigLib**

pciAutoCfgCtl()

NAME **pciAutoCfgCtl()** – set or get **autoConfigLib** options

SYNOPSIS STATUS pciAutoCfgCtl()
(
void * pCookie, /* system configuration information */
int cmd, /* command word */
void * pArg /* argument for the cmd */
)

DESCRIPTION This gets or sets the **autoConfigLib** options.

RETURNS OK, or ERROR if the command or argument is invalid.

ERRNO Not Available

SEE ALSO **vxbPci**, **vxbPciAutoCfgCtl**

pciAutoConfig()

NAME **pciAutoConfig()** – automatically configure all nonexcluded PCI headers (obsolete)

SYNOPSIS

```
void pciAutoConfig
(
    PCI_SYSTEM * pSystem /* PCI system to configure */
)
```

DESCRIPTION This routine is obsolete. It is included for backward compatibility only. It is recommended that you use the **pciAutoCfg()** interface instead of this one.

Top level function in the PCI configuration process.

For all nonexcluded PCI functions on all PCI bridges, this routine will automatically configure the PCI configuration headers for PCI devices and subbridges. The fields that are programmed are:

1. Status register.
2. Command Register.
3. Latency timer.
4. Cache Line size.
5. Memory and/or I/O base address and limit registers.
6. Primary, secondary, subordinate bus number (for PCI-PCI bridges).
7. Expansion ROM disable.
8. Interrupt Line.

ALGORITHM Probe PCI config space and create a list of available PCI functions. Call device exclusion function, if registered, to exclude/include device. Disable all devices before we initialize any. Allocate and assign PCI space to each device. Calculate and set interrupt line value. Initialize and enable each device.

RETURNS N/A

ERRNO

SEE ALSO **pciAutoConfigLib**

pciAutoConfigLibInit()

NAME	pciAutoConfigLibInit() – initialize PCI autoconfig library
SYNOPSIS	<pre>void * pciAutoConfigLibInit (void * pArg /* reserved for future use */)</pre>
DESCRIPTION	pciAutoConfigLib initialization function.
RETURNS	A cookie for use by subsequent pciAutoConfigLib function calls.
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoDevReset()

NAME	pciAutoDevReset() – quiesce a PCI device and reset all writeable status bits
SYNOPSIS	<pre>STATUS pciAutoDevReset (PCI_LOC * pPciLoc /* device to be reset */)</pre>
DESCRIPTION	This routine turns off a PCI device by disabling the Memory decoders, I/O decoders, and Bus Master capability. The routine also resets all writeable status bits in the status word that follows the command word sequentially in PCI config space by performing a longword access.
RETURNS	OK, always.
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoFuncDisable()

NAME	pciAutoFuncDisable() – disable a specific PCI function
SYNOPSIS	<pre>void pciAutoFuncDisable (PCI_LOC *pPciFunc /* input: Pointer to PCI function struct */)</pre>
DESCRIPTION	This routine clears the I/O, mem, master, & ROM space enable bits for a single PCI function. The PCI spec says that devices should normally clear these by default after reset but in actual practice, some PCI devices do not fully comply. This routine ensures that the devices have all been disabled before configuration is started.
RETURNS	N/A
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoFuncEnable()

NAME	pciAutoFuncEnable() – perform final configuration and enable a function
SYNOPSIS	<pre>void pciAutoFuncEnable (PCI_SYSTEM * pSys, /* for backwards compatibility */ PCI_LOC * pFunc /* input: Pointer to PCI function structure */)</pre>
DESCRIPTION	Depending upon whether the device is included, this routine initializes a single PCI function as follows: Initialize the cache line size register Initialize the PCI-PCI bridge latency timers Enable the master PCI bit for non-display devices Set the interrupt line value with the value from the BSP.
RETURNS	N/A
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoGetNextClass()

NAME	pciAutoGetNextClass() – find the next device of specific type from probe list
SYNOPSIS	<pre>STATUS pciAutoGetNextClass (PCI_SYSTEM *pSys, /* for backwards compatibility */ PCI_LOC *pPciFunc, /* output: Contains the BDF of the device found */ UINT *index, /* Zero-based device instance number */ UINT pciClass, /* class code field from the PCI header */ UINT mask /* mask is ANDed with the class field */)</pre>
DESCRIPTION	The function uses the probe list which was built during the probing process. Using configuration accesses, it searches for the occurrence of the device subject to the class and mask restrictions outlined below. Setting class to zero and mask to zero allows searching the entire set of devices found regardless of class.
RETURNS	TRUE if a device was found, else FALSE.
ERRNO	
SEE ALSO	pciAutoConfigLib

pciAutoRegConfig()

NAME	pciAutoRegConfig() – assign PCI space to a single PCI base address register
SYNOPSIS	<pre>UINT pciAutoRegConfig (PCI_SYSTEM *pSys, /* backwards compatibility */ PCI_LOC *pPciFunc, /* Pointer to function in device list */ UINT baseAddr, /* Offset of base PCI address */ UINT nSize, /* Size and alignment requirements */ UINT addrInfo /* PCI address type information */)</pre>
DESCRIPTION	This routine allocates and assigns PCI space (either memory or I/O) to a single PCI base address register.
RETURNS	Returns (1) if BAR supports mapping anywhere in 64-bit address space. Returns (0) otherwise.

ERRNO

SEE ALSO [pciAutoConfigLib](#)

pciBusAnnounceDevices()

NAME [**pciBusAnnounceDevices\(\)**](#) – Notify the bus subsystem of all devices on PCI

SYNOPSIS

```
void pciBusAnnounceDevices
(
    struct vxbAccessList * pArg,
    struct vxbDev *      pDev,
    void *               pCookie
)
```

DESCRIPTION none

RETURNS N/A

ERRNO

SEE ALSO [vxbPci](#)

pciConfigBdfPack()

NAME [**pciConfigBdfPack\(\)**](#) – pack parameters for the Configuration Address Register

SYNOPSIS

```
int pciConfigBdfPack
(
    int busNo,      /* bus number */
    int deviceNo,   /* device number */
    int funcNo      /* function number */
)
```

DESCRIPTION This routine packs three parameters into one integer for accessing the Configuration Address Register

RETURNS packed integer encoded version of bus, device, and function numbers.

ERRNO

SEE ALSO **pciConfigLib**

pciConfigCmdWordShow()

NAME **pciConfigCmdWordShow()** – show the decoded value of the command word

SYNOPSIS STATUS pciConfigCmdWordShow

```
(  
    int bus,      /* bus */  
    int device,   /* device */  
    int function, /* function */  
    void *pArg    /* ignored */  
)
```

DESCRIPTION This routine reads the value of the command word for the specified bus, device, function and displays the information.

RETURNS OK, always.

ERRNO

SEE ALSO **pciConfigShow**

pciConfigCmdWordShow()

NAME **pciConfigCmdWordShow()** – show the decoded value of the command word

SYNOPSIS STATUS pciConfigCmdWordShow

```
(  
    int bus,      /* bus */  
    int device,   /* device */  
    int function, /* function */  
    void *pArg    /* ignored */  
)
```

DESCRIPTION This routine reads the value of the command word for the specified bus, device, function and displays the information.

RETURNS OK, always.

ERRNO

SEE ALSO vxbPci

pciConfigEnable()

NAME pciConfigEnable() – Set the globalBusCtrlID for the other LEGACY pci

SYNOPSIS

```
void pciConfigEnable
(
    VXB_DEVICE_ID busCtrlID
)
```

DESCRIPTION routines.

RETURNS N/A

ERRNO

SEE ALSO vxbPci

pciConfigExtCapFind()

NAME pciConfigExtCapFind() – find extended capability in ECP linked list

SYNOPSIS

```
STATUS pciConfigExtCapFind
(
    UINT8 extCapFindId, /* Extended capabilities ID to search for */
    int bus,           /* PCI bus number */
    int device,        /* PCI device number */
    int function,      /* PCI function number */
    UINT8 * pOffset    /* returned config space offset */
)
```

DESCRIPTION This routine searches for an extended capability in the linked list of capabilities in config space. If found, the offset of the first byte of the capability of interest in config space is returned via pOffset.

RETURNS OK if Extended Capability found, ERROR otherwise

ERRNO

SEE ALSO **pciConfigLib**

pciConfigExtCapFind()

NAME **pciConfigExtCapFind()** – find extended capability in ECP linked list

SYNOPSIS

```
STATUS pciConfigExtCapFind
(
    UINT8 extCapFindId,
    int bus,
    int device,
    int function,
    UINT8 * pOffset
)
```

DESCRIPTION It calls the underlying vxkBciConfigExtCapFind with globalBusCtrlID. SEE ALSO
pciConfigEnable.

RETURNS OK if Extended Capability found, **ERROR** otherwise

ERRNO

SEE ALSO **vxkBci**

pciConfigForeachFunc()

NAME **pciConfigForeachFunc()** – check condition on specified bus

SYNOPSIS

```
STATUS pciConfigForeachFunc
(
    UINT8 bus, /* bus to start on */
    BOOL recurse, /* if TRUE, do subordinate busses */
    PCI_FOREACH_FUNC funcCheckRtn, /* routine to call for each PCI func */
    void *pArg /* argument to funcCheckRtn */
)
```

DESCRIPTION **pciConfigForeachFunc()** discovers the PCI functions present on the bus and calls a specified C-function for each one. If the function returns **ERROR**, further processing stops.

pciConfigForeachFunc() does not affect any HOST-PCI bridge on the system.

RETURNS	OK normally, or ERROR if funcCheckRtn() doesn't return OK.
ERRNO	not set
SEE ALSO	pciConfigLib

pciConfigForeachFunc()

NAME	pciConfigForeachFunc() – check condition on specified bus
SYNOPSIS	<pre>STATUS pciConfigForeachFunc (UINT8 bus, /* bus to start on */ BOOL recurse, /* if TRUE, do subordinate busses */ PCI_FOREACH_FUNC funcCheckRtn, /* routine to call for each PCI func */ void *pArg /* argument to funcCheckRtn */)</pre>
DESCRIPTION	pciConfigForeachFunc() discovers the PCI functions present on the bus and calls a specified C-function for each one. If the function returns ERROR , further processing stops. pciConfigForeachFunc() does not affect any HOST-PCI bridge on the system.
RETURNS	OK normally, or ERROR if funcCheckRtn() doesn't return OK.
ERRNO	not set
SEE ALSO	vxbPci

pciConfigFuncShow()

NAME	pciConfigFuncShow() – show configuration details about a function
SYNOPSIS	<pre>STATUS pciConfigFuncShow (int bus, /* bus */ int device, /* device */ int function, /* function */ void *pArg /* ignored */)</pre>

DESCRIPTION	This routine reads various information from the specified bus, device, function, and displays the information.
RETURNS	OK, always.
ERRNO	
SEE ALSO	pciConfigShow

pciConfigFuncShow()

NAME	pciConfigFuncShow() – show configuration details about a function
SYNOPSIS	<pre>STATUS pciConfigFuncShow (int bus, /* bus */ int device, /* device */ int function, /* function */ void *pArg /* ignored */)</pre>
DESCRIPTION	This routine reads various information from the specified bus, device, function, and displays the information.
RETURNS	OK, always.
ERRNO	
SEE ALSO	vxbPci

pciConfigInByte()

NAME	pciConfigInByte() – read one byte from the PCI configuration space
SYNOPSIS	<pre>STATUS pciConfigInByte (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT8 * pData /* data read from the offset */)</pre>

DESCRIPTION	This routine reads one byte from the PCI configuration space
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	pciConfigLib

pciConfigInByte()

NAME	pciConfigInByte() – read one byte from the PCI configuration space
SYNOPSIS	<pre>STATUS pciConfigInByte (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT8 * pData /* data read from the offset */)</pre>
DESCRIPTION	This routine reads one byte from the PCI configuration space
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	vxbPci

pciConfigInLong()

NAME	pciConfigInLong() – read one longword from the PCI configuration space
SYNOPSIS	<pre>STATUS pciConfigInLong (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT32 * pData /* data read from the offset */)</pre>

DESCRIPTION	This routine reads one longword from the PCI configuration space
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	pciConfigLib

pciConfigInLong()

NAME	pciConfigInLong() – read one longword from the PCI configuration space
SYNOPSIS	STATUS pciConfigInLong (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT32 * pData /* data read from the offset */)
DESCRIPTION	
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	vxbPci

pciConfigInWord()

NAME	pciConfigInWord() – read one word from the PCI configuration space
SYNOPSIS	STATUS pciConfigInWord (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT16 * pData /* data read from the offset */)
DESCRIPTION	
RETURNS	OK, or ERROR if this library is not initialized

DESCRIPTION	This routine reads one word from the PCI configuration space
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	pciConfigLib

pciConfigInWord()

NAME	pciConfigInWord() – read one word from the PCI configuration space
SYNOPSIS	<pre>STATUS pciConfigInWord (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT16 * pData /* data read from the offset */)</pre>
DESCRIPTION	This routine reads one word from the PCI configuration space
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	vxbPci

pciConfigLibInit()

NAME	pciConfigLibInit() – initialize the configuration access-method and addresses
SYNOPSIS	<pre>STATUS pciConfigLibInit (int mechanism, /* configuration mechanism: 0, 1, 2 */ ULONG addr0, /* config-addr-reg / CSE-reg */ ULONG addr1, /* config-data-reg / Forward-reg */ ULONG addr2 /* none / Base-address */)</pre>
DESCRIPTION	This routine initializes the configuration access-method and addresses.

Configuration mechanism one utilizes two 32-bit IO ports located at addresses 0x0cf8 and 0x0cfc. These two ports are:

Port 1

32-bit configuration address port, at 0x0cf8

Port 2

32-bit configuration data port, at 0x0cfc

Accessing a PCI function's configuration port is two step process.

Step 1

Write the bus number, physical device number, function number and register number to the configuration address port.

Step 2

Perform an IO read from or an write to the configuration data port.

Configuration mechanism two uses following two single-byte IO ports.

Port 1

Configuration space enable, or CSE, register, at 0x0cf8

Port 2

Forward register, at 0x0cfa

To generate a PCI configuration transaction, the following actions are performed.

- Write the target bus number into the forward register.
- Write a one byte value to the CSE register at 0x0cf8. The bit pattern written to this register has three effects: disables the generation of special cycles; enables the generation of configuration transactions; specifies the target PCI functional device.
- Perform a one, two or four byte IO read or write transaction within the IO range 0xc000 through 0xffff.

Configuration mechanism zero is for non-PC/PowerPC environments where an area of address space produces PCI configuration transactions. No support for special cycles is included.

RETURNS OK, or ERROR if a mechanism is not 0, 1, or 2.

ERRNO

SEE ALSO [pciConfigLib](#)

pciConfigLibInit()

NAME

pciConfigLibInit() – initialize the configuration access-method and addresses

SYNOPSIS

```
STATUS vxbsPciConfigLibInit
(
    struct vxbsPciConfig *pPciConfig,
    int                 pciConfAddr0,
    int                 pciConfAddr1, /* PCI config data reg */
    int                 pciConfAddr2,
    int                 pciConfigMech, /* 1=mechanism-1, 2=mechanism-2 */
    int                 pciMaxBus /* Max number of sub-busses */
)
```

DESCRIPTION

This routine initializes the configuration access-method and addresses.

Configuration mechanism one utilizes two 32-bit IO ports located at addresses 0x0cf8 and 0x0fcf. These two ports are:

Port 1

32-bit configuration address port, at 0x0cf8

Port 2

32-bit configuration data port, at 0x0fcf

Accessing a PCI function's configuration port is two step process.

Step 1

Write the bus number, physical device number, function number and register number to the configuration address port.

Step 2

Perform an IO read from or an write to the configuration data port.

Configuration mechanism two uses following two single-byte IO ports.

Port 1

Configuration space enable, or CSE, register, at 0x0cf8

Port 2

Forward register, at 0x0cfa

To generate a PCI configuration transaction, the following actions are performed.

- Write the target bus number into the forward register.
- Write a one byte value to the CSE register at 0x0cf8. The bit pattern written to this register has three effects: disables the generation of special cycles; enables the generation of configuration transactions; specifies the target PCI functional device.
- Perform a one, two or four byte IO read or write transaction within the IO range 0xc000 through 0xffff.

Configuration mechanism zero is for non-PC/PowerPC environments where an area of address space produces PCI configuration transactions. No support for special cycles is included.

RETURNS OK, or ERROR if a mechanism is not 0, 1, or 2.

ERRNO

SEE ALSO [vxPci](#)

pciConfigModifyByte()

NAME `pciConfigModifyByte()` – Perform a masked longword register update

SYNOPSIS

```
STATUS pciConfigModifyByte
(
    int    busNo,      /* bus number */
    int    deviceNo,   /* device number */
    int    funcNo,     /* function number */
    int    offset,     /* offset into the configuration space */
    UINT8 bitMask,   /* Mask which defines field to alter */
    UINT8 data        /* data written to the offset */
)
```

DESCRIPTION This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the **bitMask** parameter.

Do not use this routine to modify any register that contains 'write 1 to clear' type of status bits in the same longword. This specifically applies to the command register. Modify byte operations could potentially be implemented as longword operations with bit shifting and masking. This could have the effect of clearing status bits in registers that aren't being updated. Use `pciConfigInLong` and `pciConfigOutLong`, or `pciModifyLong`, to read and update the entire longword.

RETURNS OK if operation succeeds, ERROR if operation fails.

ERRNO

SEE ALSO [pciConfigLib](#)

pciConfigModifyLong()

NAME

pciConfigModifyLong() – Perform a masked longword register update

SYNOPSIS

```
STATUS pciConfigModifyLong
(
    int    busNo,      /* bus number */
    int    deviceNo,   /* device number */
    int    funcNo,    /* function number */
    int    offset,    /* offset into the configuration space */
    UINT32 bitMask,  /* Mask which defines field to alter */
    UINT32 data       /* data written to the offset */
)
```

DESCRIPTION

This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the **bitMask** parameter.

Be careful to using **pciConfigModifyLong** for updating the Command and status register. The status bits must be written back as zeroes, else they will be cleared. Proper use involves including the status bits in the mask value, but setting their value to zero in the data value.

The following example will set the **PCI_CMD_IO_ENABLE** bit without clearing any status bits. The macro **PCI_CMD_MASK** includes all the status bits as part of the mask. The fact that **PCI_CMD_MASTER** doesn't include these bits, causes them to be written back as zeroes, therefore they aren't cleared.

```
pciConfigModifyLong (b, d, f, PCI_CFG_COMMAND,
                    (PCI_CMD_MASK | PCI_CMD_IO_ENABLE), PCI_CMD_IO_ENABLE);
```

Use of explicit longword read and write operations for dealing with any register containing "write 1 to clear" bits is sound policy.

RETURNS

OK if operation succeeds, **ERROR** if operation fails.

ERRNO

SEE ALSO

pciConfigLib

pciConfigModifyWord()

NAME

pciConfigModifyWord() – Perform a masked longword register update

SYNOPSIS	<pre>STATUS pciConfigModifyWord (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT16 bitMask, /* Mask which defines field to alter */ UINT16 data /* data written to the offset */)</pre>
DESCRIPTION	<p>This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the bitMask parameter.</p> <p>Do not use this routine to modify any register that contains 'write 1 to clear' type of status bits in the same longword. This specifically applies to the command register. Modify byte operations could potentially be implemented as longword operations with bit shifting and masking. This could have the effect of clearing status bits in registers that aren't being updated. Use pciConfigInLong and pciConfigOutLong, or pciModifyLong, to read and update the entire longword.</p>
RETURNS	OK if operation succeeds. ERROR if operation fails.
ERRNO	
SEE ALSO	pciConfigLib

pciConfigOutByte()

NAME	pciConfigOutByte() – write one byte to the PCI configuration space
SYNOPSIS	<pre>STATUS pciConfigOutByte (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo, /* function number */ int offset, /* offset into the configuration space */ UINT8 data /* data written to the offset */)</pre>
DESCRIPTION	This routine writes one byte to the PCI configuration space.
RETURNS	OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [pciConfigLib](#)

pciConfigOutByte()

NAME [pciConfigOutByte\(\)](#) – write one byte to the PCI configuration space

SYNOPSIS

```
STATUS pciConfigOutByte
(
    int    busNo,      /* bus number */
    int    deviceNo,   /* device number */
    int    funcNo,     /* function number */
    int    offset,     /* offset into the configuration space */
    UINT8 data        /* data written to the offset */
)
```

DESCRIPTION This routine writes one byte to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [vxbPci](#)

pciConfigOutLong()

NAME [pciConfigOutLong\(\)](#) – write one longword to the PCI configuration space

SYNOPSIS

```
STATUS pciConfigOutLong
(
    int    busNo,      /* bus number */
    int    deviceNo,   /* device number */
    int    funcNo,     /* function number */
    int    offset,     /* offset into the configuration space */
    UINT32 data       /* data written to the offset */
)
```

DESCRIPTION This routine writes one longword to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [pciConfigLib](#)

pciConfigOutLong()

NAME [pciConfigOutLong\(\)](#) – write one longword to the PCI configuration space

SYNOPSIS

```
STATUS pciConfigOutLong(
    (
        int      busNo,      /* bus number */
        int      deviceNo,   /* device number */
        int      funcNo,    /* function number */
        int      offset,    /* offset into the configuration space */
        UINT32  data       /* data written to the offset */
    )
```

DESCRIPTION This routine writes one longword to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [vxbPci](#)

pciConfigOutWord()

NAME [pciConfigOutWord\(\)](#) – write one 16-bit word to the PCI configuration space

SYNOPSIS

```
STATUS pciConfigOutWord(
    (
        int      busNo,      /* bus number */
        int      deviceNo,   /* device number */
        int      funcNo,    /* function number */
        int      offset,    /* offset into the configuration space */
        UINT16  data       /* data written to the offset */
    )
```

DESCRIPTION This routine writes one 16-bit word to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [pciConfigLib](#)

pciConfigOutWord()

NAME [pciConfigOutWord\(\)](#) – write one 16-bit word to the PCI configuration space

SYNOPSIS STATUS pciConfigOutWord
(
 int busNo, /* bus number */
 int deviceNo, /* device number */
 int funcNo, /* function number */
 int offset, /* offset into the configuration space */
 UINT16 data /* data written to the offset */
)

DESCRIPTION This routine writes one 16-bit word to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [vxbPci](#)

pciConfigReset()

NAME [pciConfigReset\(\)](#) – disable cards for warm boot

SYNOPSIS STATUS pciConfigReset
(
 int startType /* for reboot hook, ignored */
)

DESCRIPTION [pciConfigReset\(\)](#) goes through the list of PCI functions at the top-level bus and disables them, preventing them from writing to memory while the system is trying to reboot.

RETURNS OK, always

ERRNO Not set

SEE ALSO

[pciConfigLib](#)

pciConfigStatusWordShow()

NAME **pciConfigStatusWordShow()** – show the decoded value of the status word

SYNOPSIS

```
STATUS pciConfigStatusWordShow
(
    int bus,          /* bus */
    int device,       /* device */
    int function,     /* function */
    void *pArg        /* ignored */
)
```

DESCRIPTION This routine reads the value of the status word for the specified bus, device, function and displays the information.

RETURNS OK, always.

ERRNO

SEE ALSO [pciConfigShow](#)

pciConfigStatusWordShow()

NAME **pciConfigStatusWordShow()** – show the decoded value of the status word

SYNOPSIS

```
STATUS pciConfigStatusWordShow
(
    int bus,          /* bus */
    int device,       /* device */
    int function,     /* function */
    void *pArg        /* ignored */
)
```

DESCRIPTION This routine reads the value of the status word for the specified bus, device, function and displays the information.

RETURNS OK, always.

ERRNO

SEE ALSO [vxPci](#)

pciConfigTopoShow()

NAME [pciConfigTopoShow\(\) – show PCI topology](#)

SYNOPSIS `void pciConfigTopoShow(void)`

DESCRIPTION This routine traverses the PCI bus and prints assorted information about every device found. The information is intended to present the topology of the PCI bus. It includes: (1) the device type, (2) the command and status words, (3) for PCI to PCI bridges the memory and I/O space configuration, and (4) the values of all implemented BARs.

RETURNS N/A

ERRNO

SEE ALSO [pciConfigShow](#)

pciConfigTopoShow()

NAME [pciConfigTopoShow\(\) – show PCI topology](#)

SYNOPSIS `STATUS pciConfigTopoShow(void)`

DESCRIPTION This routine traverses the PCI bus and prints assorted information about every device found. The information is intended to present the topology of the PCI bus. It includes: (1) the device type, (2) the command and status words, (3) for PCI to PCI bridges the memory and I/O space configuration, and (4) the values of all implemented BARs.

RETURNS N/A

ERRNO

SEE ALSO [vxPci](#)

pciConnect()

NAME	pciConnect() – connect PCI bus type to bus subsystem
SYNOPSIS	STATUS pciConnect(void)
DESCRIPTION	This function currently does nothing.
RETURNS	OK, always
ERRNO	
SEE ALSO	vxbPci

pciDevConfig()

NAME	pciDevConfig() – configure a device on a PCI bus
SYNOPSIS	STATUS pciDevConfig (int pciBusNo, /* PCI bus number */ int pciDevNo, /* PCI device number */ int pciFuncNo, /* PCI function number */ UINT32 devIoBaseAdrs, /* device I/O base address */ UINT32 devMemBaseAdrs, /* device memory base address */ UINT32 command /* command to issue */)
DESCRIPTION	This routine configures a device that is on a Peripheral Component Interconnect (PCI) bus by writing to the configuration header of the selected device. It first disables the device by clearing the command register in the configuration header. It then sets the I/O and/or memory space base address registers, the latency timer value and the cache line size. Finally, it re-enables the device by loading the command register with the specified command.
NOTE	This routine is designed for Type 0 PCI Configuration Headers ONLY. It is NOT usable for configuring, for example, a PCI-to-PCI bridge.
RETURNS	OK always.

ERRNO

SEE ALSO [pciConfigLib](#)

pciDevConfig()

NAME [pciDevConfig\(\)](#) – configure a device on a PCI bus

SYNOPSIS

```
STATUS pciDevConfig
    (
        int      pciBusNo,
        int      pciDevNo,
        int      pciFuncNo,
        UINT32  devIoBaseAdrs,
        UINT32  devMemBaseAdrs,
        UINT32  command
    )
```

DESCRIPTION It calls the underlying vxbPciDevConfig with globalBusCtrlID. See [pciConfigEnable](#).

RETURNS OK always.

ERRNO

SEE ALSO [vxbPci](#)

pciDevMatch()

NAME [pciDevMatch\(\)](#) – check whether device and driver go together

SYNOPSIS

```
BOOL pciDevMatch
    (
        struct vxbDevRegInfo * pDriver, /* Pointer to device driver */
        struct vxbDev *         pDev     /* Device information */
    )
```

DESCRIPTION This routine checks that the specified device and device driver are a matched pair.

RETURNS TRUE if the device and driver match, FALSE otherwise

ERRNO

SEE ALSO [vxPci](#)

pciDevShow()

NAME [pciDevShow\(\)](#) – show information about PCI device

SYNOPSIS

```
void pciDevShow
(
    struct vxPci * pDev /* Device information */
)
```

DESCRIPTION none

RETURNS OK, or **ERROR** if there's an error

ERRNO

SEE ALSO [vxPci](#)

pciDeviceAnnounce()

NAME [pciDeviceAnnounce\(\)](#) – notify the bus subsystem of a device on PCI

SYNOPSIS

```
STATUS pciDeviceAnnounce
(
    VXB_DEVICE_ID busCtrlID,
    UINT8        bus,          /* PCI bus number */
    UINT8        dev,          /* PCI device number */
    UINT8        func,         /* PCI function number */
    void *       pArg,         /* pDev */
)
```

DESCRIPTION This routine tells the vxBus subsystem that a PCI device has been found on the PCI bus.

RETURNS OK, or **ERROR**

ERRNO

SEE ALSO [vxPci](#)

pciDeviceShow()

NAME	pciDeviceShow() – print information about PCI devices
SYNOPSIS	STATUS pciDeviceShow (int busNo /* bus number */)
DESCRIPTION	This routine prints information about the PCI devices on a given PCI bus segment (specified by <i>busNo</i>).
RETURNS	OK, or ERROR if the library is not initialized.
ERRNO	
SEE ALSO	pciConfigShow

pciDeviceShow()

NAME	pciDeviceShow() – print information about PCI devices
SYNOPSIS	STATUS pciDeviceShow (int busNo /* bus number */)
DESCRIPTION	This routine prints information about the PCI devices on a given PCI bus segment (specified by <i>busNo</i>).
RETURNS	OK, or ERROR if the library is not initialized.
ERRNO	
SEE ALSO	vxbPci

pciFindClass()

NAME	pciFindClass() – find the nth occurrence of a device by PCI class code.
-------------	---

SYNOPSIS

```
STATUS pciFindClass
(
    int    classCode, /* 24-bit class code */
    int    index,     /* desired instance of device */
    int * pBusNo,    /* bus number */
    int * pDeviceNo, /* device number */
    int * pFuncNo    /* function number */
)
```

DESCRIPTION This routine finds the nth device with the given 24-bit PCI class code (class subclass prog_if).

The classcode arg of must be carefully constructed from class and sub-class macros.

Example : To find an ethernet class device, construct the classcode arg as follows:

```
((PCI_CLASS_NETWORK_CTLR << 16 | PCI_SUBCLASS_NET_ETHERNET << 8))
```

RETURNS OK, or ERROR if the class didn't match.

