1: Libraries

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

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

2: Routines

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

Keyword Index

This section is a “permuted index” of keywords found in the NAME line of each reference entry. The keyword for each index item is left-aligned in column 2. The remaining words in column 1 and 2 show the context for the keyword.
Libraries

aic7880Lib – Adaptec 7880 SCSI Host Adapter Library File ...................................................... 5
ambaSio – ARM AMBA UART tty driver ................................................................................. 8
ataDrv – ATA/IDE and ATAPI CDROM (LOCAL and PCMCIA) disk device driver... 11
ataShow – ATA/IDE (LOCAL and PCMCIA) disk device driver show routine .......... 14
auEnd – END style Au MAC Ethernet driver....................................................................... 14
cd2400Sio – CL-CD2400 MPCC serial driver ........................................................................ 18
cisLib – PCMCIA CIS library............................................................................................... 18
cisShow – PCMCIA CIS show library ................................................................................... 19
coldfireSio – ColdFire Serial Communications driver .......................................................... 19
ctB69000Vga – a CHIPS B69000 initialization source module........................................ 21
dec21x4xEnd – END style DEC 21x4x PCI Ethernet network interface driver.................... 24
dec21x40End – END-style DEC 21x40 PCI Ethernet network interface driver ............... 28
ei82596End – END style Intel 82596 Ethernet network interface driver............................ 35
ei3c90xEnd – END network interface driver for 3COM 3C90xB XL..................................... 38
e13c509End – END network interface driver for 3COM 3C509 ........................................... 42
endLib – support library for END-based drivers................................................................. 45
evbNs16550Sio – NS16550 serial driver for the IBM PPC403GA evaluation ..................... 45
fei82557End – END style Intel 82557 Ethernet network interface driver............................ 47
gel82543End – Intel PRO/1000 F/T/XF/XT/MT network adapter END driver ............... 50
i8250Sio – I8250 serial driver ........................................................................................... 54
if_cpm – Motorola CPM core network interface driver ..................................................... 54
if_cs – Crystal Semiconductor CS8900 network interface driver ..................................... 58
if_dc – DEC 21x4x Ethernet LAN network interface driver ............................................. 61
if_eex – Intel EtherExpress 16 network interface driver .................................................... 65
if_ei – Intel 82596 Ethernet network interface driver ....................................................... 66
if_eidve – Intel 82596 Ethernet network interface driver for DVE-SH7XXX................. 69
if_eihk – Intel 82596 Ethernet network interface driver for hkv3500............................. 73
if_elc – SMC 8013WC Ethernet network interface driver ............................................... 76
if elt – 3Com 3C509 Ethernet network interface driver .................................................. 77
if_ene – Novell/Eagle NE2000 network interface driver ............................................... 78
<table>
<thead>
<tr>
<th>Driver</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_esmc</td>
<td>Ampro Ethernet2 SMC-91c9x Ethernet network interface driver</td>
<td>80</td>
</tr>
<tr>
<td>if_fei</td>
<td>Intel 82557 Ethernet network interface driver</td>
<td>81</td>
</tr>
<tr>
<td>if_fn</td>
<td>Fujitsu MB86960 NICE Ethernet network interface driver</td>
<td>83</td>
</tr>
<tr>
<td>if_in</td>
<td>AMD Am7990 LANCE Ethernet network interface driver</td>
<td>85</td>
</tr>
<tr>
<td>if_InPci</td>
<td>AMD Am79C970 PCnet-PCI Ethernet network interface driver</td>
<td>88</td>
</tr>
<tr>
<td>if_loop</td>
<td>software loopback network interface driver</td>
<td>92</td>
</tr>
<tr>
<td>if_mbc</td>
<td>Motorola 68EN302 network-interface driver</td>
<td>92</td>
</tr>
<tr>
<td>if_nicEvb</td>
<td>National Semiconductor ST-NIC Chip network interface driver</td>
<td>95</td>
</tr>
<tr>
<td>if_sl</td>
<td>Serial Line IP (SLIP) network interface driver</td>
<td>96</td>
</tr>
<tr>
<td>if_sm</td>
<td>shared memory backplane network interface driver</td>
<td>98</td>
</tr>
<tr>
<td>if_sn</td>
<td>National Semiconductor DP83932B SONIC Ethernet network driver</td>
<td>99</td>
</tr>
<tr>
<td>if Ultra</td>
<td>SMC Elite Ultra Ethernet network interface driver</td>
<td>102</td>
</tr>
<tr>
<td>iOlicomEnd</td>
<td>END style Intel Olicom PCMCIA network interface driver</td>
<td>103</td>
</tr>
<tr>
<td>iPIIX4</td>
<td>low level initialization code for PCI ISA/IDE Xcelerator</td>
<td>106</td>
</tr>
<tr>
<td>ln97xEnd</td>
<td>END style AMD Am79C97X PCnet-PCI Ethernet driver</td>
<td>110</td>
</tr>
<tr>
<td>ln7990End</td>
<td>END style AMD 7990 LANCE Ethernet network interface driver</td>
<td>115</td>
</tr>
<tr>
<td>lptDrv</td>
<td>parallel chip device driver for the IBM-PC LPT</td>
<td>118</td>
</tr>
<tr>
<td>m68302Sio</td>
<td>Motorola MC68302 bimodal tty driver</td>
<td>119</td>
</tr>
<tr>
<td>m68332Sio</td>
<td>Motorola MC68332 tty driver</td>
<td>120</td>
</tr>
<tr>
<td>m68360Sio</td>
<td>Motorola MC68360 SCC UART serial driver</td>
<td>120</td>
</tr>
<tr>
<td>m68562Sio</td>
<td>MC68562 DUSCC serial driver</td>
<td>121</td>
</tr>
<tr>
<td>m68681Sio</td>
<td>M68681 serial communications driver</td>
<td>121</td>
</tr>
<tr>
<td>m68901Sio</td>
<td>MC68901 MFP tty driver</td>
<td>124</td>
</tr>
<tr>
<td>mb86940Sio</td>
<td>MB 86940 UART tty driver</td>
<td>124</td>
</tr>
<tr>
<td>mb86960End</td>
<td>END style Fujitsu MB86960 Ethernet network interface driver</td>
<td>125</td>
</tr>
<tr>
<td>mb87030Lib</td>
