WIND RIVER

VxWorks®

DRIVERS API REFERENCE

6.2

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

VxWorks Drivers API Reference, 6.2

19 Dec 05
Part #: DOC-15602-ND-00

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

ambaSio

NAME

ambaSio – ARM AMBA UART *tty* driver

ROUTINES

ambaDevInit( ) – initialize an AMBA channel
ambaIntTx( ) – handle a transmitter interrupt
ambaIntRx( ) – handle a receiver interrupt

DESCRIPTION

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.

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.

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.

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 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( )**.

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.

DATA STRUCTURES

An **AMBA_CHAN** data structure is used to describe each channel, this structure is described in **h/drv/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 (such as **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK ioctl** command. Likewise, a receiver interrupt handluer 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 modules are **ambaDevInit( )**, **ambaIntTx( )** and **ambaIntRx( )**.

The BSP's **sysHwInit( )** routine typically calls **sysSerialHwInit( )**, which initializes 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 initialization:

```
#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;

  /*
   * Initialize driver functions, getTxChar, putRcvChar and
   * channelMode, then initialize UART
   */

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

The BSP's **sysHwInit2( )** routine typically calls **sysSerialHwInit2( )**, which connects the chips 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 initialization 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, *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**, 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 is 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**, *"Network Protocol Toolkit User's Guide"*, *"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
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

DESCRIPTION

BLOCK DEVICE DRIVER

This is a Block Device Driver for ATA/ATAPI devices on IDE host controller. It also provides neccessary 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 ata 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, usally 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 catagories as device contol 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 **atapiBlkIcotl( )**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 managment feature set commands, host protected feature set commands, security feature set commands, media control functions etc).

Device Information Function:

In this catagory various functions are implmented 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 implmented 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.

The **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 the **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;
```

The user is expected to fill the **ATAPI_CMD** structure with the 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** structrue 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	3.3 MBps
ATA_PIO_1	PIO mode 1	5.2 MBps
ATA_PIO_2	PIO mode 2	8.3 MBps
ATA_PIO_3	PIO mode 3	11.1 MBps
ATA_PIO_4	PIO mode 4	16.6 MBps
ATA_PIO_AUTO	PIO max supported mode	
ATA_DMA_SINGLE_0	Single DMA mode 0	2.1 MBps
ATA_DMA_SINGLE_1	Single DMA mode 1	4.2 MBps
ATA_DMA_SINGLE_2	Single DMA mode 2	8.3 MBps
ATA_DMA_MULTI_0	Multi word DMA mode 0	4.2 MBps
ATA_DMA_MULTI_1	Multi word DMA mode 1	13.3 MBps
ATA_DMA_MULTI_2	Multi word DMA mode 2	16.6 MBps
ATA_DMA_ULTRA_0	Ultra DMA mode 0	16.6 MBps
ATA_DMA_ULTRA_1	Ultra DMA mode 1	25.0 MBps
ATA_DMA_ULTRA_2	Ultra DMA mode 2	33.3 MBps
ATA_DMA_ULTRA_3	Ultra DMA mode 3	44.4 MBps
ATA_DMA_ULTRA_4	Ultra DMA mode 4	66.6 MBps
ATA_DMA_ULTRA_5	Ultra DMA mode 5	100.0 MBps
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 is done at the initialization of the device from the device parameters and if 80 conductor cable is connected, 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 (that is, for UDMA, MDMA, SDMA and PIO modes).

The host IDE Bus master functions are to be mapped to follwing 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( )**.

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.

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 following 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 in 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 is 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.

INCLUDE FILES **end.h endLib.h etherMultiLib.h auEnd.h**

SEE ALSO **muxLib**, **endLib**, **netBufLib**, *Writing and Enhanced Network Driver*

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 transmit frames.

INCLUDE FILES **endLib.h etherMultiLib.h bcm1250MacEnd.h**

SEE ALSO **muxLib**, **endLib**, **netBufLib**, *"Writing and Enhanced Network Driver"*

bio

NAME **bio** – buffer I/O 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 I/O (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, **cisConifigList**. 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

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), 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. See **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 **DEC_USR_21140** 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 **DEC_USR_XEA** 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 **DEC_USR_BAR_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 **DEC_USR_TAP_***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, 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, the PHY will negotiate this technology ability, if present.

100Base-T Mode:
when the flag **DEC_USR_MII_100MB** is set, the PHY will negotiate this technology ability, if present.

Half duplex Mode:
when the flag **DEC_USR_MII_HD** is set, the PHY will negotiate this technology ability, if present.

Full duplex Mode:
when the flag **DEC_USR_MII_FD** is set, 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, 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

```
VOIDFUNCPTR       _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

```
VOIDFUNCPTR _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

Chip	Vendor ID	Device ID
dec 21140	0x1011	0x0009
dec 21143	0x1011	0x0019
dec 21145	0x8086	0x0039

INCLUDE FILES none

SEE ALSO **ifLib**, *"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
devUnmap( ) – unmap a device
devName( ) – name a device

DESCRIPTION This library provides the interface for the device infrastructure.

INCLUDE FILES **drv/manager/device.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 Xl 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 operated 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, 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 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. If both *devIoAddr* and *devMemAddr* are given, 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 I/O 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**, 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 is 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, 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)
sysEl3c90xIntEnable(pDrvCtrl->intLevel)
sysEl3c90xIntDisable(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*.

INCLUDE FILES **end.h endLib.h etherMultiLib.h el3c90xEnd.h**

SEE ALSO **muxLib**, **endLib**, **netBufLib**, *VxWorks Programmer's Guide: Writing and Enhanced Network Driver*

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

The only external interface is the **elt3c509Load( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*port*:*intVector*:*intLevel*:*attachementType*:*nRxFrames*

The **elt3c509Load( )** 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.

intVector
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.

intLevel
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.

attachmentType
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.

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 I/O-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.

INCLUDE FILES

end.h endLib.h etherMultiLib.h elt3c509End.h

SEE ALSO

muxLib, **endLib**, *Writing an Enhanced Network Driver*

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

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**

erfLib

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

erfShow

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

evbNs16550Sio

NAME	**evbNs16550Sio** – NS16550 serial driver for the IBM PPC403GA evaluation
ROUTINES	**evbNs16550HrdInit( )** – 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 MB 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 and 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 pared with. It is left to the driver to keep track of which descriptor goes with its counter part. If this synchronization is lost, 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_TAG**s 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 *nRfds* * 2. If this value is set, 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, 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*

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

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.

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.

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:

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/IPv4, UDP/IPv4, IPv4
	TCP/IPv6, UDP/IPv6	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 16 KB. 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 in 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 the 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 mblks 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, 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*

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 problems.

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**, *Intel 82595TX ISA/PCMCIA High Integration Ethernet Controller User Manual*, *Vadem VG-468 PC Card Socket Controller Data Manual*.

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 I/O (**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 SYSCLK 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, PIIX4 supports a serial interrupt scheme. PIIX4 provides full support for the use of an external I/O APIC.

Enhanced Universal Serial Bus (USB) Controller

The PIIX4 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 PIRQ[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 routine above locates and initializes the PIIX4.

```
STATUS iPIIX4KbdInit
    (
    )
```

The routine above does keyboard-specific initialization on PIIX4.

```
STATUS iPIIX4FdInit
    (
    )
```

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

```
STATUS iPIIX4AtaInit
    (
    )
```

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

```
STATUS iPIIX4IntrRoute
```

```
    (
    int pintx, char irq
    )
```

The routine above routes **PIRQ[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 routine above returns the interrupt level of a PCI interrupt previously set by **iPIIX4IntrRoute**.

INCLUDE FILES **iPIIX4.h**

ln97xEnd

NAME **ln97xEnd** – END-style AMD Am79C97X PCnet-PCI Ethernet driver

ROUTINES
ln97xEndLoad( ) – initialize the driver and device
ln97xInitParse( ) – 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 setup. 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, **ln97xEndLoad( )**. 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**, 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 is 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 multple END drivers.

ln97xIntConnect

```
IMPORT STATUS (* ln97xIntConnect)
    (
    VOIDFUNCPTR * vector,     /* interrupt vector to attach to     */
    VOIDFUNCPTR   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( )**.

ln97xIntDisconnect

```
IMPORT STATUS (* ln97xIntDisconnect)
    (
    VOIDFUNCPTR * vector,     /* interrupt vector to attach to     */
    VOIDFUNCPTR   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( )**.

ln97xIntEnable

```
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( )**.

ln97xIntDisable

```
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. Wind River Intel Pentium BSPs may implement this routine via **sysIntDisablePIC( )**.

ln97xEnetAddrGet

```
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**, *"Network Protocol Toolkit User'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"

lptDrv

NAME **lptDrv** – parallel chip device driver for the IBM-PC LPT

ROUTINES

lptDrv( ) – initialize the LPT driver
lptDevCreate( ) – create a device for an LPT port
lptShow( ) – 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: **lptDrv( )** to initialize the driver, and **lptDevCreate( )** to create devices.

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

Before the driver can be used, it must be initialized by calling **lptDrv( )**. This routine should be called exactly once, before any reads, writes, or calls to **lptDevCreate( )**. Normally, it is called from **usrRoot( )** in **usrConfig.c**. The first argument to **lptDrv( )** 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;         /* I/O 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 *VxWorks Programmer's Guide: I/O System*

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

miiLib

NAME **miiLib** – Media Independent Interface library

ROUTINES
miiPhyInit( ) – initialize and configure the PHY devices
miiPhyUnInit( ) – uninitialize 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( ) – uninitialize 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
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,

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 **FUNCPTR**s 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 **miiPhyInt** 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.

INCLUDE FILES

none

SEE ALSO

ifLib, *MPC8260 Fast Ethernet Controller (Supplement to the MPC860 User's Manual)* , *Motorola MPC860 User's Manual*

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 as follows:

"*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 the 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. It 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.

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*, *VxWorks Programmer's Guide: I/O System*

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 *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*adrs*:*vecNum*:*intLvl*:*byteAccess*:*usePromEnetAddr*:*offset*

The **ne2000EndLoad( )** 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.

adrs
Tells the driver where to find the ne2000.

vecNum
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.

intLvl
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( )**.

INCLUDE FILES **end.h endLib.h etherMultiLib.h**

SEE ALSO **muxLib**, **endLib**, *Writing and Enhanced Network Driver*

nec765Fd

NAME **nec765Fd** – NEC 765 floppy disk device driver

ROUTINES
fdDrv( ) – initialize the floppy disk driver
fdDevCreate( ) – create a device for a floppy disk
fdRawio( ) – 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 *VxWorks Programmer's Guide: I/O System*

ns16550Sio

NAME **ns16550Sio** – NS 16550 UART *tty* driver

ROUTINES
ns16550DevInit( ) – intialize an NS16550 channel
ns16550IntWr( ) – handle a transmitter interrupt
ns16550IntRd( ) – handle a receiver interrupt
ns16550IntEx( ) – miscellaneous interrupt processing
ns16550Int( ) – 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, 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 it's 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, 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, 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**, autoconfig only calls the BSP interrupt routing routine for devices on bus number 0. If a device resides on a higher numbered bus, 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**, 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, 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 the following:

- 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.