ERRNO

SEE ALSO [pciConfigLib](#)

pciFindClass()

NAME **pciFindClass()** – find the nth occurrence of a device by PCI class code.

SYNOPSIS

```
STATUS pciFindClass
(
    int    classCode, /* 24-bit class code */
    int    index,     /* desired instance of device */
    int * pBusNo,    /* bus number */
    int * pDeviceNo, /* device number */
    int * pFuncNo    /* function number */
)
```

DESCRIPTION This routine finds the nth device with the given 24-bit PCI class code (class subclass prog_if).

The classcode arg of must be carefully constructed from class and sub-class macros.

Example : To find an ethernet class device, construct the classcode arg as follows:

```
((PCI_CLASS_NETWORK_CTLR << 16 | PCI_SUBCLASS_NET_ETHERNET << 8))
```

RETURNS OK, or ERROR if the class didn't match.

ERRNO

SEE ALSO [vxPci](#)

pciFindClassShow()

NAME	pciFindClassShow() – find a device by 24-bit class code
SYNOPSIS	STATUS pciFindClassShow (int classCode, /* 24-bit class code */ int index /* desired instance of device */)
DESCRIPTION	This routine finds a device by its 24-bit PCI class code, then prints its information.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	pciConfigShow

pciFindClassShow()

NAME	pciFindClassShow() – find a device by 24-bit class code
SYNOPSIS	STATUS pciFindClassShow (int classCode, /* 24-bit class code */ int index /* desired instance of device */)
DESCRIPTION	This routine finds a device by its 24-bit PCI class code, then prints its information.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	vxbPci

pciFindDevice()

NAME	pciFindDevice() – find the nth device with the given device & vendor ID
-------------	---

SYNOPSIS

```
STATUS pciFindDevice
(
    int    vendorId,    /* vendor ID */
    int    deviceId,    /* device ID */
    int    index,        /* desired instance of device */
    int * pBusNo,       /* bus number */
    int * pDeviceNo,   /* device number */
    int * pFuncNo      /* function number */
)
```

DESCRIPTION This routine finds the nth device with the given device & vendor ID.

RETURNS OK, or ERROR if the deviceId and vendorId didn't match.

ERRNO

SEE ALSO [pciConfigLib](#)

pciFindDevice()

NAME **pciFindDevice()** – find the nth device with the given device & vendor ID

SYNOPSIS

```
STATUS pciFindDevice
(
    int    vendorId,    /* vendor ID */
    int    deviceId,    /* device ID */
    int    index,        /* desired instance of device */
    int * pBusNo,       /* bus number */
    int * pDeviceNo,   /* device number */
    int * pFuncNo      /* function number */
)
```

DESCRIPTION This routine finds the nth device with the given device & vendor ID.

RETURNS OK, or ERROR if the deviceId and vendorId didn't match.

ERRNO

SEE ALSO [vxbPci](#)

pciFindDeviceShow()

NAME	pciFindDeviceShow() – find a PCI device and display the information
SYNOPSIS	<pre>STATUS pciFindDeviceShow (int vendorId, /* vendor ID */ int deviceId, /* device ID */ int index /* desired instance of device */)</pre>
DESCRIPTION	This routine finds a device by deviceId, then displays the information.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	pciConfigShow

pciFindDeviceShow()

NAME	pciFindDeviceShow() – find a PCI device and display the information
SYNOPSIS	<pre>STATUS pciFindDeviceShow (int vendorId, /* vendor ID */ int deviceId, /* device ID */ int index /* desired instance of device */)</pre>
DESCRIPTION	This routine finds a device by deviceId, then displays the information.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	vxbPci

pciHcfRecordFind()

NAME **pciHcfRecordFind()** – find device's HCF pciSlot record

SYNOPSIS

```
HCF_DEVICE * pciHcfRecordFind
(
    int pciBus,      /* PCI Bus number */
    int pciDevice   /* PCI device number */
)
```

DESCRIPTION This routine finds HCF pciSlot record for the specified bus and device.

RETURNS Pointer to record, or **NULL**

ERRNO

SEE ALSO [vxbPci](#)

pciHeaderShow()

NAME **pciHeaderShow()** – print a header of the specified PCI device

SYNOPSIS

```
STATUS pciHeaderShow
(
    int busNo,      /* bus number */
    int deviceNo,   /* device number */
    int funcNo     /* function number */
)
```

DESCRIPTION This routine prints a header of the PCI device specified by busNo, deviceNo, and funcNo.

RETURNS OK, or **ERROR** if this library is not initialized.

ERRNO

SEE ALSO [pciConfigShow](#)

pciHeaderShow()

NAME **pciHeaderShow()** – print a header of the specified PCI device

SYNOPSIS	STATUS pciHeaderShow (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo /* function number */)
DESCRIPTION	This routine prints a header of the PCI device specified by busNo, deviceNo, and funcNo.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	vxbPci

pciInit()

NAME	pciInit() – first-pass bus type initialization
SYNOPSIS	STATUS pciInit1(void)
DESCRIPTION	This function currently does nothing.
RETURNS	OK, always
ERRNO	
SEE ALSO	vxbPci

pciInit2()

NAME	pciInit2() – second-pass bus type initialization
SYNOPSIS	STATUS pciInit2(void)
DESCRIPTION	This function currently does nothing.
RETURNS	OK, always

ERRNO

SEE ALSO [vxbPci](#)

pciInt()

NAME **pciInt()** – interrupt handler for shared PCI interrupt.

SYNOPSIS

```
VOID pciInt
(
    int irq /* IRQ associated to the PCI interrupt */
)
```

DESCRIPTION This routine executes multiple interrupt handlers for a PCI interrupt. Each interrupt handler must check the device dependent interrupt status bit to determine the source of the interrupt, since it simply execute all interrupt handlers in the link list.

This is not a user callable routine

RETURNS N/A

ERRNO

SEE ALSO [pciIntLib](#)

pciIntConnect()

NAME **pciIntConnect()** – connect the interrupt handler to the PCI interrupt.

SYNOPSIS

```
STATUS pciIntConnect
(
    VOIDFUNCPTR *vector, /* interrupt vector to attach to */
    VOIDFUNCPTR routine, /* routine to be called */
    int parameter /* parameter to be passed to routine */
)
```

DESCRIPTION This routine connects an interrupt handler to a shared PCI interrupt vector. A link list is created for each shared interrupt used in the system. It is created when the first interrupt handler is attached to the vector. Subsequent calls to pciIntConnect just add their routines to the linked list for that vector.

RETURNS OK, or ERROR if the interrupt handler cannot be built.

ERRNO

SEE ALSO **pciIntLib**

pciIntConnect()

NAME **pciIntConnect()** – connect the interrupt handler to the PCI interrupt.

SYNOPSIS STATUS pciIntConnect
(
 VOIDFUNC PTR *vector, /* interrupt vector to attach to */
 VOIDFUNC PTR routine, /* routine to be called */
 int parameter /* parameter to be passed to routine */
)

DESCRIPTION This routine connects an interrupt handler to a shared PCI interrupt vector. A link list is created for each shared interrupt used in the system. It is created when the first interrupt handler is attached to the vector. Subsequent calls to pciIntConnect just add their routines to the linked list for that vector.

RETURNS OK, or ERROR if the interrupt handler cannot be built.

ERRNO

SEE ALSO **vxbPci**

pciIntDisconnect()

NAME **pciIntDisconnect()** – disconnect the interrupt handler (OBSOLETE)

SYNOPSIS STATUS pciIntDisconnect
(
 VOIDFUNC PTR *vector, /* interrupt vector to attach to */
 VOIDFUNC PTR routine /* routine to be called */
)

DESCRIPTION This routine disconnects the interrupt handler from the PCI interrupt line.

In a system where one driver and one ISR services multiple devices, this routine removes all instances of the ISR because it completely ignores the parameter argument used to install the handler.

NOTE	Use of this routine is discouraged and will be obsoleted in the future. New code should use the pciIntDisconnect2() routine instead.
RETURNS	OK, or ERROR if the interrupt handler cannot be removed.
ERRNO	
SEE ALSO	pciIntLib

pciIntDisconnect()

NAME	pciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)
SYNOPSIS	<pre>STATUS pciIntDisconnect (VOIDFUNC PTR *vector, /* interrupt vector to attach to */ VOIDFUNC PTR routine /* routine to be called */)</pre>
DESCRIPTION	This routine disconnects the interrupt handler from the PCI interrupt line. In a system where one driver and one ISR services multiple devices, this routine removes all instances of the ISR because it completely ignores the parameter argument used to install the handler.
NOTE	Use of this routine is discouraged and will be obsoleted in the future. New code should use the pciIntDisconnect2() routine instead.
RETURNS	OK, or ERROR if the interrupt handler cannot be removed.
ERRNO	
SEE ALSO	vxbPci

pciIntDisconnect2()

NAME **pciIntDisconnect2()** – disconnect an interrupt handler from the PCI interrupt.

SYNOPSIS

```
STATUS pciIntDisconnect2
(
    VOIDFUNC PTR *vector, /* interrupt vector to attach to      */
    VOIDFUNC PTR routine, /* routine to be called           */
    int          parameter /* routine parameter            */
)
```

DESCRIPTION This routine disconnects a single instance of an interrupt handler from the PCI interrupt line.

NOTE This routine should be used in preference to the original **pciIntDisconnect()** routine. This routine is compatible with drivers that are managing multiple device instances, using the same basic ISR, but with different parameters.

RETURNS OK, or ERROR if the interrupt handler cannot be removed.

ERRNO

SEE ALSO **pciIntLib**

pciIntDisconnect2()

NAME **pciIntDisconnect2()** – disconnect the interrupt handler

SYNOPSIS

```
STATUS pciIntDisconnect2
(
    VOIDFUNC PTR *vector, /* interrupt vector to attach to      */
    VOIDFUNC PTR routine, /* routine to be called           */
    int          arg
)
```

DESCRIPTION This routine disconnects the interrupt handler from the PCI interrupt line.
Use this routine in a system where one driver and one ISR services multiple devices.

RETURNS OK, or ERROR if the interrupt handler cannot be removed.

ERRNO

SEE ALSO **vxbPci**

pciIntLibInit()

NAME	pciIntLibInit() – initialize the pciIntLib module
SYNOPSIS	STATUS pciIntLibInit (void)
DESCRIPTION	This routine initializes the linked lists used to chain together the PCI interrupt service routines.
RETURNS	OK, or ERROR upon link list failures.
ERRNO	
SEE ALSO	pciIntLib

pciRegister()

NAME	pciRegister() – register PCI bus type
SYNOPSIS	STATUS pciRegister(void)
DESCRIPTION	This routine registers the PCI bus type with the vxBus subsystem.
RETURNS	OK, always
ERRNO	
SEE ALSO	vxbPci

pciSpecialCycle()

NAME	pciSpecialCycle() – generate a special cycle with a message
SYNOPSIS	STATUS pciSpecialCycle (int busNo, /* bus number */ UINT32 message /* data driven onto AD[31:0] */)

DESCRIPTION	This routine generates a special cycle with a message.
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	pciConfigLib

pciSpecialCycle()

NAME	pciSpecialCycle() – generate a special cycle with a message
SYNOPSIS	STATUS vxbPciSpecialCycle (VXB_DEVICE_ID busCtrlID, int busNo, /* bus number */ UINT32 message /* data driven onto AD[31:0] */)
DESCRIPTION	This routine generates a special cycle with a message.
RETURNS	OK, or ERROR if this library is not initialized
ERRNO	
SEE ALSO	vxbPci

pcicInit()

NAME	pcicInit() – initialize the PCIC chip
SYNOPSIS	STATUS pcicInit (int ioBase, /* IO base address */ int intVec, /* interrupt vector */ int intLevel, /* interrupt level */ FUNCPTR showRtn /* show routine */)
DESCRIPTION	This routine initializes the PCIC chip.

RETURNS	OK, or ERROR if the PCIC chip cannot be found.
ERRNO	Not Available
SEE ALSO	pcic

pcicShow()

NAME	pcicShow() – show all configurations of the PCIC chip
SYNOPSIS	<pre>void pcicShow (int sock /* socket no. */)</pre>
DESCRIPTION	This routine shows all configurations of the PCIC chip.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	pcicShow

pcmciaInit()

NAME	pcmciaInit() – initialize the PCMCIA event-handling package
SYNOPSIS	<pre>STATUS pcmciaInit (void)</pre>
DESCRIPTION	This routine installs the PCMCIA event-handling facilities and spawns pcmciad() , which performs special PCMCIA event-handling functions that need to be done at task level. It also creates the message queue used to communicate with pcmciad() .
RETURNS	OK, or ERROR if a message queue cannot be created or pcmciad() cannot be spawned.
ERRNO	Not Available
SEE ALSO	pcmciaLib , pcmciad()

pcmciaShow()

NAME	pcmciaShow() – show all configurations of the PCMCIA chip
SYNOPSIS	void pcmciaShow (int sock /* socket no. */)
DESCRIPTION	This routine shows all configurations of the PCMCIA chip.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	pcmciaShow

pcmciaShowInit()

NAME	pcmciaShowInit() – initialize all show routines for PCMCIA drivers
SYNOPSIS	void pcmciaShowInit (void)
DESCRIPTION	This routine initializes all show routines related to PCMCIA drivers.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	pcmciaShow

pcmciad()

NAME	pcmciad() – handle task-level PCMCIA events
SYNOPSIS	void pcmciad (void)

DESCRIPTION	This routine is spawned as a task by pcmciaInit() to perform functions that cannot be performed at interrupt or trap level. It has a priority of 0. Do not suspend, delete, or change the priority of this task.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	pcmciaLib, pcmciaInit()

pentiumPciAllHeaderShow()

NAME	pentiumPciAllHeaderShow() – show PCI header for specified device
SYNOPSIS	STATUS pentiumPciAllHeaderShow() (int bus, /* Bus number */ int dev, /* Device number */ int function, /* Function number */ void * pArg /* Argument - not currently used */)
DESCRIPTION	This routine displays the PCI header for the specified device.
RETURNS	OK, always
ERRNO	
SEE ALSO	pentiumPci

pentiumPciBusDevGet()

NAME	pentiumPciBusDevGet() – find bus controller
SYNOPSIS	struct vxbDev * pentiumPciBusDevGet() (struct vxbDev * pDev /* device info */)
DESCRIPTION	none

RETURNS	a pointer to the pDev structure for the Pentium PCI bus controller
ERRNO	Not Available
SEE ALSO	pentiumPci

pentiumPciMmuMapAdd()

NAME	pentiumPciMmuMapAdd() – memory map sysPhysMemDesc
SYNOPSIS	<pre>STATUS pentiumPciMmuMapAdd (void * address, /* memory region base address */ UINT length, /* memory region length in bytes*/ UINT initialStateMask, /* PHYS_MEM_DESC state mask */ UINT initialState /* PHYS_MEM_DESC state */)</pre>
DESCRIPTION	This routine adds memory mappings to sysPhysMemDesc.
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	pentiumPci

pentiumPciPhysMemHandle()

NAME	pentiumPciPhysMemHandle() – configure PCI memory for a device
SYNOPSIS	<pre>STATUS pentiumPciPhysMemHandle (VXB_DEVICE_ID busCtrlID, int bus, /* Bus number */ int dev, /* Device number */ int func, /* Function number */ void * pArg /* Argument */)</pre>
DESCRIPTION	This routine maps memory for the specified PCI device.

RETURNS OK, always

ERRNO

SEE ALSO **pentiumPci**

pentiumPciPhysMemShow()

NAME **pentiumPciPhysMemShow()** – display sysPhysMemDesc entries

SYNOPSIS void pentiumPciPhysMemShow (void)

DESCRIPTION This routine displays the sysPhysMemDesc[] entries.

RETURNS N/A

ERRNO

SEE ALSO **pentiumPci**

pentiumPciRegAddrShow()

NAME **pentiumPciRegAddrShow()** – debug routine to print register addresses

SYNOPSIS void * pentiumPciRegAddrShow
(
 struct vxbdDev * pDev,
 UINT32 regBaseIndex,
 UINT32 byteOffset,
 UINT8 * pDataBuf,
 UINT32 * pFlags
)

DESCRIPTION This routine prints the actual processor addresses used by the access methods to read and write device registers.

RETURNS pointer to register address

ERRNO

SEE ALSO **pentiumPci**

pentiumPciRegister()

NAME	pentiumPciRegister() – register Pentium PCI host bridge device driver
SYNOPSIS	<code>void pentiumPciRegister (void)</code>
DESCRIPTION	This routine registers the Pentium PCI host bridge with vxbus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	pentiumPci

plbAccessInit()

NAME	plbAccessInit() – initialize the plb access module
SYNOPSIS	<code>void plbAccessInit (void)</code>
DESCRIPTION	This routine is used to initialize the access module of the PLB
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPlbAccess

plbConnect()

NAME	plbConnect() – third-stage PLB bus initialization
SYNOPSIS	<code>STATUS plbConnect(void)</code>
DESCRIPTION	This routine executes the third stage of the PLB bus initialization and connection.
RETURNS	OK, always

ERRNO

SEE ALSO **vxPlb**

plbDevMatch()

NAME **plbDevMatch()** – check whether device and driver go together

SYNOPSIS

```
BOOL plbDevMatch
(
    struct vxbDevRegInfo * pDriver,
    struct vxbDev *          pDev
)
```

DESCRIPTION This routine matches a device with a device driver.

RETURNS TRUE if the device and driver match up, FALSE otherwise

ERRNO

SEE ALSO **vxPlb**

plbInit1()

NAME **plbInit1()** – first-stage PLB bus initialization

SYNOPSIS

```
STATUS plbInit1
(
    struct vxbDev * pCtlr
)
```

DESCRIPTION This routine executes the first stage of the PLB bus initialization.

RETURNS OK, or ERROR

ERRNO

SEE ALSO **vxPlb**

plbInit2()

NAME	plbInit2() – second-stage PLB bus initialization
SYNOPSIS	STATUS plbInit2(void)
DESCRIPTION	This routine executes the second stage of the PLB bus initialization.
RETURNS	OK, always
ERRNO	
SEE ALSO	vxbPlb

plbIntrGet()

NAME	plbIntrGet() – Get interrupt controller for device
SYNOPSIS	STATUS plbIntrGet (VXB_DEVICE_ID pDev, int indx, VXB_DEVICE_ID * ppIntCtrlr, /* out */ int * pInputPin /* out */)
DESCRIPTION	This routine retrieves the interrupt controller for a specific device interrupt output. The values for interrupt controller and input pin on the interrupt controller are set in the variables pointed to by the 3rd and 4th arguments.
RETURNS	OK, or ERROR if the interrupt controller is not known.
ERRNO	Not Available
SEE ALSO	vxbPlb

plbIntrSet()

NAME	plbIntrSet() – set interrupt controller for device
-------------	--

SYNOPSIS	<pre>STATUS plbIntrSet (VXB_DEVICE_ID pDev, int idx, VXB_DEVICE_ID pIntCtlr, int inputPin)</pre>
DESCRIPTION	This routine modifies the interrupt information structure associated with the device, to indicate which interrupt controller it is connected to.
RETURNS	OK, or ERROR if structures cannot be allocated or if parameters are illegal
ERRNO	Not Available
SEE ALSO	vxbPlb

plbIntrShow()

NAME	plbIntrShow() – Show interrupt controllers for device's interrupts
SYNOPSIS	<pre>int plbIntrShow (VXB_DEVICE_ID pDev, int verboseLevel)</pre>
DESCRIPTION	This routine shows the interrupt connectivity information for the specified device.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbPlb

plbRegister()

NAME	plbRegister() – register PLB with bus subsystem
-------------	---

SYNOPSIS	void plbRegister(void)
DESCRIPTION	This routine registers the PLB with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPlb

ppc403DevInit()

NAME	ppc403DevInit() – initialize the serial port unit
SYNOPSIS	void ppc403DevInit (PPC403_CHAN * pChan)
DESCRIPTION	The BSP must already have initialized all the device addresses in the PPC403_CHAN structure. This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state.
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	ppc403Sio

ppc403DummyCallback()

NAME	ppc403DummyCallback() – dummy callback routine
SYNOPSIS	STATUS ppc403DummyCallback (void)
DESCRIPTION	none
RETURNS	ERROR (always).
ERRNO	Not Available

SEE ALSO [ppc403Sio](#)

ppc403IntEx()

NAME [ppc403IntEx\(\)](#) – handle error interrupts

SYNOPSIS

```
void ppc403IntEx
(
    PPC403_CHAN * pChan
)
```

DESCRIPTION This routine handles miscellaneous interrupts on the serial communication controller.

RETURNS N/A

ERRNO Not Available

SEE ALSO [ppc403Sio](#)

ppc403IntRd()

NAME [ppc403IntRd\(\)](#) – handle a receiver interrupt

SYNOPSIS

```
void ppc403IntRd
(
    PPC403_CHAN * pChan
)
```

DESCRIPTION This routine handles read interrupts from the serial communication controller.

RETURNS N/A

ERRNO Not Available

SEE ALSO [ppc403Sio](#)

ppc403IntWr()

NAME	ppc403IntWr() – handle a transmitter interrupt
SYNOPSIS	void ppc403IntWr <pre>(PPC403_CHAN * pChan)</pre>
DESCRIPTION	This routine handles write interrupts from the serial communication controller.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	ppc403Sio

ppc440gpPciHostBridgeInit()

NAME	ppc440gpPciHostBridgeInit() – Initialize the PCI-X Host Bridge																								
SYNOPSIS	STATUS ppc440gpPciHostBridgeInit(void)																								
DESCRIPTION	Initializes the PCI bridge so it can operate as both a PCI master and slave. Parameters set are: <ul style="list-style-type: none"> - CPU->PCI (master/initiator/outbound) address translation - PCI->CPU (slave/target/inbound) address translation <p>This routine allocates virtual memory from the BSP-supplied virtual memory pools named pVmPref and pVmNoPref. The physical addresses for non-prefetchable and prefetchable PCI regions are hard-coded in this file to the values 0x3.0000.0000, 0x3.1000.0000, respectively. The pVmIO and pVmIOExtra pools are ignored, since addresses and configuration for the I/O and "Extra" I/O regions (in POM2) are hardwired in the device. In addition, the upper 32-bits of all 64-bit addresses are hard-coded to 0x00000000.</p> <p>The inbound window is hard-coded to start at 0x00000000 and have size of 1GB. See the macros in config.h for details.</p> <p>32 bit PCI bus memory map:</p> <table> <thead> <tr> <th>size</th> <th>CPU (virt)</th> <th>PLB (real)</th> <th>PCI</th> <th>usage</th> <th>map by</th> </tr> </thead> <tbody> <tr> <td>0x40000000</td> <td>0x00000000</td> <td>0x0.00000000</td> <td>0x00000000</td> <td>440GP slave</td> <td>PIM0</td> </tr> <tr> <td>0x10000000</td> <td>pVmNoPref</td> <td>0x3.00000000</td> <td>pVmNoPref</td> <td>Mem no prefetch</td> <td>POM1</td> </tr> <tr> <td>0x10000000</td> <td>pVmPref</td> <td>0x3.10000000</td> <td>pVmPref</td> <td>Mem w/ prefetch</td> <td>POM0</td> </tr> </tbody> </table>	size	CPU (virt)	PLB (real)	PCI	usage	map by	0x40000000	0x00000000	0x0.00000000	0x00000000	440GP slave	PIM0	0x10000000	pVmNoPref	0x3.00000000	pVmNoPref	Mem no prefetch	POM1	0x10000000	pVmPref	0x3.10000000	pVmPref	Mem w/ prefetch	POM0
size	CPU (virt)	PLB (real)	PCI	usage	map by																				
0x40000000	0x00000000	0x0.00000000	0x00000000	440GP slave	PIM0																				
0x10000000	pVmNoPref	0x3.00000000	pVmNoPref	Mem no prefetch	POM1																				
0x10000000	pVmPref	0x3.10000000	pVmPref	Mem w/ prefetch	POM0																				

0x00010000	0xD8000000	0x2.08000000	0x00000000	I/O	fixed
0x03800000	0xD8800000	0x2.08800000	0x00800000	"Extra" I/O	fixed

RETURNS OK, always

ERRNO Not Available

SEE ALSO [ppc440gpPci](#)

ppc440gpPciRegAddrShow()

NAME [ppc440gpPciRegAddrShow\(\)](#) – debug routine to print register addresses

SYNOPSIS

```
void * ppc440gpPciRegAddrShow
(
    struct vxbdDev * pDev,
    UINT32          regBaseIndex,
    UINT32          byteOffset,
    UINT8 *         pDataBuf,
    UINT32 *        flags
)
```

DESCRIPTION This routine prints the actual processor addresses used by the access methods to read and write device registers.

RETURNS N/A

ERRNO Not Available

SEE ALSO [ppc440gpPci](#)

ppc440gpPciRegister()

NAME [ppc440gpPciRegister\(\)](#) – register PowerPC 440GP host bridge

SYNOPSIS

```
void ppc440gpPciRegister(void)
```

DESCRIPTION This routine registers the PPC440GP host bridge with the vxBus subsystem.

RETURNS N/A

ERRNO Not Available

SEE ALSO **ppc440gpPci**

2

ppc860DevInit()

NAME **ppc860DevInit()** – initialize the SMC

SYNOPSIS

```
void ppc860DevInit
(
    PPC860SMC_CHAN *pChan
)
```

DESCRIPTION This routine is called to initialize the chip to a quiescent state. Note that the **smcNum** field of PPC860SMC_CHAN must be either 1 or 2.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **ppc860Sio**

ppc860Int()

NAME **ppc860Int()** – handle an SMC interrupt

SYNOPSIS

```
void ppc860Int
(
    PPC860SMC_CHAN *pChan
)
```

DESCRIPTION This routine is called to handle SMC interrupts.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **ppc860Sio**

ppcDecTimerDrvRegister()

NAME	ppcDecTimerDrvRegister() – register ppcDec timer driver
SYNOPSIS	<code>void ppcDecTimerDrvRegister(void)</code>
DESCRIPTION	This routine registers the ppcDec timer driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPpcDecTimer

qeAnd32()

NAME	qeAnd32() – Read then Write 32 bit register usign and mask
SYNOPSIS	<code>void qeAnd32 (void* handle, UINT32 *offset, UINT32 andMask)</code>
DESCRIPTION	none
RETURNS	NONE.
ERRNO	
SEE ALSO	vxbQeIntCtlr

qeIvecToInum()

NAME	qeIvecToInum() – get the relevant interrupt number
SYNOPSIS	<code>int qeIvecToInum</code>

```
(  
VOIDFUNC PTR * vector /* interrupt vector to attach to */  
)
```

DESCRIPTION	This routine finds out the interrupt number associated with the vector in <i>vector</i> . <i>vector</i> types are defined in h/drv/intrClt/vxbQeIntr.h .
RETURNS	the interrupt number for the vector
ERRNO	Not Available
SEE ALSO	vxbQeIntCtlr, vxbQeIntr.h

qeOr32()

NAME	qeOr32() – Read then Write 32 bit register using or mask
SYNOPSIS	<pre>void qeOr32 (void* handle, UINT32 *offset, UINT32 orMask)</pre>
DESCRIPTION	none
RETURNS	NONE.
ERRNO	
SEE ALSO	vxbQeIntCtlr

quiccAnd32()

NAME	quiccAnd32() – Read then Write 32 bit register usign and mask
SYNOPSIS	<pre>void quiccAnd32 (void* handle, UINT32 *offset,</pre>

```
    UINT32 andMask
)
```

DESCRIPTION none

RETURNS NONE.

ERRNO

SEE ALSO [vxbQuiccIntCtlr](#)

quiccIntrInit()

NAME **quiccIntrInit()** – initialize the interrupt manager for the PowerPC 83XX

SYNOPSIS STATUS vxbSysQuiccIntrInit
(
 VXB_DEVICE_ID pIntCtlr
)

DESCRIPTION This routine connects the default demultiplexer, **quiccIntrDeMux()**, to the external interrupt vector and associates all interrupt sources with the default interrupt handler. This routine is called by **sysHwInit()** in **sysLib.c**.

NOTE All interrupt from the SIU unit are enabled, CICR is setup so that SCC1 has the highest relative interrupt priority, through SCC4 with the lowest.