<td>Fujitsu MB87030 SCSi Protocol Controller (SPC) library</td>
<td>126</td>
</tr>
<tr>
<td>mbcEnd</td>
<td>Motorola 68302fads END network interface driver</td>
<td>127</td>
</tr>
<tr>
<td>miiLib</td>
<td>Media Independent Interface library</td>
<td>130</td>
</tr>
<tr>
<td>motCpmEnd</td>
<td>END style Motorola MC68EN360/MPC800 network interface driver</td>
<td>132</td>
</tr>
<tr>
<td>motFccEnd</td>
<td>END style Motorola FCC Ethernet network interface driver</td>
<td>135</td>
</tr>
<tr>
<td>motFecEnd</td>
<td>END style Motorola FEC Ethernet network interface driver</td>
<td>143</td>
</tr>
<tr>
<td>n72001Sio</td>
<td>NEC PD72001 MPSC (Multiprotocol Serial Communications Controller)</td>
<td>150</td>
</tr>
<tr>
<td>ncr710ComLib</td>
<td>common library for ncr710Lib.c and ncr710lib2.c</td>
<td>150</td>
</tr>
<tr>
<td>ncr710Lib</td>
<td>NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-1)</td>
<td>151</td>
</tr>
<tr>
<td>ncr710Lib2</td>
<td>NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-2)</td>
<td>151</td>
</tr>
<tr>
<td>ncr810Lib</td>
<td>NCR 53C8xx PCI SCSI I/O Processor (SIOP) library (SCSI-2)</td>
<td>152</td>
</tr>
<tr>
<td>ncr5390Lib</td>
<td>NCR5390 SCSI-Bus Interface Controller library (SBIC)</td>
<td>153</td>
</tr>
<tr>
<td>ncr5390Lib1</td>
<td>NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-1)</td>
<td>154</td>
</tr>
<tr>
<td>ncr5390Lib2</td>
<td>NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-2)</td>
<td>154</td>
</tr>
<tr>
<td>ne2000End</td>
<td>NE2000 END network interface driver</td>
<td>155</td>
</tr>
<tr>
<td>nec765Fd</td>
<td>NEC 765 floppy disk device driver</td>
<td>157</td>
</tr>
<tr>
<td>nicEvbEnd</td>
<td>National Semiconductor ST-NIC Chip network interface driver</td>
<td>157</td>
</tr>
<tr>
<td>ns16550Sio</td>
<td>NS 16550 UART tty driver</td>
<td>159</td>
</tr>
<tr>
<td>ns83902End</td>
<td>National Semiconductor DP83902A ST-NIC..................................................</td>
<td>160</td>
</tr>
<tr>
<td>nvr4101DSIU</td>
<td>NEC VR4101 DSII UART tty driver</td>
<td>161</td>
</tr>
</tbody>
</table>
nvr4101SIU/Sio – NEC VR4101 SIU UART tty driver ......................................................... 162
nvr4102DSIU/Sio – NEC VR4102 DSIU UART tty driver .................................................. 162
pcardLib – PC CARD enabler library ........................................................................... 164
pcAutoConfigLib – PCI bus scan and resource allocation facility ......................... 165
pcic – Intel 82365SL PCMCIA host adaptor chip library ......................................... 173
pciConfigLib – PCI Configuration space access support for PCI drivers ............... 174
pciConfigShow – show routines of PCI bus (IO mapped) library ............................. 185
pcicShow – Intel 82365SL PCMCIA host bus adaptor chip show library ................ 185
pciIntLib – PCI Shared Interrupt support ................................................................. 186
pcmciaLib – generic PCMCIA event-handling facilities ........................................... 186
pcmciaShow – PCMCIA show library ........................................................................ 187
ppc403Sio – ppc403GA serial driver ......................................................................... 188
ppc555SciSio – MPC555 SCI serial driver .................................................................. 188
ppc860Sio – Motorola MPC800 SMC UART serial driver ......................................... 189
sa1100Sio – Digital Semiconductor SA-1100 UART tty driver................................ 190
sab82532 – Siemens SAB 82532 UART tty driver ...................................................... 192
sh7615End – sh7615End END network interface driver ........................................... 193
shScifSio – Hitachi SH SCIF (Serial Communications Interface) driver .................... 195
shSciSio – Hitachi SH SCI (Serial Communications Interface) driver ....................... 195
smcFdc37b78x – a super IO (fdc37b78x) initialization source module ..................... 196
smNetLib – VxWorks interface to shared memory network (backplane) driver .... 198
smNetShow – shared memory network driver show routines .................................... 199
sn83932End – Nat. Semi DP83932B SONIC Ethernet driver ...................................... 199
sramDrv – PCMCIA SRAM device driver .................................................................... 201
sta16552Sio – ST 16C552 DUART tty driver ................................................................. 202
sym895Lib – SCSI-2 driver for Symbios SYM895 SCSI Controller ......................... 204
tcic – Databook TCIC/2 PCMCIA host bus adaptor chip driver .............................. 207
tcicShow – Databook TCIC/2 PCMCIA host bus adaptor chip show library ............ 207
ultraEnd – SMC Ultra Elite END network interface driver ....................................... 208
vgaInit – a VGA 3+ mode initialization source module ............................................. 210
wd33c93Lib – WD33C93 SCSI-Bus Interface Controller (SBIC) library ..................... 211
wd33c93Lib1 – WD33C93 SCSI-Bus Interface Controller library (SCSI-1) ............... 211
wd33c93Lib2 – WD33C93 SCSI-Bus Interface Controller library (SCSI-2) ............... 212
wdbEndPktDrv – END based packet driver for lightweight UDP/IP .......................... 212
wdbNetromPktDrv – NETROM packet driver for the WDB agent ............................. 213
wdbPipePktDrv – pipe packet driver for lightweight UDP/IP ................................... 213
wdbSlipPktDrv – serial line packet-size for the WDB agent ....................................... 215
wdbTsfsDrv – virtual generic file I/O driver for the WDB agent ............................... 216
wdbUlipPktDrv – WDB communication interface for the ULIP driver .................... 219
wdbVioDrv – virtual tty I/O driver for the WDB agent ............................................. 220
z8530Sio – Z8530 SCC Serial Communications Controller driver .......................... 221
aic7880Lib