```
                  ---------
                  |       |
                  |  CPU  |
                  |       |
                  ---------
                      |
   Host bus           |
----------------------+------------------------- ...
                      |
                 ------------
                 | Bridge 0 |
                 |  (host   |
                 | adapter) |
                 ------------
                      |
   PCI bus segment 0  |  (primary bus segment)
----------------------+------------------------- ...
         |            |    |                  |
       dev 0          |   dev 1              dev 2
                      |
                 ------------
                 |          |
                 | Bridge 1 |
                 |  (P2P)   |
                 ------------
                      |
   PCI bus segment 1  |  (secondary bus segment)
----------------------+------------------------- ...
         |            |    |                  |
       dev 0          |   dev 1              dev 2
                      |
                 ------------
                 |          |
                 | Bridge 2 |
                 |  (P2P)   |
                 ------------
                      |
   PCI bus segment 2  |  (tertiary bus segment)
----------------------+------------------------- ...
         |                 |                  |
       dev 0              dev 1              dev 2
```

For further details, see 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 signifiant 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, see the PCI Local Bus Specification, Revision 2.1. See 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, 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).

For example:

```
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).

For example:

```
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 0xffff0000 in the bit mask for ModifyLong */

pciConfigModifyLong (bus, dev, func, PCI_CFG_COMMAND,
                 (0xffff0000 | 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 I/O-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 argument *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, 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(I/O mapped) library

ROUTINES

pciDeviceShow( ) – print information about PCI devices
pciHeaderShow( ) – print a header of the specified PCI device
pciFindDeviceShow( ) – find a PCI device and display the information
pciFindClassShow( ) – find a device by 24-bit class code
pciConfigStatusWordShow( ) – show the decoded value of the status word
pciConfigCmdWordShow( ) – show the decoded value of the command word
pciConfigFuncShow( ) – show configuration details about a function
pciConfigTopoShow( ) – 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 **pciConfigLib.o**. There are two ways to find out an empty device.

- 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.

pciIntLib

NAME

pciIntLib – PCI Shared Interrupt support

ROUTINES

pciIntLibInit( ) – initialize the **pciIntLib** module
pciInt( ) – interrupt handler for shared PCI interrupt.
pciIntConnect( ) – connect the interrupt handler to the PCI interrupt.
pciIntDisconnect( ) – disconnect the interrupt handler (OBSOLETE)
pciIntDisconnect2( ) – disconnect an interrupt handler from the PCI interrupt.

DESCRIPTION

This component is PCI Revision 2.1 compliant.

The functions addressed here include:

- 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: **pciInt( )**, **pciIntConnect( )**, and **pciIntDisconnect2( )**. **pciIntConnect( )** adds the specified interrupt handler to the link list and **pciIntDisconnect2( )** removes it from the link list. The master interrupt handler **pciInt( )** 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. **pciInt( )** should be attached by **intConnect( )** function in the BSP initialization with its parameter. The parameter is an vector number associated to the PCI interrupt.

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:

- 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

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, see the *PPC403GA Embedded Controller User's Manual*.

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

ppc860Sio

NAME **ppc860Sio** – Motorola MPC800 SMC UART serial driver

ROUTINES **ppc860DevInit( )** – initialize the SMC
ppc860Int( ) – 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 **sysHwInit( )** routine typically calls **sysSerialHwInit( )**, which initializes all the values in the PPC860SMC_CHAN structure (except the **SIO_DRV_FUNCS**) before calling **ppc860DevInit( )**.

The BSP's **sysHwInit2( )** routine typically calls **sysSerialHwInit2( )** which connects the chip's interrupts via **intConnect( )**.

INCLUDE FILES **drv/sio/ppc860Sio.h**

rm9000x2glSio

NAME **rm9000x2glSio** – RM9000 *tty* driver

ROUTINES **rm9000x2glDevInit( )** – intialize an NS16550 channel
rm9000x2glIntWr( ) – handle a transmitter interrupt
rm9000x2glIntRd( ) – handle a receiver interrupt
rm9000x2glIntEx( ) – miscellaneous interrupt processing
rm9000x2glInt( ) – interrupt-level processing
rm9000x2glIntMod( ) – 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 **rm9000x2glSio.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.

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

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]
EIOCPOLLSTART	**N/A**	start polled operation
EIOCPOLLSTOP	**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. Used by the master CPU when building SM.
SM_TAS_TYPE	10	Test-and-set type (**SM_TAS_HARD** or **SM_TAS_SOFT**)
SM_CPUS_MAX	10	Maximum number of CPUs supported in SM (0 = default number)
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)
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 **mBlk**s in driver memory pool (if < 16, a default value is used)
SM_NUM_CBLKS	16	Number of **clBlk**s 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.

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.

INCLUDE FILES **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 *VxWorks AE Network Programmer's Guide*

smcFdc37b78x

NAME **smcFdc37b78x** – a superIO (fdc37b78x) initialization source module

ROUTINES **smcFdc37b78xDevCreate( )** – set correct I/O 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 intialize 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 thie facility can selectively intialize a set of devices on superIO chip.
- Intializing keyboard by sending commands to its controller embedded in superIO chip.

INTERNAL DATABASES

This library provides its user to changes 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 the very first routine that should be called by the user of this library. This routine sets up I/O port address that will subsequentally be used later on. The I/O 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 intialization mask and intializes 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 can be bitwise ORed to enable more than one device at a time. For example, if you want COM1 and COM2 to be enabled on superIO chip, call the following:

```
smcFdc37b78xInit (SMCFDC37B78X_COM1_EN | SMCFDC37B78X_COM2_EN);
```

As prerequisites for the above call, the superIO chip library should be intialized 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 *VxWorks Programmer's Guide: I/O System*

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 diagnotics 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_REG*x*_READ( )** and **SYM895_REG*x*_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 I/O or 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 I/O 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 undefine **INCLUDE_SHOW_ROUTINES**. You can exclude **tffsBootImagePut( )** if you edit **sysTffs.c** and undefine **INCLUDE_TFFS_BOOT_IMAGE**. (If you find these utilities are missing and you want them included, edit **config.h** and define **INCLUDE_SHOW_ROUTINES** and **INCLUDE_TFFS_BOOT_IMAGE**.)

If you wish to include only the TrueFFS-specific show routines you could define **INCLUDE_TFFS_SHOW** instead of **INCLUDE_SHOW_ROUTINES** 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**, undefine the appropriate constant, 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

USAGE 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

```
STATUS vgaInit
    (
    VOID
    )
```

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 *Programmer's Guide to the EGA, VGA, and Super VGA Cards - Ferraro. Programmer's Guide to PC & PS/2 Video Systems - Richard Wilton.*

INCLUDE FILES None.

wancomEnd

NAME **wancomEnd** – END style Marvell/Galileo GT642xx Ethernet network interface driver

ROUTINES **wancomEndLoad( )** – initialize the driver and device
wancomEndDbg( ) – 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

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 "-".

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 as follows:
"*memBase*:*memSize*:*nCfds*:*nRfds*:*flags*"

TARGET-SPECIFIC PARAMETERS

memBase

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( )**.

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 Command Frame Descriptor, Receive Frame Descriptor and reception buffers.

nTfds

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

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.

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 amout 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 independant of the OS, and thus can be used to support an external mode (e.g., monitor style) debug agent.

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.

PACKET READY CALLBACK

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.

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**, your driver's 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 ultimatly 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:

- **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, 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, 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 incompatabilities 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 Tornado 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 **/tgtsvr**. 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 *WindView User's Guide: Data Upload* and the target server documentation.

TSFS SOCKETS

TSFS provides all of the functionality of other VxWorks file systems. For details, see the *VxWorks Programmer's Guide: I/O System and Local File Systems*. 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:

```
TCP:server_name | server_ip:port_number
```

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 ("/tgtsvr/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 ("/tgtsvr/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, **WDB_TSFS_O_TEXT** can be included in the mode argument to **open( )**. For example:

```
fd = open ("/tgtsvr/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.

TGTSVR

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

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 descritptor 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_DONTROUTE** 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 errnos 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 *Tornado User's Guide, VxWorks Programmer's Guide: I/O System, Local File Systems*

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
1-0xffffff	Dynamically created on the host
>= 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 *Tornado 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. See 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

ambaDevInit( )

NAME
ambaDevInit( ) – initialize an AMBA channel

SYNOPSIS
```
void ambaDevInit
    (
    AMBA_CHAN * pChan  /* ptr to AMBA_CHAN describing this channel */
    )
```

DESCRIPTION
This routine initializes some **SIO_CHAN** function pointers and then resets the chip to a quiescent state. Before this routine is called, the BSP must already have initialized 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
```
void ambaIntRx
    (
    AMBA_CHAN * pChan  /* ptr to AMBA_CHAN describing this channel */
    )
```

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

```
void ambaIntTx
    (
    AMBA_CHAN * pChan  /* ptr to AMBA_CHAN describing this channel */
    )
```

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

```
STATUS amd8111LanDumpPrint
    (
    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

```
STATUS amd8111LanErrCounterDump
    (
    AMD8111_LAN_DRV_CTRL * pDrvCtrl,  /* pointer to DRV_CTRL structure */
    UINT32 *               memAddr    /* pointer to receive stat data */
    )
```

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

amd8111LanEnd

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 for 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
ATA_CMD_INITP	0	0	0	0	0	0
ATA_CMD_RECALIB	0	0	0	0	0	0
ATA_PI_CMD_SRST	0	0	0	0	0	0
ATA_CMD_EXECUTE_DEVICE_DIAGNOSTIC	0	0	0	0	0	0
ATA_CMD_SEEK	cylinder or LBA high	head LBA low	0	0	0	0
ATA_CMD_SET_FEATURE	FR (SUBCOMMAND)	SC (SubCommand Specific Value)	0	0	0	0
ATA_CMD_SET_MULTI	sectors per block	0	0	0	0	0
ATA_CMD_IDLE	SC (Timer Period)	0	0	0	0	0
ATA_CMD_STANDBY	SC (Timer Period)	0	0	0	0	0
ATA_CMD_STANDBY_IMMEDIATE	0	0	0	0	0	0
ATA_CMD_SLEEP	0	0	0	0	0	0
ATA_CMD_CHECK_POWER_MODE	0	0	0	0	0	0
ATA_CMD_IDLE_IMMEDIATE	0	0	0	0	0	0
ATA_CMD_SECURITY_DISABLE_PASSWORD	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_SECURITY_ERASE_PREPARE	0	0	0	0	0	0
ATA_CMD_SECURITY_ERASE_UNIT	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO	ATA_ZERO
ATA_CMD_SECURITY_FREEZE_LOCK	0	0	0	0	0	0
ATA_CMD_SECURITY_SET_PASSWORD	0	0	0	0	0	0

```
ATA_CMD_SECURITY_UNLOCK
       0             0           0          0          0         0

ATA_CMD_SMART  (not implemented)
      FR          SC          SN        ATA_ZERO   ATA_ZERO     ATA_ZERO
 (SUBCOMMAND) (SubCommand   (SubCommand
            Specific Value)  Specific Value)

ATA_CMD_GET_MEDIA_STATUS
       0             0           0          0          0         0

ATA_CMD_MEDIA_EJECT
       0             0           0          0          0         0

ATA_CMD_MEDIA_LOCK
       0             0           0          0          0         0

ATA_CMD_MEDIA_UNLOCK
       0             0           0          0          0         0

ATA_CMD_CFA_ERASE_SECTORS
       0             0           0          0          0         0

ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE
    ATA_ZERO        SC      ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_CFA_WRITE_SECTORS_WITHOUT_ERASE
    ATA_ZERO        SC      ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_CFA_TRANSLATE_SECTOR
    ATA_ZERO    ATA_ZERO        SN         cylLo     cylHi       DH