RETURNS OK always

ERRNO

SEE ALSO [vxbQuiccIntCtlr](#)

quiccIvecToInum()

NAME **quiccIvecToInum()** – get the relevant interrupt number

SYNOPSIS int quiccIvecToInum
(
 VOIDFUNC PTR * vector /* interrupt vector to attach to */
)

DESCRIPTION	This routine finds out the interrupt number associated with the vector in <i>vector</i> . <i>vector</i> types are defined in h/drv/intrClt/quiccIntr.h .
RETURNS	the interrupt number for the vector
ERRNO	
SEE ALSO	vxbQuiccIntCtlr , quiccIntr.h

quiccOr32()

NAME	quiccOr32() – Read then Write 32 bit register using or mask
SYNOPSIS	<pre>void quiccOr32 (void* handle, UINT32 *offset, UINT32 orMask)</pre>
DESCRIPTION	none
RETURNS	NONE.
ERRNO	
SEE ALSO	vxbQuiccIntCtlr

quiccTimerDrvRegister()

NAME	quiccTimerDrvRegister() – register coldFire timer driver
SYNOPSIS	<pre>void quiccTimerDrvRegister(void)</pre>
DESCRIPTION	This routine registers the quicc timer driver with the vxBus subsystem.
RETURNS	N/A

ERRNO

SEE ALSO [vxbPpcQuiccTimer](#)

rapidIoRegister()

NAME [**rapidIoRegister\(\)**](#) – register RapidIO bus type

SYNOPSIS STATUS rapidIoRegister(void)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxRapidIO](#)

rioConnect()

NAME [**rioConnect\(\)**](#) – connect RapidIO bus type to bus subsystem

SYNOPSIS STATUS rioConnect(void)

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxRapidIO](#)

rioDevMatch()

NAME [**rioDevMatch\(\)**](#) – check whether device and driver go together

SYNOPSIS	BOOL rioDevMatch (DRIVER_REGISTRATION * pVxbDriver, VXB_DEVICE_ID pDev)
DESCRIPTION	none
RETURNS	TRUE if the device and driver match up, FALSE otherwise
ERRNO	Not Available
SEE ALSO	vxbRapidIO

rioInit1()

NAME	rioInit1() – first-pass bus type initialization
SYNOPSIS	STATUS rioInit1 (VXB_DEVICE_ID pInst)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbRapidIO

rioInit2()

NAME	rioInit2() – second-pass bus type initialization
SYNOPSIS	STATUS rioInit2(void)
DESCRIPTION	none
RETURNS	N/A

ERRNO Not Available

SEE ALSO [vxbRapidIO](#)

rm9000x2glDevInit()

NAME [**rm9000x2glDevInit\(\)**](#) – intialize an NS16550 channel

SYNOPSIS

```
void rm9000x2glDevInit
(
    RM9000x2gl_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine initializes some **SIO_CHAN** function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialized all the device addresses, etc. in the RM9000x2gl_CHAN structure.

RETURNS N/A

ERRNO Not Available

SEE ALSO [**rm9000x2glSio**](#)

rm9000x2glInt()

NAME [**rm9000x2glInt\(\)**](#) – interrupt level processing

SYNOPSIS

```
void rm9000x2glInt
(
    RM9000x2gl_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status.

When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

ERRNO Not Available

SEE ALSO **rm9000x2glSio**

rm9000x2glIntEx()

NAME **rm9000x2glIntEx()** – miscellaneous interrupt processing

SYNOPSIS

```
void rm9000x2glIntEx
(
    RM9000x2gl_CHAN *pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles miscellaneous interrupts on the UART. Not implemented yet.

RETURNS N/A

ERRNO Not Available

SEE ALSO **rm9000x2glSio**

rm9000x2glIntMod()

NAME **rm9000x2glIntMod()** – interrupt level processing

SYNOPSIS

```
void rm9000x2glIntMod
(
    RM9000x2gl_CHAN * pChan,      /* pointer to channel */
    char             intStatus
)
```

DESCRIPTION This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status.

When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

ERRNO Not Available

SEE ALSO [rm9000x2glSio](#)

rm9000x2glIntRd()

NAME [**rm9000x2glIntRd\(\)**](#) – handle a receiver interrupt

SYNOPSIS

```
void rm9000x2glIntRd
(
    RM9000x2gl_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles read interrupts from the UART.

RETURNS N/A

ERRNO Not Available

SEE ALSO [rm9000x2glSio](#)

rm9000x2glIntWr()

NAME [**rm9000x2glIntWr\(\)**](#) – handle a transmitter interrupt

SYNOPSIS

```
void rm9000x2glIntWr
(
    RM9000x2gl_CHAN * pChan /* pointer to channel */
)
```

DESCRIPTION This routine handles write interrupts from the UART. It reads a character and puts it in the transmit holding register of the device for transfer.

If there are no more characters to transmit, transmission is disabled by clearing the transmit interrupt enable bit in the IER(int enable register).

RETURNS N/A

ERRNO Not Available

SEE ALSO [rm9000x2glSio](#)

sh7700TimerDrvRegister()

NAME	sh7700TimerDrvRegister() – register sh7700 timer driver
SYNOPSIS	void sh7700TimerDrvRegister(void)
DESCRIPTION	This routine registers the sh7700 timer driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbSh7700Timer

sh77xxPciRegister()

NAME	sh77xxPciRegister() – register sh77xxPci driver
SYNOPSIS	void sh77xxPciRegister(void)
DESCRIPTION	We use a two-function method here, to delay registration of the driver during debug. This should be left in place when the driver is released.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbSh77xxPci

sh77xxPcipDrvCtrlShow()

NAME	sh77xxPcipDrvCtrlShow() – show pDrvCtrl for sh77xxPci bus controller
SYNOPSIS	int sh77xxPcipDrvCtrlShow (VXB_DEVICE_ID pInst)

DESCRIPTION	none
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbsH77xxPci

shSciDevInit()

NAME	shSciDevInit() – initialize a on-chip serial communication interface
SYNOPSIS	<pre>void shSciDevInit (SCI_CHAN * pChan)</pre>
DESCRIPTION	This routine initializes the driver function pointers and then resets the chip in a quiescent state. The BSP must have already initialized all the device addresses and the baudFreq fields in the SCI_CHAN structure before passing it to this routine.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	shSciSio

shSciIntErr()

NAME	shSciIntErr() – handle a channel's error interrupt.
SYNOPSIS	<pre>void shSciIntErr (SCI_CHAN * pChan /* channel generating the interrupt */)</pre>
DESCRIPTION	none
RETURNS	N/A

ERRNO Not Available

SEE ALSO shSciSio

shSciIntRcv()

NAME shSciIntRcv() – handle a channel's receive-character interrupt.

SYNOPSIS

```
void shSciIntRcv
(
    SCI_CHAN * pChan /* channel generating the interrupt */
)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO shSciSio

shSciIntTx()

NAME shSciIntTx() – handle a channels transmitter-ready interrupt.

SYNOPSIS

```
void shSciIntTx
(
    SCI_CHAN * pChan /* channel generating the interrupt */
)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO shSciSio

shScifDevInit()

NAME	shScifDevInit() – initialize a on-chip serial communication interface
SYNOPSIS	void shScifDevInit (SCIF_CHAN * pChan)
DESCRIPTION	This routine initializes the driver function pointers and then resets the chip in a quiescent state. The BSP must have already initialized all the device addresses and the baudFreq fields in the SCIF_CHAN structure before passing it to this routine.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	shScifSio

shScifIntErr()

NAME	shScifIntErr() – handle a channel's error interrupt.
SYNOPSIS	void shScifIntErr (SCIF_CHAN * pChan /* channel generating the interrupt */)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	shScifSio

shScifIntRcv()

NAME	shScifIntRcv() – handle a channel's receive-character interrupt.
-------------	--

SYNOPSIS	void shScifIntRcv (SCIF_CHAN * pChan /* channel generating the interrupt */)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	shScifSio

shScifIntTx()

NAME	shScifIntTx() – handle a channels transmitter-ready interrupt.
SYNOPSIS	void shScifIntTx (SCIF_CHAN * pChan /* channel generating the interrupt */)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	shScifSio

shScifSioRegister()

NAME	shScifSioRegister() – register shScif driver
SYNOPSIS	void shScifSioRegister (void)
DESCRIPTION	This routine registers the shScif driver and device recognition data with the vxBus subsystem.

NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbShScifSio

sioNextChannelNumberAssign()

NAME	sioNextChannelNumberAssign() – assign a new serial channel number
SYNOPSIS	<code>int sioNextChannelNumberAssign(void)</code>
DESCRIPTION	This routine assigns the next serial channel number available.
RETURNS	assigned channel number.
ERRNO	
SEE ALSO	sioChanUtil

smEndLoad()

NAME	smEndLoad() – attach the SM interface to the MUX, initialize driver and device
SYNOPSIS	<code>END_OBJ * smEndLoad (char * pParamStr /* ptr to initialization parameter string */)</code>
DESCRIPTION	This routine attaches an SM Ethernet interface to the network MUX. This routine makes the interface available by allocating and filling in an END_OBJ structure, a driver entry table, and a MIB2 interface table. Calls to this routine evoke different results depending upon the parameter string it receives. If the string is empty, the MUX is requesting that the device name be returned, not an initialized END_OBJ pointer. If the string is not empty, a load operation is being requested with initialization being done with the parameters parsed from the string.

Upon successful completion of a load operation by this routine, the driver will be ready to be started, not active. The system will start the driver when it is ready to accept packets.

The shared memory region will be initialized, via **smPktSetup()**, during the call to this routine if it is executing on the designated master CPU. The **smEndLoad()** routine can be called to load only one device unit at a time.

Input parameters are specified in the form of an ASCII string of colon (:) delimited values of the form:

```
"unit:pAnchor:smAddr:memSize:tasType:  
maxCpus:masterCpu:localCpu:maxPktBytes:maxInputPkts:  
intType:intArg1:intArg2:intArg3:mbNum:cbNum:  
configFlg:pBootParams"
```

The *unit* parameter denotes the logical device unit number assigned by the operating system. Specified using radix 10.

The *pAnchor* parameter is the address of the SM anchor in the given *adrsSpace*. If *adrsSpace* is **SM_M_LOCAL**, this is the local virtual address on the SM master node by which the local CPU may access the shared memory anchor. Specified using radix 16.

The *smAddr* parameter specify the shared memory address; It could be in the master node, or in the off-board memory. The address is the local address of the master CPU. If *smAddr* is **NONE**, the driver may allocate a cache-safe memory region from the system memory in the master node as the shared memory region; and Currently, it is users' responsibility to make sure slave nodes can access this memory, and maintain atomic operations on this region.

The *memSize* parameter is the size, in bytes, of the shared memory region. Specified using radix 16.

The *tasType* parameter specifies the test-and-set operation to be used to obtain exclusive access to the shared data structures. It is preferable to use a genuine test-and-set instruction, if the hardware permits it. In this case, *tasType* should be **SM_TAS_HARD**. If any of the CPUs on the SM network do not support the test-and-set instruction, *tasType* should be **SM_TAS_SOFT**. Specified using radix 10.

The *maxCpus* parameter specifies the maximum number of CPUs that may use the shared memory region. Specified using radix 10.

The *masterCpu* parameter indicates the shared memory master CPU number. Specified in radix 10.

The *localCpu* parameter specifies this CPU's number in the SM subnet.

The *maxPktBytes* parameter specifies the size, in bytes, of the data buffer in shared memory packets. This is the largest amount of data that may be sent in a single packet. If this value is not an exact multiple of 4 bytes, it will be rounded up to the next multiple of 4. If zero, the default size specified in **DEFAULT_PKT_SIZE** is used. Specified using radix 10.

The *maxInputPkts* parameter specifies the maximum number of incoming shared memory packets which may be queued to this CPU at one time. If zero, the default value is used. Specified using radix 10.

The *intType* parameter allows a CPU to announce the method by which it is to be notified of input packets which have been queued to it. Specified using radix 10.

The *intArg1*, *intArg2*, and *intArg3* parameters are arguments chosen based on, and required by, the interrupt method specified. They are used to generate an interrupt of type *intType*. Specified using radix 16.

If *mbNum* is non-zero, it specifies the number of mBlks to allocate in the driver memory pool. If *mbNum* is less than 0x10, a default value is used. Specified using radix 16.

If *cbNum* is non-zero, it specifies the number of cBlks and, therefore, the number of clusters, to allocate in the driver memory pool. If *cbNum* is less than 0x10, a default value is used. Specified using radix 16.

The number of cBlks is also the number of clusters which will be allocated. The clusters allocated in the driver memory pool all have a size of *maxPktBytes* bytes.

The *configFlg* parameter indicate some configuration flags for smEnd. The flag includes, but not limited to, **SMEND_PROXY_SERVER**, **SMEND_PROXY_CLIENT**, **SMEND_PROXY_DEFAULT_ADDR**, and **SMEND_INCLUDE_SEQ_ADDR**.

The *pBootParams* parameter is the address of a **BOOT_PARAMS**. The smEnd will use this structure to get the backplane IP address, and/or anchor address.

RETURNS return values are dependent upon the context implied by the input parameter string length as shown below.

Length	Return Value
0	OK and device name copied to input string pointer or ERROR if NULL string pointer.
non-0	END_OBJ * to initialized object or NULL if bogus string or an internal error occurs.

ERRNO Not Available

SEE ALSO **smEnd**

smEndRegister()

NAME **smEndRegister()** – register smEnd driver

SYNOPSIS void smEndRegister(void)

DESCRIPTION	This routine registers the smEnd driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbSmEnd

smNetShow()

NAME	smNetShow() – show information about a shared memory network
SYNOPSIS	STATUS smNetShow (char * endName, /* shared memory device name (NULL = current) */ BOOL zero /* TRUE = zero the totals */)
DESCRIPTION	This routine displays information about the different CPUs configured in a shared memory network specified by <i>endName</i> . It prints error statistics and zeros these fields if <i>zero</i> is set to TRUE.
EXAMPLE	<pre>-> smNetShow Anchor Local Addr: 0x10000800, Hard TAS Sequential addressing enabled. Master IP address: 192.168.207.1 Local IP address: 192.168.207.2 heartbeat = 56, header at 0x1071d5b4, free pkts = 29. cpu int type arg1 arg2 arg3 queued pkts ----- ----- 0 mbox-4 0xd 0x807ffffc 0 0 1 poll 0xd 0xfb001000 0x80 0 PACKETS ERRORS Unicast Brdcast Input Output Input Output + Input Output ===== ====== ===== ===== + ===== ===== 3 2 0 2 0 0 value = 0 = 0x0</pre>
RETURNS	OK, or ERROR if there is a hardware setup problem or the routine cannot be initialized.

ERRNO Not Available

SEE ALSO **smEndShow**

smcFdc37b78xDevCreate()

NAME **smcFdc37b78xDevCreate()** – set correct IO port addresses for Super I/O chip

SYNOPSIS

```
VOID smcFdc37b78xDevCreate
(
    SMCFDC37B78X_IOPORTS *smcFdc37b78x_iop
)
```

DESCRIPTION This routine will initialize smcFdc37b78xIoPorts data structure. These ioports can either be changed on-the-fly or overriding SMCFDC37B78X_CONFIG_PORT, SMCFDC37B78X_INDEX_PORT and SMCFDC37B78X_DATA_PORT. This is a necessary step in intialization of superIO chip and logical devices embedded in it.

RETURNS NONE

ERRNO Not Available

SEE ALSO **smcFdc37b78x**

smcFdc37b78xInit()

NAME **smcFdc37b78xInit()** – initializes Super I/O chip Library

SYNOPSIS

```
VOID smcFdc37b78xInit
(
    int devInitMask
)
```

DESCRIPTION This routine will initialize serial, keyboard, floppy disk, parallel port and gpio pins as a part super i/o intialization

RETURNS NONE

ERRNO Not Available

SEE ALSO [smcFdc37b78x](#)

smcFdc37b78xKbdInit()

NAME [smcFdc37b78xKbdInit\(\)](#) – initializes the keyboard controller

SYNOPSIS

```
STATUS smcFdc37b78xKbdInit
(
    VOID
)
```

DESCRIPTION This routine will initialize keyboard controller

RETURNS OK/ERROR

ERRNO Not Available

SEE ALSO [smcFdc37b78x](#)

sramDevCreate()

NAME [sramDevCreate\(\)](#) – create a PCMCIA memory disk device

SYNOPSIS

```
BLK_DEV *sramDevCreate
(
    int sock,           /* socket no. */
    int bytesPerBlk,   /* number of bytes per block */
    int blksPerTrack,  /* number of blocks per track */
    int nBlocks,        /* number of blocks on this device */
    int blkOffset       /* no. of blks to skip at start of device */
)
```

DESCRIPTION This routine creates a PCMCIA memory disk device.

RETURNS A pointer to a block device structure (**BLK_DEV**), or **NULL** if memory cannot be allocated for the device structure.

ERRNO Not Available

SEE ALSO [sramDrv](#), [ramDevCreate\(\)](#)

sramDrv()

NAME	sramDrv() – install a PCMCIA SRAM memory driver
SYNOPSIS	STATUS sramDrv (int sock /* socket no. */)
DESCRIPTION	This routine initializes a PCMCIA SRAM memory driver. It must be called once, before any other routines in the driver.
RETURNS	OK, or ERROR if the I/O system cannot install the driver.
ERRNO	Not Available
SEE ALSO	sramDrv

sramMap()

NAME	sramMap() – map PCMCIA memory onto a specified ISA address space
SYNOPSIS	STATUS sramMap (int sock, /* socket no. */ int type, /* 0: common 1: attribute */ int start, /* ISA start address */ int stop, /* ISA stop address */ int offset, /* card offset address */ int extraws /* extra wait state */)
DESCRIPTION	This routine maps PCMCIA memory onto a specified ISA address space.
RETURNS	OK, or ERROR if the memory cannot be mapped.
ERRNO	Not Available
SEE ALSO	sramDrv

sym895CtrlCreate()

NAME

sym895CtrlCreate() – create a structure for a SYM895 device

SYNOPSIS

```
SYM895_SCSI_CTRL * sym895CtrlCreate
(
    UINT8 * siopBaseAdrs,      /* base address of the SCSI Controller */
    UINT    clkPeriod,        /* clock controller period (nsec*100) */
    UINT16 devType,          /* SCSI device type */
    UINT8 * siopRamBaseAdrs, /* on Chip Ram Address */
    UINT16 flags             /* options */
)
```

DESCRIPTION

This routine creates a SCSI Controller data structure and must be called before using a SCSI Controller chip. It should be called once and only once for a specified SCSI Controller. Since it allocates memory for a structure needed by all routines in **sym895Lib**, it must be called before any other routines in the library. After calling this routine, **sym895CtrlInit()** should be called at least once before any SCSI transactions are initiated using the SCSI Controller.

A Detailed description of parameters follows.

siopBaseAdrs

base address of the SCSI controller.

clkPeriod

clock controller period (nsec*100). This is used to determine the clock period for the SCSI core and affects the timing of both asynchronous and synchronous transfers.
Several Commonly used values are

SYM895_1667MHZ	6000	16.67Mhz chip
SYM895_20MHZ	5000	20Mhz chip
SYM895_25MHZ	4000	25Mhz chip
SYM895_3750MHZ	2667	37.50Mhz chip
SYM895_40MHZ	2500	40Mhz chip
SYM895_50MHZ	2000	50Mhz chip
SYM895_66MHZ	1515	66Mhz chip
SYM895_6666MHZ	1500	66Mhz chip
SYM895_75MHZ	1333	75Mhz chip
SYM895_80MHZ	1250	80Mhz chip
SYM895_160MHZ	625	40Mhz chip with Quadrupler

devType

SCSI sym8xx device type

siopRamBaseAdrs

base address of the internal scripts RAM

flags

various device/debug options for the driver. Commonly used values are

SYM895_ENABLE_PARITY_CHECK	0x01
SYM895_ENABLE_SINGLE_STEP	0x02

SYM895_COPY_SCRIPTS	0x04
---------------------	------

RETURNS	A pointer to SYM895_SCSI_CTRL structure, or NULL if memory is unavailable or there are invalid parameters.
ERRORS	N/A
SEE ALSO	sym895Lib

sym895CtrlInit()

NAME	<i>sym895CtrlInit()</i> – initialize a SCSI Controller Structure
-------------	--

SYNOPSIS	<pre>STATUS sym895CtrlInit (FAST SIOP * pSiop, /* pointer to SCSI Controller structure */ UINT scsiCtrlBusId /* SCSI bus ID of this SCSI Controller */)</pre>
-----------------	---

DESCRIPTION	This routine initializes an SCSI Controller structure, after the structure is created with <i>sym895CtrlCreate()</i> . This structure must be initialized before the SCSI Controller can be used. It may be called more than once if needed; however, it should only be called while there is no activity on the SCSI interface.
--------------------	---

A Detailed description of parameters follows.

pSiop

pointer to the SCSI controller structure created with ***sym895CtrlCreate()***

scsiCtrlBusId

SCSI Bus Id of the SIOP.

RETURNS	OK, or ERROR if parameters are out of range.
----------------	--

ERRORS	N/A
---------------	-----

SEE ALSO	sym895Lib
-----------------	----------------------------------

sym895GPIOConfig()

NAME	<i>sym895GPIOConfig()</i> – configures general purpose pins GPIO 0-4
-------------	---

SYNOPSIS	<pre>STATUS sym895GPIOConfig (SIOP * pSiop, /* pointer to SIOP structure */ UINT8 ioEnable, /* bits indicate input/output */ UINT8 mask /* mask for ioEnable parameter */)</pre>
DESCRIPTION	This routine uses the GPCNTL register to configure the general purpose pins available on Sym895 chip. Bits 0-4 of GPCNTL register map to GPIO 0-4 pins. A bit set in GPCNTL configures corresponding pin as input and a bit reset configures the pins as output.
<i>pSiop</i>	pointer to the SIOP structure.
<i>ioEnable</i>	bits 0-4 of this parameter configure GPIO 0-4 pins. 1 => input, 0 => output.
<i>mask</i>	bits 0-4 of this parameter identify valid bits in <i>ioEnable</i> parameter. Only those pins are configured, which have a corresponding bit set in this parameter.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	sym895Lib

sym895GPIOCtrl()

NAME	<code>sym895GPIOCtrl()</code> – controls general purpose pins GPIO 0-4
SYNOPSIS	<pre>STATUS sym895GPIOCtrl (SIOP * pSiop, /* pointer to SIOP structure */ UINT8 ioState, /* bits indicate set/reset */ UINT8 mask /* mask for ioState parameter */)</pre>
DESCRIPTION	This routine uses the GPREG register to set/reset of the general purpose pins available on Sym895 chip.
<i>pSiop</i>	pointer to the SIOP structure.
<i>ioState</i>	bits 0-4 of this parameter controls GPIO 0-4 pins. 1 => set, 0 => reset.

mask

bits 0-4 of this parameter identify valid bits in *ioState* parameter. Only those pins are activated, which have a corresponding bit set in this parameter.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **sym895Lib**

sym895Intr()

NAME **sym895Intr()** – interrupt service routine for the SCSI Controller

SYNOPSIS

```
void sym895Intr
(
    SIOP * pSiop /* pointer to the SIOP structure */
)
```

DESCRIPTION The first thing to determine is whether the device is generating an interrupt. If not, then this routine must exit as quickly as possible.

Find the event type corresponding to this interrupt, and carry out any actions which must be done before the SCSI Controller is re-started. Determine whether or not the SCSI Controller is connected to the bus (depending on the event type - see note below). If not, start a client script if possible or else just make the SCSI Controller wait for something else to happen.

The "connected" variable, at the end of switch statement, reflects the status of the currently executing thread. If it is TRUE it means that the thread is suspended and must be processed at the task level. Set the state of SIOP to IDLE and leave the control to the SCSI Manager. The SCSI Manager, in turn invokes the driver through a "resume" call.

Notify the SCSI manager of a controller event.

RETURNS N/A

RETURNS Not Available

ERRNO Not Available

SEE ALSO **sym895Lib**

sym895Loopback()

NAME	sym895Loopback() – this routine performs loopback diagnostics on 895 chip
SYNOPSIS	STATUS sym895Loopback (SIOP * pSiop /* pointer to SIOP controller structure */)
DESCRIPTION	Loopback mode allows 895 chip to control all signals, regardless of whether it is in initiator or target role. This mode insures proper SCRIPTS instruction fetches and data paths. SYM895 executes initiator instructions through the SCRIPTS, and this routine implements the target role by asserting and polling the appropriate SCSI signals in the SOCL, SODL, SBCL, and SBDL registers. To configure 895 in loopback mode, (1) Bits 3 and 4 of STEST2 should be set to put SCSI pins in High-Impedance mode, so that signals are not asserted on to the SCSI bus. (2) Bit 4 of DCNTL should be set to turn on single step mode. This allows the target program (this routine) to monitor when an initiator SCRIPTS instruction has completed. In this routine, SELECTION, MSG_OUT and DATA_OUT phases are checked. This ensures that data and control paths are proper.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	sym895Lib

sym895SetHwOptions()

NAME	sym895SetHwOptions() – sets the Sym895 chip Options
SYNOPSIS	STATUS sym895SetHwOptions (FAST SIOP * pSiop, /* pointer to the SIOP structure */ SYM895_HW_OPTIONS * pHwOptions /* pointer to the Options Structure */)

DESCRIPTION This function sets optional bits required for tweaking the performance of 895 to the Ultra2 SCSI. The routine should be called with SYM895_HW_OPTIONS structure as defined in **sym895.h** file.

The input parameters are

pSiop

pointer to the SIOP structure

pHwOptions

pointer to the a SYM895_HW_OPTIONS structure.

```
struct sym895HWOptions
{
    int    SCLK    : 1;      /* STEST1:b7,if false, uses PCI Clock for SCSI */
    int    SCE     : 1;      /* STEST2:b7, enable assertion of SCSI thro SOCL*/
                           /* and SODL registers */
    int    DIF     : 1;      /* STEST2:b5, enable differential SCSI */
    int    AWS     : 1;      /* STEST2:b2, Always Wide SCSI */
    int    EWS     : 1;      /* SCNTL3:b3, Enable Wide SCSI */
    int    EXT     : 1;      /* STEST2:b1, Extend SREQ/SACK filtering */
    int    TE      : 1;      /* STEST3:b7, TolerANT Enable */
    int    BL      : 3;      /* DMODE:b7,b6, CTEST5:b2 : Burst length
                           /* when set to any of 32/64/128 burst length
                           /* transfers, requires the DMA Fifo size to be
                           /* 816 bytes (ctest5:b5 = 1).
    int    SIOM    : 1;      /* DMODE:b5, Source I/O Memory Enable */
    int    DIOM    : 1;      /* DMODE:b4, Destination I/O Memory Enable */
    int    EXC     : 1;      /* SCNTL1:b7, Slow Cable Mode */
    int    ULTRA   : 1;      /* SCNTL3:b7, Ultra Enable */
    int    DFS     : 1;      /* CTEST5:b5, DMA Fifo size 112/816 bytes
} SYM895_HW_OPTIONS;
```

This routine should not be called when there is SCSI Bus Activity as this modifies the SIOP Registers.

RETURNS OK or ERROR if any of the input parameters is not valid.

ERRORS N/A

SEE ALSO [sym895Lib](#), [sym895.h](#), [sym895CtrlCreate\(\)](#)

sym895Show()

NAME ***sym895Show()*** – display values of all readable SYM 53C8xx SIOP registers

SYNOPSIS STATUS sym895Show

```

(
    SIOP * pSiop /* pointer to SCSI controller */
)

```

DESCRIPTION This routine displays the state of the SIOP registers in a user-friendly way. It is useful primarily for debugging. The input parameter is the pointer to the SIOP information structure returned by the **sym895CtrlCreate()** call.

NOTE The only readable register during a script execution is the Istat register. If you use this routine during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE

```

-> sym895Show
SYM895 Registers
-----
Scntl0 = 0xd0 Scntl1 = 0x00 Scntl2 = 0x00 Scntl3 = 0x00
Scid = 0x67 Sxfer = 0x00 Sdid = 0x00 Gpreg = 0x0f
Sfbr = 0x0f Socl = 0x00 Ssid = 0x00 Sbcl = 0x00
Dstat = 0x80 Sstat0 = 0x00 Sstat1 = 0x0f Sstat2 = 0x02
Dsa = 0x07ea9538
Istat = 0x00
Ctest0 = 0x00 Ctest1 = 0xf0 Ctest2 = 0x35 Ctest3 = 0x10
Temp = 0x001d0c54
Dfifo = 0x00
Dbc0:23-Dcmd24:31 = 0x54000000
Dnad = 0x001d0c5c
Dsp = 0x001d0c5c
Dsps = 0x000000a0
Scratch0 = 0x01 Scratch1 = 0x00 Scratch2 = 0x00 Scratch3 = 0x00
Dmode = 0x81 Dien = 0x35 Dwt = 0x00 Dcntl = 0x01
Sien0 = 0x0f Sien1 = 0x17 Sist0 = 0x00 Sist1 = 0x00
Slpar = 0x4c Swide = 0x00 Macntl = 0xd0 Gpcntl = 0x0f
Stime0 = 0x00 Stime1 = 0x00 Respid0 = 0x80 Respid1 = 0x00
Stest0 = 0x07 Stest1 = 0x00 Stest2 = 0x00 Stest3 = 0x80
Sid1 = 0x0000 Sodl = 0x0000 Sbd1 = 0x0000
Scratchb = 0x00000200
value = 0 = 0x0

```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

ERRNO Not Available

SEE ALSO **sym895Lib**, **sym895CtrlCreate()**

sysAuxClkConnect()

NAME **sysAuxClkConnect()** – connect a routine to the auxiliary clock interrupt

SYNOPSIS	<pre>STATUS sysAuxClkConnect (FUNCPTR routine, /* routine called at each aux clock interrupt */ int arg /* argument to auxiliary clock interrupt routine */)</pre>
DESCRIPTION	This routine later specifies the interrupt service routine to be called at each auxiliary clock interrupt. It does not enable auxiliary clock interrupts.
RETURNS	OK , or ERROR if the routine cannot be connected to the interrupt.
ERRNO	Not Available
SEE ALSO	vxbAuxClkLib , sysAuxClkEnable()

sysAuxClkDisable()

NAME	sysAuxClkDisable() – turn off auxiliary clock interrupts
SYNOPSIS	<pre>void sysAuxClkDisable (void)</pre>
DESCRIPTION	This routine disables auxiliary clock interrupts.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbAuxClkLib , sysAuxClkEnable()

sysAuxClkEnable()

NAME	sysAuxClkEnable() – turn on auxiliary clock interrupts
SYNOPSIS	<pre>void sysAuxClkEnable (void)</pre>
DESCRIPTION	This routine enables auxiliary clock interrupts.
RETURNS	N/A
ERRNO	Not Available

SEE ALSO vxbAuxClkLib, sysAuxClkConnect(), sysAuxClkDisable(), sysAuxClkRateSet()

sysAuxClkHandleGet()

NAME sysAuxClkHandleGet() – get the timer handle for auxiliary clock

SYNOPSIS timerHandle_t sysAuxClkHandleGet (void)

DESCRIPTION This routine returns the timer handle for the auxiliary clock.