NAME

aic7880Lib – Adaptec 7880 SCSI Host Adapter Library File

ROUTINES

aic7880CtrlCreate() – create a control structure for the AIC 7880
aic7880sCompleted() – successfully completed execution of a client thread
aic7880EnableFast20() – enable double speed SCSI data transfers
aic7880dFifoThresholdSet() – set the data FIFO threshold
aic7880GetNumOfBuses() – perform a PCI bus scan
aic7880ReadConfig() – read from PCI config space
aic7880WriteConfig() – read to PCI config space

DESCRIPTION

This is the I/O driver for the Adaptec AIC 7880 PCI Bus Master Single Chip SCSI Host Adapter. It is designed to work with scsi2Lib. This driver runs in conjunction with the HIM (Hardware Interface Module) supplied by Adaptec. The AIC 7880 SCSI Host Adapter driver supports the following features:

Fast, Double Speed, 20 MHz data transfers.
16 bit Wide Synchronous Data transfers.
Tagged Command Queueing.
Data FIFO threshold selection.
Disconnect / Reconnect support.
Multiple Initiator support.
Multiple Controller support.

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).

To use this driver, enable the INCLUDE_AIC7880_SCSI component (VxAE).

OPERATIONS OVERVIEW

The host processor initiates a SCSI I/O operation by programming a data structure called SCB (SCSI Command Block). The SCB contains all the relevant information needed by the Host Adapter to carry out the requested SCSI operation. SCSI SCBs are passed to the HIM by this module which are then sent to the AIC-7880 for execution. The AIC-7880 Sequencer or PhaseEngine comprises the on-chip intelligence that allows the AIC-7880 to execute SCB commands. The Sequencer is programmable and uses its own microcode program which is downloaded to AIC-7880 by the host at initialization.

The following is an example of how an SCB is delivered to the AIC-7880:
Memory is allocated for the SCB structure and it is programmed with the necessary information required to execute a SCSI transaction.

The SCB is then sent to HIM.

The HIM pauses the Sequencer.

The Sequencer has internal registers that point to the area in system memory where the SCB resides.

The HIM unpauses the Sequencer.

The AIC-7880 Sequencer uses DMA to transfer the SCB into its internal memory.

The AIC-7880 executes the SCB.

Upon completion of the SCB command, the AIC-7880 Sequencer posts the pointer of the completed SCB into system memory.

The AIC-7880 generates an interrupt.

The status of the completed SCB is then read by the host.

**SCB PROCESSING**

The AIC-7880 Sequencer uses DMA to transfer the SCB into its internal memory. The Sequencer processes SCBs in the order they are received with new SCBs being started when older SCB operations are idle due to wait for selection or a SCSI bus disconnect. When operations for an Idle SCB reactivate, the sequencer scans the SCB array for the SCB corresponding to the Target/LUN reactivating. The Sequencer then restarts the SCB found until the next disconnect or SCB completion.