ATA_CMD_CFA_REQUEST_EXTENDED_ERROR_CODE
    ATA_ZERO    ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO

ATA_CMD_SET_MAX
      FR        ATA_ZERO    ATA_ZERO    ATA_ZERO   ATA_ZERO     ATA_ZERO
    (SUBCOMMAND)
```

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      sector no   cylLo       cylHi       head + modebit
(SET_MAX_VOLATILE
      or
SET_MAX_NON_VOLATILE)

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 the **ATA_CMD_SET_FEATURE** subcommand, only *arg0* and *arg1* are valid; all others 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)
ATA_SUB_ENB_ADV_POW_MNGMNT	0x90
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_REVE	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_REVE	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 *ctrl*. 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.

ctrl is a controller number for the ATA controller; the primary controller is 0. The maximum is specified via **ATA_MAX_CTRLS**.

drive is the drive number for the ATA hard drive; the master drive is 0. The maximum is specified via **ATA_MAX_DRIVES**.

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 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

```
STATUS ataDevIdentify
    (
    int ctrl,
    int dev
    )
```

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

```
STATUS ataDmaRW
    (
    int    ctrl,
    int    drive,
    UINT32 cylinder,
    UINT32 head,
    UINT32 sector,
    void * buffer,
    UINT32 nSecs,
    int    direction
    )
```

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( )**

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

```
STATUS ataParamRead
    (
    int  ctrl,
    int  drive,
    void *buffer,
    int  command
    )
```

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

```
STATUS ataPiInit
    (
    int ctrl,
    int drive
    )
```

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

```
STATUS ataRW
    (
    int       ctrl,
    int       drive,
    UINT32    cylinder,
    UINT32    head,
    UINT32    sector,
    void    * buffer,
    UINT32    nSecs,
    int       direction
    )
```

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**

ataShow( )

NAME **ataShow( )** – show the ATA/IDE disk parameters

SYNOPSIS

```
STATUS ataShow
    (
    int ctrl,
    int drive
    )
```

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 **OK**, or **ERROR** if the parameters are invalid.

ERRNO Not Available

SEE ALSO **ataShow**

ataShowInit( )

NAME **ataShowInit( )** – initialize the ATA/IDE disk driver show routine

SYNOPSIS

```
STATUS ataShowInit (void)
```

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:

- 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

```
STATUS ataStatusChk
    (
    ATA_CTRL  * pCtrl,
    UINT8       mask,
    UINT8       status
    )
```

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

```
device_t ataXbdDevCreate
    (
    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

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

```
STATUS atapiCtrlMediumRemoval
    (
    ATA_DEV   * pAtapiDev,
    int         arg0
    )
```

DESCRIPTION This function issues a command to drive to PREVENT or ALLOW MEDIA removal. Argument *arg0* selects to **LOCK_EJECT** or **UNLOCK_EJECT**.

To lock media eject *arg0* should be **LOCK_EJECT** To unload media eject *arg0* 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

```
UINT16 atapiCurrentCylinderCountGet
    (
    int ctrl,
    int drive
    )
```

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

```
UINT8 atapiCurrentHeadCountGet
    (
    int ctrl,
    int drive
    )
```

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

```
UINT8 atapiCurrentMDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive MDMA mode enabled in the ATA/ATAPI drive that is specified by *ctrl* and *drive* from the **drive** structure. The following bit is set for corresponding mode selected:

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

```
UINT8 atapiCurrentSDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive SDMA mode enable in the ATA/ATAPI drive specified by *ctrl* and *drive* 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

```
UINT8 atapiCurrentUDmaModeGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION This function is used to get drive UDMA mode enable in the ATA/ATAPI drive specified by *ctrl* and *drive* from drive structure The following bit is set for corresponding mode selected.

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 cyclinder 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**

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

```
UINT32 atapiFeatureEnabledGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function is used to get drive Features Enabled by the ATA/ATAPI drive specified by *ctrl* and *drive* 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

```
UINT32 atapiFeatureSupportedGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function is used to get drive Feature supported by the ATA/ATAPI drive specified by *ctrl* and *drive* 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 firmware 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 *ctrl* and *drive* 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

```
STATUS atapiInit
    (
    int ctrl,
    int drive
    )
```

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

atapiIoctl( )

NAME

atapiIoctl( ) – Control the drive.

SYNOPSIS

```
STATUS atapiIoctl
    (
    int              function,   /* The I/O operation to do */
    int                  ctrl,   /* Controller number of the drive */
    int                 drive,   /* Drive number */
    int         password [16],   /* Password to set. NULL if not applicable*/
    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    - it returns address in <arg1>
ATA_ZERO          (ATA_SDH_IBM or    LBA/CHS add          ATA_ZERO
                   ATA_SDH_LBA )   ( LBA 27:24 / Head
                                     LBA 23:16 / cylHi
                                     LBA 15:8  / cylLow
                                     LBA 7:0   / sector no )

IOCTL_SET_MAX_ADDRESS           - <arg1> is pointer to LBA address
ATA_ZERO     SET_MAX_VOLATILE or     LBA address          ATA_ZERO
             SET_MAX_NON_VOLATILE

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           - returns power mode in <arg1>
ATA_ZERO            ATA_ZERO         returns power        ATA_ZERO
                                     mode
         power modes  :-1)     0x00  Device in standby mode
                         2)    0x80  Device in Idle mode
                3)   0xff  Device in Active or Idle mode

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
ATA_ZERO            ATA_ZERO          ATA_ZERO            ATA_ZERO

IOCTL_GET_MEDIA_STATUS      - returns status in <arg1>
ATA_ZERO            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          ATA_ZERO            ATA_ZERO
      (refer to ref1 page no 190)

IOCTL_SMART_READ_DATA   - returns pointer to pointer <ppBuf> of read data
ATA_ZERO            ATA_ZERO          ATA_ZERO            read data

IOCTL_SMART_READ_LOG_SECTOR  - returns pointer to pointer <ppBuf>of read data
ATA_ZERO      no of sector to     log Address    read data
                  be read

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 - <ppbuf> returns pointer to data pointer.
ATA_ZERO          ATA_ZERO         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

```
UINT8 atapiPktCmd
    (
    ATA_DEV   * pAtapiDev,
    ATAPI_CMD * pComPack
    )
```

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

```
UINT8  atapiPktCmdSend
    (
    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

```
STATUS atapiRead10
    (
    ATA_DEV   * pAtapiDev,
    UINT32      startBlk,
    UINT32      nBlks,
    UINT32      transferLength,
    char      * pBuf
    )
```

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

```
STATUS atapiReadCapacity
    (
    ATA_DEV * pAtapiDev
    )
```

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

```
STATUS atapiReadTocPmaAtip
    (
    ATA_DEV * pAtapiDev,
    UINT32    transferLength,
    char    * resultBuf
    )
```

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

```
UINT16 atapiRemovMediaStatusNotifyVerGet
    (
    int ctrl,
    int drive
    )
```

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

```
STATUS atapiScan
    (
    ATA_DEV   * pAtapiDev,
    UINT32      startAddressField,
    int         function
    )
```

DESCRIPTION This function issues SCAN packet command to ATAPI drive. The *function* 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

```
STATUS atapiSeek
    (
    ATA_DEV * pAtapiDev,
    UINT32    addressLBA
    )
```

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

```
STATUS atapiSetCDSpeed
    (
    ATA_DEV   * pAtapiDev,
    int         readDriveSpeed,
    int         writeDriveSpeed
    )
```

DESCRIPTION This function issues SET CD SPEED packet command to ATAPI drive while reading and writing of ATAPI drive(CD-ROM) data. The arguments *readDriveSpeed* and *writeDriveSpeed* 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 *arg0* selects to EJECT or LOAD.

To eject media *arg0* should be **EJECT_DISK** To load media *arg0* 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

```
STATUS atapiTestUnitRdy
    (
    ATA_DEV    * pAtapiDev
    )
```

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

```
UINT32 atapiVersionNumberGet
    (
    int ctrl,
    int drive
    )
```

DESCRIPTION

This function will return the ATA/ATAPI version number of the drive. Most significant 16 bits represent the Major Version Number and the Lease significant 16 bits represents the minor Version Number.

Major Version Number

Bit 22 ATA/ATAPI-6

Bit 21 ATA/ATAPI-5

Bit 20 ATA/ATAPI-4

Bit 19 ATA-3

Bit 18 ATA-2

Minor Version Number (bit 15 through bit 0)

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
0015h	ATA/ATAPI-5 T13 1321D revision 1
0016h	Reserved
0017h	ATA/ATAPI-4 T13 1153D revision 17
0018h-FFFFh	Reserved

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 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:*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: *unit*:*devMemAddr*:*devIoAddr*:*vecNum*:*intLvl*:*offset*:*flags*

unit
    Device unit number, a small integer.

devMemAddr
Device register base memory address

devIoAddr
I/O register base memory address

enableAddr
Address of MAC enable register

vecNum
Interrupt vector number.

intLvl
Interrupt level.

offset
Offset of starting of data in the device buffers.

qtyCluster
Number of clusters to allocate

flags
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

```
END_OBJ * bcm1250MacEndLoad
    (
    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:

The initialization string format is:
"*unit*:*hwunit*:*vecnum*:*flags*:*numRds0*:*numTds0*:*numRds1*:*numTds1*"

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 *initString* 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

```
void bcm1250MacPhyShow
    (
    int inst  /* driver instance */
    )
```

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
```
void * bio_alloc
    (
    device_t xbd,       /* XBD for which to allocate */
    int      numBlocks  /* number of blocks to allocate */
    )
```

DESCRIPTION This routine allocates *numBlocks* of memory. The size of the block is extracted from the *xbd*.

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

```
STATUS cisConfigregGet
    (
    int sock,     /* socket no. */
    int reg,      /* configuration register no. */
    int *pValue  /* content of the register */
    )
```

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

```
STATUS cisConfigregSet
    (
    int sock,   /* socket no. */
    int reg,    /* register no. */
    int value  /* content of the register */
    )
```

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

```
void cisFree
    (
    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

ctB69000VgaInit( )

NAME
ctB69000VgaInit( ) – initializes the B69000 chip and loads font in memory.

SYNOPSIS
```
STATUS ctB69000VgaInit
    (
    void
    )
```

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 *ndevs* 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

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

devUnmap( )

NAME

devUnmap( ) – unmap a device

SYNOPSIS

```
int devUnmap
    (
    struct device * dev  /* pointer to device to unmap */
    )
```

ERRNO

Not Available

SEE ALSO

device

el3c90xEndLoad( )

NAME

el3c90xEndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * el3c90xEndLoad
    (
    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:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecnum*:*intLvl*:*memAdrs*
:*memSize*:*memWidth*:*flags*:*buffMultiplier*

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 *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

el3c90xEnd

el3c90xInitParse( )

NAME

el3c90xInitParse( ) – parse the initialization string

SYNOPSIS

```
STATUS el3c90xInitParse
    (
    EL3C90X_DEVICE * pDrvCtrl,   /* pointer to the control structure */
    char *           initString  /* initialization string */
    )
```

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 I/O 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**.

SEE ALSO **emacEnd**

emacTimerDebugDump( )

NAME **emacTimerDebugDump( )** – Enable debugging output in timer handler

SYNOPSIS

```
void emacTimerDebugDump
    (
    int  unit,
    BOOL enable  /* 0 for disable */
    )
```

DESCRIPTION This function will enable/diable 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