RETURNS auxiliary clock timer handle.

ERRNO Not Available

SEE ALSO vxbAuxClkLib

sysAuxClkRateGet()

NAME sysAuxClkRateGet() – get the auxiliary clock rate

SYNOPSIS int sysAuxClkRateGet (void)

DESCRIPTION This routine returns the interrupt rate of the auxiliary clock.

RETURNS The number of ticks per second of the auxiliary clock.

ERRNO Not Available

SEE ALSO vxbAuxClkLib, sysAuxClkEnable(), sysAuxClkRateSet()

sysAuxClkRateSet()

NAME sysAuxClkRateSet() – set the auxiliary clock rate

SYNOPSIS STATUS sysAuxClkRateSet

```
(  
    int ticksPerSecond /* number of clock interrupts per second */  
)
```

DESCRIPTION	This routine sets the interrupt rate of the auxiliary clock. It does not enable auxiliary clock interrupts.
RETURNS	OK, or ERROR if the tick rate is invalid or the timer cannot be set.
ERRNO	Not Available
SEE ALSO	vxbAuxClkLib , sysAuxClkEnable() , sysAuxClkRateGet()

sysBusIntAck()

NAME	sysBusIntAck() – acknowledge an interrupt
SYNOPSIS	<pre>int sysBusIntAck (int intLevel /* interrupt level to acknowledge*/)</pre>
DESCRIPTION	This routine calls a hook or helper function to acknowledge interrupt.
RETURNS	OK
ERRNO	
SEE ALSO	vxbSmSupport

sysBusIntAckHelper()

NAME	sysBusIntAckHelper() – locate the correct call to acknowledge an interrupt
SYNOPSIS	<pre>void sysBusIntAckHelper (VXB_DEVICE_ID pInst, int intLevel)</pre>

DESCRIPTION	This routine iterates through all devices looking for the correct call to acknowledge an interrupt.
RETURNS	NONE
ERRNO	
SEE ALSO	vxbSmSupport

sysBusIntGen()

NAME	sysBusIntGen() – call interrupt generator function hook
SYNOPSIS	<pre>STATUS sysBusIntGen (int level, /* interrupt level to generate */ int vector /* interrupt vector for interrupt*/)</pre>
DESCRIPTION	This routine either calls interrupt generator function hook directly or calls VxBus helper to find the correct device and then calls the interrupt generator function hook.
RETURNS	STATUS
ERRNO	
SEE ALSO	vxbSmSupport

sysBusIntGenHelper()

NAME	sysBusIntGenHelper() – locate the correct call to generate an interrupt
SYNOPSIS	<pre>void sysBusIntGenHelper (VXB_DEVICE_ID pInst, void * pArgs)</pre>
DESCRIPTION	This routine iterates through all devices looking for the correct call to generate the interrupt.
RETURNS	NONE

ERRNO

SEE ALSO [vxbsmSupport](#)

sysBusTas()

NAME [sysBusTas\(\)](#) – establish a reservation for a particular address

SYNOPSIS

```
BOOL sysBusTas
(
    char * adrs
)
```

DESCRIPTION This routine establishes a reservation for a particular address. If no direct hook is available, it calls the VxBus helper function to find the correct call.

RETURNS NONE

ERRNO

SEE ALSO [vxbsmSupport](#)

sysBusTasClear()

NAME [sysBusTasClear\(\)](#) – clear a reservation for a particular address

SYNOPSIS

```
void sysBusTasClear
(
    volatile char * adrs
)
```

DESCRIPTION This routine clears a reservation for a particular address. If no direct hook is available, it calls the VxBus helper function to find the correct call.

RETURNS NONE

ERRNO

SEE ALSO [vxbsmSupport](#)

sysBusTasClearHelper()

NAME	sysBusTasClearHelper() – locate and clear a reservation for a particular address
SYNOPSIS	<pre>void sysBusTasClearHelper (VXB_DEVICE_ID pInst, void * adrs)</pre>
DESCRIPTION	This routine clears a reservation for a particular address by calling all devices which support this functionality.
RETURNS	NONE
ERRNO	
SEE ALSO	vxbSmSupport

sysBusTasHelper()

NAME	sysBusTasHelper() – locate the correct device to do TAS
SYNOPSIS	<pre>void sysBusTasHelper (VXB_DEVICE_ID pInst, UINT32* tasArgs)</pre>
DESCRIPTION	This routine iterates through all VxBus devices to find the correct device to do TAS.
RETURNS	TRUE/FALSE via tasArgs
ERRNO	
SEE ALSO	vxbSmSupport

sysClkConnect()

NAME	sysClkConnect() – connect a routine to the system clock interrupt
-------------	---

SYNOPSIS	<pre>STATUS sysClkConnect (FUNCPTR routine, /* routine called at each system clock interrupt */ int arg /* argument with which to call routine */)</pre>
DESCRIPTION	This routine specifies the interrupt service routine to be called at each clock interrupt. Normally, it is called from usrRoot() in usrConfig.c to connect usrClock() to the system clock interrupt.
RETURN	OK , or ERROR if the routine cannot be connected to the interrupt.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbSysClkLib , sysClkEnable()

sysClkDisable()

NAME	sysClkDisable() – turn off system clock interrupts
SYNOPSIS	<pre>void sysClkDisable (void)</pre>
DESCRIPTION	This routine disables system clock interrupts.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbSysClkLib , sysClkEnable()

sysClkEnable()

NAME	sysClkEnable() – turn on system clock interrupts
SYNOPSIS	<pre>void sysClkEnable (void)</pre>
DESCRIPTION	This routine enables system clock interrupts.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxbSysClkLib](#), [sysClkConnect\(\)](#), [sysClkDisable\(\)](#), [sysClkRateSet\(\)](#)

sysClkHandleGet()

NAME [sysClkHandleGet\(\)](#) – get the timer handle for system clock

SYNOPSIS `timerHandle_t sysClkHandleGet (void)`

DESCRIPTION This routine returns the timer handle for the system clock.

RETURNS system clock timer handle.

ERRNO Not Available

SEE ALSO [vxbSysClkLib](#)

sysClkRateGet()

NAME [sysClkRateGet\(\)](#) – get the system clock rate

SYNOPSIS `int sysClkRateGet (void)`

DESCRIPTION This routine returns the system clock rate.

RETURNS The number of ticks per second of the system clock.

ERRNO Not Available

SEE ALSO [vxbSysClkLib](#), [sysClkEnable](#), [sysClkRateSet\(\)](#)

sysClkRateSet()

NAME	sysClkRateSet() – set the system clock rate
SYNOPSIS	STATUS sysClkRateSet (int ticksPerSecond /* number of clock interrupts per second */)
DESCRIPTION	This routine sets the interrupt rate of the system clock. It is called by usrRoot() in usrConfig.c .
RETURNS	OK, or ERROR if the tick rate is invalid or the timer cannot be set.
ERRNO	Not Available
SEE ALSO	vxbSysClkLib , sysClkEnable , sysClkRateGet()

sysMailboxConnect()

NAME	sysMailboxConnect() – TBD
SYNOPSIS	STATUS sysMailboxConnect (FUNCPTR isr, int level)
DESCRIPTION	none
RETURNS	STATUS
ERRNO	
SEE ALSO	vxbSmSupport

sysMailboxEnable()

NAME	sysMailboxEnable() – TBD
-------------	----------------------------------

SYNOPSIS STATUS sysMailboxEnable
 (
 char * level
)

DESCRIPTION none

RETURNS STATUS

ERRNO

SEE ALSO vxbSmSupport

sysPciHostBridgeInit()

NAME sysPciHostBridgeInit() – Initialize the PCI Host Bridge

SYNOPSIS STATUS sysPciHostBridgeInit
 (
 VXB_DEVICE_ID pDev
)

DESCRIPTION Initializes the PCI bridge so it can operate as both a PCI master and slave.

Parameters set are:

CPU->PCI (master/initiator/outbound) address translation
PCI->CPU (slave/target/inbound) address translation

RETURNS OK, always

ERRNO

SEE ALSO mcf5475Pci

sysPciHostBridgeInit()

NAME sysPciHostBridgeInit() – initialize the PCI Host Bridge

SYNOPSIS STATUS sysPciHostBridgeInit (void)

DESCRIPTION	This routine initializes the PCI Host Bridge.
RETURNS	OK, always
ERRNO	
SEE ALSO	pentiumPci

sysPciHostBridgeInit()

NAME	sysPciHostBridgeInit() – Initialize the PCI-X Host Bridge
SYNOPSIS	<code>STATUS sysPciHostBridgeInit(void)</code>
DESCRIPTION	Initializes the PCI bridge so it can operate as both a PCI master and slave. Parameters set are: <ul style="list-style-type: none">- CPU->PCI (master/initiator/outbound) address translation- PCI->CPU (slave/target/inbound) address translation
RETURNS	OK, always
ERRNO	Not Available
SEE ALSO	ppc440gpPci

sysSerialChanConnect()

NAME	sysSerialChanConnect() – connect the SIO_CHAN device
SYNOPSIS	<code>STATUS sysSerialChanConnect (int sioChan)</code>
DESCRIPTION	This routine connects the specified serial channel to the I/O system. This routine first checks for BSP-supplied serial channels. For BSP-supplied serial channels, this routine does nothing. If the specified channel number is not supplied by the BSP, then this routine runs the connect method on the appropriate serial device instance.

RETURNS OK if the specified channel can be connected, or **ERROR**

ERRNO

SEE ALSO **sioChanUtil**

sysSerialChanGet()

NAME **sysSerialChanGet()** – get the **SIO_CHAN** device associated with a serial channel

SYNOPSIS

```
SIO_CHAN * sysSerialChanGet
(
    int sioChan
)
```

DESCRIPTION This routine returns a pointer to the **SIO_CHAN** device associated with a specified serial channel. It is called by **usrRoot()** to obtain pointers when creating the system serial devices, **/tyCo/x**. It is also used by the WDB agent to locate its serial channel.

This routine first checks for BSP-supplied serial channels, then queries the bus subsystem for a serial channel matching the specified channel number.

RETURNS pointer to the **SIO_CHAN** structure for the channel, or **ERROR** if an invalid channel

ERRNO

SEE ALSO **sioChanUtil**

sysSerialConnectAll()

NAME **sysSerialConnectAll()** – connect all **SIO_CHAN** devices

SYNOPSIS

```
void sysSerialConnectAll(void)
```

DESCRIPTION This routine connects all serial channel to the I/O system.

This routine first checks for BSP-supplied serial channels. For BSP-supplied serial channels, this routine does nothing. If the specified channel number is not supplied by the BSP, then this routine runs the connect method for all serial device instances.

RETURNS N/A

ERRNO

SEE ALSO **sioChanUtil**

sysSmEndLoad()

NAME **sysSmEndLoad()** – load the shared memory END driver

SYNOPSIS

```
END_OBJ * sysSmEndLoad
(
    char * pParamStr, /* ptr to the initialization parameter string */
    void * unused     /* unused argument */
)
```

DESCRIPTION This function packs the loadstring, and calls the smEndLoad to load the the shared memory END driver

RETURN pointer to the END object or NULL

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxbSmEnd**, **smEndLoad()**

sysTimestamp()

NAME **sysTimestamp()** – get the timestamp timer tick count

SYNOPSIS **UINT32 sysTimestamp (void)**

DESCRIPTION This routine returns the current value of the timestamp timer tick counter. The tick count can be converted to seconds by dividing by the return of **sysTimestampFreq()**.

This routine should be called with interrupts locked. If interrupts are not already locked, **sysTimestampLock()** should be used instead.

RETURNS The current timestamp timer tick count.

ERRNO Not Available

SEE ALSO vxbTimestampLib, sysTimestampLock()

sysTimestampConnect()

NAME sysTimestampConnect() – connect a user routine to the timestamp timer interrupt

SYNOPSIS

```
STATUS sysTimestampConnect
(
    FUNCPTR routine, /* routine called at each timestamp timer interrupt */
    int      arg      /* argument with which to call routine */
)
```

DESCRIPTION This routine specifies the user interrupt routine to be called at each timestamp timer interrupt. It does not enable the timestamp timer itself. If the timestamp timer is same as the system clock the ISR should not be replaced Hence returns **ERROR**.

RETURNS OK, or **ERROR** if system clock is used.

ERRNO Not Available

SEE ALSO vxbTimestampLib

sysTimestampDisable()

NAME sysTimestampDisable() – disable the timestamp timer

SYNOPSIS

```
STATUS sysTimestampDisable (void)
```

DESCRIPTION This routine disables the timestamp timer. Interrupts are not disabled, although the tick counter will not increment after the timestamp timer is disabled, thus interrupts will no longer be generated.

RETURNS OK, or **ERROR** if the timestamp timer cannot be disabled.

ERRNO Not Available

SEE ALSO vxbTimestampLib

sysTimestampEnable()

NAME	sysTimestampEnable() – initialize and enable the timestamp timer
SYNOPSIS	<code>STATUS sysTimestampEnable (void)</code>
DESCRIPTION	This routine connects the timestamp timer interrupt and initializes the counter registers. If the timestamp timer is already running, this routine merely resets the timer counter. The rate of the timestamp timer should be set explicitly within the BSP, in the sysHwInit() routine. This routine does not intialize the timer rate.
RETURNS	OK, or ERROR if the timestamp timer cannot be enabled.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

sysTimestampFreq()

NAME	sysTimestampFreq() – get the timestamp timer clock frequency
SYNOPSIS	<code>UINT32 sysTimestampFreq (void)</code>
DESCRIPTION	This routine returns the frequency of the timer clock, in ticks per second. The rate of the timestamp timer should be set explicitly within the BSP, in the sysHwInit() routine.
RETURNS	The timestamp timer clock frequency, in ticks per second.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

sysTimestampHandleGet()

NAME	sysTimestampHandleGet() – get the timer handle for timestamp timer
SYNOPSIS	<code>timerHandle_t sysTimestampHandleGet (void)</code>

DESCRIPTION	This routine returns the timer handle for the timestamp timer.
RETURNS	timestamp timer handle.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

sysTimestampLock()

NAME	sysTimestampLock() – get the timestamp timer tick count
SYNOPSIS	<code>UINT32 sysTimestampLock (void)</code>
DESCRIPTION	This routine returns the current value of the timestamp timer tick counter. The tick count can be converted to seconds by dividing by the return of sysTimestampFreq() . This routine locks interrupts for cases where it is necessary to stop the tick counter in order to read it, or when two independent counters must be read. If interrupts are already locked, sysTimestamp() should be used instead.
RETURNS	The current timestamp timer tick count.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib , sysTimestamp()

sysTimestampPeriod()

NAME	sysTimestampPeriod() – get the timestamp timer period
SYNOPSIS	<code>UINT32 sysTimestampPeriod (void)</code>
DESCRIPTION	This routine returns the period of the timestamp timer in ticks. The period, or terminal count, is the number of ticks to which the timestamp timer will count before rolling over and restarting the counting process.
RETURNS	The period of the timestamp timer in counter ticks.

ERRNO	Not Available
SEE ALSO	vxbTimestampLib

tcicInit()

NAME	tcicInit() – initialize the TCIC chip
SYNOPSIS	<pre>STATUS tcicInit (int ioBase, /* IO base address */ int intVec, /* interrupt vector */ int intLevel, /* interrupt level */ FUNCPTR showRtn /* show routine */)</pre>
DESCRIPTION	This routine initializes the TCIC chip.
RETURNS	OK, or ERROR if the TCIC chip cannot be found.
ERRNO	Not Available
SEE ALSO	tcic

tcicShow()

NAME	tcicShow() – show all configurations of the TCIC chip
SYNOPSIS	<pre>void tcicShow (int sock /* socket no. */)</pre>
DESCRIPTION	This routine shows all configurations of the TCIC chip.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	tcicShow

tffsBootImagePut()

NAME

tffsBootImagePut() – write to the boot-image region of the flash device

SYNOPSIS

```
STATUS tffsBootImagePut
(
    int     driveNo, /* TFFS drive number */
    int     offset,  /* offset in the flash chip/card */
    char * filename /* binary format of the bootimage */
)
```

DESCRIPTION

This routine writes an input stream to the boot-image region (if any) of a flash memory device. Typically, the input stream contains a boot image, such as the VxWorks boot image, but you are free to use this function to write any data needed. The size of the boot-image region is set by the **tffsDevFormat()** call (or the **sysTffsFormat()** call, a BSP-specific helper function that calls **tffsDevFormat()** internally) that formats the flash device for use with TrueFFS.

If **tffsBootImagePut()** is used to put a VxWorks boot image in flash, you should not use the s-record version of the boot image typically produced by make. Instead, you should take the pre s-record version (usually called **bootrom** instead of **bootrom.hex**), and filter out its loader header information using an **xxxToBin** utility. For example:

```
elfToBin < bootrom > bootrom.bin
```

Use the resulting **bootrom.bin** as input to **tffsBootImagePut()**.

The discussion above assumes that you want only to use the flash device to store a VxWorks image that is retrieved from the flash device and then run out of RAM. However, because it is possible to map many flash devices directly into the target's memory, it is also possible run the VxWorks image from flash memory, although there are some restrictions:

- The flash device must be non-NAND.
- Only the text segment of the VxWorks image (**vxWorks.res_rom**) may run out of flash memory. The data segment of the image must reside in standard RAM.
- No part of the flash device may be erased while the VxWorks image is running from flash memory.

Because TrueFFS garbage collection triggers an erase, this last restriction means that you cannot run a VxWorks boot image out of a flash device that must also support a writable file system (although a read-only file system is OK).

This last restriction arises from the way in which flash devices are constructed. The current physical construction of flash memory devices does not allow access to the device while an erase is in progress anywhere on the flash device. As a result, if TrueFFS tries to erase a portion of the flash device, the entire device becomes inaccessible to all other users. If that other user happens to be the VxWorks image looking for its next instruction, the VxWorks image crashes.

RETURNS OK or ERROR

ERRNO Not Available

SEE ALSO [tffsConfig](#)

tffsShow()

NAME **tffsShow()** – show device information on a specific socket interface

SYNOPSIS

```
void tffsShow
    (
        int driveNo /* TFFS drive number */
    )
```

DESCRIPTION This routine prints device information on the specified socket interface. This information is particularly useful when trying to determine the number of Erase Units required to contain a boot image. The field called unitSize reports the size of an Erase Unit.

If the process of getting physical information fails, an error code is printed. The error codes can be found in **flbase.h**.

RETURNS N/A

ERRNO Not Available

SEE ALSO [tffsConfig](#)

tffsShowAll()

NAME **tffsShowAll()** – show device information on all socket interfaces

SYNOPSIS

```
void tffsShowAll (void)
```

DESCRIPTION This routine prints device information on all socket interfaces.

RETURNS N/A

ERRNO Not Available

SEE ALSO

tffsConfig

vgaInit()

NAME

vgaInit() – initializes the VGA chip and loads font in memory.

SYNOPSIS

```
STATUS vgaInit
(
    void
)
```

DESCRIPTION

This routine will initialize the VGA specific register set to bring a VGA card in VGA 3+ mode and loads the font in plane 2.

RETURNS

OK/ERROR

ERRNO

Not Available

SEE ALSO

vgaInit

vxBusShow()

NAME

vxBusShow() – show vxBus subsystem

SYNOPSIS

```
void vxBusShow
(
    int verboseLevel
)
```

DESCRIPTION

This routine shows the vxBus configuration.

While this routine is being executed, no elements must be removed from the lists by any other thread. If any element is added by another thread, the new element may or may not be displayed.

RETURNS

N/A

ERRNO

SEE ALSO

vxbShow

vxbAccessMethodGet()

NAME vxbAccessMethodGet() – find specific method for accessing device

SYNOPSIS FUNCPTR vxbAccessMethodGet
(
 struct vxbDev * pDev, /* Device Information */
 UINT32 accessType /* Access method to find */
)

DESCRIPTION This routine finds the specified method for accessing the specified device.

RETURNS pointer to method, or NULL

ERRNO

SEE ALSO vxBus

vxbAuxClkLibInit()

NAME vxbAuxClkLibInit() – initialize the auxiliary clock library

SYNOPSIS STATUS vxbAuxClkLibInit (void)

DESCRIPTION This routine initializes the auxiliary clock library by selecting the timer which is best suited for use as auxiliary clock from the timers available in the system.

RETURNS OK or ERROR.

ERRNO Not Available

SEE ALSO vxbAuxClkLib

vxbAuxClkShow()

NAME vxbAuxClkShow() – show the auxiliary clock information

SYNOPSIS void vxbAuxClkShow (void)

DESCRIPTION	This routine is used to display the auxiliary clock details
RETURNS	None.
ERRNO	Not Available
SEE ALSO	vxbAuxClkLib

vxbBusAnnounce()

NAME	vxbBusAnnounce() – announce bus discovery to bus subsystem
SYNOPSIS	STATUS vxbBusAnnounce (struct vxbDev * pBusDev, /* bus controller */ UINT32 busID /* bus type */)
DESCRIPTION	This routine is called by bus controller drivers, whenever a new bus is found. The pBusDev->pParentBus must be populated before calling this function. Otherwise, the bus shows up as a lost bus when vxBusShow() is called
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	vxBus

vxbBusListPrint()

NAME	vxbBusListPrint() – Show bus topology
SYNOPSIS	int vxbBusListPrint (struct vxbBusPresent * pBusPres)
DESCRIPTION	This routine prints information about the bus hierarchy.
RETURNS	N/A

ERRNO

SEE ALSO [vxbShow](#)

vxbBusTypeRegister()

NAME **vxbBusTypeRegister()** – register a bus type

SYNOPSIS STATUS vxbBusTypeRegister
 (
 struct vxbBusTypeInfo * pBusType
)

DESCRIPTION This routine registers a bus type with the vxBus subsystem.

RETURNS OK if successfully registered, else ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbBusTypeString()

NAME **vxbBusTypeString()** – retrieve bus type string

SYNOPSIS char * vxbBusTypeString
 (
 int busType
)

DESCRIPTION This routine returns the bus type string based on the bus type integer value

RETURNS pointer to char

ERRNO

SEE ALSO [vxBus](#)

vxbBusTypeUnregister()

NAME **vxbBusTypeUnregister()** – unregister a bus type

SYNOPSIS **STATUS vxbBusTypeUnregister(struct vxbBusTypeInfo * pBusType)**

DESCRIPTION This routine unregisters a bus type with the vxBus subsystem.

RETURNS OK is success, else ERROR

ERRNO

SEE ALSO **vxBus**

vxbCn3xxxTimerDrvRegister()

NAME **vxbCn3xxxTimerDrvRegister()** – register cn3xxx timer driver

SYNOPSIS **void vxbCn3xxxTimerDrvRegister (void)**

DESCRIPTION This routine registers the cn3xxx timer driver with the vxBus subsystem.

RETURNS N/A

ERRNO none

SEE ALSO **vxbCn3xxxTimer**

vxbDelay()

NAME **vxbDelay()** – delay for a moment

SYNOPSIS **void vxbDelay (void)**

DESCRIPTION This routine introduces a busy wait of 1 ms.

RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	vxbDelayLib

vxbDelayLibInit()

NAME	vxbDelayLibInit() – initialize the delay library
SYNOPSIS	<code>STATUS vxbDelayLibInit (void)</code>
DESCRIPTION	This routine initializes the delay library by selecting the timers which are best suited for use as delay timers from the timers available in the system.
RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	vxbDelayLib

vxbDelayTimersShow()

NAME	vxbDelayTimersShow() – show the delay timers information
SYNOPSIS	<code>void vxbDelayTimerShow (void)</code>
DESCRIPTION	This routine is used to display the delay timer details
RETURNS	None.
ERRNO	Not Available
SEE ALSO	vxbDelayLib

vxbDevAccessShow()

NAME **vxbDevAccessShow()** – Show bus access methods

SYNOPSIS

```
void vxbDevAccessShow
(
    struct vxbDev * pDev
)
```

DESCRIPTION This routine prints the access methods for the specified device.

RETURNS N/A

ERRNO

SEE ALSO **vxbShow**

vxbDevConnect()

NAME **vxbDevConnect()** – HWIF Post-Kernel Connection

SYNOPSIS STATUS vxbDevConnect (void)

DESCRIPTION This function performs the hardware interface post kernel device connection.

RETURNS OK or ERROR

ERRNO None

SEE ALSO **cmdLineBuild**

vxbDevConnectInternal()

NAME **vxbDevConnectInternal()** – third-pass initialization of devices

SYNOPSIS STATUS vxbDevConnectInternal (void)

DESCRIPTION Note: This is called from the **vxbUsrCmdLine.c** in command line builds

This routine executes the third stage of device initialization.

RETURNS OK, always

ERRNO

SEE ALSO **vxBus**

vxbDevError()

NAME **vxbDevError()** – driver does not support specified functionality

SYNOPSIS STATUS vxbDevError(void)

DESCRIPTION This routine provides an error if the device driver does not support the specified functionality.

RETURNS ERROR, always

ERRNO

SEE ALSO **vxBus**

vxbDevInit()

NAME **vxbDevInit()** – HWIF Post-Kernel Init

SYNOPSIS STATUS vxbDevInit (void)

DESCRIPTION This function performs the hardware interface post kernel initialization

RETURNS OK or ERROR

ERRNO None

SEE ALSO **cmdLineBuild**

vxbDevInitInternal()

NAME	vxbDevInitInternal() – second-pass initialization of devices
SYNOPSIS	STATUS vxbDevInitInternal (void)
DESCRIPTION	This routine executes the second stage of device initialization. Note: This is called from the vxbUsrCmdLine.c in command line builds
RETURNS	OK, always
ERRNO	
SEE ALSO	vxBus

vxbDevIterate()

NAME	vxbDevIterate() – perform specified action for each device
SYNOPSIS	STATUS vxbDevIterate (FUNCPTR func, /* function to call */ void * pArg, /* 2nd argument to func */ UINT32 flags /* flags to determine what to do */)
DESCRIPTION	This routine performs the specified action for each device.
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	vxBus

vxbDevMethodGet()

NAME	vxbDevMethodGet() – find entry point of method
SYNOPSIS	FUNCPTR vxbDevMethodGet

```
(  
    struct vxbDev * pDev, /* Device information */  
    UINT32         method /* Specified method */  
)
```

DESCRIPTION This routine finds an entry point for specified method.

RETURNS a pointer to the entry point routine for accessing the specified functionality of the specified device, or NULL if no such functionality is available.

ERRNO

SEE ALSO **vxBus**

vxbDevMethodRun()

NAME **vxbDevMethodRun()** – run method on devices

SYNOPSIS STATUS vxbDevMethodRun
(
 UINT32 method, /* Method to run */
 void * pArg /* Argument to routine */
)

DESCRIPTION This routine runs the specified method for all instances which provide a specified method.

RETURNS OK, always

ERRNO

SEE ALSO **vxBus**

vxbDevParent()

NAME **vxbDevParent()** – find parent device

SYNOPSIS struct vxbDev * vxbDevParent
(
 struct vxbDev * pDev /* Device Information */
)

DESCRIPTION	This routine finds the parent device of the specified device.
RETURNS	pointer to parent device, or NULL
ERRNO	
SEE ALSO	vxBus

vxbDevPath()

NAME	vxbDevPath() – trace from device to nexus
SYNOPSIS	<pre>STATUS vxbDevPath (struct vxbDev * pDev, /* device */ BOOL (*func)(struct vxbDev * pDev, void * pArg), /* func @ each ctrlr */ void * pArg /* 2nd arg to func */)</pre>
DESCRIPTION	This routine traces the specified device to the PLB bus.
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	vxBus

vxbDevPathShow()

NAME	vxbDevPathShow() – Show bus hierarchy
SYNOPSIS	<pre>void vxbDevPathShow (struct vxbDev * pDev)</pre>
DESCRIPTION	This routine prints the path of busses between a specified device and the nexus
RETURNS	N/A

ERRNO

SEE ALSO [vxbShow](#)

vxbDevRegister()

NAME **vxbDevRegister()** – register a device driver

SYNOPSIS STATUS vxbDevRegister
(
 struct vxbDevRegInfo * pDeviceInfo /* per-bus recognition info */
)

DESCRIPTION This routine registers a device driver with the vxBus subsystem.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbDevRemovalAnnounce()

NAME **vxbDevRemovalAnnounce()** – announce device removal to bus subsystem

SYNOPSIS STATUS vxbDevRemovalAnnounce
(
 struct vxbDev * pDev
)

DESCRIPTION This routine removes the device from the bus subsystem. If there is an instance associated with this device, then the driver's method for unlink is called and device is removed from the instances list. If the driver does not implement this method, then this function returns an ERROR. If the device is present in the orphans list, it is removed from the orphans list.