**MAXIMUM NUMBER OF TAGGED SCBs**

The number of tagged SCBs per SCSI target that is handled by the Sequencer, range from 1-32. The HIM supports only the External SCB Access mode. The default number of tags handled by the Sequencer in this mode is 32. Changing the field `Cf_MaxTagScbs` in the `cfp_struct` changes the maximum number of tagged SCBs.

**MAXIMUM NUMBER OF SCBs**

The number of SCBs that can be queued to the Sequencer, range from 1-254. This value can be changed before calling the HIM routine `PH_GetConfig()`. Changing the field `Cf_NumberScbs` in `cfp_struct` changes the maximum number of SCBs to be used. The default max number of SCBs is 254.

**SYNCHRONOUS TRANSFER SUPPORT**

If double speed SCSI mode is enabled, this driver supports transfer periods of 50, 64 and 76 ns. In standard fast SCSI mode transfer periods of 100, 125, 150, 175, 200, 225, 250 and 275 are supported. Synchronous transfer parameters for a target can be set using the SCSI library function `scsiTargetOptionsSet()`.
DOUBLE SPEED SCSI MODE

To enable/disable double speed SCSI mode, the routine \texttt{aic7880EnableFast20()} needs to be invoked with the following two parameters:

(1) A pointer to the appropriate SCSI Controller structure
(2) A BOOLEAN value which enables or disable double speed SCSI mode.

With double speed SCSI mode enabled the host adapter may be capable of transferring data at theoretical transfer rates of 20 MB/s for an 8-bit device and 40 MB/s for a 16-bit device. Double Speed SCSI is disabled by default.

DATA FIFO THRESHOLD

To set the data FIFO threshold the routine \texttt{aic7880dFifoThresholdSet()} needs to be invoked with the following two parameters:

(1) A pointer to the appropriate SCSI Controller structure
(2) The data FIFO threshold value.

For more information about the data FIFO threshold value refer the \texttt{aic7880dFifoThresholdSet()} routine

In order to initialize the driver from the BSP the following needs to be done in the BSP specific routine \texttt{sysScsiInit()} in file \texttt{sysScsi.c}:

(1) Find the SCSI Host Adapter.
(2) Create the SCSI Controller Structure.
(3) Connect the interrupt to Interrupt Service Routine (ISR).
(4) Enable the SCSI interrupt.

The following example shows the SCSI initialization sequence that need to be done in the BSP.

```c
STATUS sysScsiInit ()
{
    int busNo;    /* PCI bus number */
    int devNo;    /* PCI device number */
    UWORD found = FALSE; /* host adapter found */
    int numHa = 0; /* number of host adapters */

    for (busNo=0; busNo < MAX_NO_OF_PCI_BUSES && !found; busNo++)
        for (devNo = 0; devNo < MAX_NO_OF_PCI_DEVICES; devNo++)
        {
            if ((found = sysScsiHostAdapterFind (busNo, devNo)) == HA_FOUND)
                {
                    numHa++;

                    /* Create the SCSI controller */
                    if ((pSysScsiCtrl = (SCSI_CTRL *) aic7880CtrlCreate
                     (busNo, devNo, SCSI_DEF_CTRL_BUS_ID)) == NULL)
```
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.

SEE ALSO

The UART design can generate four interrupts: Rx, Tx, modem status change and a UART disabled interrupt (which is asserted when a start bit is detected on the receive line when the UART is disabled). The implementation in the 21285 chip has only two interrupts: Rx and Tx, but the Rx interrupt is a combination of the normal Rx interrupt status and the UART disabled interrupt status.

Only asynchronous serial operation is supported by the UART which supports 5- to 8-bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity. The 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 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 (e.g. ttyDrv) will install its own callback routine using the SIO_INSTALL_CALLBACK ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

**MODES**

This driver supports both polled and interrupt modes.

**USAGE**

The driver is typically only called by the BSP. The directly callable routines in this 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:

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

LOCAL AMBA_CHAN ambaChan[N_AMBA_UART_CHANS];
```
void sysSerialHwInit (void)
{
    int i;
    for (i = 0; i < N_AMBA_UART_CHANS; i++)
    {
        ambaChan[i].regs = devParas[i].baseAdrs;
        ambaChan[i].baudRate = CONSOLE_BAUD_RATE;
        ambaChan[i].xtal = UART_XTAL_FREQ;
        ambaChan[i].levelRx = devParas[i].intLevelRx;
        ambaChan[i].levelTx = devParas[i].intLevelTx;
        /*
         * Initialise driver functions, getTxChar, putRcvChar and
         * channelMode, then initialise UART
         */
        ambaDevInit(&ambaChan[i]);
    }
}

The BSP's sysHwInit2() routine typically calls sysSerialHwInit2(), which connects the
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.
     */
}

By convention all the BSP-specific serial initialization is performed in a file called
sysSerial.c, which is include'ed 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**


---

**ataDrv**

**NAME**

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

**ROUTINES**

*ataDriveInit()* – initialize ATA drive  
*ataDrv()* – initialize the ATA driver  
*ataDevCreate()* – create a device for a ATA/IDE disk  
*ataRawio()* – do raw I/O access

**DESCRIPTION**

This is a driver for ATA/IDE and ATAPI CDROM devices on PCMCIA, ISA, and other buses. The driver can be customized via various macros to run on a variety of boards and both big-endian, and little-endian CPUs.

**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: *ataDrv()* to initialize the driver and *ataDevCreate()* to create devices.

Before the driver can be used, it must be initialized by calling *ataDrv(). This routine must be called exactly once, before any reads, writes, or calls to *ataDevCreate(). Normally, it is called from *usrRoot()* in *usrConfig.c.*

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

**NOTE**

Format is not supported, because ATA/IDE disks are already formatted, and bad sectors are mapped.

During initialization this driver queries each disk to determine if the disk supports LBA. 16 bit words 0x60 and 0x61 (returned from the *ATA IDENTIFY DEVICE* command) may report a larger value than the product of the CHS fields on newer large disks (8.4Gb+). The driver will use strict LBA access commands and LBA geometry for drives reporting “total LBA sectors” greater than the product of CHS. Although everyone should also be using strict LBA on LBA disks, some older systems (mostly PC’s) do not and use only
CHS. Such system cannot view drives larger than 8GB. VxWorks does not have such limitations. However, it may be desirable to force VxWorks ignore the LBA information in favor of CHS in order to mount a file system originally formatted on a CHS-only system. Setting the boolean ataForceCHSonLBA to TRUE will force the use of CHS parameters on all drives and the LBA parameters are ignored. Again, setting this boolean may prevent access to the drive's full capacity, since some manufacturers have stopped setting a drive's CHS accurately in favor of LBA.

**PARAMETERS**

The ataDrv() function requires a configuration flag as a parameter. The configuration flag is one of the following:

**Transfer mode**

- **ATA_PIO_DEF_0**  
  PIO default mode
- **ATA_PIO_DEF_1**  
  PIO default mode, no IORDY
- **ATA_PIO_0**  
  PIO mode 0
- **ATA_PIO_1**  
  PIO mode 1
- **ATA_PIO_2**  
  PIO mode 2
- **ATA_PIO_3**  
  PIO mode 3
- **ATA_PIO_4**  
  PIO mode 4
- **ATA_PIO_AUTO**  
  PIO max supported mode
- **ATA_DMA_0**  
  DMA mode 0
- **ATA_DMA_1**  
  DMA mode 1
- **ATA_DMA_2**  
  DMA mode 2
- **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
- **ATA_DMA_SINGLE**  
  RW DMA single word
- **ATA_DMA_MULTI**  
  RW DMA multi word

**Geometry parameters**

- **ATA_GEO_FORCE**  
  set geometry in the table
- **ATA_GEO_PHYSICAL**  
  set physical geometry
- **ATA_GEO_CURRENT**  
  set current geometry

DMA transfer is not supported in this release. If **ATA_PIO_AUTO** or **ATA_DMA_AUTO** is specified, the driver automatically chooses the maximum mode supported by the device. If **ATA_PIO_MULTI** or **ATA_DMA_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.
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:

```c
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 */
```

This driver does not access the PCI-chip-set IDE interface, but rather takes advantage of BIOS or VxWorks initialization. Thus, the BIOS setting should match the modes specified by the configuration flag.

The BSP may provide a sysAtaInit() routine for situations where an ATA controller RESET(0x1f6 or 0x3f6, bit 2 is set) clears ATA specific functionality in a chipset that is not re-enabled per the ATA-2 spec.

This BSP routine should be declared in sysLib.c or sysAta.c as follows:

```c
void sysAtaInit (BOOL ctrl)
{
    /* BSP SPECIFIC CODE HERE */
}
```

Then the BSP should perform the following operation before ataDrv() is called, in sysHwInit for example:

```c
IMPORT VOIDFUNCPTR _func_sysAtaInit;
/* setup during initialization */
_func_sysAtaInit = (VOIDFUNCPTR) sysAtaInit;
```

It should contain chipset specific reset code, such as code which re-enables PCI write posting for an integrated PCI-IDE device, for example. This will be executed during every ataDrv(), ataInit(), and ataReset() or equivalent block device routine. If the sysAtaInit() routine is not provided by the BSP it is ignored by the driver, therefore it is not a required BSP routine.

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

**DESCRIPTION**
This library contains a driver show routine for the ATA/IDE (PCMCIA and LOCAL) devices supported on the IBM PC.

**auEnd**

**NAME**
auEnd – END style Au MAC Ethernet driver

**ROUTINES**
- auEndLoad() – initialize the driver and device
- auInitParse() – parse the initialization string
- auDump() – display device status

**DESCRIPTION**
This module implements the Alchemey Semiconductor Au on-chip Ethernet MACs. The software interface to the driver is divided into three parts. The first part is the interrupt registers and their setup. This part is done at the BSP level in the various BSPs which use this driver. The second and third part are addressed in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the buffers.

This driver is designed to be moderately generic. Though it currently is implemented on one processor, in the future it may be added to other Alchemey product offerings. Thus, it would be desirable to use the same driver with no source level changes. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These target-specific values and the external support routines are described below.

This driver supports multiple units per CPU. The driver can be configured to support big-endian or little-endian architectures.

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

**EXTERNAL INTERFACE**
The only external interface is the auEndLoad() routine, which expects the initString parameter as input. This parameter passes in a colon-delimited string of the format:
The `auEndLoad()` function uses `strtok()` to parse the string.

**TARGET-SPECIFIC PARAMETERS**

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

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

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

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

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

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

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

**qtyCluster**
This parameter is for explicitly allocating 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 to get the Ethernet hardware of the chip. This macro calls an external board level routine namely `sysLanAuEnetAddrGet()` to get the ethernet address.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

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

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

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

**SEE ALSO**
muxLib, endLib, netBufLib, *Writing An Enhanced Network Driver*
cd2400Sio

NAME
cd2400Sio – CL-CD2400 MPCC serial driver

ROUTINES
cd2400HrdInit() – initialize the chip
cd2400IntRx() – handle receiver interrupts
cd2400IntTx() – handle transmitter interrupts
cd2400Int() – handle special status interrupts

DESCRIPTION
This is the driver for the Cirus Logic CD2400 MPCC. It uses the SCC’s in asynchronous mode.

USAGE
A CD2400_QUSART structure is used to describe the chip. This data structure contains four CD2400_CHAN structure which describe the chip’s four serial channels. The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the CD2400_QUSART structure (except the SIO_DRV_FUNCS) before calling cd2400HrdInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupts (cd2400Int, cd2400IntRx, and cd2400IntTx) via intConnect().

IOCTL FUNCTIONS
This driver responds to the same ioctl() codes as a normal serial driver; for more information, see the comments in sioLib.h. The available baud rates are: 50, 110, 150, 300, 600, 1200, 2400, 3600, 4800, 7200, 9600, 19200, and 38400.

INCLUDE FILES
drv/sio/cd2400Sio.h

cisLib

NAME
cisLib – PCMCIA CIS library

ROUTINES
cisGet() – get information from a PC card’s CIS
cisFree() – free tuples from the linked list
cisConfigregGet() – get the PCMCIA configuration register
cisConfigregSet() – set the PCMCIA configuration register

DESCRIPTION
This library contains routines to manipulate the CIS (Configuration Information Structure) tuples and the card configuration registers. The library uses a memory window which is defined in pcmciaMemwin to access the CIS of a PC card. All CIS tuples in a PC card are read and stored in a linked list, cisTupleList. If there are configuration tuples, they are...
interpreted and stored in another link list, cisConfigList. After the CIS is read, the PC card’s enabler routine allocates resources and initializes a device driver for the PC card.

If a PC card is inserted, the CSC (Card Status Change) interrupt handler gets a CSC event from the PCMCIA chip and adds a cisGet() job to the PCMCIA daemon. The PCMCIA daemon initiates the cisGet() work. The CIS library reads the CIS from the PC card and makes a linked list of CIS tuples. It then enables the card.

If the PC card is removed, the CSC interrupt handler gets a CSC event from the PCMCIA chip and adds a cisFree() job to the PCMCIA daemon. The PCMCIA daemon initiates the cisFree() work. The CIS library frees allocated memory for the linked list of CIS tuples.

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.

coldfireSio

NAME  coldfireSio – ColdFire Serial Communications driver
ROUTINES  coldfireDevInit() – initialize a COLDFIRE_CHAN
coldfireDevInit2() – initialize a COLDFIRE_CHAN, part 2
coldfireImrSetClr() – set and clear bits in the UART’s interrupt mask register
coldfireImr() – return current interrupt mask register contents
coldfireAcrSetClr() – set and clear bits in the UART’s aux control register
coldfireAcr() – return aux control register contents
coldfireOprSetClr() – set and clear bits in the output port register
coldfireOpr() – return the current state of the output register
coldfireInt() – handle all interrupts in one vector

DESCRIPTION  This is the driver for the UART contained in the ColdFire Microcontroller.
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. These default settings can be overridden by setting the COLDFIRE_CHAN options and baudRate fields to the desired values before calling coldfireDevInit(). See sioLib.h for options values. The defaults for the module can be changed by redefining the macros COLDFIRE_DEFAULT_OPTIONS and COLDFIRE_DEFAULT_BAUD and recompiling this driver.

This driver uses the system clock as the input to the baud rate generator. The clkRate field must be set to the system clock rate in HZ for the baud rate calculations to work correctly. The actual range of supported baud rates depends on the system clock speed. For example, with a 25MHz system clock, the lowest baud rate is 24, and the highest is over 38400. Because the baud rate values are calculated on request, there is no checking that the requested baud rate is standard, you can set the UART to operate at 4357 baud if you wish.

**USAGE**

A COLDFIRE_CHAN structure is used to describe the chip.

The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the H/W addresses in the COLDFIRE_CHAN structure before calling coldfireDevInit(). This enables the chip to operate in polled mode, but not interrupt mode. Calling coldfireDevInit2() from the sysSerialHwInit2() routine allows interrupts to be enabled and interrupt mode operation to be used.

**i.e.**