```
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? */
    )
```

DESCRIPTION This routine accepts the source and destination addressing information through *pSrcAddr* and *pDstAddr* 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 *pMblk* if there is enough space available in the cluster. It then returns a pointer to the pointer referenced in *pMblk*. However, if there is not enough space in the cluster associated with *pMblk*, this routine reserves a new **mBlk**-'clBlk'-cluster construct for the header information. It then prepends the new **mBlk** to the **mBlk** passed in *pMblk*. As the function value, this routine then returns a pointer to the new **mBlk**, which the head of a chain of **mBlk** structures. The second element of this chain is the **mBlk** referenced in *pMblk*.

RETURNS **M_BLK_ID** or **NULL**.

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**

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**.

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 *pEnd* parameter) and a flags value (the *flags* parameter). This routine sets the **flags** member of the **END_OBJ** structure to the value of the *flags* 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

SEE ALSO
endLib

endObjInit( )

NAME
endObjInit( ) – initialize an **END_OBJ** structure

SYNOPSIS
```
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
    )
```

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**.

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**.

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

SEE ALSO **erfLib**

erfCategoriesAvailable( )

NAME **erfCategoriesAvailable( )** – Get the maximum number of Categories.

SYNOPSIS `UINT16 erfMaxCategoriesGet(void)`

DESCRIPTION none

RETURNS Number of categories

SEE ALSO **erfShow**

erfCategoriesAvailable( )

NAME **erfCategoriesAvailable( )** – Get the maximum number of Types.

SYNOPSIS

```
UINT16 erfMaxTypesGet(void)
```

DESCRIPTION none

RETURNS Number of types

SEE ALSO **erfShow**

erfCategoryAllocate( )

NAME **erfCategoryAllocate( )** – allocates a user-defined Event Category

SYNOPSIS

```
STATUS erfCategoryAllocate
    (
    UINT16 * pEventCat  /* variable to return Event Category */
    )
```

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
```
STATUS erfCategoryQueueCreate
    (
    UINT16 eventCat,   /* variable to return Event Category */
    int    queueSize,  /* max number of events for queue */
    int    priority    /* priority of event processing task */
    )
```

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
```
UINT16 erfDefaultQueueSizeGet(void)
```

DESCRIPTION
none

RETURNS
Number of elements that can be stored on the default queue.

SEE ALSO
erfShow

erfEventRaise( )

NAME

erfEventRaise( ) – Raises an event.

SYNOPSIS

```
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 */
    )
```

DESCRIPTION

This function will directly call all the handlers that are registed 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

```
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 */
    )
```

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

```
STATUS     erfHandlerUnregister
    (
    UINT16                 eventCat,      /* Event Category */
    UINT16                 eventType,     /* Event Type */
    erfHandlerPrototype * pEventHandler, /* Pointer to Event Handler */
    void *                 pUserData      /* pointer to User data structure */
    )
```

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

```
STATUS erfLibInit
    (
    UINT16 maxUserCat,  /* Maximum number of User Categories */
    UINT16 maxUserType  /* Maximum number of User Types */
    )
```

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 occured 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

```
void erfShow
    (
    )
```

DESCRIPTION none

RETURNS Nothing

SEE ALSO **erfShow**

erfTypeAllocate( )

NAME

erfTypeAllocate( ) – allocates a user-defined Type for this Category

SYNOPSIS

```
STATUS erfTypeAllocate
    (
    UINT16   eventCat,   /* Event Category */
    UINT16 * pEventType  /* pointer to returned Event Type */
    )
```

DESCRIPTION

This routine allocates a user-defined Event Type for a Category. Once defined, the 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

```
UINT16 erfTypesAvailable
    (
    UINT16 eventCat  /* Event Category */
    )
```

DESCRIPTION

none

RETURNS

Number of types or 0 for a bad category

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 *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).

The *pFdRaw* 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 *initString*.

The parameter string *initString* is an ordered list of parameters each separated by a colon. The format of *initString* is as follows:

"*unit*:*memBase*:*memSize*:*nCfds*:*nRfds*:*flags*:*offset*:*deviceId*:
maxRxFrames:*clToRfdRatio*:*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**, 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 *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, the driver assumes that this region is non-cached.

If the caller indicates that this routine must allocate the shared memory region, 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 *nRfds* * 2. If this value is set, 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**

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

```
void gei82543RegSet
    (
    int    unit,    /* device unit */
    UINT32 offset,  /* register offset */
    UINT32 regVal   /* value to write */
    )
```

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 workaround

SYNOPSIS

```
void gei82543TbiCompWr
    (
    int unit,  /* device unit */
    int flag   /* 0 - off, and others on */
    )
```

DESCRIPTION This routine enables/disables TBI compatibility workaround 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
```
END_DEVICE * gei82543Unit
    (
    int unit
    )
```

DESCRIPTION
none

RETURNS
A pointer the **END_DEVICE** for the unit, or **NULL**.

ERRNO
Not Available

SEE ALSO
gei82543End

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

```
END_OBJ * iOlicomEndLoad
    (
    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.

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 (i.e. oli) into the initString and returns 0.

If the string is allocated, the routine attempts to perform its load functionality.

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

```
void iOlicomIntHandle
    (
    END_DEVICE * pDrvCtrl  /* pointer to END_DEVICE structure */
    )
```

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**.

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**.

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

SEE ALSO **iPIIX4**

iPIIX4Init( )

NAME **iPIIX4Init( )** – initialize PIIX4

SYNOPSIS

```
STATUS iPIIX4Init
    (
    )
```

DESCRIPTION initialize PIIX4

RETURNS **OK** or **ERROR**.

SEE ALSO **iPIIX4**

iPIIX4IntrRoute( )

NAME **iPIIX4IntrRoute( )** – Route PIRQ[A:D]

SYNOPSIS

```
STATUS iPIIX4IntrRoute
    (
    int  pintx,
    char irq
    )
```

DESCRIPTION This routine will connect an irq to a pci interrupt.

RETURNS **OK** or **ERROR**.

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**.

SEE ALSO **iPIIX4**

ln97xEndLoad( )

NAME **ln97xEndLoad( )** – initialize the driver and device

SYNOPSIS

```
END_OBJ * ln97xEndLoad
    (
    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: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

ln97xEnd

ln97xInitParse( )

NAME

ln97xInitParse( ) – parse the initialization string

SYNOPSIS

```
STATUS ln97xInitParse
    (
    LN_97X_DRV_CTRL * pDrvCtrl,   /* pointer to the control structure */
    char *            initString  /* initialization string */
    )
```

DESCRIPTION

Parse the input string. This routine is called from **ln97xEndLoad( )** 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:csr3b:offset:flags>

unit
The device unit number. Unit numbers are integers starting at zero and increasing for each device controlled by the driver.

devMemAddr
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.

devIoAddr
Device register base I/O address (obsolete).

pciMemBase
Base address of PCI memory space.

vecNum
Interrupt vector number.

intLvl
Interrupt level. Generally, this value specifies an interrupt level defined for an external interrupt controller.

memAdrs
Memory pool address or **NONE**.

memSize
Memory pool size or zero.

memWidth
Memory system size, 1, 2, or 4 bytes (optional).

CSR3
Control and Status Register 3 (CSR3) options.

offset
Memory alignment offset.

flags
Device-specific flags reserved for future use.

RETURNS **OK**, or **ERROR** if any arguments are invalid.

ERRNO Not Available

SEE ALSO **ln97xEnd**

lptDevCreate( )

NAME **lptDevCreate( )** – create a device for an LPT port

SYNOPSIS

```
STATUS lptDevCreate
    (
    char *name,   /* name to use for this device */
    int  channel  /* physical channel for this device (0 - 2) */
    )
```

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:

```
    lptDevCreate ("/lpt/0", 0);
```

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

```
STATUS lptDrv
    (
    int          channels,   /* LPT channels */
    LPT_RESOURCE *pResource  /* LPT resources */
    )
```

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

```
void lptShow
    (
    UINT channel  /* channel (0 - 2) */
    )
```

DESCRIPTION This routine shows statistics for a specified LPT port.

RETURNS N/A

ERRNO Not Available

SEE ALSO **lptDrv**

m8260SccEndLoad( )

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

SYNOPSIS

```
END_OBJ * m8260SccEndLoad
    (
    char * initString
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in the *initString*, which is of the following format:

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

The parameters of this string are individually described in the **motCpmEnd** man page.

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 *bufBase*. 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.

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 *txBdNum* and *rxBdNum*, 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.

The user must specify the location of the transmit and receive BDs in the processor's dual ported RAM. *txBdBase* and *rxBdBase* give the offsets from *motCpmAddr* 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.

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 *sccNum* parameter is used to explicitly state which SCC is being used. SCC1 is most commonly used, thus this parameter most often equals "1".

Before this routine returns, it connects up the interrupt vector *ivec*.

RETURNS An END object pointer or **NULL** on error.

ERRNO Not Available

SEE ALSO **m8260SccEnd**, *Motorola MPC8260 User's Manual*

mib2ErrorAdd( )

NAME **mib2ErrorAdd( )** – change a MIB-II error count

SYNOPSIS

```
STATUS mib2ErrorAdd
    (
    M2_INTERFACETBL * 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**

SEE ALSO **endLib**

mib2Init( )

NAME

mib2Init( ) – initialize a MIB-II structure

SYNOPSIS

```
STATUS mib2Init
    (
    M2_INTERFACETBL *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 */
    )
```

DESCRIPTION

Initialize a MIB-II structure. Set all error counts to zero. Assume a 10Mbps Ethernet device.

RETURNS

OK or **ERROR**.

SEE ALSO

endLib

miiAnCheck( )

NAME

miiAnCheck( ) – check the auto-negotiation process result

SYNOPSIS

```
STATUS miiAnCheck
    (
    PHY_INFO * pPhyInfo,  /* pointer to PHY_INFO structure */
    UINT8      phyAddr    /* address of a PHY */
    )
```

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. Users 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( )** – uninitialize the MII library

SYNOPSIS

```
STATUS miiLibUnInit
    (
    )
```

DESCRIPTION This routine uninitializes 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** and **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 ORed 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, 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 on type of network device requiring a PHY-specific registers handler use this routine instead of **miiPhyOptFuncSet( )** to ensure device-specific handlers and to avoid overwritting 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

```
void miiPhyOptFuncSet
    (
    FUNCPTR optRegsFunc  /* function pointer */
    )
```

DESCRIPTION This routine sets the function pointer in *optRegsFunc* 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( )** – uninitialize a PHY

SYNOPSIS

```
STATUS miiPhyUnInit
    (
    PHY_INFO * pPhyInfo  /* pointer to PHY_INFO structure */
    )
```

DESCRIPTION This routine uninitializes the PHY specified in *pPhyInfo*. 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

```
STATUS miiRegsGet
    (
    PHY_INFO    * pPhyInfo,  /* pointer to PHY_INFO structure */
    UINT          regNum,    /* number of registers to display */
    UCHAR *       buff       /* where to read registers to */
    )
```

DESCRIPTION This routine gets the contents of the first *regNum* MII registers, and, if *buff* is not **NULL**, copies them to the space pointed to by *buff*.

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

```
void miiShow
    (
    PHY_INFO    * pPhyInfo  /* pointer to PHY_INFO structure */
    )
```

DESCRIPTION This is a show routine for the MII library

RETURNS **OK**, always.

ERRNO Not Available

SEE ALSO **miiLib**

motFccDrvShow( )

NAME **motFccDrvShow( )** – Debug Function to show FCC parameter ram addresses, initial BD and cluster settings

SYNOPSIS

```
void motFccDrvShow
    (
    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**

motFccDumpRxRing( )

NAME **motFccDumpRxRing( )** – Show the Receive Ring details

SYNOPSIS

```
void motFccDumpRxRing
    (
    int fccNum
    )
```

DESCRIPTION This routine displays the receive ring descriptors.