RETURNS OK, or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbDevStructAlloc()

NAME	vxbDevStructAlloc() – allocate VXB_DEVICE structure
SYNOPSIS	struct vxbDev * vxbDevStructAlloc (int flags /* Flags */)
DESCRIPTION	This routine allocates the VXB_DEVICE structure.
RETURNS	pointer to structure, or NULL
ERRNO	
SEE ALSO	vxBus

vxbDevStructFree()

NAME	vxbDevStructFree() – free VXB_DEVICE structure
SYNOPSIS	void vxbDevStructFree (struct vxbDev * pDev /* Device Information */)
DESCRIPTION	This routine free the previously allocated VXB_DEVICE structure.
RETURNS	N/A
ERRNO	
SEE ALSO	vxBus

vxbDevStructShow()

NAME	vxbDevStructShow() – Show device information
SYNOPSIS	STATUS vxbDevStructShow

```
(  
    struct vxbDev * pDev  
)
```

DESCRIPTION This routine prints information about the specified device.

RETURNS N/A

ERRNO

SEE ALSO [vxbShow](#)

vxbDevTblEnumerate()

NAME **vxbDevTblEnumerate()** – enumerate VxBus devices from hwconf device table

SYNOPSIS

```
int vxbDevTblEnumerate  
(  
    VXB_DEVICE_ID     pParent,          /* parent bus controller */  
    VXB_ACCESS_LIST * pAccess,         /* device access API */  
    UINT32            busType,          /* bus type */  
    STATUS             (*busOverride) /* bus-specific init rtn */  
    (VXB_DEVICE_ID    pDev)           /* (init rtn arg) */  
)
```

DESCRIPTION This function searches through the `hcfDeviceList[]` array for devices matching the specified bus type. For each such device, the routine creates a device entry, fills the relevant information into the `VXB_DEVICE` structure, calls the specified bus-specific initialization routine, and then announces the device to VxBus.

NOTE: This routine puts the HCF record pointer into the `VXB_DEVICE` field `pBusSpecificDevInfo`. This allows the bus controller or bus type access to all configuration information which might be kept there. The bus controller and/or bus type code are free to use this field for their own purposes and override the HCF record pointer, but they should save the HCF pointer or extract all information from the record first, if they want to have access to that information later.

RETURNS: the number of devices found, or **ERROR** if an error occurs

ERRNO: not set

RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbDevTable

vxbDeviceAnnounce()

NAME	vxbDeviceAnnounce() – announce device discovery to bus subsystem
SYNOPSIS	STATUS vxbDeviceAnnounce (struct vxbDev * pDev /* device information */)
DESCRIPTION	This routine goes through existing bus types and drivers, attempting to match the device which has been discovered with a driver. If a driver is found, then an instance is created, and the instance is added to the bus on which the device resides. If no driver is found for the device, then the device is added to a list of unattached devices. At some future time, a driver matching this device may be added to the system. The pDev->pParentBus must be populated before calling this function. Otherwise, this device shows up in the lost Devices list(under "Lost devices in vxBus system") when vxBusShow() is called
RETURNS	OK, or FALSE
ERRNO	
SEE ALSO	vxBus

vxbDeviceDriverRelease()

NAME	vxbDeviceDriverRelease() – turn an instance into an orphan
SYNOPSIS	STATUS vxbDeviceDriverRelease

```
(  
    struct vxbDev *pDev  
)
```

DESCRIPTION This routine dissociates the specified device from its driver.

RETURNS OK or ERROR

ERRNO

SEE ALSO **vxBus**

vxbDeviceMethodRun()

NAME **vxbDeviceMethodRun()** – run method on device

SYNOPSIS STATUS vxbDeviceMethodRun
(
 struct vxbDev * pDev, /* Device Information */
 void * pArg /* Parameter to method */
)

DESCRIPTION This routine runs a method on a device if it provides specified method.

RETURNS OK or ERROR

ERRNO

SEE ALSO **vxBus**

vxbDmaBufFlush()

NAME **vxbDmaBufFlush()** – partial cache flush for DMA map

SYNOPSIS STATUS vxbDmaBufFlush
(
 VXB_DEVICE_ID pInst,
 VXB_DMA_TAG_ID dmaTagID,
 VXB_DMA_MAP_ID map,
 int index,
 int offset, /* within fragment */

```
    int             length      /* within fragment */  
)
```

DESCRIPTION	This routine does cache flush for a portion of the specified DMA map within one fragment.
RETURNS	OK if successful, else ERROR
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufInit()

NAME	vxbDmaBufInit() – initialize buffer and DMA system
SYNOPSIS	<code>void vxbDmaBufInit (void)</code>
DESCRIPTION	This routine initializes the buffer and DMA system.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufInvalidate()

NAME	vxbDmaBufInvalidate() – partial cache invalidate for DMA map
SYNOPSIS	<code>STATUS vxbDmaBufInvalidate (VXB_DEVICE_ID pInst, VXB_DMA_TAG_ID dmaTagID, VXB_DMA_MAP_ID map, int index, int offset, /* within fragment */ int length /* within fragment */)</code>
DESCRIPTION	This routine does cache invalidate for a portion of the specified DMA map within one fragment.

RETURNS OK if successful, else ERROR

ERRNO N/A

SEE ALSO **vxbDmaBufLib**

vxbDmaBufMapCreate()

NAME **vxbDmaBufMapCreate()** – create/allocate a DMA map

SYNOPSIS

```
VXB_DMA_MAP_ID vxbDmaBufMapCreate
(
    VXB_DEVICE_ID      pInst,
    VXB_DMA_TAG_ID    dmaTagID,
    int                flags,
    VXB_DMA_MAP_ID *  mapp
)
```

DESCRIPTION This routine creates/allocates a DMA map for a transfer (used as an argument for **vxbDmaBufMapLoad()**/**vxbDmaBufMapUnload()** later -- a map holds state like pointers to allocated bounce buffers).

NOTE Allocate a handle for mapping from kva/uva/physical address space into bus device space.

RETURNS pointer to the created map ID

ERRNO N/A

SEE ALSO **vxbDmaBufLib**

vxbDmaBufMapDestroy()

NAME **vxbDmaBufMapDestroy()** – release a DMA map

SYNOPSIS

```
STATUS vxbDmaBufMapDestroy
(
    VXB_DMA_TAG_ID dmaTagID,
    VXB_DMA_MAP_ID map
)
```

DESCRIPTION This routine releases a DMA map and zeros the memory associated with it

NOTE	Destroy a handle for mapping from kva/uva/physical address space into bus device space.
RETURNS	OK if the map was valid and the storage could be freed, ERROR otherwise
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufMapFlush()

NAME **vxbDmaBufMapFlush()** – flush DMA Map cache

SYNOPSIS

```
STATUS vxbDmaBufMapFlush
(
    VXB_DEVICE_ID pInst,
    VXB_DMA_TAG_ID dmaTagID,
    VXB_DMA_MAP_ID map
)
```

DESCRIPTION This routine does cache flush for the specified DMA map.

RETURNS OK if successful, else ERROR

ERRNO N/A

SEE ALSO **vxbDmaBufLib**

vxbDmaBufMapInvalidate()

NAME **vxbDmaBufMapInvalidate()** – invalidate DMA Map cache

SYNOPSIS

```
STATUS vxbDmaBufMapInvalidate
(
    VXB_DEVICE_ID pInst,
    VXB_DMA_TAG_ID dmaTagID,
    VXB_DMA_MAP_ID map
)
```

DESCRIPTION This routine does cache invalidate for the specified DMA map.

RETURNS OK if successful, else ERROR

ERRNO N/A

SEE ALSO **`vxbDmaBufLib`**

`vxbDmaBufMapIoVecLoad()`

NAME **`vxbDmaBufMapIoVecLoad()`** – map a virtual buffer with scatter/gather

SYNOPSIS STATUS vxbDmaBufMapIoVecLoad
(
 VXB_DEVICE_ID pInst,
 VXB_DMA_TAG_ID dmaTagID,
 VXB_DMA_MAP_ID map,
 struct uio * uio,
 int flags
)

DESCRIPTION This routine maps a virtual buffer. Its behavior is like **`vxbDmaBufMapLoad()`**, but works on a scatter/gather array of virtual buffers (called a struct iovec in BSD) -- useful for disk transfers of multiple data blocks.

RETURNS OK if successful, else ERROR

ERRNO N/A

SEE ALSO **`vxbDmaBufLib`**

`vxbDmaBufMapLoad()`

NAME **`vxbDmaBufMapLoad()`** – map a virtual buffer

SYNOPSIS STATUS vxbDmaBufMapLoad
(
 VXB_DEVICE_ID pInst,
 VXB_DMA_TAG_ID dmaTagID,
 VXB_DMA_MAP_ID map,
 void * buf,
 bus_size_t bufLen,
 int flags
)

DESCRIPTION	This routine maps a virtual buffer into a physical address and length, using info in DMA tag to decided what sort of address translation and possible bounce-buffering may need to be done. Consumes a DMA map allocated with vxbDmaBufMapCreate() .
NOTE	Map the buffer buf into bus space using the dmamap map.
RETURNS	OK if successful, else ERROR
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufMapMblkLoad()

NAME	vxbDmaBufMapMblkLoad() – map a virtual buffer with mBlk
SYNOPSIS	STATUS vxbDmaBufMapMblkLoad (VXB_DEVICE_ID pInst, VXB_DMA_TAG_ID dmaTagID, VXB_DMA_MAP_ID map, M_BLK_ID pMblk, int flags)
DESCRIPTION	This routine maps a virtual buffer. Its behavior is like vxbDmaBufMapLoad() , but uses an mBlk tuple.
RETURNS	OK if successful, else ERROR
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufMapUnload()

NAME	vxbDmaBufMapUnload() – unmap/destroy a previous virtual buffer mapping
SYNOPSIS	STATUS vxbDmaBufMapUnload

```
(  
VXB_DMA_TAG_ID dmaTagID,  
VXB_DMA_MAP_ID map  
)
```

DESCRIPTION	This routine unmaps/destroys a previous virtual buffer mapping after a transfer completes, possibly releasing any bounce buffers or other system resources consumed by mapping the virtual buffer.
RETURNS	OK, always
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufMemAlloc()

NAME	vxbDmaBufMemAlloc() – allocate DMA-able memory
SYNOPSIS	<pre>void * vxbDmaBufMemAlloc (VXB_DEVICE_ID pInst, VXB_DMA_TAG_ID dmaTagID, void ** vaddr, int flags, VXB_DMA_MAP_ID * pMap)</pre>
DESCRIPTION	This routine allocates DMA-able memory that satisfies requirements encoded in the supplied DMA tag -- note: in the BSD API, the DMA tag controls the size of the memory block returned by this function. So if you need, say, 1536 byte blocks, you would create a tag that lets you allocate 1536-byte chunks of memory from whatever pool you need to satisfy the DMA constraints of your device. Note also that this offers the possibility of using a slab allocator to speed up allocation time
NOTE	Allocate a piece of memory that can be efficiently mapped into bus device space based on the constraints listed in the DMA tag. A dmamap to for use with dmamap_load is also allocated.
RETURNS	a buffer pointer
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufMemFree()

NAME	vxbDmaBufMemFree() – release DMA-able memory
SYNOPSIS	STATUS vxbDmaBufMemFree((VXB_DMA_TAG_ID dmaTagID, void * vaddr, VXB_DMA_MAP_ID map >)
DESCRIPTION	This routine releases DMA-able memory allocated with vxbDmaBufMemAlloc() .
NOTE	Free a piece of memory and it's allocated dmamap, that was allocated via bus_dmamem_alloc() . Make the same choice for free/contigfree.
RETURNS	OK, always
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufSync()

NAME	vxbDmaBufSync() – do cache flushes or invalidates
SYNOPSIS	STATUS vxbDmaBufSync((VXB_DEVICE_ID pInst, VXB_DMA_TAG_ID dmaTagID, VXB_DMA_MAP_ID map, bus_dmasync_op_t op >)
DESCRIPTION	This routine does cache flushes or invalidates before/after transfers.
RETURNS	OK if successful, else ERROR
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufTagCreate()

NAME vxbDmaBufTagCreate() – create a DMA tag

SYNOPSIS

```
VXB_DMA_TAG_ID vxbDmaBufTagCreate
(
    VXB_DEVICE_ID      pInst,
    VXB_DMA_TAG_ID     parent,
    bus_size_t          alignment,
    bus_size_t          boundary,
    bus_addr_t          lowAddr,
    bus_addr_t          highAddr,
    bus_dma_filter_t * filter,
    void *              filterArg,
    bus_size_t          maxSize,
    int                 nSegments,
    bus_size_t          maxSegSz,
    int                 flags,
    bus_dma_lock_t *    lockFunc,
    void *              lockFuncArg,
    VXB_DMA_TAG_ID *   ppDmaTag
)
```

DESCRIPTION This function creates a device-specific DMA tag with the given characteristics (type of allocation (general heap, uncached, dual-ported RAM, specific partition), required alignment, max and min physical address boundaries (can the PCI bridge see all memory in a 64-bit system, or just a 4GB window?), DMAable memory allocation size, max number of allowed entries in a DMA fragment list, etc...)

RETURNS pointer to the created DMA tag

ERRNO N/A

SEE ALSO vxbDmaBufLib

vxbDmaBufTagDestroy()

NAME vxbDmaBufTagDestroy() – destroy a DMA tag

SYNOPSIS

```
STATUS vxbDmaBufTagDestroy
(
    VXB_DMA_TAG_ID dmaTagID
)
```

DESCRIPTION This routine destroys the tag when we're done with it (usually during driver unload)

RETURNS	OK if the tag was valid and the tag structure's memory was freed, ERROR otherwise.
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaBufTagParentGet()

NAME	vxbDmaBufTagParentGet() – retrieve parent DMA tag
SYNOPSIS	VXB_DMA_TAG_ID vxbDmaBufTagParentGet (VXB_DEVICE_ID pInst, UINT32 pRegBaseIndex)
DESCRIPTION	This function retrieves the DMA tag associated with the upstream bus controller closest to the device, which provides a DMA tag.
RETURNS	NULL
ERRNO	N/A
SEE ALSO	vxbDmaBufLib

vxbDmaChanAlloc()

NAME	vxbDmaChanAlloc() – allocate and initialize a DMA channel
SYNOPSIS	VXB_DMA_RESOURCE_ID vxbDmaChanAlloc (VXB_DEVICE_ID pDev, UINT32 minQueueDepth, UINT32 flags, void * pDedicatedChanInfo)
DESCRIPTION	This routine allocates and initializes a DMA channel for use by an instance. If a dedicated channel is available, this routine searches through all the instances in the system to locate the DMA engine which provides the dedicated channel. If the DMA engine

device is located, the method **vxbDmaResDedicatedGet_desc** allocates the dedicated channel resources and returns **VXB_DMA_RESOURCE_ID** for the dedicated DMA channel.

If no dedicated channel is available, this routine searches through devices, starting at the bus controller immediately upstream from the instance. If a DMA channel is allocated, this routine returns the **VXB_DMA_RESOURCE_ID** for the DMA channel.

Whether a channel will be allocated depends on the flags and minimum queue depth specified by the caller. If the available queue depth is larger than the specified minimum, and if the flags specify services which can be provided by available hardware, then the call will succeed.

RETURNS **VXB_DMA_RESOURCE_ID** if successful or NULL on error

ERRNO

SEE ALSO **vxbDmaLib**

vxbDmaChanFree()

NAME **vxbDmaChanFree()** – free a DMA channel

SYNOPSIS **VOID vxbDmaChanFree(**
 `(`
 VXB_DMA_RESOURCE_ID **dmaChan**
 `)`

DESCRIPTION This routine frees a DMA channel for allocation by another instance

RETURNS None

ERRNO

SEE ALSO **vxbDmaLib**

vxbDmaLibInit()

NAME **vxbDmaLibInit()** – initialize the VxBus slave DMA library

SYNOPSIS **void vxbDmaLibInit (void)**

DESCRIPTION	This routine initializes the buffer and DMA system.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	vxbDmaLib

vxbDriverUnregister()

NAME	vxbDriverUnregister() – remove a device driver from the bus subsystem
SYNOPSIS	STATUS vxbDriverUnregister (struct vxbDevRegInfo * pDriver)
DESCRIPTION	This routine is the final stage of driver removal. The driver must disconnect itself from the OS, free resources, and then finally call this routine. If the vxbDevRegInfo structure was dynamically allocated by the driver, the driver is responsible for freeing it. This routine dissociates the driver from each device to which it has been connected, and moves each such device onto the orphan list of the bus it resides on.
RETURNS	OK or ERROR
ERRNO	
SEE ALSO	vxBus

vxbDrvRescan()

NAME	vxbDrvRescan() – rescan all orphans to match against driver
SYNOPSIS	BOOL vxbDrvRescan (struct vxbDevRegInfo * pDriver)
DESCRIPTION	This routine re-scans all orphan device, checking to see whether they match the specified driver.

RETURNS OK, or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbEndQctrlInit()

NAME **vxbEndQctrlInit()** – initialize the vxbEndQctrl library

SYNOPSIS STATUS vxbEndQctrlInit (void)

DESCRIPTION This routine is called automatically if component INCLUDE_VXBUS_END_QCTRL is included in the VxWorks image.

RETURNS OK or ERROR.

ERRNO N/A

SEE ALSO [vxbEndQctrl](#)

vxbEndQnumSet()

NAME **vxbEndQnumSet()** – set the network job queue for a VxBus END device

SYNOPSIS STATUS vxbEndQnumSet
(
 char * instName,
 int unit,
 unsigned int qnum
)

DESCRIPTION This function sets the receive job queue used by a VxBus END device, identified by its VxBus instance name, to the job queue corresponding to network daemon number *qnum*.

If the END device is already started, then in order to change the job queue, the END device must be stopped and restarted after a short delay. In the interim, some traffic may be dropped. If you call this routine when the END device is stopped, the change does not take effect until you subsequently call **muxDevStart()** to start the device.

WARNING	This routine is not protected against concurrent access to the device by other tasks attempting to stop, start, or change the queue parameters of the device.
RETURNS	OK if the routine successfully sets the new job queue ID; ERROR otherwise.
ERRNO	N/A
SEE ALSO	vxbEndQctrl , muxEndQnumGet()

vxbEpicIntCtlrRegister()

NAME	vxbEpicIntCtlrRegister() – register epicIntCtlr driver
SYNOPSIS	<code>void vxbEpicIntCtlrRegister(void)</code>
DESCRIPTION	This routine registers the EPCI driver and device recognition data with the vxBus subsystem.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbEpicIntCtlr

vxbEpicIntCtlrpDrvCtrlShow()

NAME	vxbEpicIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller
SYNOPSIS	<code>int vxbEpicIntCtlrpDrvCtrlShow (VXB_DEVICE_ID pInst, int verboseLevel)</code>
DESCRIPTION	none
RETURNS	0 always

ERRNO Not Available

SEE ALSO [vxbEpicIntCtlr](#)

vxbEpicSharedMsgHandler()

NAME [vxbEpicSharedMsgHandler\(\)](#) – Handles the interrupt and calls the relevant ISRs

SYNOPSIS

```
void vxbEpicSharedMsgHandler
(
    struct intCtlrHwConf *isrHandle
)
```

DESCRIPTION none

RETURNS NONE

ERRNO None

SEE ALSO [vxbEpicIntCtlr](#)

vxbGetPciDevice()

NAME [vxbGetPciDevice\(\)](#) – Locates a PCI device

SYNOPSIS

```
STATUS vxbGetPciDevice
(
    VXB_DEVICE_ID childID,
    VXB_DEVICE_ID busCtrlID,
    char *          pArg
)
```

DESCRIPTION Called using vxbSubDevAction it locates a PCI device and then returns the vxBus device ID.

RETURNS OK, or ERROR upon a device which isn't.

ERRNO

SEE ALSO [vxbPci](#)

vxbI8253TimerDrvRegister()

NAME	vxbI8253TimerDrvRegister() – registers the driver for i8253 timer
SYNOPSIS	void vxbI8253TimerDrvRegister(void)
DESCRIPTION	This routine registers the driver for the i8253 timer with the <code>vxBus</code> subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbI8253Timer

vxbI8259IntCtlrRegister()

NAME	vxbI8259IntCtlrRegister() – registers I8259 driver with <code>vxBus</code>
SYNOPSIS	VOID vxbI8259IntCtlrRegister (void)
DESCRIPTION	This routine registers the Intel 8259 Programmer Interrupt Controller Driver with <code>vxBus</code> .
RETURNS	NA
	ERRNO L none
ERRNO	Not Available
SEE ALSO	vxbI8259IntCtlr

vxbI8259IntDisablePIC()

NAME	vxbI8259IntDisablePIC() – disables a PIC interrupt level
SYNOPSIS	STATUS vxbI8259IntDisablePIC (int intLevel /* interrupt level to disable */)

DESCRIPTION	This routine disables a specified PIC interrupt level. This function implements the disabling of PIC level for legacy (non-vxBus) devices.
RETURNS	OK, always.
ERRNO	none
SEE ALSO	vxbI8259IntCtlr , vxbI8259IntEnablePIC()

vxbI8259IntEOI()

NAME	vxbI8259IntEOI() – to send an EOI signal
SYNOPSIS	void vxbI8259IntEOI (void)
DESCRIPTION	This routine send the EOI for all instances of i8259 Interrupt Controller driver. The function implements the legacy support for same.
RETURNS	OK, always.
ERRNO	Not Available
SEE ALSO	vxbI8259IntCtlr

vxbI8259IntEnablePIC()

NAME	vxbI8259IntEnablePIC() – enable a PIC interrupt level
SYNOPSIS	STATUS vxbI8259IntEnablePIC (int intLevel /* interrupt level to enable */)
DESCRIPTION	This routine enables a specified PIC interrupt level. This function implements the enabling of PIC level for legacy (non-vxBus) devices.
RETURNS	OK, always.
ERRNO	none

SEE ALSO vxbI8259IntCtlr, vxbI8259IntDisablePIC()

vxbIaTimestampDrvRegister()

NAME vxbIaTimestampDrvRegister() – register intel timestamp driver

SYNOPSIS void vxbIaTimestampDrvRegister(void)

DESCRIPTION This routine registers the intel timestamp driver with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO vxbIntelTimestamp

vxbInit()

NAME vxbInit() – initialize vxBus

SYNOPSIS STATUS vxbInit (void)

DESCRIPTION This routine initializes the vxBus subsystem.

RETURNS OK, always

ERRNO

SEE ALSO vxBus

vxbInstByNameFind()

NAME vxbInstByNameFind() – retrieve the VXB_DEVICE_ID for an instance

SYNOPSIS VXB_DEVICE_ID vxbInstByNameFind

```
(  
char * instName,  
int    unit  
)
```

DESCRIPTION	This routine returns the VXB_DEVICE_ID for a given instance identified by name and unit number.
RETURNS	VXB_DEVICE_ID for the instance, or NULL if no instance is found
ERRNO	N/A
SEE ALSO	vxblParamSys

vxblInstParamByIndexGet()

NAME	vxblInstParamByIndexGet() – retrieve driver parameter value
SYNOPSIS	<pre>STATUS vxblInstParamByIndexGet (VXB_DEVICE_ID pInst, int paramInt, VXB_INST_PARAM_VALUE * pValue)</pre>
DESCRIPTION	<p>This routine retrieves the value of a parameter specified by index. The driver calls this routine to find the parameter value at a specified offset into the parameter table. The pvc field points to pre-allocated storage. When called, vxblInstParamByIndexGet() fills in the value into the memory pointed to by pValue.</p>
RETURNS	OK, or ERROR if no default parameter table has been registered
ERRNO	N/A
SEE ALSO	vxblParamSys

vxbInstParamByNameGet()

NAME **vxbInstParamByNameGet()** – retrieve driver parameter value

SYNOPSIS

```
STATUS vxbInstParamByNameGet
(
    VXB_DEVICE_ID      pInst,
    char *             paramName,
    UINT32              paramType,
    VXB_INST_PARAM_VALUE * pValue
)
```

DESCRIPTION This routine retrieves the value of a parameter specified by name.

The driver calls this routine to find the parameter value associated with a given named parameter for the specified instance.

The parsonage and paramType specify the name and type of the parameter whose value is to be found. The pValue field points to pre-allocated storage. When called, **vxbInstParamByNameGet()** fills in the value into the memory pointed to by pValue.

RETURNS OK, or ERROR if no parameter of that name exists

ERRNO N/A

SEE ALSO **vxbParamSys**

vxbInstParamSet()

NAME **vxbInstParamSet()** – set driver parameter for specified instance

SYNOPSIS

```
STATUS vxbInstParamSet
(
    VXB_DEVICE_ID      pInst,
    char *             paramName,
    UINT32              paramType,
    VXB_INST_PARAM_VALUE * value
)
```

DESCRIPTION This routine provides a programmatic API for middleware modules to set a parameter for a specific instance.

The instance is identified by a **VXB_DEVICE_ID**.

The parameter consists of three fields: parameter value, parameter name, and parameter type. The parameter name is specific to the driver. The supported parameter types of storage are unsigned integer 32, unsigned integer 64, character string, function pointer, and generic "void" pointer.

Note that when 64-bit values are specified, the value must be specified with a pointer. That is, value is interpreted in that case as "UINT64 * pValue" rather than as "UINT64 value".

This routine should not be used from any BSP code, but only by application or middleware code. Parameter setting in the BSP should be done in the BSP-resident parameter override table.

RETURNS OK, or ERROR if the specified parameter could not be set for the instance

ERRNO N/A

SEE ALSO [vxbParamSys](#)

vxbInstUnitGet()

NAME **vxbInstUnitGet()** – get the unit number

SYNOPSIS STATUS vxbInstUnitGet
(
 VXB_DEVICE_ID pDev,
 UINT32 * pUnitNumber
)

DESCRIPTION This routine gets the unit number for the device.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbInstUnitSet()

NAME **vxbInstUnitSet()** – set the unit number

SYNOPSIS STATUS vxbInstUnitSet

```
(  
VXB_DEVICE_ID pDev,  
UINT32          unitNumber  
)
```

DESCRIPTION This routine sets the unit number for the device.

RETURNS OK or ERROR

ERRNO

SEE ALSO **vxBus**

vxbIntAcknowledge()

NAME **vxbIntAcknowledge()** – Acknowledge device's interrupt

SYNOPSIS STATUS vxbIntAcknowledge
(
 struct vxbDev * pDev, /* Device Information */
 int index, /* index of interrupt vector */
 VOIDFUNCPTR pIsr, /* ISR */
 void * pArg /* parameter */
)

DESCRIPTION This routine acknowledges and clears the specified interrupt on any interrupt controller intervening between the processor and the device. It does not affect the interrupt source or the processor.

RETURNS OK or ERROR

ERRNO

SEE ALSO **vxBus**

vxbIntConnect()

NAME **vxbIntConnect()** – connect device's interrupt

SYNOPSIS STATUS vxbIntConnect

```
(  
    struct vxbDev * pDev,      /* Device Information */  
    int             index,     /* index of interrupt vector */  
    VOIDFUNC PTR   pIsr,      /* ISR */  
    void *          pArg       /* parameter */  
)
```

DESCRIPTION This routine connects the specified ISR to the interrupt source.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxblntDisable()

NAME **vxblntDisable()** – disable device's interrupt

SYNOPSIS STATUS vxblntDisable
(
 struct vxbDev * pDev, /* Device Information */
 int index, /* index of interrupt vector */
 VOIDFUNC PTR pIsr, /* ISR */
 void * pArg /* parameter */
)

DESCRIPTION This routine disables the specified interrupt on the lowest-level interrupt controller between the processor and the device. It does not affect the interrupt source nor the processor.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxblntDisconnect()

NAME **vxblntDisconnect()** – disconnect device's interrupt

SYNOPSIS STATUS vxblntDisconnect

```
(  
    struct vxbDev * pDev,      /* Device Information */  
    int             index,     /* index of interrupt vector */  
    VOIDFUNCPTR    pIsr,      /* ISR */  
    void *          pArg       /* parameter */  
)
```

DESCRIPTION This routine disconnects the specified ISR from the interrupt source.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbIntDynaConnect()

NAME [**vxbIntDynaConnect\(\)**](#) – Initializes/connects the dynamic interrupt

SYNOPSIS STATUS vxbIntDynaConnect
(
 VXB_DEVICE_ID pDev,
 int vecCount,
 struct vxbIntDynaVecInfo * pDynaVecIntr
)

DESCRIPTION This routine finds the interrupt controller then calls the method for connecting dynamic interrupt.

RETURNS STATUS

ERRNO

SEE ALSO [vxbIntDynaCtlrLib](#)

vxbIntDynaCtlrInit()

NAME [**vxbIntDynaCtlrInit\(\)**](#) – Intializes function pointers to enable library use.