```c
#include "drv/multi/coldfireSio.h"
COLDFIRE_CHAN coldfireUart; /* my device structure */
#define INT_VEC_UART   (24+3) /* use single vector, #27 */
sysSerialHwInit()
{
    /* initialize the register pointers/data for uart */
    coldfireUart.clkRate = MASTER_CLOCK;
    coldfireUart.intVec = INT_VEC_UART;
    coldfireUart.mr = COLDFIRE_UART_MR(SIM_BASE);
    coldfireUart.sr = COLDFIRE_UART_SR(SIM_BASE);
    coldfireUart.csr = COLDFIRE_UART_CSR(SIM_BASE);
    coldfireUart.cr = COLDFIRE_UART_CR(SIM_BASE);
    coldfireUart.rb = COLDFIRE_UART_RB(SIM_BASE);
    coldfireUart.tb = COLDFIRE_UART_TB(SIM_BASE);
    coldfireUart.ipcr = COLDFIRE_UART_IPCR(SIM_BASE);
    coldfireUart.acr = COLDFIRE_UART_ACR(SIM_BASE);
    coldfireUart.isr = COLDFIRE_UART_ISR(SIM_BASE);
    coldfireUart.imr = COLDFIRE_UART_IMR(SIM_BASE);
    coldfireUart.bg1 = COLDFIRE_UART_BG1(SIM_BASE);
    coldfireUart.bg2 = COLDFIRE_UART_BG2(SIM_BASE);
    coldfireUart.ivr = COLDFIRE_UART_IVR(SIM_BASE);
    coldfireUart.ip = COLDFIRE_UART_IP(SIM_BASE);
```

20
coldfireUart.op1 = COLDFIRE_UART_OP1(SIM_BASE);
coldfireUart.op2 = COLDFIRE_UART_OP2(SIM_BASE);
coldfireDevInit (&coldfireUart);
}

The BSP's `sysHwInit2()` routine typically calls `sysSerialHwInit2()` which connects the chips interrupts via `intConnect()` to the single interrupt handler `coldfireInt`. After the interrupt service routines are connected, the user then calls `coldfireDevInit2()` to allow the driver to turn on interrupt enable bits. That is:

```
sysSerialHwInit2 ()
{
    /* connect single vector for 5204 */
    intConnect (INUM_TO_IVEC(MY_VEC), coldfireInt, (int)&coldfireUart);
    ...
    /* allow interrupts to be enabled */
    coldfireDevInit2 (&coldfireUart);
}
```

**SPECIAL CONSIDERATIONS**

The CLOCAL hardware option presumes that CTS outputs are not wired as necessary. CLOCAL is one of the default options for this reason.

As to the output port, this driver does not manipulate the output port, or it's configuration register in any way. As stated above, if the user does not select the CLOCAL option then the output port bit must be wired correctly or the hardware flow control will not function as desired.

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

cB69000Vga

**NAME**
cB69000Vga – a CHIPS B69000 initialization source module

**ROUTINES**
cB69000VgaInit() – initialize 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:

i - Initialization of CHIPS B69000 IC.

**USER INTERFACE**

```c
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
NAME
dec21x4xEnd – END style DEC 21x4x PCI Ethernet network interface driver

ROUTINES
dec21x4xEndLoad() – initialize the driver and device

DESCRIPTION
This module implements a DEC 21x4x PCI Ethernet network interface driver and supports 21040, 21140 and 21143 versions of the chip.

The DEC 21x4x PCI Ethernet controller is little-endian because it interfaces with a little-endian PCI bus. Although PCI configuration for a device is handled in the BSP, all other device programming and initialization are handled in this module.

This driver is designed to be moderately generic. Without modification, it can operate across the range of architectures and targets supported by VxWorks. To achieve this, the driver requires a few external support routines as well as several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, you need to modify the driver before it can operate correctly on your hardware.

On 21040, the driver configures the 10BASE-T interface by default, waits for two seconds, and checks the status of the link. If the link status indicates failure, AUI interface is configured.

On other versions of the 2114x family, the driver reads media information from a DEC serial ROM and configures the media. On targets that do not support a DEC format serial ROM, the driver calls a target-specific media select routine using the hook, _func_dec2114xMediaSelect, to configure the media.

The driver supports big-endian or little-endian architectures (as a configurable option). The driver also and contains error recovery code that handles known device errata related to DMA activity.

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

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

EXTERNAL INTERFACE
The driver provides one standard external interface, dec21x4xEndLoad(), which a takes 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 `strtok_r()`, each parameter is converted from string to binary by a call to:

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

The format of the `parameter` string is:

"unit number:device addr:PCI addr:ivec:ilevel:mem base:mem size:user flags:offset"

**TARGET-SPECIFIC PARAMETERS**

- **unit number**
  This represents the device instance number relative to this driver. That is, a value of zero represents the first dec21x4x device, a value of 1 represents the second dec21x4x device.