RETURNS N/A

ERRNO Not Available

SEE ALSO **motFcc2End**, **motFccDumpTxRing( )**

motFccDumpTxRing( )

NAME
motFccDumpTxRing( ) – 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
motFcc2End, **motFccDumpRxRing( )**

motFccEndLoad( )

NAME
motFccEndLoad( ) – 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 *initString*.

The parameter string, *initString*, is an ordered list of parameters each separated by a colon. The format of *initString* is:

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

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 *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**, 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, 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, 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

```
void motFccIramShow
    (
    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

motFccMibShow( )

NAME
motFccMibShow( ) – Debug Function to show MIB statistics.

SYNOPSIS

```
void motFccMibShow
    (
    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

motFccMiiShow( )

NAME **motFccMiiShow( )** – Debug Function to show the Mii settings in the Phy Info structure.

SYNOPSIS

```
void motFccMiiShow
    (
    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**

motFccPramShow( )

NAME **motFccPramShow( )** – Debug Function to show FCC CP parameter ram.

SYNOPSIS

```
void motFccPramShow
    (
    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**

motFccShow( )

NAME **motFccShow( )** – Debug Function to show driver-specific control data.

SYNOPSIS

```
void motFccShow
    (
    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**

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**, 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, 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, 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

```
STATUS ncr810CtrlInit
    (
    FAST NCR_810_SCSI_CTRL *pSiop,        /* ptr to SIOP struct */
    int                    scsiCtrlBusId  /* SCSI bus ID of this SIOP */
    )
```

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:

pSiop
: a pointer to the **NCR_810_SCSI_CTRL** structure created with **ncr810CtrlCreate( )**.

scsiCtrlBusId
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 **ncr810CtlrCreate( )** and **ncr810CrtlInit( )** is {0,0,0,0,0,1,0,0,0,0,0}.

```
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 scntl1Bit7;             /* 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**, **ncr810CtlrCreate( )**

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
----------------
0xfff47000: Sien    = 0xa5 Sdid    = 0x00 Scntl1  = 0x00 Scntl0  = 0x04
0xfff47004: Socl    = 0x00 Sodl    = 0x00 Sxfer   = 0x80 Scid    = 0x80
0xfff47008: Sbcl    = 0x00 Sbdl    = 0x00 Sidl    = 0x00 Sfbr    = 0x00
0xfff4700c: Sstat2  = 0x00 Sstat1  = 0x00 Sstat0  = 0x00 Dstat   = 0x80
0xfff47010: Dsa     = 0x00000000
0xfff47014: Ctest3  = ???? Ctest2  = 0x21 Ctest1  = 0xf0 Ctest0  = 0x00
0xfff47018: Ctest7  = 0x32 Ctest6  = ???? Ctest5  = 0x00 Ctest4  = 0x00
0xfff4701c: Temp    = 0x00000000
0xfff47020: Lcrc    = 0x00 Ctest8  = 0x00 Istat   = 0x00 Dfifo   = 0x00
0xfff47024: Dcmd/Ddc= 0x50000000
0xfff47028: Dnad    = 0x00066144
```

```
0xfff4702c: Dsp     = 0x00066144
0xfff47030: Dsps    = 0x00066174
0xfff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
0xfff47038: Dcntl   = 0x21 Dwt     = 0x00 Dien    = 0x37 Dmode   = 0x01
0xfff4703c: Adder   = 0x000cc2b8
value = 0 = 0x0
```

RETURNS **OK**, or **ERROR** if *pScsiCtrl* and *pSysScsiCtrl* are both **NULL**.

ERRNO Not Available

SEE ALSO **ncr810Lib**, **ncr810CtrlCreate( )**

ne2000EndLoad( )

NAME **ne2000EndLoad( )** – initialize the driver and device

SYNOPSIS

```
END_OBJ* ne2000EndLoad
    (
    char* initString,  /* String to be parsed by the driver. */
    void* pBSP         /* for BSP group */
    )
```

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 **ne2000End**

ns16550DevInit( )

NAME

ns16550DevInit( ) – intialize an NS16550 channel

SYNOPSIS

```
void ns16550DevInit
    (
    NS16550_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 **NS16550_CHAN** structure.

RETURNS

N/A

ERRNO

Not Available

SEE ALSO

ns16550Sio

ns16550Int( )

NAME

ns16550Int( ) – interrupt-level processing

SYNOPSIS

```
void ns16550Int
    (
    NS16550_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

ns16550Sio

ns16550IntEx( )

NAME

ns16550IntEx( ) – miscellaneous interrupt processing

SYNOPSIS

```
void ns16550IntEx
    (
    NS16550_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

ns16550Sio

ns16550IntRd( )

NAME

ns16550IntRd( ) – handle a receiver interrupt

SYNOPSIS

```
void ns16550IntRd
    (
    NS16550_CHAN * pChan  /* pointer to channel */
    )
```

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

```
void ns16550IntWr
    (
    NS16550_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

ns16550Sio

ns83902EndLoad( )

NAME

ns83902EndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* ns83902EndLoad
    (
    char* initString  /* string to be parsed */
    )
```

DESCRIPTION

This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*. 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, "ln") 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 **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**

pccardAtaEnabler( )

NAME **pccardAtaEnabler( )** – enable the PCMCIA-ATA device

SYNOPSIS

```
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 */
    )
```

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

```
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 */
    )
```

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**

pciAutoAddrAlign( )

NAME **pciAutoAddrAlign( )** – align a PCI address and check boundary conditions

SYNOPSIS

```
STATUS pciAutoAddrAlign
    (
    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.

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.

SEE ALSO **pciAutoConfigLib**

pciAutoCardBusConfig( )

NAME **pciAutoCardBusConfig( )** – set mem and I/O registers for a single PCI-Cardbus bridge

SYNOPSIS

```
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 */
    )
```

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 I/O 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:

```
Memory #0:                    Prefetch memory
Memory #1:                    Non-prefetch memory
IO #0:                        IO
IO #1:                        Unused
```

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

SEE ALSO **pciAutoConfigLib**

pciAutoCfg( )

NAME **pciAutoCfg( )** – Automatically configure all nonexcluded PCI headers

SYNOPSIS

```
STATUS pciAutoCfg
    (
    void *pCookie  /* cookie returned by pciAutoConfigLibInit() */
    )
```

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 will initialize 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 will automatically configure the PCI configuration headers for PCI devices and subbridges. The fields that are programmed are as follows:

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.

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 as follows:

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 will first disable FBB on all functions, then enable 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** will be 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** will be set to **TRUE** if **PCI_FBB_ENABLE** has been called and FBB is enabled, even if FBB is not activated on any card. It will be set to **FALSE** otherwise.

Note that in the current implementation, FBB will be enabled or disabled on the entire bus. If any device anywhere on the bus cannot support FBB, it is not enabled, even if specific sub-busses could support it.

PCI_MAX_LATENCY_FUNC_SET - FUNCPTR * pArg
This routine will be 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 will be 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 will go 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)
    {
    sysLedOn(4);
    return(sprintf(sysExcMsg,fmt,a1,a2,a3,a4,a5,a6));
    }
```

PCI_MAX_BUS_GET - int * pArg
During autoconfiguration, the library will maintain 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 will be 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

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 **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 if *cmd* is not recognized

SEE ALSO

pciAutoConfigLib

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 as follows:

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.

SEE ALSO **pciAutoConfigLib**

pciAutoConfigLibInit( )

NAME **pciAutoConfigLibInit( )** – initialize PCI autoconfig library

SYNOPSIS

```
void * pciAutoConfigLibInit
    (
    void * pArg  /* reserved for future use */
    )
```

DESCRIPTION **pciAutoConfigLib** initialization function.

RETURNS A cookie for use by subsequent **pciAutoConfigLib** function calls.

SEE ALSO **pciAutoConfigLib**

pciAutoDevReset( )

NAME **pciAutoDevReset( )** – quiesce a PCI device and reset all writeable status bits

SYNOPSIS

```
STATUS pciAutoDevReset
    (
    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.

SEE ALSO **pciAutoConfigLib**

pciAutoFuncDisable( )

NAME **pciAutoFuncDisable( )** – disable a specific PCI function

SYNOPSIS

```
void pciAutoFuncDisable
    (
    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.

SEE ALSO **pciAutoConfigLib**

pciAutoFuncEnable( )

NAME **pciAutoFuncEnable( )** – perform final configuration and enable a function

SYNOPSIS

```
void pciAutoFuncEnable
    (
    PCI_SYSTEM * pSys,  /* 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.

SEE ALSO **pciAutoConfigLib**

pciAutoGetNextClass( )

NAME

pciAutoGetNextClass( ) – find the next device of specific type from probe list

SYNOPSIS

```
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 */
    )
```

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**.

SEE ALSO

pciAutoConfigLib

pciAutoRegConfig( )

NAME

pciAutoRegConfig( ) – assign PCI space to a single PCI base address register

SYNOPSIS

```
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 */
    )
```

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.

SEE ALSO

pciAutoConfigLib

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.

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.

SEE ALSO **pciConfigShow**

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

SEE ALSO **pciConfigLib**

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**

pciConfigFuncShow( )

NAME

pciConfigFuncShow( ) – show configuration details about a function

SYNOPSIS

```
STATUS pciConfigFuncShow
    (
    int  bus,       /* bus */
    int  device,    /* device */
    int  function,  /* function */
    void *pArg      /* ignored */
    )
```

DESCRIPTION

This routine reads various information from the specified bus, device, function, and displays the information.

RETURNS

OK, always.

SEE ALSO

pciConfigShow

pciConfigInByte( )

NAME

pciConfigInByte( ) – read one byte from the PCI configuration space

SYNOPSIS

```
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 */
    )
```

DESCRIPTION

This routine reads one byte from the PCI configuration space

RETURNS

OK, or **ERROR** if this library is not initialized

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 This routine reads one longword from the PCI configuration space

RETURNS **OK**, or **ERROR** if this library is not initialized

SEE ALSO **pciConfigLib**

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 This routine reads one word from the PCI configuration space

RETURNS **OK**, or **ERROR** if this library is not initialized

SEE ALSO **pciConfigLib**

pciConfigLibInit( )

NAME

pciConfigLibInit( ) – initialize the configuration access-method and addresses

SYNOPSIS

```
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 */
    )
```

DESCRIPTION

This routine initializes the configuration access-method and addresses.

Configuration mechanism one utilizes two 32-bit I/O 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 I/O read from or an write to the configuration data port.

Configuration mechanism two uses following two single-byte I/O 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 I/O read or write transaction within the I/O range 0xc000 through 0xcfff.

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.

SEE ALSO **pciConfigLib**

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.

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.

SEE ALSO

pciConfigLib

pciConfigModifyWord( )

NAME

pciConfigModifyWord( ) – Perform a masked longword register update

SYNOPSIS

```
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 */
    )
```

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.

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

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

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

SEE ALSO **pciConfigLib**

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.

SEE ALSO **pciConfigShow**

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. In 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

SEE ALSO **pciConfigShow**

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.

SEE ALSO **pciConfigLib**

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 *busNo*).

RETURNS **OK**, or **ERROR** if the library is not initialized.

SEE ALSO **pciConfigShow**

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.

SEE ALSO **pciConfigLib**

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.

SEE ALSO **pciConfigShow**

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 and vendor ID.

RETURNS **OK**, or **ERROR** if the *deviceId* and *vendorId* didn't match.

SEE ALSO **pciConfigLib**

pciFindDeviceShow( )

NAME **pciFindDeviceShow( )** – find a PCI device and display the information

SYNOPSIS

```
STATUS pciFindDeviceShow
    (
    int vendorId,  /* vendor ID */
    int deviceId,  /* device ID */
    int index      /* desired instance of device */
    )
```

DESCRIPTION This routine finds a device by *deviceId*, then displays the information.

RETURNS **OK**, or **ERROR** if this library is not initialized.

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.

SEE ALSO **pciConfigShow**

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

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.

SEE ALSO

pciIntLib

pciIntDisconnect( )

NAME

pciIntDisconnect( ) – disconnect the interrupt handler (OBSOLETE)

SYNOPSIS

```
STATUS pciIntDisconnect
    (
    VOIDFUNCPTR *vector,  /* interrupt vector to attach to     */
    VOIDFUNCPTR 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.