SYNOPSIS void vxbIntDynaCtlrInit(void)

DESCRIPTION	none
RETURNS	NONE
ERRNO	
SEE ALSO	vxbIntDynaCtlrLib

`vxbIntDynaVecAlloc()`

NAME	<code>vxbIntDynaVecAlloc()</code> – Finds the the next vector for dynamic interrupt
SYNOPSIS	<pre>STATUS vxbIntDynaVecAlloc (VXB_DEVICE_ID vecOwner, VXB_DEVICE_ID serviceInstance, int numVecs, struct vxbIntDynaVecInfo * pDynaVec)</pre>
DESCRIPTION	This routine calls the vector owner to allocate a vector for the dynamic interrupt. Only needed if interrupt controller isn't the vector owner.
RETURNS	NONE
ERRNO	
SEE ALSO	vxbIntDynaCtlrLib

`vxbIntDynaVecDevMultiProgram()`

NAME	<code>vxbIntDynaVecDevMultiProgram()</code> – program multiple vectors
SYNOPSIS	<pre>STATUS vxbIntDynaVecDevMultiProgram (VXB_DEVICE_ID pVectorOwner, VXB_DEVICE_ID serviceInstance, int numVectors, struct vxbIntDynaVecInfo * pDynaVec)</pre>
DESCRIPTION	none

RETURNS	STATUS
ERRNO	
SEE ALSO	vxbIntDynaCtlrLib

vxbIntDynaVecMultiConnect()

NAME	vxbIntDynaVecMultiConnect() – connect to multiple vectors
SYNOPSIS	STATUS vxbIntDynaVecMultiConnect (VXB_DEVICE_ID intCtrlrID, VXB_DEVICE_ID instID, int numVecs, struct vxbIntDynaVecInfo * pDynaVec)
DESCRIPTION	none
RETURNS	STATUS
ERRNO	
SEE ALSO	vxbIntDynaCtlrLib

vxbIntDynaVecOwnerFind()

NAME	vxbIntDynaVecOwnerFind() – Finds the owner of dynnamic vector alloc
SYNOPSIS	VXB_DEVICE_ID vxbIntDynaVecOwnerFind (VXB_DEVICE_ID instID)
DESCRIPTION	This routine finds the vector owner that can allocate a vector for the dynamic interrupt. Only needed if interrupt controller isn't the vector owner.
RETURNS	NONE

ERRNO

SEE ALSO [vxbIntDynaCtlrLib](#)

vxbIntEnable()

NAME **vxbIntEnable()** – Enable device's interrupt

SYNOPSIS STATUS vxbIntEnable
(
 struct vxbDev * pDev, /* Device Information */
 int index, /* index of interrupt vector */
 VOIDFUNCPTR pIsr, /* ISR */
 void * pArg /* parameter */
)

DESCRIPTION This routine enables the specified interrupt on any interrupt controller intervening between the processor and the device. It does not affect the interrupt source nor the processor.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbIntReroute()

NAME **vxbIntReroute()** – Route specified interrupt to destination CPU

SYNOPSIS STATUS vxbIntReroute
(
 VXB_DEVICE_ID pRequestor,
 int interruptIndex,
 cpuset_t destCpu
)

DESCRIPTION This routine routes the specified interrupt to the specified CPU.

RETURNS OK always

ERRNO

SEE ALSO **vxBus**

vxbIntToCpuRoute()

NAME **vxbIntToCpuRoute()** – reroute all interrupts for a specified CPU

SYNOPSIS STATUS vxbIntToCpuRoute
(
 unsigned int destCpu /* destination CPU for reroute */
)

DESCRIPTION This routine calls all interrupt controllers that support interrupt routing functions (method `vxbIntCpuReroute` declared by the controller) passing this function the specified CPU for the reroute operation.

RETURNS OK always

ERRNO

SEE ALSO **vxBus**

vxbIntVectorGet()

NAME **vxbIntVectorGet()** – get device's interrupt vector

SYNOPSIS VOIDFUNC PTR * vxbIntVectorGet
(
 struct vxbDev * pDev, /* Device Information */
 int index /* index of interrupt vector */
)

DESCRIPTION This routine returns the vector for the specified interrupt on the specified device. The return value of this routine can be passed as the vector to `intConnect()` and other system-level interrupt support routines.

RETURNS vector, or NULL

ERRNO

SEE ALSO **vxBus**

vxbIoApicIntrDataShow()

NAME **vxbIoApicIntrDataShow()** – show IO APIC register data acquired by vxBus

SYNOPSIS

```
void vxbIoApicIntrDataShow
(
    VXB_DEVICE_ID pInst,
    int            verboseLevel
)
```

DESCRIPTION This routine shows IO APIC register data acquired by vxBus.

RETURNS N/A

ERRNO Not Available

SEE ALSO **vxbIoApicIntr**

vxbIoApicIntrDrvRegister()

NAME **vxbIoApicIntrDrvRegister()** – register ioApic driver

SYNOPSIS

```
void vxbIoApicIntrDrvRegister(void)
```

DESCRIPTION This routine registers the ioApic driver with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO **vxbIoApicIntr**

vxbIoApicIntrShowAll()

NAME	vxbIoApicIntrShowAll() – show All IO APIC registers
SYNOPSIS	void vxbIoApicIntrShowAll (void)
DESCRIPTION	temporary -since it depends on MP table being present This routine shows all IO APIC registers
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbIoApicIntr

vxbLibError()

NAME	vxbLibError() – handle error conditions
SYNOPSIS	STATUS vxbLibError (FUNCPTR pAddr, /* calling routine */ char * pMsg /* error message */)
DESCRIPTION	This is a generic routine to handle error conditions.
RETURNS	ERROR, always
ERRNO	
SEE ALSO	vxBus

vxbLibInit()

NAME	vxbLibInit() – initialize vxBus library
SYNOPSIS	STATUS vxbLibInit (void)

DESCRIPTION This routine initializes vxBus library.

RETURNS OK, always

ERRNO

SEE ALSO **vxBus**

vxbLoApicIntrDrvRegister()

NAME **vxbLoApicIntrDrvRegister()** – register loApic driver

SYNOPSIS void vxbLoApicIntrDrvRegister(void)

DESCRIPTION This routine registers the loApic driver with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO **vxbLoApicIntr**

vxbLoApicIntrShow()

NAME **vxbLoApicIntrShow()** – show Local APIC registers

SYNOPSIS STATUS vxbLoApicIntrShow
(
 VXB_DEVICE_ID pInst,
 int verboseLevel
)

DESCRIPTION This routine shows Local APIC registers

RETURNS OK, or ERROR if not compliant

ERRNO Not Available

SEE ALSO **vxbLoApicIntr**

vxbLoApicIpiConnect()

NAME vxbLoApicIpiConnect() – Connect ISR to specified IPI

SYNOPSIS

```
STATUS vxbLoApicIpiConnect
(
    VXB_DEVICE_ID     pCtlr,
    INT32             ipiId,
    IPI_HANDLER_FUNC  ipiHandler,
    void *            ipiArg
)
```

DESCRIPTION This function connects the specified ISR and argument to the specified IPI.

FIXME

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbLoApicIntr](#)

vxbLoApicIpiDisable()

NAME vxbLoApicIpiDisable() – Disable specified IPI

SYNOPSIS

```
STATUS vxbLoApicIpiDisable
(
    VXB_DEVICE_ID pCtlr,
    INT32         ipiId
)
```

DESCRIPTION This function disables the specified IPI.

Currently, this function models the [cpcIpiIntDisable\(\)](#) routine from the IA32 arch code, which simply returns **OK**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO

[vxbLoApicIntr](#)

vxbLoApicIpiDisconnect()

NAME **vxbLoApicIpiDisconnect()** – Disconnect ISR from specified IPI

SYNOPSIS

```
STATUS vxbLoApicIpiDisconnect
(
    VXB_DEVICE_ID     pCtlr,
    INT32             ipiId,
    IPI_HANDLER_FUNC  ipiHandler,
    void *            ipiArg
)
```

DESCRIPTION This function is not currently supported.

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbLoApicIntr](#)

vxbLoApicIpiEnable()

NAME **vxbLoApicIpiEnable()** – Enable ISR for specified IPI

SYNOPSIS

```
STATUS vxbLoApicIpiEnable
(
    VXB_DEVICE_ID pCtlr,
    INT32         ipiId
)
```

DESCRIPTION This function enables the specified IPI.

Currently, this function models the **cpcIpiIntEnable()** routine from the IA32 arch code, which simply returns **OK**.

RETURNS Not Available

ERRNO Not Available

SEE ALSO

vxbLoApicIntr

vxbLoApicIpiGen()

NAME

vxbLoApicIpiGen() – Generate specified IPI

SYNOPSIS

```
STATUS vxbLoApicIpiGen
(
    VXB_DEVICE_ID pCtldr,
    INT32         ipiId,
    cpuset_t      cpus
)
```

DESCRIPTION

This function emits an IPI according to the cpus argument.

RETURNS

Inter-Processor Interrupt info

ERRNO

Not Available

SEE ALSO

vxbLoApicIntr

vxbLoApicIpiPrioGet()

NAME

vxbLoApicIpiPrioGet() – Find priority of specified IPI

SYNOPSIS

```
INT32 vxbLoApicIpiPrioGet
(
    VXB_DEVICE_ID pCtldr,
    INT32         ipiId
)
```

DESCRIPTION

This function returns the priority of the specified IPI.

IPIs available, and their uses:

0xe8:	Shutdown	prio 7
0xe9:	TscReset	prio 6
0xea:	TlbFlush	prio 5
0xeb:	available	prio 4
0xec:	available	prio 3
0xed:	available	prio 2

0xee: Debug prio 1
0xef: CPC prio 0

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbLoApicIntr](#)

vxbLoApicIpiPrioSet()

NAME **vxbLoApicIpiPrioSet()** – Set priority of specified IPI

SYNOPSIS STATUS vxbLoApicIpiPrioSet
(
 VXB_DEVICE_ID pCtlr,
 INT32 ipiId,
 INT32 prio
)

DESCRIPTION This function is not implemented for IA32 architecture. The IPI priority is determined by the ipiID used, and cannot be set after the IPI has been connected.

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbLoApicIntr](#)

vxbLoApicStatusShow()

NAME **vxbLoApicStatusShow()** – show Local APIC TMR, IRR, ISR registers

SYNOPSIS STATUS vxbLoApicIntrStatusShow
(
 VXB_DEVICE_ID pInst,

	int verboseLevel)
DESCRIPTION	This routine shows Local APIC TMR, IRR, ISR registers
RETURNS	OK, or ERROR if not compliant
ERRNO	Not Available
SEE ALSO	vxbLoApicIntr

vxbLoApicTimerDrvRegister()

NAME	<code>vxbLoApicTimerDrvRegister()</code> – register loApic timer driver
SYNOPSIS	<code>void vxbLoApicTimerDrvRegister(void)</code>
DESCRIPTION	This routine registers the loApic timer driver with the <code>vxBus</code> subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbLoApicTimer

vxbLockGive()

NAME	<code>vxbLockGive()</code> – release a VxBus lock
SYNOPSIS	<code>STATUS vxbLockGive (VXB_LOCK lockId, BOOL reader)</code>
DESCRIPTION	none
RETURNS	Not Available
ERRNO	Not Available

SEE ALSO

vxBus

vxbLockInit()

NAME **vxbLockInit()** – initialize a VxBus lock

SYNOPSIS STATUS vxbLockInit
(
 VXB_LOCK lockId
)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxBus**

vxbLockTake()

NAME **vxbLockTake()** – take a VxBus lock

SYNOPSIS STATUS vxbLockTake
(
 VXB_LOCK lockId,
 BOOL reader
)

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO **vxBus**

vxbMalRegister()

NAME	vxbMalRegister() – register the vxbMalDma driver
SYNOPSIS	<code>void vxbMalRegister(void)</code>
DESCRIPTION	This routine registers the vxbMalDma driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbIbmMalDma

vxbMc146818RtcDrvRegister()

NAME	vxbMc146818RtcDrvRegister() – registers the driver for MC146818 RTC
SYNOPSIS	<code>void vxbMc146818RtcDrvRegister(void)</code>
DESCRIPTION	This routine registers the driver for MC146818 Real Time Clock with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbMc146818Rtc

vxbMipsCavIntCtlrRegister()

NAME	vxbMipsCavIntCtlrRegister() – register the mipsIntCtlr driver
SYNOPSIS	<code>void vxbMipsCavIntCtlrRegister(void)</code>
DESCRIPTION	This routine registers the Cavium interrupt controller driver and device recognition data with the vxBus subsystem.
RETURNS	N/A

ERRNO

SEE ALSO [vxbMipsCavIntCtlr](#)

`vxbMipsIntCtlrRegister()`

NAME `vxbMipsIntCtlrRegister()` – register the mips cpu driver

SYNOPSIS `void vxbMipsIntCtlrRegister(void)`

DESCRIPTION This routine registers the mips Core driver and device recognition data with the vxBus subsystem.

RETURNS N/A

ERRNO N/A

SEE ALSO [vxbMipsIntCtlr](#)

`vxbMipsSbIntCtlrRegister()`

NAME `vxbMipsSbIntCtlrRegister()` – register the mipsIntCtlr driver

SYNOPSIS `void vxbMipsSbIntCtlrRegister(void)`

DESCRIPTION This routine registers the SB interrupt controller driver and device recognition data with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO [vxbMipsSbIntCtlr](#)

vxbMpApicDataShow()

NAME	vxbMpApicDataShow() – show MP Configuration Data acquired by vxBus
SYNOPSIS	void vxbMpApicDataShow (VXB_DEVICE_ID pInst, int *dummy)
DESCRIPTION	This routine shows MP Configuration Data acquired by vxBus.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbMpApic

vxbMpApicDrvRegister()

NAME	vxbMpApicDrvRegister() – register mpApic driver
SYNOPSIS	void vxbMpApicDrvRegister(void)
DESCRIPTION	This routine registers the mpApic driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbMpApic

vxbMpApicpDrvCtrlShow()

NAME	vxbMpApicpDrvCtrlShow() – show pDrvCtrl for template controller
SYNOPSIS	int vxbMpApicpDrvCtrlShow

```
(  
VXB_DEVICE_ID pInst  
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbMpApic](#)

vxbMpBiosIoIntMapShow()

NAME [vxbMpBiosIoIntMapShow\(\)](#) – show MP IO interrupt mapping

SYNOPSIS void vxbMpBiosIoIntMapShow (void)

DESCRIPTION This routine shows MP configuration for BIOS I/O Interrupt Configuration Table.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxbMpApic](#)

vxbMpBiosLocalIntMapShow()

NAME [vxbMpBiosLocalIntMapShow\(\)](#) – show MP local interrupt mapping

SYNOPSIS void vxbMpBiosLocalIntMapShow (void)

DESCRIPTION This routine shows MP configuration for BIOS Local Interrupt Configuration Table.

RETURNS N/A

ERRNO Not Available

SEE ALSO [vxbMpApic](#)

vxbMpBiosShow()

NAME	vxbMpBiosShow() – show MP configuration table
SYNOPSIS	void vxbMpBiosShow (void)
DESCRIPTION	This routine shows MP configuration table.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbMpApic

vxbMsDelay()

NAME	vx_bMsDelay() – delay for delayTime milliseconds
SYNOPSIS	<pre>void vx_bMsDelay (int delayTime)</pre>
DESCRIPTION	This routine introduces a busy wait of delayTime milliseconds.
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	vx_bDelayLib

vxbMsiConnect()

NAME vxbMsiConnect() – calls vxbIntDynaConnect if available

SYNOPSIS

```
STATUS vxbMsiConnect
(
    VXB_DEVICE_ID           instance,
    int                      vecCount,
```

```
    struct vxbIntDynaVecInfo * pDynaVecIntr
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbIntDynaCtlrLib](#)

vxbNextUnitGet()

NAME **vxbNextUnitGet()** – get the next available unit number for a driver

SYNOPSIS

```
STATUS vxbNextUnitGet
(
    VXB_DEVICE_ID pDev
)
```

DESCRIPTION This routine searches the instances associated with a given driver and finds the next available free unit number for a new device. The search starts at unit zero and works up until an unused unit number is found.

This is a convenience function for drivers where simply using a monotonically increasing unit number would be unsuitable, such as for pluggable devices where instances are constantly being created and destroyed.

RETURNS OK or ERROR

ERRNO

SEE ALSO [vxBus](#)

vxbPciAccessCopy()

NAME **vxbPciAccessCopy()** – copy access function pointers

SYNOPSIS

```
void vxbPciAccessCopy
```

```

(
    struct vxbAccessList *pAccess
)

```

DESCRIPTION This routine copies the access function pointers.

RETURNS N/A

ERRNO

SEE ALSO [vxbPciAccess](#)

vxbPciAutoAddrAlign()

NAME [vxbPciAutoAddrAlign\(\)](#) – align a PCI address and check boundary conditions

SYNOPSIS STATUS vxbPciAutoAddrAlign

```

(
    UINT32 base,           /* base of available memory */
    UINT32 limit,          /* last addr of available memory */
    UINT32 reqSize,         /* required size */
    UINT32 *pAlignedBase   /* output: aligned address put here */
)

```

DESCRIPTION This routine handles address alignment/checking.

RETURNS OK, or **ERROR** if available memory has been exceeded.

ERRNO

SEE ALSO [vxbPci](#)

vxbPciAutoBusNumberSet()

NAME [vxbPciAutoBusNumberSet\(\)](#) – set the primary, secondary, and subordinate bus number

SYNOPSIS STATUS vxbPciAutoBusNumberSet

```

(
    PCI_AUTO_CONFIG_OPTS * pSystem,      /* needed for config info */
    PCI_LOC *             pPciLoc,        /* device affected */
    UINT                 primary,        /* primary bus specification */
    UINT                 secondary,       /* secondary bus specification */
)

```

```
    UINT                      subordinate /* subordinate bus specification */  
)
```

DESCRIPTION	This routine sets the primary, secondary, and subordinate bus numbers for a device that implements the Type 1 PCI Configuration Space Header.
	This routine has external visibility to enable it to be used by BSP Developers for initialization of PCI Host Bridges that may implement registers similar to those found in the Type 1 Header.
RETURNS	OK, always
ERRNO	
SEE ALSO	vxbPci

vxbPciAutoCardBusConfig()

NAME	vxbPciAutoCardBusConfig() – set mem and I/O registers for a single PCI-Cardbus bridge
SYNOPSIS	<pre>LOCAL void vxbPciAutoCardBusConfig (PCI_AUTO_CONFIG_OPTS * pSystem, /* PCI system info */ PCI_LOC * pPciLoc, /* PCI address of this bridge */ PCI_LOC ** ppPciList, /* Pointer to function list pointer */ */ UINT * nSize /* Number of remaining functions */)</pre>
DESCRIPTION	This routine sets up memory and I/O base/limit registers for an individual PCI-Cardbus bridge. Cardbus bridges have four windows - 2 memory windows and 2 IO windows. The 2 memory windows can be setup individually for either prefetchable or non-prefetchable memory accesses. Since PC Cards can be inserted at any time, and are not necessarily present when this code is run, the code does not probe any further after encountering a Cardbus bridge. Instead, the code allocates default window sizes for the Cardbus bridge. Three windows are used: Warning: do not sort the include function list before this routine is called. This routine requires each function in the list to be in the same order as the probe occurred.
RETURNS	N/A

ERRNO

SEE ALSO

vxbPci

vxbPciAutoCfg()

NAME **vxbPciAutoCfg()** – Automatically configure all nonexcluded PCI headers

SYNOPSIS

```
STATUS vxbPciAutoCfg
(
    void *pCookie /* cookie returned by vxbPciAutoConfigLibInit() */
)
```

DESCRIPTION Top level function in the PCI configuration process.

CALLING SEQUENCE

```
pCookie = vxbPciAutoConfigLibInit(NULL);
vxbPciAutoCfgCtl(pCookie, COMMAND, VALUE);
...
vxbPciAutoCfgCtl(pCookie, COMMAND, VALUE);
vxbPciAutoCfg(pCookie);
```

For ease in converting from the old interface to the new one, a **vxbPciAutoCfgCtl()** command **PCI_PSYSTEM_STRUCT_COPY** has been implemented. This can be used just like any other **vxbPciAutoCfgCtl()** command, and it initializes all the values in pSystem. If used, it should be the first call to **vxbPciAutoCfgCtl()**.

For a description of the COMMANDs and VALUEs to **vxbPciAutoCfgCtl()**, see the **vxbPciAutoCfgCtl()** documentation.

For all nonexcluded PCI functions on all PCI bridges, this routine automatically configures the PCI configuration headers for PCI devices and subbridges. The fields that are programmed are:

1. Status register.
2. Command Register.
3. Latency timer.
4. Cache Line size.
5. Memory and/or I/O base address and limit registers.
6. Primary, secondary, subordinate bus number (for PCI-PCI bridges).
7. Expansion ROM disable.
8. Interrupt Line.

ALGORITHM	Probe PCI config space and create a list of available PCI functions. Call device exclusion function, if registered, to exclude/include device. Disable all devices before we initialize any. Allocate and assign PCI space to each device. Calculate and set interrupt line value. Initialize and enable each device.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPci

vxbPciAutoCfgCtl()

NAME	vxbPciAutoCfgCtl() – set or get vxbPciAutoConfigLib options
SYNOPSIS	<pre>STATUS vxbPciAutoCfgCtl (void * pCookie, /* system configuration information */ int cmd, /* command word */ void * pArg /* argument for the cmd */)</pre>
DESCRIPTION	<p>vxbPciAutoCfgCtl() can be considered analogous to ioctl() calls: the call takes arguments of (1) a pCookie, returned by vxbPciAutoConfigLibInit(). (2) A command, macros for which are defined in vxbPciAutoConfigLib.h. And, (3) an argument, the type of which depends on the specific command, but will always fit in a pointer variable. Currently, only globally effective commands are implemented.</p> <p>The commands available are:</p> <p>PCI_FBB_ENABLE - BOOL * pArg</p> <p>PCI_FBB_DISABLE - void</p> <p>PCI_FBB_UPDATE - BOOL * pArg</p> <p>PCI_FBB_STATUS_GET - BOOL * pArg</p> <p>Enable and disable the functions which check Fast Back To Back functionality.</p> <p>PCI_FBB_UPDATE is for use with dynamic/HA applications. It first disables FBB on all functions, then enables FBB on all functions, if appropriate. In HA applications, it should be called any time a card is added or removed. The BOOL pointed to by pArg for PCI_FBB_ENABLE and PCI_FBB_UPDATE is set to TRUE if all cards allow FBB functionality and FALSE if either any card does not allow FBB functionality or if FBB is disabled. The BOOL pointed to by pArg for PCI_FBB_STATUS_GET is set to TRUE if PCI_FBB_ENABLE has been called and FBB is enabled, even if FBB is not activated on any card. It is set to FALSE otherwise.</p>

NOTE: In the current implementation, FBB is enabled or disabled on the entire bus. If any device anywhere on the bus cannot support FBB, then it is not enabled, even if specific sub-busses could support it.

PCI_MAX_LATENCY_FUNC_SET - FUNCPTR * pArg

This routine is called for each function present on the bus when discovery takes place. The routine must accept four arguments, specifying bus, device, function, and a user-supplied argument of type void *. See **PCI_MAX_LATENCY_ARG_SET**. The routine should return a **UINT8** value, which will be put into the **MAX_LAT** field of the header structure. The user supplied routine must return a valid value each time it is called. There is no mechanism for any **ERROR** condition, but a default value can be returned in such a case. Default = **NULL**.

PCI_MAX_LATENCY_ARG_SET - void * pArg

When the routine specified in **PCI_MAX_LATENCY_FUNC_SET** is called, this will be passed to it as the fourth argument.

PCI_MAX_LAT_ALL_SET - int pArg

Specifies a constant max latency value for all cards, if no function has been specified with **PCI_MAX_LATENCY_FUNC_SET**..

PCI_MAX_LAT_ALL_GET - UINT * pArg

Retrieves the value of max latency for all cards, if no function has been specified with **PCI_MAX_LATENCY_FUNC_SET**. Otherwise, the integer pointed to by pArg is set to the value 0xffffffff.

PCI_MSG_LOG_SET - FUNCPTR * pArg

The argument specifies a routine which is called to print warning or error messages from **vxbPciAutoConfigLib** if **logMsg()** has not been initialized at the time **vxbPciAutoConfigLib** is used. The specified routine must accept arguments in the same format as **logMsg()**, but it does not necessarily need to print the actual message. An example of this routine is presented below, which saves the message into a safe memory space and turns on an LED. This command is useful for BSPs which call **vxbPciAutoCfg()** before message logging is enabled. Note that after **logMsg()** is configured, output goes to **logMsg()** even if this command has been called. Default = **NULL**.

```
/* sample PCI_MSG_LOG_SET function */
int pciLogMsg(char *fmt,int a1,int a2,int a3,int a4,int a5,int a6)
{
    int charsPrinted;

    sysLedOn(4);
    charsPrinted = sprintf (sysExcMsg, fmt, a1, a2, a3, a4, a5, a6);
    sysExcMsg += charsPrinted;
    return (charsPrinted);
}
```

PCI_MAX_BUS_GET - int * pArg

During autoconfiguration, the library maintains a counter with the highest numbered bus. This can be retrieved by

```
vxkBciAutoCfgCtl(pCookie, PCI_MAX_BUS_GET, &maxBus)
```

PCI_CACHE_SIZE_SET - int pArg

Sets the pci cache line size to the specified value. See CONFIGURATION SPACE PARAMETERS in the **vxkBciAutoConfigLib** documentation for more details.

PCI_CACHE_SIZE_GET - int * pArg

Retrieves the value of the pci cache line size.

PCI_AUTO_INT_ROUTE_SET - BOOL pArg

Enables or disables automatic interrupt routing across bridges during the autoconfig process. See "INTERRUPT ROUTING ACROSS PCI-TO-PCI BRIDGES" in the **vxkBciAutoConfigLib** documentation for more details.

PCI_AUTO_INT_ROUTE_GET - BOOL * pArg

Retrieves the status of automatic interrupt routing.

PCI_MEM32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit memory space. Normally, this is given by the BSP constant **PCI_MEM_ADRS**.

PCI_MEM32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit memory space. Normally, this is given by the BSP constant **PCI_MEM_SIZE**.

PCI_MEM32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit memory space.

PCI_MEMIO32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit non-prefetch memory space. Normally, this is given by the BSP constant **PCI_MEMIO_ADRS**.

PCI_MEMIO32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit non-prefetch memory space. Normally, this is given by the BSP constant **PCI_MEMIO_SIZE**.

PCI_MEMIO32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit non-prefetch memory space.

PCI_IO32_LOC_SET - UINT32 pArg

Sets the base address of the PCI 32-bit I/O space. Normally, this is given by the BSP constant **PCI_IO_ADRS**.

PCI_IO32_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 32-bit I/O space. Normally, this is given by the BSP constant **PCI_IO_SIZE**.

PCI_IO32_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 32-bit I/O space.

PCI_IO16_LOC_SET - UINT32 pArg

Sets the base address of the PCI 16-bit I/O space. Normally, this is given by the BSP constant **PCI_ISA_IO_ADDRS**

PCI_IO16_SIZE_SET - UINT32 pArg

Sets the maximum size to use for the PCI 16-bit I/O space. Normally, this is given by the BSP constant **PCI_ISA_IO_SIZE**

PCI_IO16_SIZE_GET - UINT32 * pArg

After autoconfiguration has been completed, this retrieves the actual amount of space which has been used for the PCI 16-bit I/O space.

PCI_INCLUDE_FUNC_SET - FUNCPTR * pArg

The device inclusion routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET vxbPciAutoCfgCtl()** command:

```
vxbPciAutoCfgCtl(pSystem, PCI_INCLUDE_FUNC_SET, sysPciAutoconfigInclude);
```

This optional user-supplied routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI vendorID and deviceID of the function. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigInclude
(
    PCI_AUTO_CONFIG_OPTS *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This optional user-specified routine is called by PCI AutoConfig for each and every function encountered in the scan phase. The BSP developer may use any combination of the input data to ascertain whether a device is to be excluded from the autoconfig process. The exclusion routine then returns **ERROR** if a device is to be excluded, and **OK** if a device is to be included in the autoconfiguration process.

Note that PCI-to-PCI Bridges may not be excluded, regardless of the value returned by the BSP device inclusion routine. The return value is ignored for PCI-to-PCI bridges.

The Bridge device will be always be configured with proper primary, secondary, and subordinate bus numbers in the device scanning phase and proper I/O and Memory aperture settings in the configuration phase of autoconfig regardless of the value returned by the BSP device inclusion routine.