SEE ALSO

pciIntLib

pciIntDisconnect2( )

NAME

pciIntDisconnect2( ) – disconnect an interrupt handler from the PCI interrupt.

SYNOPSIS

```
STATUS pciIntDisconnect2
    (
    VOIDFUNCPTR *vector,   /* interrupt vector to attach to     */
    VOIDFUNCPTR 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.

SEE ALSO **pciIntLib**

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.

SEE ALSO **pciIntLib**

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

SEE ALSO **pciConfigLib**

pcicInit( )

NAME

pcicInit( ) – initialize the PCIC chip

SYNOPSIS

```
STATUS pcicInit
    (
    int     ioBase,    /* I/O 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

```
void pcicShow
    (
    int sock  /* socket no. */
    )
```

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

```
STATUS pcmciaInit (void)
```

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( )**

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 seial 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 commonication controller.

RETURNS N/A

ERRNO Not Available

SEE ALSO **ppc403Sio**

ppc403IntWr( )

NAME **ppc403IntWr( )** – handle a transmitter interrupt

SYNOPSIS

```
void ppc403IntWr
    (
    PPC403_CHAN * pChan
    )
```

DESCRIPTION This routine handles write interrupts from the serial communication controller.

RETURNS N/A

ERRNO Not Available

SEE ALSO **ppc403Sio**

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

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

shSciDevInit( )

NAME **shSciDevInit( )** – initialize a on-chip serial communication interface

SYNOPSIS

```
void shSciDevInit
    (
    SCI_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 **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

```
void shSciIntErr
    (
    SCI_CHAN * pChan  /* channel generating the interrupt */
    )
```

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

smEndLoad( )

NAME

smEndLoad( ) – attach the SM interface to the MUX, initialize driver and device

SYNOPSIS

```
END_OBJ * smEndLoad
    (
    char * pParamStr  /* ptr to initialization parameter string */
    )
```

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 following 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 clBlks 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 clBlks 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

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 *endName*. It prints error statistics and zeros these fields if *zero* is set to **TRUE**.

EXAMPLE

```
-> 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
```

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 I/O port addresses for Super I/O chip

SYNOPSIS

```
VOID smcFdc37b78xDevCreate
    (
    SMCFDC37B78X_IOPORTS *smcFdc37b78x_iop
    )
```

DESCRIPTION This routine will initialize the **smcFdc37b78xIoPorts** data structure. These I/O ports 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 initialization 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 **sym895CtrlInit( )** – initialize a SCSI Controller Structure.

SYNOPSIS

```
STATUS sym895CtrlInit
    (
    FAST SIOP * pSiop,         /* pointer to SCSI Controller structure  */
    UINT        scsiCtrlBusId  /* SCSI bus ID of this SCSI Controller   */
    )
```

DESCRIPTION This routine initializes an SCSI Controller structure, after the structure is created with **sym895CtrlCreate( )**. 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.

RETURNS Not Available

ERRORS N/A

SEE ALSO **sym895Lib**

sym895GPIOConfig( )

NAME **sym895GPIOConfig( )** – configures general purpose pins GPIO 0-4.

SYNOPSIS

```
STATUS sym895GPIOConfig
    (
    SIOP * pSiop,      /* pointer to SIOP structure   */
    UINT8  ioEnable,  /* bits indicate input/output  */
    UINT8  mask       /* mask for ioEnable parameter */
    )
```

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.

pSiop
pointer to the SIOP structure.

ioEnable
bits 0-4 of this parameter configure GPIO 0-4 pins. 1 => input, 0 => output.

mask
bits 0-4 of this parameter identify valid bits in *ioEnable* parameter. Only those pins are configured, which have a corresonding bit set in this parameter.

RETURNS Not Available

ERRNO Not Available

SEE ALSO **sym895Lib**

sym895GPIOCtrl( )

NAME

sym895GPIOCtrl( ) – controls general purpose pins GPIO 0-4.

SYNOPSIS

```
STATUS sym895GPIOCtrl
    (
    SIOP * pSiop,    /* pointer to SIOP structure   */
    UINT8  ioState,  /* bits indicate set/reset */
    UINT8  mask      /* mask for ioState parameter */
    )
```

DESCRIPTION

This routine uses the GPREG register to set/reset of the general purpose pins available on Sym895 chip.

pSiop
pointer to the SIOP structure.

ioState
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 corresonding 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, 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 Not Available

ERRNO Not Available

SEE ALSO **sym895Lib**

sym895Loopback( )

NAME **sym895Loopback( )** – This routine performs loopback diagnotics 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, the **SELECTION**, **MSG_OUT** and **DATA_OUT** phases are checked. This will ensure 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.

ERRNO 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
Sidl     = 0x0000 Sodl     = 0x0000 Sbdl     = 0x0000
Scratchb = 0x00000200
value = 0 = 0x0
```

RETURNS **OK**, or **ERROR** if *pScsiCtrl* and *pSysScsiCtrl* are both **NULL**.

ERRNO Not Available

SEE ALSO **sym895Lib**, **sym895CtrlCreate( )**

tcicInit( )

NAME **tcicInit( )** – initialize the TCIC chip

SYNOPSIS

```
STATUS tcicInit
    (
    int      ioBase,    /* I/O base address */
    int      intVec,    /* interrupt vector */
    int      intLevel,  /* interrupt level */
    FUNCPTR showRtn     /* show routine */
    )
```

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

```
void tcicShow
    (
    int sock   /* socket no. */
    )
```

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 *xxx***ToBin** 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**

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 *initString*.

The parameter string, *initString*, is an ordered list of parameters each separated by a colon. The format of *initString* 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 *nTfds* and *nRfds*. In other cases, the number of frames selected should be greater than two.

The *memBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant **NONE**, 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 descriptors.

If the caller provides the shared memory region, 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, 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 the 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) (),       /* recv 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 respectivly (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 incomming 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
```
void wdbSlipPktDevInit
    (
    WDB_SLIP_PKT_DEV *pPktDev,      /* SLIP packetizer device */
    SIO_CHAN *       pSioChan,      /* underlying serial channel */
    void             (*stackRcv)()  /* callback when a packet arrives */
    )
```

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             nblocks,
    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 **z8530DevInit( )** – 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

z8530Int( ) – 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

z8530IntEx( ) – handle error interrupts

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 **z8530IntRd( )** – handle a reciever interrupt

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**

Keyword Index

display	device status.	**auDump( )**	187
vendor ID. find nth	device with given device &	**pciFindDevice( )**	288
initialize and configure PHY	devices.	**miiPhyInit( )**	240
print information about PCI	devices.	**pciDeviceShow( )**	287
This routine performs loopback	diagnotics on 895 chip.	**sym895Loopback( )**	316
from PCI interrupt.	disconnect interrupt handler	**pciIntDisconnect2( )**	291
(OBSOLETE).	disconnect interrupt handler	**pciIntDisconnect( )**	291
create device for ATA/IDE	disk.	**ataDevCreate( )**	154
create XBD device for ATA/IDE	disk.	**ataXbdDevCreate( )**	161
create device for floppy	disk.	**fdDevCreate( )**	218
reset specified ATA/IDE	disk controller.	**ataCtrlReset( )**	154
init ATAPI CD-ROM	disk controller.	**ataPiInit( )**	158
init ATAPI CD-ROM	disk controller.	**atapiInit( )**	171
initializes floppy	disk device.	**iPIIX4FdInit( )**	230
create PCMCIA memory	disk device.	**sramDevCreate( )**	310
NEC 765 floppy	disk device driver.	**nec765Fd**	83
ATAPI CDROM (LOCAL and PCMCIA)	disk device driver. /and	**ataDrv**	8
ATA/IDE (LOCAL and PCMCIA)	disk device driver show/	**ataShow**	13
initialize floppy	disk driver.	**fdDrv( )**	219
initialize ATA/IDE	disk driver show routine.	**ataShowInit( )**	160
show ATA/IDE	disk parameters.	**ataShow( )**	159
of sectors on current track in	DMA mode. read/write number	**ataDmaRW( )**	156
get enabled Multi word	DMA mode.	**atapiCurrentMDmaModeGet( )**	164
get enabled Single word	DMA mode.	**atapiCurrentSDmaModeGet( )**	165
get enabled Ultra	DMA mode.	**atapiCurrentUDmaModeGet( )**	166
support /get Maximum Ultra	DMA mode drive can	**atapiMaxUDmaModeGet( )**	178
get Maximum Multi word	DMA mode drive supports.	**atapiMaxMDmaModeGet( )**	176
get Maximum Single word	DMA mode drive supports.	**atapiMaxSDmaModeGet( )**	177
display RX	DMA register values.	**bcm1250MacRxDmaShow( )**	190
display TX	DMA register values.	**bcm1250MacTxDmaShow( )**	191
or off individual controllers	dma support. turn on	**ataDmaToggle( )**	156
initialize device and mount	DOS file system.	**pccardMkfs( )**	260
mount	DOS file system.	**pccardMount( )**	261
National Semiconductor	DP83902A ST-NIC.	**ns83902End**	84
logical number of cylinders in	drive. get	**atapiCurrentCylinderCountGet( )**	163
number of read/write heads in	drive. get	**atapiCurrentHeadCountGet( )**	163
get number of cylinders in	drive.	**atapiCylinderCountGet( )**	166
get features supported by	drive.	**atapiFeatureSupportedGet( )**	169
get firmware revision of	drive.	**atapiFirmwareRevisionGet( )**	170
get number heads in	drive.	**atapiHeadCountGet( )**	171
Control	drive.	**atapiIoctl( )**	172
get model number of	drive.	**atapiModelNumberGet( )**	178
SCAN packet command to ATAPI	drive. issue	**atapiScan( )**	182
issues SEEK packet command to	drive.	**atapiSeek( )**	183
SPEED packet command to ATAPI	drive. issue SET CD	**atapiSetCDSpeed( )**	183
packet command to ATAPI	drive. issue STOP PLAY/SCAN	**atapiStopPlayScan( )**	185
UNIT READY command to ATAPI	drive. issue TEST	**atapiTestUnitRdy( )**	185
ATA/ATAPI version number of	drive. get	**atapiVersionNumberGet( )**	186
status. Check status of	drive and compare to requested	**ataStatusChk( )**	160
get Maximum PIO mode that	drive can support.	**atapiMaxPioModeGet( )**	177

configures	general purpose pins GPIO 0-4.	**sym895GPIOConfig( )**	314
controls	general purpose pins GPIO 0-4.	**sym895GPIOCtrl( )**	315
Ethernet network/ Second	Generation Motorola FCC	**motFcc2End**	65
general purpose pins	GPIO 0-4. configures	**sym895GPIOConfig( )**	314
controls general purpose pins	GPIO 0-4.	**sym895GPIOCtrl( )**	315
END style Marvell/Galileo	GT642xx Ethernet network/	**wancomEnd**	126
to MII optional registers	handlers. set pointers	**miiPhyOptFuncMultiSet( )**	242
for NCR 53C8xx SIOP. set	hardware-dependent registers	**ncr810SetHwRegister( )**	253
device. print	header of specified PCI	**pciHeaderShow( )**	289
configure all nonexcluded PCI	headers. Automatically	**pciAutoCfg( )**	264
/configure all nonexcluded PCI	headers (obsolete).	**pciAutoConfig( )**	272
get number of read/write	heads in drive.	**atapiCurrentHeadCountGet( )**	163
get number	heads in drive.	**atapiHeadCountGet( )**	171
Communications Interface)/	Hitachi SH SCI (Serial	**shSciSio**	112
Databook TCIC/2 PCMCIA	host bus adaptor chip driver.	**tcic**	124
Intel 82365SL PCMCIA	host bus adaptor chip library.	**pcic**	108
library. Intel 82365SL PCMCIA	host bus adaptor chip show	**pcicShow**	108
Databook TCIC/2 PCMCIA	host bus adaptor chip show/	**tcicShow**	124
	I8250 serial driver.	**i8250Sio**	46
NS16550 serial driver for	IBM PPC403GA evaluation.	**evbNs16550Sio**	37
chip device driver for	IBM-PC LPT. parallel	**lptDrv**	59
	identify device.	**ataDevIdentify( )**	155
enable PCMCIA Etherlink	III card.	**pccardEltEnabler( )**	260
buffer I/O	Implementation.	**bio**	18
Media	Independent Interface library.	**miiLib**	63
support. turn on or off	individual controllers dma	**ataDmaToggle( )**	156
Shows debug	info for this library.	**erfShow( )**	215
to show Mii settings in Phy	Info structure. /Function	**motFccMiiShow( )**	249
Device	Infrastructure Library.	**device**	28
controller.	init ATAPI CD-ROM disk	**ataPiInit( )**	158
controller.	init ATAPI CD-ROM disk	**atapiInit( )**	171
parse	init string.	**elt3c509Parse( )**	204
FCC parameter ram addresses,	initial BD and cluster/ /show	**motFccDrvShow( )**	245
Return current RU status and	int mask.	**fei82557GetRUStatus( )**	223
adaptor chip library.	Intel 82365SL PCMCIA host bus	**pcic**	108
adaptor chip show library.	Intel 82365SL PCMCIA host bus	**pcicShow**	108
	Intel/	**gei82543End**	42
interface driver. END-style	Intel 82557 Ethernet network	**fei82557End**	37
interface driver. END style	Intel Olicom PCMCIA network	**iOlicomEnd**	47
handle receiver	interrupt.	**ambaIntRx( )**	147
handle transmitter	interrupt.	**ambaIntTx( )**	148
handle receiver/transmitter	interrupt.	**i8250Int( )**	229
handle receiver	interrupt.	**ns16550IntRd( )**	257
handle transmitter	interrupt.	**ns16550IntWr( )**	258
handler for shared PCI	interrupt. interrupt	**pciInt( )**	290
interrupt handler to PCI	interrupt. connect	**pciIntConnect( )**	290
interrupt handler from PCI	interrupt. disconnect	**pciIntDisconnect2( )**	291
handle receiver	interrupt.	**ppc403IntRd( )**	297
handle transmitter	interrupt.	**ppc403IntWr( )**	297
handle SMC	interrupt.	**ppc860Int( )**	298

END based packet driver for	lightweight UDP/IP.	**wdbEndPktDrv**	129
pipe packet driver for	lightweight UDP/IP.	**wdbPipePktDrv**	130
free tuples from	linked list.	**cisFree( )**	195
extended capability in ECP	linked list. find	**pciConfigExtCapFind( )**	277
free tuples from linked	list.	**cisFree( )**	195
of specific type from probe	list. find next device	**pciAutoGetNextClass( )**	275
capability in ECP linked	list. find extended	**pciConfigExtCapFind( )**	277
initializes B69000 chip and	loads font in memory.	**ctB69000VgaInit( )**	196
initializes VGA chip and	loads font in memory.	**vgaInit( )**	322
ATA/IDE and ATAPI CDROM	(LOCAL and PCMCIA) disk device/	**ataDrv**	8
driver show routine. ATA/IDE	(LOCAL and PCMCIA) disk device	**ataShow**	13
	locate addresses in packet.	**endEtherPacketAddrGet( )**	207
drive. get	logical number of cylinders in ..	**atapiCurrentCylinderCountGet( )**	163
configuration space. read one	longword from PCI	**pciConfigInLong( )**	279
Perform masked	longword register update.	**pciConfigModifyByte( )**	281
Perform masked	longword register update.	**pciConfigModifyLong( )**	282
Perform masked	longword register update.	**pciConfigModifyWord( )**	283
space. write one	longword to PCI configuration	**pciConfigOutLong( )**	284
chip. This routine performs	loopback diagnotics on 895	**sym895Loopback( )**	316
for PCI ISA/IDE Xcelerator.	low-level initialization code	**iPIIX4**	50
ATA device.	low-level initialization of	**iPIIX4AtaInit( )**	230
chip device driver for IBM-PC	LPT. parallel	**lptDrv**	59
initialize	LPT driver.	**lptDrv( )**	235
create device for	LPT port.	**lptDevCreate( )**	234
show	LPT statistics.	**lptShow( )**	236
82541/82543/82544/82545/82546/	MAC driver. Intel 82540/	**gei82543End**	42
END style Au	MAC Ethernet driver.	**auEnd**	14
END style BCM1250	MAC Ethernet driver.	**bcm1250MacEnd**	17
display	MAC register values.	**bcm1250MacShow( )**	191
	map device.	**devMap( )**	200
specified ISA address space.	map PCMCIA memory onto	**sramMap( )**	311
Show routines of PCI bus(I/O	mapped) library.	**pciConfigShow**	106
Ethernet network/ END style	Marvell/Galileo GT642xx	**wancomEnd**	126
current RU status and int	mask. Return	**fei82557GetRUStatus( )**	223
update. Perform	masked longword register	**pciConfigModifyByte( )**	281
update. Perform	masked longword register	**pciConfigModifyLong( )**	282
update. Perform	masked longword register	**pciConfigModifyWord( )**	283
drive supports. get	Maximum Multi word DMA mode ...	**atapiMaxMDmaModeGet( )**	176
Get	maximum number of Categories.	**erfCategoriesAvailable( )**	210
Get	maximum number of Types.	**erfCategoriesAvailable( )**	211
can support. get	Maximum PIO mode that drive	**atapiMaxPioModeGet( )**	177
drive supports. get	Maximum Single word DMA mode	**atapiMaxSDmaModeGet( )**	177
can support. get	Maximum Ultra DMA mode drive	**atapiMaxUDmaModeGet( )**	178
library.	Media Independent Interface	**miiLib**	63
Version. get	Media Stat Notification ..	**atapiRemovMediaStatusNotifyVerGet( )**	182
Issues prevent/allow	medium removal packet command ...	**atapiCtrlMediumRemoval( )**	162
single PCI-Cardbus/ set	mem and I/O registers for	**pciAutoCardBusConfig( )**	263
set flags	member of END_OBJ structure.	**endObjFlagSet( )**	208
free bio	memory.	**bio_free( )**	193
B69000 chip and loads font in	memory. initializes	**ctB69000VgaInit( )**	196

conditions. align	PCI address and check boundary	**pciAutoAddrAlign( )**	262
initialize	PCI autoconfig library.	**pciAutoConfigLibInit( )**	273
assign PCI space to single	PCI base address register.	**pciAutoRegConfig( )**	275
configure device on	PCI bus.	**pciDevConfig( )**	286
allocation facility.	PCI bus scan and resource	**pciAutoConfigLib**	87
Show routines of	PCI bus(I/O mapped) library.	**pciConfigShow**	106
nth occurrence of device by	PCI class code. find	**pciFindClass( )**	287
read one byte from	PCI configuration space.	**pciConfigInByte( )**	278
read one longword from	PCI configuration space.	**pciConfigInLong( )**	279
read one word from	PCI configuration space.	**pciConfigInWord( )**	279
write one byte to	PCI configuration space.	**pciConfigOutByte( )**	283
write one longword to	PCI configuration space.	**pciConfigOutLong( )**	284
write one 16-bit word to	PCI configuration space.	**pciConfigOutWord( )**	284
support for PCI drivers.	PCI Configuration space access	**pciConfigLib**	95
print header of specified	PCI device.	**pciHeaderShow( )**	289
information. find	PCI device and display	**pciFindDeviceShow( )**	289
writeable status/ quiesce	PCI device and reset all	**pciAutoDevReset( )**	273
print information about	PCI devices.	**pciDeviceShow( )**	287
space access support for	PCI drivers. /Configuration	**pciConfigLib**	95
driver. END-style DEC 21x40	PCI Ethernet network interface	**dec21x40End**	22
disable specific	PCI function.	**pciAutoFuncDisable( )**	274
configure all nonexcluded	PCI headers. Automatically	**pciAutoCfg( )**	264
/configure all nonexcluded	PCI headers (obsolete).	**pciAutoConfig( )**	272
interrupt handler for shared	PCI interrupt.	**pciInt( )**	290
connect interrupt handler to	PCI interrupt.	**pciIntConnect( )**	290
interrupt handler from	PCI interrupt. disconnect	**pciIntDisconnect2( )**	291
/initialization code for	PCI ISA/IDE Xcelerator.	**iPIIX4**	50
library (SCSI-2). NCR 53C8xx	PCI SCSI I/O Processor (SIOP)	**ncr810Lib**	80
	PCI Shared Interrupt support.	**pciIntLib**	107
address register. assign	PCI space to single PCI base	**pciAutoRegConfig( )**	275
show	PCI topology.	**pciConfigTopoShow( )**	286
set or get	**pciAutoConfigLib** options	**pciAutoCfgCtl( )**	265
initialize	PCIC chip.	**pcicInit( )**	293
show all configurations of	PCIC chip.	**pcicShow( )**	293
and I/O registers for single	PCI-Cardbus bridge. set mem	**pciAutoCardBusConfig( )**	263
initialize	**pciIntLib** module.	**pciIntLibInit( )**	292
initializes	PCI-ISA/IDE bridge.	**iPIIX4KbdInit( )**	232
show all configurations of	PCMCIA chip.	**pcmciaShow( )**	294
	PCMCIA CIS library.	**cisLib**	19
	PCMCIA CIS show library.	**cisShow**	19
get	PCMCIA configuration register.	**cisConfigregGet( )**	194
set	PCMCIA configuration register.	**cisConfigregSet( )**	194
/and ATAPI CDROM (LOCAL and	PCMCIA) disk device driver.	**ataDrv**	8
show/ ATA/IDE (LOCAL and	PCMCIA) disk device driver	**ataShow**	13
all show routines for	PCMCIA drivers. initialize	**pcmciaShowInit( )**	295
enable	PCMCIA Etherlink III card.	**pccardEltEnabler( )**	260
facilities. generic	PCMCIA event-handling	**pcmciaLib**	108
initialize	PCMCIA event-handling package.	**pcmciaInit( )**	294
handle task-level	PCMCIA events.	**pcmciad( )**	295
driver. Databook TCIC/2	PCMCIA host bus adaptor chip	**tcic**	124