PCI_INT_ASSIGN_FUNC_SET - FUNCPTR * pArg

The interrupt assignment routine is specified by assigning a function pointer with the **PCI_INCLUDE_FUNC_SET vxbPciAutoCfgCtl()** command:

```
vxbPciAutoCfgCtl(pCookie, PCI_INT_ASSIGN_FUNC_SET,
    sysPciAutoconfigIntrAssign);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and an 8-bit quantity containing the contents of the interrupt Pin register from the PCI configuration header of the device under consideration. The interrupt pin register

specifies which of the four PCI Interrupt request lines available are connected. The function prototype for this function is shown below:

```
UCHAR sysPciAutoconfigIntrAssign
(
    PCI_AUTO_CONFIG_OPTS *pSys,
    PCI_LOC *pLoc,
    UCHAR pin
);
```

This routine may use any combination of these data to ascertain the interrupt level. This value is returned from the function, and is programmed into the interrupt line register of the function's PCI configuration header. In this manner, device drivers may subsequently read this register in order to calculate the appropriate interrupt vector which to attach an interrupt service routine.

PCI_BRIDGE_PRE_CONFIG_FUNC_SET - FUNCPTR * pArg

The bridge pre-configuration pass initialization routine is provided so that the BSP Developer can initialize a bridge device prior to the configuration pass on the bus that the bridge implements. This routine is specified by calling ***vxbPciAutoCfgCtl()*** with the ***PCI_BRIDGE_PRE_CONFIG_FUNC_SET*** command:

```
vxbPciAutoCfgCtl(pCookie, PCI_BRIDGE_PRE_CONFIG_FUNC_SET,
                  sysPciAutoconfigPreEnumBridgeInit);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI deviceID and vendorID of the device. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigPreEnumBridgeInit
(
    PCI_AUTO_CONFIG_OPTS *pSys,
    PCI_LOC *pLoc,
    UINT devVend
);
```

This routine may use any combination of these input data to ascertain any special initialization requirements of a particular type of bridge at a specified geographic location.

PCI_BRIDGE_POST_CONFIG_FUNC_SET - FUNCPTR * pArg

The bridge post-configuration pass initialization routine is provided so that the BSP Developer can initialize the bridge device after the bus that the bridge implements has been enumerated. This routine is specified by calling ***vxbPciAutoCfgCtl()*** with the ***PCI_BRIDGE_POST_CONFIG_FUNC_SET*** command

```
vxbPciAutoCfgCtl(pCookie, PCI_BRIDGE_POST_CONFIG_FUNC_SET,
                  sysPciAutoconfigPostEnumBridgeInit);
```

This optional user-specified routine takes as input both the bus-device-function tuple, and a 32-bit quantity containing both the PCI deviceID and vendorID of the device. The function prototype for this function is shown below:

```
STATUS sysPciAutoconfigPostEnumBridgeInit
```

```
(  
PCI_AUTO_CONFIG_OPTS *pSys,  
PCI_LOC *pLoc,  
UINT devVend  
);
```

This routine may use any combination of these input data to ascertain any special initialization requirements of a particular type of bridge at a specified geographic location.

PCI_ROLLCALL_FUNC_SET - FUNCPTR * pArg

The specified routine will be configured as a roll call routine.

If a roll call routine has been configured, before any configuration is actually done, the roll call routine is called repeatedly until it returns TRUE. A return value of TRUE indicates that either (1) the specified number and type of devices named in the roll call list have been found during PCI bus enumeration or (2) the timeout has expired without finding all of the specified number and type of devices. In either case, it is assumed that all of the PCI devices which are going to appear on the busses have appeared and we can proceed with PCI bus configuration.

PCI_TEMP_SPACE_SET - char * pArg

This command is not currently implemented. It allows the user to set aside memory for use during **vxbPciAutoConfigLib** execution, e.g. memory set aside using **USER_RESERVED_MEM**. After PCI configuration has been completed, the memory can be added to the system memory pool using **memAddToPool()**.

PCI_MINIMIZE_RESOURCES

This command is not currently implemented. It specifies that **vxbPciAutoConfigLib** minimize requirements for memory and I/O space.

PCI_PSYSTEM_STRUCT_COPY - PCI_AUTO_CONFIG_OPTS * pArg

This command has been added for ease of converting from the old interface to the new one. This will set each value as specified in the pSystem structure. If the **PCI_AUTO_CONFIG_OPTS** structure has already been filled, the **vxbPciAutoConfig(pSystem)** call can be changed to:

```
void *pCookie;  
pCookie = vxbPciAutoConfigLibInit(NULL);  
vxbPciAutoCfgCtl(pCookie, PCI_PSYSTEM_STRUCT_COPY, (void *)pSystem);  
vxbPciAutoCfgFunc(pCookie);
```

The fields of the **PCI_AUTO_CONFIG_OPTS** structure are defined below. For more information about each one, see the paragraphs above and the documentation for **vxbPciAutoConfigLib**.

pciMem32

Specifies the 32-bit prefetchable memory pool base address.

pciMem32Size

Specifies the 32-bit prefetchable memory pool size.

pciMemIo32

Specifies the 32-bit non-prefetchable memory pool base address.

pciMemIo32Size

Specifies the 32-bit non-prefetchable memory pool size

pciIo32

Specifies the 32-bit I/O pool base address.

pciIo32Size

Specifies the 32-bit I/O pool size.

pciIo16

Specifies the 16-bit I/O pool base address.

pciIo16Size

Specifies the 16-bit I/O pool size.

includeRtn

Specifies the device inclusion routine.

intAssignRtn

Specifies the interrupt assignment routine.

autoIntRouting

Can be set to TRUE to configure **vxbPciAutoConfig()** only to call the BSP interrupt routing routine for devices on bus number 0. Setting autoIntRoutine to FALSE will configure **vxbPciAutoConfig()** to call the BSP interrupt routing routine for every device regardless of the bus on which the device resides.

RETURNS OK, or ERROR if the command or argument is invalid.

ERRNO EINVAL
if pCookie is not NULL or cmd is not recognized

SEE ALSO **vxbPci**

vxbPciAutoConfig()

NAME **vxbPciAutoConfig()** – set standard parameters & initialize PCI

SYNOPSIS STATUS vxbPciAutoConfig
(
 VXB_DEVICE_ID busCtrlID
)

DESCRIPTION	This function reads PCI parameters from a PCI bridge controller's HCF record in the BSP-provided hwconf.c file. It calls the appropriate vxbPciAutoConfigLib routines to initialize the PCI bus.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbPci

vxbPciAutoConfigLibInit()

NAME	vxbPciAutoConfigLibInit() – initialize PCI autoconfig library
SYNOPSIS	<pre>void * vxbPciAutoConfigLibInit (void * pArg /* reserved for future use */)</pre>
DESCRIPTION	vxbPciAutoConfigLib initialization function.
RETURNS	A cookie for use by subsequent vxbPciAutoConfigLib function calls.
ERRNO	
SEE ALSO	vxbPci

vxbPciAutoConfigListShow()

NAME	vxbPciAutoConfigListShow() – show the devices in the list
SYNOPSIS	<pre>void vxbPciAutoConfigListShow (PCI_LOC *pLoc, int num)</pre>
DESCRIPTION	This routine shows the devices which are available in the list.

RETURNS OK, or ERROR if pCookie is not valid.

ERRNO

SEE ALSO vxbPci

vxbPciAutoDevReset()

NAME **vxbPciAutoDevReset()** – quiesce a PCI device and reset all writeable status bits

SYNOPSIS

```
STATUS vxbPciAutoDevReset
(
    PCI_AUTO_CONFIG_OPTS * pSystem, /* needed for config info */
    PCI_LOC *             pPciLoc   /* device to be reset */
)
```

DESCRIPTION This routine turns **off** a PCI device by disabling the Memory decoders, I/O decoders, and Bus Master capability. The routine also resets all writeable status bits in the status word that follows the command word sequentially in PCI config space by performing a longword access.

RETURNS OK, always.

ERRNO

SEE ALSO vxbPci

vxbPciAutoFuncDisable()

NAME **vxbPciAutoFuncDisable()** – disable a specific PCI function

SYNOPSIS

```
void vxbPciAutoFuncDisable
(
    PCI_AUTO_CONFIG_OPTS * pSystem,
    PCI_LOC *             *pPciFunc /* input: Pointer to PCI function
struct */
)
```

DESCRIPTION This routine clears the I/O, mem, master, & ROM space enable bits for a single PCI function.

The PCI spec says that devices should normally clear these by default after reset but in actual practice, some PCI devices do not fully comply. This routine ensures that the devices have all been disabled before configuration is started.

RETURNS N/A

ERRNO

SEE ALSO **vxbPci**

vxbPciAutoFuncEnable()

NAME **vxbPciAutoFuncEnable()** – perform final configuration and enable a function

SYNOPSIS

```
void vxbPciAutoFuncEnable
(
    PCI_AUTO_CONFIG_OPTS * pSystem, /* for backwards compatibility */
    PCI_LOC *             pFunc   /* input: Pointer to PCI function
structure */
)
```

DESCRIPTION Depending upon whether the device is included, this routine initializes a single PCI function as follows:

Initialize the cache line size register Initialize the PCI-PCI bridge latency timers Enable the master PCI bit for non-display devices Set the interrupt line value with the value from the BSP.

RETURNS N/A

ERRNO

SEE ALSO **vxbPci**

vxbPciAutoGetNextClass()

NAME **vxbPciAutoGetNextClass()** – find the next device of specific type from probe list

SYNOPSIS

```
STATUS vxbPciAutoGetNextClass
(
    PCI_AUTO_CONFIG_OPTS *pSystem, /* for backwards compatibility */
```

```

PCI_LOC          *pPciFunc, /* output: Contains the BDF of the
device found */
UINT             *index,   /* Zero-based device instance number */
UINT             pciClass, /* class code field from the PCI header
*/
UINT             mask      /* mask is ANDed with the class field */
)

```

DESCRIPTION	The function uses the probe list which was built during the probing process. Using configuration accesses, it searches for the occurrence of the device subject to the class and mask restrictions outlined below. Setting class to zero and mask to zero allows searching the entire set of devices found regardless of class.
RETURNS	TRUE if a device was found, else FALSE.
ERRNO	
SEE ALSO	vxbPci

vxbPciAutoRegConfig()

NAME	vxbPciAutoRegConfig() – assign PCI space to a single PCI base address register
SYNOPSIS	<pre> UINT vxbPciAutoRegConfig (PCI_AUTO_CONFIG_OPTS *pSystem, /* backwards compatibility */ PCI_LOC *pPciFunc, /* Pointer to function in device list */ UINT baseAddr, /* Offset of base PCI address */ UINT nSize, /* Size and alignment requirements */ UINT addrInfo /* PCI address type information */) </pre>
DESCRIPTION	This routine allocates and assigns PCI space (either memory or I/O) to a single PCI base address register.
RETURNS	Returns (1) if BAR supports mapping anywhere in 64-bit address space. Returns (0) otherwise.
ERRNO	
SEE ALSO	vxbPci

vxbPciBusTypeInit()

NAME	vxbPciBusTypeInit() – initialize the PCI bus type
SYNOPSIS	STATUS vxbPciBusTypeInit (struct vxbDev * pDev)
DESCRIPTION	none
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	vxbPciAccess

vxbPciConfigBdfPack()

NAME	vxbPciConfigBdfPack() – pack parameters for the Configuration Address Register
SYNOPSIS	int vxbPciConfigBdfPack (int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo /* function number */)
DESCRIPTION	This routine packs three parameters into one integer for accessing the Configuration Address Register
RETURNS	packed integer encoded version of bus, device, and function numbers.
ERRNO	
SEE ALSO	vxbPci

vxbPciConfigCmdWordShow()

NAME	vxbPciConfigCmdWordShow() – show the decoded value of the command word
-------------	--

SYNOPSIS	<pre>STATUS vxbPciConfigCmdWordShow (VXB_DEVICE_ID busCtrlID, int bus, /* bus */ int device, /* device */ int function, /* function */ void *pArg /* ignored */)</pre>
DESCRIPTION	This routine reads the value of the command word for the specified bus, device, function and displays the information.
RETURNS	OK, always.
ERRNO	
SEE ALSO	vxbPci

vxbPciConfigExtCapFind()

NAME	vxbPciConfigExtCapFind() – find extended capability in ECP linked list
SYNOPSIS	<pre>STATUS vxbPciConfigExtCapFind (VXB_DEVICE_ID busCtrlID, UINT8 extCapFindId, /* Extended capabilities ID to search for */ int bus, /* PCI bus number */ int device, /* PCI device number */ int function, /* PCI function number */ UINT8 * pOffset /* returned config space offset */)</pre>
DESCRIPTION	This routine searches for an extended capability in the linked list of capabilities in config space. If found, the offset of the first byte of the capability of interest in config space is returned via pOffset.
RETURNS	OK if Extended Capability found, ERROR otherwise
ERRNO	
SEE ALSO	vxbPci

vxbPciConfigForeachFunc()

NAME	vxbPciConfigForeachFunc() – check condition on specified bus
SYNOPSIS	<pre>STATUS vxbPciConfigForeachFunc (VXB_DEVICE_ID busCtrlID, UINT8 bus, /* bus to start on */ BOOL recurse, /* if TRUE, do subordinate busses */ VXB_PCI_FOREACH_FUNC funcCheckRtn, /* routine to call for each PCI func */ void *pArg /* argument to funcCheckRtn */)</pre>
DESCRIPTION	<p>vxbPciConfigForeachFunc() discovers the PCI functions present on the bus and calls a specified C-function for each one. If the function returns ERROR, further processing stops.</p> <p>vxbPciConfigForeachFunc() does not affect any HOST-PCI bridge on the system.</p>
RETURNS	OK normally, or ERROR if funcCheckRtn() doesn't return OK.
ERRNO	not set
SEE ALSO	vxbPci

vxbPciConfigFuncShow()

NAME	vxbPciConfigFuncShow() – show configuration details about a function
SYNOPSIS	<pre>STATUS vxbPciConfigFuncShow (VXB_DEVICE_ID busCtrlID, int bus, /* bus */ int device, /* device */ int function, /* function */ void *pArg /* ignored */)</pre>
DESCRIPTION	This routine reads various information from the specified bus, device, function, and displays the information.
RETURNS	OK, always.

ERRNO

SEE ALSO [vxbPci](#)

vxbPciConfigInByte()

NAME [vxbPciConfigInByte\(\)](#) – read one byte from the PCI configuration space

SYNOPSIS

```
STATUS vxbPciConfigInByte
(
    VXB_DEVICE_ID busCtrlID,
    int           busNo,          /* bus number */
    int           deviceNo,       /* device number */
    int           funcNo,         /* function number */
    int           offset,         /* offset into the configuration space */
    UINT8 *      pData,          /* data read from the offset */
)
```

DESCRIPTION This routine reads one byte from the PCI configuration space

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO [vxbPci](#)

vxbPciConfigInLong()

NAME [vxbPciConfigInLong\(\)](#) – read one longword from the PCI configuration space

SYNOPSIS

```
STATUS vxbPciConfigInLong
(
    VXB_DEVICE_ID busCtrlID,
    int           busNo,          /* bus number */
    int           deviceNo,       /* device number */
    int           funcNo,         /* function number */
    int           offset,         /* offset into the configuration space */
    UINT32 *      pData,          /* data read from the offset */
)
```

DESCRIPTION This routine reads one longword from the PCI configuration space

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO vxbPci

vxbPciConfigInWord()

NAME vxbPciConfigInWord() – read one word from the PCI configuration space

SYNOPSIS STATUS vxbPciConfigInWord(
 (
 VXB_DEVICE_ID busCtrlID,
 int busNo, /* bus number */
 int deviceNo, /* device number */
 int funcNo, /* function number */
 int offset, /* offset into the configuration space */
 UINT16 * pData /* data read from the offset */
))

DESCRIPTION This routine reads one word from the PCI configuration space

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO vxbPci

vxbPciConfigModifyByte()

NAME vxbPciConfigModifyByte() – Perform a masked longword register update

SYNOPSIS STATUS vxbPciConfigModifyByte(
 (
 VXB_DEVICE_ID busCtrlID,
 int busNo, /* bus number */
 int deviceNo, /* device number */
 int funcNo, /* function number */
 int offset, /* offset into the configuration space */
 UINT8 bitMask, /* Mask which defines field to alter */
 UINT8 data /* data written to the offset */
))

DESCRIPTION

This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the **bitMask** parameter.

Do not use this routine to modify any register that contains 'write 1 to clear' type of status bits in the same longword. This specifically applies to the command register. Modify byte operations could potentially be implemented as longword operations with bit shifting and masking. This could have the effect of clearing status bits in registers that aren't being updated. Use vxbPciConfigInLong and vxbPciConfigOutLong, or pciModifyLong, to read and update the entire longword.

RETURNS

OK if operation succeeds, **ERROR** if operation fails.

ERRNO**SEE ALSO**

vxbPci

vxbPciConfigModifyLong()

NAME

vxbPciConfigModifyLong() – Perform a masked longword register update

SYNOPSIS

```
STATUS vxbPciConfigModifyLong
(
    VXB_DEVICE_ID busCtrlID,
    int           busNo,      /* bus number */
    int           deviceNo,   /* device number */
    int           funcNo,    /* function number */
    int           offset,    /* offset into the configuration space */
    UINT32        bitMask,   /* Mask which defines field to alter */
    UINT32        data       /* data written to the offset */
)
```

DESCRIPTION

This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the **bitMask** parameter.

Be careful to using vxbPciConfigModifyLong for updating the Command and status register. The status bits must be written back as zeroes, else they will be cleared. Proper use

involves including the status bits in the mask value, but setting their value to zero in the data value.

The following example will set the PCI_CMD_IO_ENABLE bit without clearing any status bits. The macro PCI_CMD_MASK includes all the status bits as part of the mask. The fact that PCI_CMD_MASTER doesn't include these bits, causes them to be written back as zeroes, therefore they aren't cleared.

```
vxbPciConfigModifyLong (b, d, f, PCI_CFG_COMMAND,  
                      (PCI_CMD_MASK | PCI_CMD_IO_ENABLE), PCI_CMD_IO_ENABLE);
```

Use of explicit longword read and write operations for dealing with any register containing "write 1 to clear" bits is sound policy.

RETURNS OK if operation succeeds, **ERROR** if operation fails.

ERRNO

SEE ALSO vxbPci

vxbPciConfigModifyWord()

NAME vxbPciConfigModifyWord() – Perform a masked longword register update

SYNOPSIS

```
STATUS vxbPciConfigModifyWord  
(  
    VXB_DEVICE_ID busCtrlID,  
    int           busNo,      /* bus number */  
    int           deviceNo,   /* device number */  
    int           funcNo,    /* function number */  
    int           offset,    /* offset into the configuration space */  
    UINT16        bitMask,   /* Mask which defines field to alter */  
    UINT16        data       /* data written to the offset */  
)
```

DESCRIPTION This function writes a field into a PCI configuration header without altering any bits not present in the field. It does this by first doing a PCI configuration read (into a temporary location) of the PCI configuration header word which contains the field to be altered. It then alters the bits in the temporary location to match the desired value of the field. It then writes back the temporary location with a configuration write. All configuration accesses are long and the field to alter is specified by the "1" bits in the **bitMask** parameter.

Do not use this routine to modify any register that contains 'write 1 to clear' type of status bits in the same longword. This specifically applies to the command register. Modify byte operations could potentially be implemented as longword operations with bit shifting and masking. This could have the effect of clearing status bits in registers that aren't being

updated. Use vxbPciConfigInLong and vxbPciConfigOutLong, or pciModifyLong, to read and update the entire longword.

RETURNS OK if operation succeeds. ERROR if operation fails.

ERRNO

SEE ALSO vxbPci

vxbPciConfigOutByte()

NAME vxbPciConfigOutByte() – write one byte to the PCI configuration space

SYNOPSIS

```
STATUS vxbPciConfigOutByte
(
    VXB_DEVICE_ID busCtrlID,
    int           busNo,          /* bus number */
    int           deviceNo,       /* device number */
    int           funcNo,         /* function number */
    int           offset,         /* offset into the configuration space */
    UINT8         data,          /* data written to the offset */
)
```

DESCRIPTION This routine writes one byte to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO vxbPci

vxbPciConfigOutLong()

NAME vxbPciConfigOutLong() – write one longword to the PCI configuration space

SYNOPSIS

```
STATUS vxbPciConfigOutLong
(
    VXB_DEVICE_ID busCtrlID,
    int           busNo,          /* bus number */
    int           deviceNo,       /* device number */
    int           funcNo,         /* function number */
    int           offset,         /* offset into the configuration space */
```

```
    UINT32          data      /* data written to the offset */  
)
```

DESCRIPTION This routine writes one longword to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO vxbPci

vxbPciConfigOutWord()

NAME vxbPciConfigOutWord() – write one 16-bit word to the PCI configuration space

SYNOPSIS STATUS vxbPciConfigOutWord
(
 VXB_DEVICE_ID busCtrlID,
 int busNo, /* bus number */
 int deviceNo, /* device number */
 int funcNo, /* function number */
 int offset, /* offset into the configuration space */
 UINT16 data, /* data written to the offset */
)

DESCRIPTION This routine writes one 16-bit word to the PCI configuration space.

RETURNS OK, or ERROR if this library is not initialized

ERRNO

SEE ALSO vxbPci

vxbPciConfigReset()

NAME vxbPciConfigReset() – disable cards for warm boot

SYNOPSIS STATUS vxbPciConfigReset
(
 VXB_DEVICE_ID busCtrlID,
 int startType /* for reboot hook, ignored */
)

DESCRIPTION	vxbPciConfigReset() goes through the list of PCI functions at the top-level bus and disables them, preventing them from writing to memory while the system is trying to reboot.
RETURNS	OK, always
ERRNO	Not set
SEE ALSO	vxbPci

vxbPciConfigStatusWordShow()

NAME	vxbPciConfigStatusWordShow() – show the decoded value of the status word
SYNOPSIS	STATUS vxbPciConfigStatusWordShow (VXB_DEVICE_ID busCtrlID, int bus, /* bus */ int device, /* device */ int function, /* function */ void *pArg /* ignored */)
DESCRIPTION	This routine reads the value of the status word for the specified bus, device, function and displays the information.
RETURNS	OK, always.
ERRNO	
SEE ALSO	vxbPci

vxbPciConfigTopoShow()

NAME	vxbPciConfigTopoShow() – show PCI topology
SYNOPSIS	void vxbPciConfigTopoShow (VXB_DEVICE_ID busCtrlID)

DESCRIPTION	This routine traverses the PCI bus and prints assorted information about every device found. The information is intended to present the topology of the PCI bus. It includes: (1) the device type, (2) the command and status words, (3) for PCI to PCI bridges the memory and I/O space configuration, and (4) the values of all implemented BARs.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPci

vxbPciDevConfig()

NAME	vxbPciDevConfig() – configure a device on a PCI bus
SYNOPSIS	<pre>STATUS vxbPciDevConfig (VXB_DEVICE_ID busCtrlID, int pciBusNo, /* PCI bus number */ int pciDevNo, /* PCI device number */ int pciFuncNo, /* PCI function number */ UINT32 devIoBaseAdrs, /* device IO base address */ UINT32 devMemBaseAdrs, /* device memory base address */ UINT32 command /* command to issue */)</pre>
DESCRIPTION	This routine configures a device that is on a Peripheral Component Interconnect (PCI) bus by writing to the configuration header of the selected device. It first disables the device by clearing the command register in the configuration header. It then sets the I/O and/or memory space base address registers, the latency timer value and the cache line size. Finally, it re-enables the device by loading the command register with the specified command.
NOTE	This routine is designed for Type 0 PCI Configuration Headers ONLY. It is NOT usable for configuring, for example, a PCI-to-PCI bridge.
RETURNS	OK always.
ERRNO	
SEE ALSO	vxbPci

vxbPciDeviceAnnounce()

NAME	vxbPciDeviceAnnounce() – Update vxBus to PCI bus orphans
SYNOPSIS	STATUS vxbPciDeviceAnnounce (VXB_DEVICE_ID busCtrlID)
DESCRIPTION	none
RETURNS	OK
ERRNO	
SEE ALSO	vxbPci

vxbPciDeviceShow()

NAME	vxbPciDeviceShow() – print information about PCI devices
SYNOPSIS	STATUS vxbPciDeviceShow (VXB_DEVICE_ID busCtrlID, int busNo /* bus number */)
DESCRIPTION	This routine prints information about the PCI devices on a given PCI bus segment (specified by <i>busNo</i>).
RETURNS	OK, or ERROR if the library is not initialized.
ERRNO	
SEE ALSO	vxbPci

vxbPciFindClass()

NAME	vxbPciFindClass() – find the nth occurrence of a device by PCI class code.
-------------	--

SYNOPSIS

```
STATUS vxbPciFindClass
(
    VXB_DEVICE_ID busCtrlID,
    int           classCode, /* 24-bit class code */
    int           index,    /* desired instance of device */
    int *         pBusNo,   /* bus number */
    int *         pDeviceNo,/* device number */
    int *         pFuncNo,  /* function number */
)
```

DESCRIPTION

This routine finds the nth device with the given 24-bit PCI class code (class subclass prog_if).

The classcode arg of must be carefully constructed from class and sub-class macros.

Example : To find an ethernet class device, construct the classcode arg as follows:

```
((PCI_CLASS_NETWORK_CTRL << 16 | PCI_SUBCLASS_NET_ETHERNET << 8))
```

RETURNS

OK, or ERROR if the class didn't match.

ERRNO**SEE ALSO**

[vxbPci](#)

vxbPciFindClassShow()

NAME

vxbPciFindClassShow() – find a device by 24-bit class code

SYNOPSIS

```
STATUS vxbPciFindClassShow
(
    VXB_DEVICE_ID busCtrlID,
    int           classCode, /* 24-bit class code */
    int           index,    /* desired instance of device */
)
```

DESCRIPTION

This routine finds a device by its 24-bit PCI class code, then prints its information.

RETURNS

OK, or ERROR if this library is not initialized.

ERRNO**SEE ALSO**

[vxbPci](#)

vxbPciFindDevice()

NAME	vxbPciFindDevice() – find the nth device with the given device & vendor ID
SYNOPSIS	<pre>STATUS vxbPciFindDevice (VXB_DEVICE_ID busCtrlID, int vendorId, /* vendor ID */ int deviceId, /* device ID */ int index, /* desired instance of device */ int * pBusNo, /* bus number */ int * pDeviceNo, /* device number */ int * pFuncNo /* function number */)</pre>
DESCRIPTION	This routine finds the nth device with the given device & vendor ID.
RETURNS	OK, or ERROR if the deviceId and vendorId didn't match.
ERRNO	
SEE ALSO	vxbPci

vxbPciFindDeviceShow()

NAME	vxbPciFindDeviceShow() – find a PCI device and display the information
SYNOPSIS	<pre>STATUS vxbPciFindDeviceShow (VXB_DEVICE_ID busCtrlID, int vendorId, /* vendor ID */ int deviceId, /* device ID */ int index, /* desired instance of device */)</pre>
DESCRIPTION	This routine finds a device by deviceId, then displays the information.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	vxbPci

vxbPciHeaderShow()

NAME	vxbPciHeaderShow() – print a header of the specified PCI device
SYNOPSIS	STATUS vxbPciHeaderShow (VXB_DEVICE_ID busCtrlID, int busNo, /* bus number */ int deviceNo, /* device number */ int funcNo /* function number */)
DESCRIPTION	This routine prints a header of the PCI device specified by busNo, deviceNo, and funcNo.
RETURNS	OK, or ERROR if this library is not initialized.
ERRNO	
SEE ALSO	vxbPci

vxbPciInt()

NAME	vxbPciInt() – interrupt handler for shared PCI interrupt.
SYNOPSIS	VOID vxbPciInt (struct vxbPciIntHandlerInfo *pIntHandlerInfo)
DESCRIPTION	This routine executes multiple interrupt handlers for a PCI interrupt. Each interrupt handler must check the device dependent interrupt status bit to determine the source of the interrupt, since it simply execute all interrupt handlers in the link list. This is not a user callable routine
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPci

vxbPciIntConnect()

NAME	vxbPciIntConnect() – connect the interrupt handler to the PCI interrupt.
SYNOPSIS	<pre>STATUS vxbPciIntConnect (VXB_DEVICE_ID pDev, VOIDFUNC PTR *vector, /* interrupt vector to attach to */ VOIDFUNC PTR routine, /* routine to be called */ int parameter /* parameter to be passed to routine */)</pre>
DESCRIPTION	This routine connects an interrupt handler to a shared PCI interrupt vector. A link list is created for each shared interrupt used in the system. It is created when the first interrupt handler is attached to the vector. Subsequent calls to vxbPciIntConnect just add their routines to the linked list for that vector.
RETURNS	OK, or ERROR if the interrupt handler cannot be built.
ERRNO	
SEE ALSO	vxbPci

vxbPciIntDisconnect()

NAME	vxbPciIntDisconnect() – disconnect the interrupt handler (OBSOLETE)
SYNOPSIS	<pre>STATUS vxbPciIntDisconnect (VXB_DEVICE_ID pDev, VOIDFUNC PTR *vector, /* interrupt vector to attach to */ VOIDFUNC PTR routine /* routine to be called */)</pre>
DESCRIPTION	This routine disconnects the interrupt handler from the PCI interrupt line. In a system where one driver and one ISR services multiple devices, this routine removes all instances of the ISR because it completely ignores the parameter argument used to install the handler.
NOTE	Use of this routine is discouraged and will be obsoleted in the future. New code should use the vxbPciIntDisconnect2() routine instead.
RETURNS	OK, or ERROR if the interrupt handler cannot be removed.

ERRNO

SEE ALSO

vxbPci

vxbPciIntDisconnect2()

NAME **vxbPciIntDisconnect2()** – disconnect an interrupt handler from the PCI interrupt.

SYNOPSIS

```
STATUS vxbPciIntDisconnect2
(
    VXB_DEVICE_ID pDev,
    VOIDFUNC PTR *vector, /* interrupt vector to attach to */ */
    VOIDFUNC PTR routine, /* routine to be called */ */
    int parameter /* routine parameter */ */
)
```

DESCRIPTION This routine disconnects a single instance of an interrupt handler from the PCI interrupt line.

NOTE This routine should be used in preference to the original **pciIntDisconnect()** routine. This routine is compatible with drivers that are managing multiple device instances, using the same basic ISR, but with different parameters.

RETURNS OK, or ERROR if the interrupt handler cannot be removed.

ERRNO

SEE ALSO

vxbPci

vxbPciIntLibInit()

NAME **vxbPciIntLibInit()** – initialize the **vxbPciIntLib** module

SYNOPSIS

```
STATUS vxbPciIntLibInit
(
    struct vxbPciInt * pIntInfo
)
```

DESCRIPTION This routine initializes the linked lists used to chain together the PCI interrupt service routines.

RETURNS OK, or ERROR upon link list failures.

ERRNO

SEE ALSO [vxbPci](#)

vxbPlbAccessCopy()

NAME **vxbPlbAccessCopy()** – copy the access data structure

SYNOPSIS