registers.	prints current value of NIC	**ns83902RegShow( )**	259
device of specific type from	probe list. find next	**pciAutoGetNextClass( )**	275
NCR 53C8xx PCI SCSI I/O	Processor (SIOP) library/	**ncr810Lib**	80
configures general	purpose pins GPIO 0-4.	**sym895GPIOConfig( )**	314
controls general	purpose pins GPIO 0-4.	**sym895GPIOCtrl( )**	315
Category Event Processing	Queue. Creates	**erfCategoryQueueCreate( )**	212
Get size of default	queue.	**erfDefaultQueueSizeGet( )**	212
regarding Tx ring and Rx	queue desc. /information	**wancomEndDbg( )**	322
all writeable status bits.	quiesce PCI device and reset	**pciAutoDevReset( )**	273
	Raises event.	**erfEventRaise( )**	213
show FCC CP ethernet parameter	ram. Debug Function to	**motFccEramShow( )**	247
to show FCC CP parameter	ram. Debug Function	**motFccPramShow( )**	249
/Function to show FCC parameter	ram addresses, initial BD and/	**motFccDrvShow( )**	245
to show FCC CP internal	ram parameters. /Function	**motFccIramShow( )**	248
provide	raw I/O access.	**fdRawio( )**	219
	Read all PHY registers out.	**dec21145SPIReadBack( )**	197
to ATAPI device. issue	READ CD-ROM CAPACITY command	**atapiReadCapacity( )**	181
	Read drive parameters.	**ataParamRead( )**	158
configuration space.	read one byte from PCI	**pciConfigInByte( )**	278
configuration space.	read one longword from PCI	**pciConfigInLong( )**	279
ATAPI Device.	read one or more blocks from	**atapiRead10( )**	180
configuration space.	read one word from PCI	**pciConfigInWord( )**	279
device. issue	READ TOC command to ATAPI	**atapiReadTocPmaAtip( )**	181
ROM.	read two bytes from serial	**dec21140SromWordRead( )**	197
display values of all	readable NCR 53C8xx SIOP/	**ncr810Show( )**	254
display values of all	readable SYM 53C8xx SIOP/	**sym895Show( )**	318
required sector.	read/write data from/to	**ataRW( )**	159
get number of	read/write heads in drive.	**atapiCurrentHeadCountGet( )**	163
on current track in DMA mode.	read/write number of sectors	**ataDmaRW( )**	156
issue TEST UNIT	READY command to ATAPI drive.	**atapiTestUnitRdy( )**	185
handle channel's	receive-character interrupt.	**shSciIntRcv( )**	303
handle channel's	receive-character interrupt.	**shScifIntRcv( )**	305
handle	receiver interrupt.	**ambaIntRx( )**	147
handle	receiver interrupt.	**ns16550IntRd( )**	257
handle	receiver interrupt.	**ppc403IntRd( )**	297
handle	receiver interrupt.	**rm9000x2glIntRd( )**	301
interrupt. handle	receiver/transmitter	**i8250Int( )**	229
for NS 16550 chip. handle	receiver/transmitter interrupt	**evbNs16550Int( )**	217
handle	reciever interrupt.	**z8530IntRd( )**	334
Print pDrvCtrl information	regarding Tx ring and Rx queue/	**wancomEndDbg( )**	322
get PCMCIA configuration	register.	**cisConfigregGet( )**	194
set PCMCIA configuration	register.	**cisConfigregSet( )**	194
to single PCI base address	register. assign PCI space	**pciAutoRegConfig( )**	275
for Configuration Address	Register. pack parameters	**pciConfigBdfPack( )**	276
Perform masked longword	register update.	**pciConfigModifyByte( )**	281
Perform masked longword	register update.	**pciConfigModifyLong( )**	282
Perform masked longword	register update.	**pciConfigModifyWord( )**	283
set specified	register value.	**gei82543RegSet( )**	227
get specified	register value in 82543 chip.	**gei82543RegGet( )**	226
get	register value in PHY.	**gei82543PhyRegGet( )**	225

issues	SEEK packet command to drive.	**atapiSeek( )**	183
National	Semiconductor DP83902A ST-NIC.	**ns83902End**	84
interface. initialize on-chip	serial communication	**shSciDevInit( )**	302
interface. initialize on-chip	serial communication	**shScifDevInit( )**	304
Controller driver. Z8530 SCC	Serial Communications	**z8530Sio**	137
Interface)/ Hitachi SH SCI	(Serial Communications	**shSciSio**	112
Interface)/ Renesas SH SCIF	(Serial Communications	**shScifSio**	113
ppc403GA	serial driver.	**ppc403Sio**	110
Motorola MPC800 SMC UART	serial driver.	**ppc860Sio**	111
I8250	serial driver.	**i8250Sio**	46
evaluation. NS16550	serial driver for IBM PPC403GA	**evbNs16550Sio**	37
agent.	serial line packetizer for WDB	**wdbSlipPktDrv**	131
get drive	serial number.	**atapiDriveSerialNumberGet( )**	167
initialize	serial port unit.	**ppc403DevInit( )**	295
read two bytes from	serial ROM.	**dec21140SromWordRead( )**	197
interrupt	service for card interrupts.	**iOlicomIntHandle( )**	230
Controller. interrupt	service routine for SCSI	**sym895Intr( )**	315
Interface) driver. Hitachi	SH SCI (Serial Communications	**shSciSio**	112
Interface) driver. Renesas	SH SCIF (Serial Communications	**shScifSio**	113
PCI	Shared Interrupt support.	**pciIntLib**	107
show information about	shared memory network.	**smNetShow( )**	308
driver show routines.	shared memory network END	**smEndShow**	118
interface driver. END	shared memory (SM) network	**smEnd**	113
interrupt handler for	shared PCI interrupt.	**pciInt( )**	290
PCIC chip.	show all configurations of	**pcicShow( )**	293
PCMCIA chip.	show all configurations of	**pcmciaShow( )**	294
TCIC chip.	show all configurations of	**tcicShow( )**	319
	show ATA/IDE disk parameters.	**ataShow( )**	159
	show CIS information.	**cisShow( )**	196
about function.	show configuration details	**pciConfigFuncShow( )**	278
word.	show decoded value of command	**pciConfigCmdWordShow( )**	276
word.	show decoded value of status	**pciConfigStatusWordShow( )**	285
socket interfaces.	show device information on all	**tffsShowAll( )**	321
specific socket interface.	show device information on	**tffsShow( )**	321
data. Debug Function to	show driver-specific control	**motFccShow( )**	250
ram. Debug Function to	show FCC CP ethernet parameter	**motFccEramShow( )**	247
parameters. Debug Function to	show FCC CP internal ram	**motFccIramShow( )**	248
Debug Function to	show FCC CP parameter ram	**motFccPramShow( )**	249
addresses,/ Debug Function to	show FCC parameter ram	**motFccDrvShow( )**	245
memory network.	show information about shared	**smNetShow( )**	308
PCMCIA host bus adaptor chip	show library. Intel 82365SL	**pcicShow**	108
PCMCIA	show library.	**pcmciaShow**	110
PCMCIA host bus adaptor chip	show library. Databook TCIC/2	**tcicShow**	124
PCMCIA CIS	show library.	**cisShow**	19
	show LPT statistics.	**lptShow( )**	236
Debug Function to	show MIB statistics.	**motFccMibShow( )**	248
structure. Debug Function to	show Mii settings in Phy Info	**motFccMiiShow( )**	249
	show PCI topology.	**pciConfigTopoShow( )**	286
	Show Receive ring.	**fei82557ShowRxRing( )**	223
	Show Receive Ring details.	**motFccDumpRxRing( )**	245

and PCMCIA) disk device driver	show routine. ATA/IDE (LOCAL	**ataShow**	13
initialize ATA/IDE disk driver	show routine.	**ataShowInit( )**	160
	show routine for MII library.	**miiShow( )**	244
memory network END driver	show routines. shared	**smEndShow**	118
Reporting Framework Library	**Show** routines. Event	**erfShow**	36
drivers. initialize all	show routines for PCMCIA	**pcmciaShowInit( )**	295
mapped) library.	**Show** routines of PCI bus(I/O	**pciConfigShow**	106
	Show Transmit Ring details.	**motFccDumpTxRing( )**	246
library.	Shows debug info for this	**erfShow( )**	215
structure for NCR 53C8xx	SIOP. create control	**ncr810CtrlCreate( )**	251
structure for NCR 53C8xx	SIOP. initialize control	**ncr810CtrlInit( )**	252
registers for NCR 53C8xx	SIOP. set hardware-dependent	**ncr810SetHwRegister( )**	253
53C8xx PCI SCSI I/O Processor	(SIOP) library (SCSI-2). NCR	**ncr810Lib**	80
of all readable NCR 53C8xx	SIOP registers. /values	**ncr810Show( )**	254
of all readable SYM 53C8xx	SIOP registers. /values	**sym895Show( )**	318
agent. initialize	SLIP packet device for WDB	**wdbSlipPktDevInit( )**	326
initialize driver and/ attach	SM interface to MUX,	**smEndLoad( )**	306
END shared memory	(SM) network interface driver.	**smEnd**	113
initialize	SMC.	**ppc860DevInit( )**	298
handle	SMC interrupt.	**ppc860Int( )**	298
Motorola MPC800	SMC UART serial driver.	**ppc860Sio**	111
device information on specific	socket interface. show	**tffsShow( )**	321
show device information on all	socket interfaces.	**tffsShowAll( )**	321
(fdc37b78x) initialization	source module. superIO	**smcFdc37b78x**	118
VGA 3+ mode initialization	source module.	**vgaInit**	126
CHIPS B69000 initialization	source module.	**ctB69000Vga**	20
byte from PCI configuration	space. read one	**pciConfigInByte( )**	278
from PCI configuration	space. read one longword	**pciConfigInLong( )**	279
word from PCI configuration	space. read one	**pciConfigInWord( )**	279
one byte to PCI configuration	space. write	**pciConfigOutByte( )**	283
longword to PCI configuration	space. write one	**pciConfigOutLong( )**	284
word to PCI configuration	space. write one 16-bit	**pciConfigOutWord( )**	284
onto specified ISA address	space. map PCMCIA memory	**sramMap( )**	311
drivers. PCI Configuration	space access support for PCI	**pciConfigLib**	95
address register. assign PCI	space to single PCI base	**pciAutoRegConfig( )**	275
generate	special cycle with message.	**pciSpecialCycle( )**	292
drive. issue SET CD	SPEED packet command to ATAPI	**atapiSetCDSpeed( )**	183
PCMCIA	SRAM device driver.	**sramDrv**	121
install PCMCIA	SRAM memory driver.	**sramDrv( )**	311
get Media	Stat Notification Version. .	**atapiRemovMediaStatusNotifyVerGet( )**	182
Display	statistical counters.	**amd8111LanDumpPrint( )**	148
dump	statistical counters.	**amd8111LanErrCounterDump( )**	149
Display	statistical counters.	**fei82557DumpPrint( )**	220
dump	statistical counters.	**fei82557ErrCounterDump( )**	222
show LPT	statistics.	**lptShow( )**	236
Debug Function to show MIB	statistics.	**motFccMibShow( )**	248
initialize polling	statistics updates.	**endPollStatsInit( )**	209
drive and compare to requested	status. Check status of	**ataStatusChk( )**	160
display device	status.	**auDump( )**	187
Return current RU	status and int mask.	**fei82557GetRUStatus( )**	223