```
void vxbPlbAccessCopy
(
    struct vxbAccessList *pAccess
)
```

DESCRIPTION This routine is used to copy the contents of the PLB access array onto the parameter

RETURNS N/A

ERRNO

SEE ALSO [vxbPlbAccess](#)

vxbPresStructShow()

NAME **vxbPresStructShow()** – Show bus information

SYNOPSIS

```
STATUS vxbPresStructShow
(
    struct vxbBusPresent * pPres
)
```

DESCRIPTION This routine prints information about the specified bus.

RETURNS N/A

ERRNO

SEE ALSO [vxbShow](#)

vxbPrimeCellSioRegister()

NAME	vxbPrimeCellSioRegister() – register vxbAmBaSio driver
SYNOPSIS	<code>void vxbPrimeCellSioRegister (void)</code>
DESCRIPTION	This routine registers the vxbAmBaSio driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A
ERRNO	
SEE ALSO	vxbPrimeCellSio

vxbQeIntCtlrRegister()

NAME	vxbQeIntCtlrRegister() – register qeIntCtlr driver
SYNOPSIS	<code>void vxbQeIntCtlrRegister(void)</code>
DESCRIPTION	This routine registers the EPCI driver and device recognition data with the vxBus subsystem.
RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbQeIntCtlr

vxbQeIntCtlrpDrvCtrlShow()

NAME	vxbQeIntCtlrpDrvCtrlShow() – show pDrvCtrl for template controller
SYNOPSIS	<code>int vxbQeIntCtlrpDrvCtrlShow</code>

```
(  
    VXB_DEVICE_ID pIntCtlr,  
    int           verboseLevel  
)
```

DESCRIPTION none

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbQeIntCtlr](#)

vxbQuiccIntCtlrRegister()

NAME [vxbQuiccIntCtlrRegister\(\)](#) – register quiccIntCtlr driver

SYNOPSIS void vxbQuiccIntCtlrRegister(void)

DESCRIPTION This routine registers the EPCI driver and device recognition data with the vxBus subsystem.

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbQuiccIntCtlr](#)

vxbQuiccIntCtlrpDrvCtrlShow()

NAME [vxbQuiccIntCtlrpDrvCtrlShow\(\)](#) – show pDrvCtrl for template controller

SYNOPSIS int vxbQuiccIntCtlrpDrvCtrlShow
(
 VXB_DEVICE_ID pIntCtlr,
 int verboseLevel
)

DESCRIPTION none

RETURNS	Not Available
ERRNO	Not Available
SEE ALSO	vxbQuiccIntCtlr

vxbR4KTimerDrvRegister()

NAME	vxbR4KTimerDrvRegister() – register mips R4K timer driver
SYNOPSIS	<code>void vxbR4KTimerDrvRegister (void)</code>
DESCRIPTION	This routine registers the mips R4k timer driver with the vxBus subsystem.
RETURNS	N/A
ERRNO	none
SEE ALSO	vxbMipsR4KTimer

vxbRapidIOBusTypeInit()

NAME	vxbRapidIOBusTypeInit() – initialize RapidIO bus type
SYNOPSIS	<code>STATUS vxbRapidIOBusTypeInit (VXB_DEVICE_ID pInst)</code>
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	vxbRapidIO

vxbRapidIOCfg()

NAME vxbRapidIOCfg() – RapidIO Table-based device enumeration

SYNOPSIS

```
int vxbRapidIOCfg
(
    VXB_DEVICE_ID      pParent,          /* parent bus controller */
    VXB_ACCESS_LIST * pAccess,           /* device access API */
    STATUS             (*busOverride)   /* bus-specific init rtn */
    (VXB_DEVICE_ID     pDev)            /* (init rtn arg) */
)
```

DESCRIPTION This routine causes RapidIO device information to be read from hcfDeviceList[] in order to enumerate RapidIO devices.

RETURNS: none

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO vxbRapidIOCfgTable

vxbRapidIOTableOverride()

NAME vxbRapidIOTableOverride() – RapidIO Table-based override function

SYNOPSIS

```
STATUS vxbRapidIOHcfTableOverride
(
    VXB_DEVICE_ID pDev
)
```

DESCRIPTION This routine extracts entries from the hcfResources table and puts them in the device identification information. Drivers may override this function, but if they do so, they must perform this action, which they can do by calling this function.

This function expects to find the hcfDevices table pointer in the pBusSpecificDeviceInfo entry of the **VXB_DEVICE** structure. It replaces this pointer with a **RAPIDIO_DEVICE_INFORMATION** pointer.

RETURNS: none

ERRNO: not set

RETURNS Not Available

ERRNO Not Available

SEE ALSO [vxbRapidIOCfgTable](#)

vxbRegMap()

NAME **vxbRegMap()** – obtain an access handle for a given register mapping

SYNOPSIS STATUS vxbRegMap
(
 VXB_DEVICE_ID pDev,
 int index,
 void ** ppHandle
)

DESCRIPTION This function returns a handle suitable for accessing a given BAR with the **vxbReadXX()**/**vxbWriteXX()** register access API. The handle is opaque to the caller, but is used by the API to determine if the BAR is I/O mapped, memory mapped, or some special case so that it can choose which access method to use.

The **vxbRegMap()** routine will first try to query the parent bus controller of a given device to see if it has a custom mapping routine. If not, an attempt is made to return a reasonable default handle value.

RETURNS OK the BAR can be mapped to a reasonable handle, otherwise **ERROR**

ERRNO N/A

SEE ALSO [vxbArchAccess](#)

vxbResourceFind()

NAME	vxbResourceFind() – find and allocate a vxBus resource
SYNOPSIS	<pre>STATUS vxbResourceFind (struct vxbDev * instance, UINT32 method, char * pArg, UINT flags)</pre>
DESCRIPTION	<p>This routine searches for a vxBus resource to allocate. Resources are identified by a driver method. The driver method, when called, must allocate and initialize the specified resource, fill in the appropriate fields in the structure pointed to by the pArg variable, and return OK. The flags are currently ignored. The flags field is present for a future enhancement to allow different behaviors:</p> <ul style="list-style-type: none">1) Check only PLB2) Check only bus on which device resides3) Check all devices if none found on direct path
RETURNS	OK , or ERROR if the resource could not be allocated
ERRNO	
SEE ALSO	vxBus

vxbSb1DuartSioRegister()

NAME	vxbSb1DuartSioRegister() – register vxbSb1DuartSio driver
SYNOPSIS	<pre>void vxbSb1DuartSioRegister (void)</pre>
DESCRIPTION	This routine registers the vxbSb1DuartSio driver and device recognition data with the vxBus subsystem.
NOTE	This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.
RETURNS	N/A

ERRNO

SEE ALSO [vxbSb1DuartSio](#)

2

vxbSb1TimerDrvRegister()

NAME [vxbSb1TimerDrvRegister\(\)](#) – register sb1 timer driver

SYNOPSIS void vxbSb1TimerDrvRegister (void)

DESCRIPTION This routine registers the sb1 timer driver with the vxBus subsystem.

RETURNS N/A

ERRNO none

SEE ALSO [vxbSb1Timer](#)

vxbSmcFdc37xRegister()

NAME [vxbSmcFdc37xRegister\(\)](#) – register smcFdc37x driver

SYNOPSIS void vxbSmcFdc37xRegister(void)

DESCRIPTION This routine registers the smcFdc37x driver and device recognition data with the vxBus subsystem.

NOTE This routine is called early during system initialization, and *MUST NOT* make calls to OS facilities such as memory allocation and I/O.

RETURNS N/A

ERRNO

SEE ALSO [vxbSmcFdc37x](#)

vxbSubDevAction()

NAME	vxbSubDevAction() – perform an action on all devs on bus controller
SYNOPSIS	<pre>STATUS vxbSubDevAction (struct vxbDev * pBusCtrlr, VXB_SUBDEV_ACTION_FUNC actionFunc, char * pArg, UINT32 flags)</pre>
DESCRIPTION	This routine performs the specified action (that is, calls the specified function) for each device directly connected to the bus downstream from the specified bus controller.
RETURNS	OK, or ERROR
ERRNO	
SEE ALSO	vxBus

vxbSysClkLibInit()

NAME	vxbSysClkLibInit() – initialize the system clock library
SYNOPSIS	<pre>STATUS vxbSysClkLibInit (void)</pre>
DESCRIPTION	This routine initializes the system clock library by selecting the timer which is best suited for use as system clock from the timers available in the system.
RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	vxbSysClkLib

vxbSysClkShow()

NAME	vxbSysClkShow() – show the system clock information
-------------	---

SYNOPSIS	void vxbSysClkShow (void)
DESCRIPTION	This routine is used to display the system clock details
RETURNS	None.
ERRNO	Not Available
SEE ALSO	vxbSysClkLib

vxbSysQeIntDisable()

NAME	vxbSysQeIntDisable() – Disable one of the Level or IRQ interrupts into the SIU
SYNOPSIS	int vxbSysQeIntDisable (int intNum /* interrupt level to disable */)
DESCRIPTION	This routine will mask the bit in the SIMASK register corresponding to the requested interrupt level.
RETURNS	0, always.
ERRNO	Not Available
SEE ALSO	vxbQeIntCtlr

vxbSysQeIntEnable()

NAME	vxbSysQeIntEnable() – enable the indicated interrupt
SYNOPSIS	int vxbSysQeIntEnable (int intNum /* interrupt level to enable */)
DESCRIPTION	This routine will enable the indicated interrupt by setting the appropriate bit in the SIU Interrupt Mask Registers. The design of the 8260 presents the following design requirements:

1. the mapping from interrupt number to mask bit can not be represented by a function. An array, indexed by interrupt number (QE_INUM), is used to map the interrupt number to the appropriate mask.
2. There are two 32 bit mask registers (CIMR and CRIMR). The interrupt number must be compared to 4 ranges to determine which register contains its mask bit:

interrupt number in vxbQeIntCtlr.h	register
-----	-----
0-15	CIMR
16-31	CRIMR
32-47	CIMR
48-63	CRIMR

RETURNS 0, always.

ERRNO Not Available

SEE ALSO [vxbQeIntCtlr](#)

vxbSysQuiccInit()

NAME [**vxbSysQuiccInit\(\)**](#) – initialize the interrupt manager for the PowerPC 83XX

SYNOPSIS STATUS vxbSysQuiccInit
(
 VXB_DEVICE_ID pIntCtlr
)

DESCRIPTION Disables interrupts for initialization

RETURNS OK always

ERRNO

SEE ALSO [vxbQuiccIntCtlr](#)

vxbSysQuiccIntDisable()

NAME	<i>vxbSysQuiccIntDisable()</i> – Disable one of the Level or IRQ interrupts into the SIU
SYNOPSIS	int vxbSysQuiccIntDisable (int intNum /* interrupt level to disable */)
DESCRIPTION	This routine will mask the bit in the SIMASK register corresponding to the requested interrupt level.
RETURNS	0, always.
ERRNO	
SEE ALSO	<i>vxbQuiccIntCtlr</i>

vxbSysQuiccIntEnable()

NAME	<i>vxbSysQuiccIntEnable()</i> – enable the indicated interrupt
SYNOPSIS	int vxbSysQuiccIntEnable (int intNum /* interrupt level to enable */)
DESCRIPTION	This routine will enable the indicated interrupt by setting the appropriate bit in the SIU Interrupt Mask Registers. The design of the 83XX presents the following design requirements: 1. the mapping from interrupt number to mask bit can not be represented by a function. An array, indexed by interrupt number (INUM), is used to map the interrupt number to the appropriate mask. 2. There are two 32 bit mask registers (SIMR_L and SIMR_H). The interrupt number must be compared to 4 ranges to determine which register contains its mask bit:

interrupt number in quiccIntrCtl.h	register
-----	-----

0-15	SIMR_L
16-31	SIMR_H
32-47	SIMR_L
48-63	SIMR_H

RETURNS 0, always.

ERRNO Not Available

SEE ALSO vxbQuiccIntCtlr

vxbTimerAlloc()

NAME **vxbTimerAlloc()** – allocate timer for the requested characteristics

SYNOPSIS

```
timerHandle_t vxbTimerAlloc
(
    UINT32          freq,           /* requested frequency */
    UINT32          minFreq,        /* max acceptable min freq */
    UINT32          maxFreq,        /* min acceptable max freq */
    UINT32          features,       /* timer features */
    UINT32          featureMask,   /* timer features specified */
    void    (*pIsr)(int arg),     /* ISR */
    int              arg            /* argument to ISR */
)
```

DESCRIPTION This routine is used to allocate a timer which has the frequency and the features specified. If a timer of the requested frequency is not available, the timer which has the minimum frequency which is greater than **minFreq** and maximum frequency which is lesser than **maxFreq** will be selected.

RETURNS valid timerHandle if timer is allocated, else NULL

ERRNO Not Available

SEE ALSO **vxbTimerLib**

vxbTimerFeaturesGet()

NAME **vxbTimerFeaturesGet()** – get the timer features

SYNOPSIS

```
STATUS vxbTimerFeaturesGet
(
    timerHandle_t timer,
    UINT32 *      pMinFreq,
    UINT32 *      pMaxFreq,
    UINT32 *      pFeatures
)
```

DESCRIPTION This routine is used to determine the minimum frequency, maximum frequency, and features flags of the specified timer. Any of the pointer parameters may be set to NULL if the information is not required.

RETURNS OK on success, ERROR otherwise

ERRNO Not Available

SEE ALSO **vxbTimerLib**

vxbTimerLibInit()

NAME **vxbTimerLibInit()** – initialize the timer library

SYNOPSIS

```
void vxbTimerLibInit (void)
```

DESCRIPTION This routine initializes the timer library. Currently, this routine does nothing.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **vxbTimerLib**

vxbTimerRelease()

NAME **vxbTimerRelease()** – release the allocated timer

SYNOPSIS

```
void vxbTimerRelease
(
    timerHandle_t timer
)
```

DESCRIPTION	This routine is used to release a timer which was allocated earlier using vxbTimerAlloc.
RETURNS	None
ERRNO	Not Available
SEE ALSO	vxbTimerLib

vxbTimerStubInit()

NAME	vxbTimerStubInit() – initialization function for the timer stub
SYNOPSIS	<code>void vxbTimerStubInit (void)</code>
DESCRIPTION	This routine updates the _Delay and _us_Delay function pointers with sysDelay and sysUsDelay () functions respectively.
RETURNS	N/A.
ERRNO	Not Available
SEE ALSO	vxbTimerStub

vxbTimestampCookieGet()

NAME	vxbTimestampCookieGet() – to get the cookie information
SYNOPSIS	<code>VOID * vxbTimestampCookieGet (void)</code>
DESCRIPTION	This routine returns the timestamp cookie information. This function is only for test purpose.
RETURNS	<i>pTimestampCookie</i>
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

vxbTimestampLibInit()

NAME	vxbTimestampLibInit() – initialize the timestamp library
SYNOPSIS	STATUS vxbTimestampLibInit (void)
DESCRIPTION	This routine initializes the timestamp library by selecting the timer which is best suited for use as timestamp timer from the timers available in the system.
RETURNS	OK or ERROR.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

vxbTimestampShow()

NAME	vxbTimestampShow() – show the timestamp information
SYNOPSIS	void vxbTimestampShow (void)
DESCRIPTION	This routine is used to display the timestamp clock details
RETURNS	None.
ERRNO	Not Available
SEE ALSO	vxbTimestampLib

vxbTopoShow()

NAME	vxbTopoShow() – Show bus topology
SYNOPSIS	int vxbTopoShow(void)
DESCRIPTION	This routine prints information about the bus hierarchy

First, it shows the bus type for the top-level bus controller, then it recursively goes through the devices on each bus, and shows some information about each one.

RETURNS N/A

ERRNO

SEE ALSO [vxbShow](#)

vxbUpdateDeviceInfo()

NAME **vxbUpdateDeviceInfo()** – Update vxBus device Information

SYNOPSIS STATUS vxbUpdateDeviceInfo
(
 VXB_DEVICE_ID childID,
 VXB_DEVICE_ID busCtrlID,
 char * pArg
)

DESCRIPTION This routine updates the vxbus information on the device, specifically BARs and interrupt info.

RETURNS OK

ERRNO

SEE ALSO [vxbPci](#)

vxbUsDelay()

NAME **vxbUsDelay()** – delay for **delayTime** microseconds

SYNOPSIS void vxbUsDelay
(
 int delayTime
)

DESCRIPTION This routine introduces a busy wait of **delayTime** microseconds.

RETURNS N/A.

ERRNO Not Available

SEE ALSO **vxbDelayLib**

vxbVolRegWrite()

NAME **vxbVolRegWrite()** – volatile register writes

SYNOPSIS

```
STATUS vxbVolRegWrite
(
    struct vxbDev * pDev,           /* Device Information */
    UINT32      regBaseIndex,      /* which pRegBase to use */
    UINT32      byteOffset,        /* offset, in bytes, of register */
    UINT32      transactionSize,   /* transaction size, in bytes */
    void *      pDataBuf,          /* buffer to read-from/write-to */
    UINT32 *    pFlags            /* flags */
)
```

DESCRIPTION This routine performs volatile register writes of the size specified in transactionSize.

RETURNS OK, or ERROR

ERRNO

SEE ALSO **vxBus**

vxbVxSimIntCtlrRegister()

NAME **vxbVxSimIntCtlrRegister()** – register vxbVxSimIntCtlr driver

SYNOPSIS

```
void vxbVxSimIntCtlrRegister(void)
```

DESCRIPTION This routine registers the vxbVxSimIntCtlr driver and device recognition data with the vxBus subsystem.

RETURNS N/A

ERRNO

SEE ALSO **vxbVxSimIpi**

wancomEndDbg()

NAME	wancomEndDbg() – Print pDrvCtrl information regarding Tx ring and Rx queue desc.
SYNOPSIS	void wancomEndDbg (WANCOM_DRV_CTRL * pDrvCtrl /* pointer to WANCOM_DRV_CTRL structure */)
DESCRIPTION	none
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	wancomEnd

wancomEndLoad()

NAME	wancomEndLoad() – initialize the driver and device
SYNOPSIS	END_OBJ* wancomEndLoad (char *initString /* parameter string */)
DESCRIPTION	This routine initializes both, driver and device to an operational state using device specific parameters specified by <i>initString</i> . The parameter string, <i>initString</i> , is an ordered list of parameters each separated by a colon. The format of <i>initString</i> is, "memBase:memSize:nCFDs:nRFDs:flags" The GT642xx shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources. A default number of transmit/receive frames of 32 can be selected by passing zero in the parameters <i>nTfds</i> and <i>nRfds</i> . In other cases, the number of frames selected should be greater than two. The <i>memBase</i> parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The <i>memSize</i> parameter is used to

check that this region is large enough with respect to the provided values of both transmit/receive descriptors.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use **cacheDmaMalloc()** to obtain some non-cacheable memory.

The *flags* parameter is used to select if packet is copied to one buffer when CFDs are short of multiple fragmented data sent through multiple CFDs and at least one CFD is available which can be used to transfer the packet with copying the fragmented data to the buffer. Setting the bit 0 enables this capability and requires 1536 bytes (depends on **WANCOM_BUF_DEF_SIZE** defined in **wancomEnd.h** and **_CACHE_ALIGN_SIZE** defined in architecture specific header file) per CFD from system memory.

RETURNS an END object pointer, or NULL on error.

ERRNO Not Available

SEE ALSO **wancomEnd**, **ifLib**, *Marvell GT64240 Data Sheet*, *Marvell GT64260 Data Sheet*, *Marvell GT64260 Errata*

wdbEndPktDevInit()

NAME **wdbEndPktDevInit()** – initialize an END packet device

SYNOPSIS

```
STATUS wdbEndPktDevInit
(
    WDB_END_PKT_DEV *      pPktDev, /* device structure to init */
    void                  (*stackRcv) (), /* receive packet callback
(udpRcv) */
    char *                pDevice, /* Device (ln, ie, etc.) that we */
                                /* wish to bind to. */
    int                   unit     /* unit number (0, 1, etc.) */
)
```

DESCRIPTION This routine initializes an END packet device. It is typically called from configlette **wdbEnd.c** when the WDB agent's lightweight END communication path (**INCLUDE_WDB_COMM_END**) is selected.

RETURNS OK or ERROR

ERRNO Not Available

SEE ALSO

wdbEndPktDrv

wdbNetromPktDevInit()

NAME **wdbNetromPktDevInit()** – initialize a NETROM packet device for the WDB agent

SYNOPSIS

```
void wdbNetromPktDevInit
(
    WDB_NETROM_PKT_DEV *pPktDev,           /* packet device to initialize */
    caddr_t             dpBase,            /* address of dualport memory */
    int                 width,              /* number of bytes in a ROM word */
    int                 index,              /* pod zero's index in a ROM word */
    int                 numAccess,          /* to pod zero per byte read */
    void                (*stackRcv)(),      /* callback when packet arrives */
    int                 pollDelay           /* poll task delay */
)
```

DESCRIPTION

This routine initializes a NETROM packet device. It is typically called from **usrWdb.c** when the WDB agents NETROM communication path is selected. The *dpBase* parameter is the address of NetROM's dualport RAM. The *width* parameter is the width of a word in ROM space, and can be 1, 2, or 4 to select 8-bit, 16-bit, or 32-bit width respectively (use the macro **WDB_NETROM_WIDTH** in **configAll.h** for this parameter). The *index* parameter refers to which byte of the ROM contains pod zero. The *numAccess* parameter should be set to the number of accesses to POD zero that are required to read a byte. It is typically one, but some boards actually read a word at a time. This routine spawns a task which polls the NetROM for incoming packets every *pollDelay* clock ticks.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

wdbNetromPktDrv

wdbPipePktDevInit()

NAME **wdbPipePktDevInit()** – initialize a pipe packet device

SYNOPSIS

```
STATUS wdbPipePktDevInit
(
    WDB_PIPE_PKT_DEV *           pPktDev,   /* pipe device structure to init */
```

```
void (*stackRcv) () /* receive packet callback
(udpRcv) */
```

DESCRIPTION	This routine initializes a pipe device. It is typically called from configlette wdbPipe.c when the WDB agent's lightweight pipe communication path (INCLUDE_WDB_COMM_PIPE) is selected.
RETURNS	OK or ERROR
ERRNO	Not Available
SEE ALSO	wdbPipePktDrv

wdbSlipPktDevInit()

NAME	wdbSlipPktDevInit() – initialize a SLIP packet device for a WDB agent
SYNOPSIS	<pre>void wdbSlipPktDevInit (WDB_SLIP_PKT_DEV *pPktDev, /* SLIP packetizer device */ SIO_CHAN * pSioChan, /* underlying serial channel */ void (*stackRcv) () /* callback when a packet arrives */)</pre>
DESCRIPTION	This routine initializes a SLIP packet device on one of the BSP's serial channels. It is typically called from usrWdb.c when the WDB agent's lightweight SLIP communication path is selected.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	wdbSlipPktDrv

wdbTsfsDrv()

NAME	wdbTsfsDrv() – initialize the TSFS device driver for a WDB agent
SYNOPSIS	STATUS wdbTsfsDrv

```
(  
char * name /* root name in i/o system */  
)
```

DESCRIPTION This routine initializes the VxWorks virtual I/O "2" driver and creates a TSFS device of the specified name.

This routine should be called exactly once, before any reads, writes, or opens. Normally, it is automatically called if the component **INCLUDE_WDB_TSFS** is added at configuration time and the device name created is **/tgtsvr**.

After this routine has been called, individual virtual I/O channels can be opened by appending the host file name to the virtual I/O device name. For example, to get a file descriptor for the host file **/etc/passwd**, call **open()** as follows:

```
fd = open ("/tgtsvr/etc/passwd", O_RDWR, 0)
```

RETURNS OK, or ERROR if the driver cannot be installed.

ERRNO Not Available

SEE ALSO **wdbTsfsDrv**

wdbVioDrv()

NAME **wdbVioDrv()** – initialize the *tty* driver for a WDB agent

SYNOPSIS STATUS wdbVioDrv
(
char * name /* device name */
)

DESCRIPTION This routine initializes the VxWorks virtual I/O driver and creates a virtual I/O device of the specified name.

This routine should be called exactly once, before any reads, writes, or opens. Normally, it is automatically called when the component **INCLUDE_WDB_VIO_DRV** is included at configuration time and the device name created is **"/vio"**.

After this routine has been called, individual virtual I/O channels can be opened by appending the channel number to the virtual I/O device name. For example, to get a file descriptor for virtual I/O channel 0x1000017, call **open()** as follows:

```
fd = open ("/vio/0x1000017", O_RDWR, 0)
```

RETURNS OK, or ERROR if the driver cannot be installed.

ERRNO Not Available

SEE ALSO **wdbVioDrv**

xbdAttach()

NAME **xbdAttach()** – attach an XBD device

SYNOPSIS

```
int xbdAttach
(
    XBD *           xbd,
    struct xbd_funcs * funcs,
    const char *     name,
    unsigned         blocksize,
    sector_t         nbblocks,
    device_t *       result
)
```

DESCRIPTION none

RETURNS 0 upon success, non-zero otherwise

ERRNO Not Available

SEE ALSO **xbd**

xbdBlockSize()

NAME **xbdBlockSize()** – retrieve the block size

SYNOPSIS

```
int xbdBlockSize
(
    device_t   d,
    unsigned * result
)
```

DESCRIPTION none

RETURNS 0 on success, error code otherwise

ERRNO Not Available

SEE ALSO **xbd**

xbdDetach()

NAME **xbdDetach()** – detach an XBD device

SYNOPSIS

```
void xbdDetach
(
    XBD * xbd /* pointer to XBD device to detach */
)
```

DESCRIPTION none

RETURNS N/A

ERRNO Not Available

SEE ALSO **xbd**

xbdDump()

NAME **xbdDump()** – XBD dump routine

SYNOPSIS

```
int xbdDump
(
    device_t d,
    sector_t pos,
    void * data,
    size_t   size
)
```

DESCRIPTION none

RETURNS 0 on success, error code otherwise

ERRNO Not Available

SEE ALSO **xbd**

xbdInit()

NAME	xbdInit() – initialize the XBD library
SYNOPSIS	STATUS xbdInit (void)
DESCRIPTION	This routine initializes the XBD libarary.
RETURNS	OK upon success, ERROR otherwise
ERRNO	Not Available
SEE ALSO	xbd

xbdIoctl()

NAME	xbdIoctl() – XBD device ioctl routine
SYNOPSIS	int xbdIoctl (device_t d, int cmd, void * arg)
DESCRIPTION	none
RETURNS	varies
ERRNO	Not Available
SEE ALSO	xbd

xbdNBlocks()

NAME	xbdNBlocks() – retrieve the total number of blocks
SYNOPSIS	int xbdNBlocks

```
(  
    device_t    d,  
    sector_t * result  
)
```

DESCRIPTION none

RETURNS 0 on success, error code otherwise

ERRNO Not Available

SEE ALSO **xbd**

xbdSize()

NAME **xbdSize()** – retrieve the total number of bytes

SYNOPSIS

```
int xbdSize  
(  
    device_t    d,  
    long long * result  
)
```

DESCRIPTION This routine retrieves the total number of bytes on the backing media.

RETURNS 0 on success, error code otherwise

ERRNO Not Available

SEE ALSO **xbd**

xbdStrategy()

NAME **xbdStrategy()** – XBD strategy routine

SYNOPSIS

```
int xbdStrategy  
(  
    device_t    d,  
    struct bio * bio  
)
```

DESCRIPTION	none
RETURNS	0 upon success, error code otherwise
ERRNO	Not Available
SEE ALSO	xbd

z8530DevInit()

NAME	<i>z8530DevInit()</i> – intialize a Z8530_DUSART
SYNOPSIS	void z8530DevInit (Z8530_DUSART * pDusart)
DESCRIPTION	The BSP must have already initialized all the device addresses, etc in Z8530_DUSART structure. This routine initializes some SIO_CHAN function pointers and then resets the chip to a quiescent state.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	z8530Sio

z8530Int()

NAME	<i>z8530Int()</i> – handle all interrupts in one vector
SYNOPSIS	void z8530Int (Z8530_DUSART * pDusart)
DESCRIPTION	On some boards, all SCC interrupts for both ports share a single interrupt vector. This is the ISR for such boards. We determine from the parameter which SCC interrupted, then look at the code to find out which channel and what kind of interrupt.

RETURNS	N/A
ERRNO	Not Available
SEE ALSO	z8530Sio

z8530IntEx()

NAME	<i>z8530IntEx() – handle error interrupts</i>
SYNOPSIS	void z8530IntEx (Z8530_CHAN * pChan)
DESCRIPTION	This routine handles miscellaneous interrupts on the SCC.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	z8530Sio

z8530IntRd()

NAME	<i>z8530IntRd() – handle a receiver interrupt</i>
SYNOPSIS	void z8530IntRd (Z8530_CHAN * pChan)
DESCRIPTION	This routine handles read interrupts from the SCC.
RETURNS	N/A
ERRNO	Not Available
SEE ALSO	z8530Sio

z8530IntWr()

NAME **z8530IntWr()** – handle a transmitter interrupt

SYNOPSIS

```
void z8530IntWr
(
    Z8530_CHAN * pChan
)
```

DESCRIPTION This routine handles write interrupts from the SCC.

RETURNS N/A

ERRNO Not Available

SEE ALSO **z8530Sio**