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

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

- **ivec**
  This is the interrupt vector number of the hardware interrupt generated by this Ethernet device. The driver uses `intConnect()`, or `pciIntConnect()` (x86 arch), to attach an interrupt handler for this interrupt.

- **ilevel**
  This parameter defines the level of the hardware interrupt.

- **mem base**
  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.

  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()`.

- **mem size**
  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.

- **user flags**
  User flags control the run-time characteristics of the Ethernet chip. Most flags specify non default CSR0 bit values. Refer to `dec21x4xEnd.h` for the bit values of the flags, and to the device hardware reference manual for details about device capabilities, and CSR 0.
Some of them are worth mentioning:

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.

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.

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

Device Type
Although the default device type is DEC 21040, specifying the DEC_USR_21140 or DEC_USR_21143 flag bit turns on DEC 21140 or DEC_USR_21143 functionality.

Ethernet Address
The Ethernet address is retrieved from standard serial ROM on DEC 21040, DEC 21140 and DEC 21143 devices. If retrieve from ROM fails, the driver calls the BSP routine, sysDec21x4xEnetAddrGet(). Specifying DEC_USR_XEA flag bit tells the driver should, by default, retrieve the Ethernet address using the BSP routine, sysDec21x4xEnetAddrGet().

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. Alternately, 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.

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.
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 sysDec21x4xEnetAddrGet (int unit, char *enetAdrs)`
This routine provides a target-specific interface for accessing a device Ethernet address.

`STATUS sysDec21143Init (DRV_CTRL * pDrvCtrl)`
This routine performs any target-specific initialization required before the dec21143 device is initialized by the driver. The driver calls this routine every time it wants to load the device. This routine returns OK, or ERROR if it fails.

`FUNCPTR _func_dec2114xMediaSelect`  
This driver provides a default media select routine, when `_func_dec2114xMediaSelect` is NULL, to read and set up physical media with configuration information from a Version 3 DEC Serial ROM. Any other media configuration can be supported by initializing `_func_dec2114xMediaSelect`, typically in `sysHwInit()`, to a target-specific media select routine.

A media select routine is typically defined as:

```c
STATUS decMediaSelect
{
    DEC21X4X_DRV_CTRL * pDrvCtrl, /* Driver control */
    UINT * pCsr6Val    /* CSR6 return value */
}
```

Parameter `pDrvCtrl` is a pointer to the driver control structure which this routine may use to access the Ethernet device. The driver control structure field `mediaCount`, is initialized to 0xff at startup, while the other media control fields (`mediaDefault`, `mediaCurrent`, and `gprModeVal`) are initialized to zero. This routine may use these fields in any manner, however all other driver control fields should be considered read-only and should not be modified.

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

`FUNCPTR _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.

SEE ALSO

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

ROUTINES
edTok_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

DESCRIPTION
This module implements a DEC 21x40 PCI Ethernet network interface driver and supports both the 21040, 21140, 21143, 21145 versions of the chip.

The DEC 21x40 PCI Ethernet controller is little-endian because it interfaces with a little-endian PCI bus. Although PCI configuration for a device is handled in the BSP, all other device programming and initialization needs are handled in this module.

This driver is designed to be moderately generic. Without modification, it can operate across the full range of architectures and targets supported by VxWorks. To achieve this, the driver requires a few external support routines as well as several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, you need to modify the driver before it can operate correctly on your hardware.

On the 21040, the driver configures the 10BASE-T interface by default, waits for two seconds, and checks the status of the link. If the link status indicates failure, AUI interface is configured.

On other versions of the 21x40 family, the driver reads media information from a DEC serial ROM and configures the media. To configure the media on targets that do not support a DEC format serial ROM, the driver calls the target-specific media-select routine referenced in the _func_dec21x40MediaSelect hook.

The 21145 supports HomePNA 1.0 (Home Phone Line) Networking as well as 10Base-T. The HomePNA port can be forced to 1 MB/sec or 0.7 MB/sec mode via the
DEC_USR_HPNA_FORCE_FAST and DEC_USR_HPNA_FORCE_SLOW user flags, respectively. If these flags are not set then the speed is set using the SRM settings. Unlike the Ethernet phys, the HomePNA phy can not determine link failure and therefore will never notify the driver when the HomePNA port is disconnected. However, to allow media change, the driver can be configured to ALWAYS prefer 10Base-T over HomePNA by interrupting on 10Base-T link pass interrupt. Upon 10Base-T link failure, the driver will revert back to HomePNA. Since this method violates the preference rules outlined in Intel/DEC SRM format spec, this is not the default mode of operation. The driver must be started with DEC_USR_HPNA_PREFER_10BT user flag set to set the driver into this mode.

The driver supports big-endian or little-endian architectures (as a configurable option). The driver also contains error recovery code that handles known device errata related to DMA activity.

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

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

EXTERNAL INTERFACE
The driver provides one standard external interface, Load(). 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:

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

The format of the parameter string is:

```
"deviceAddr:pciAddr:iVec:iLevel:numRds:numTds:
memBase:memSize:userFlags:phyAddr:phyTbl:phyFlags:offset:loanBufs"
```

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.
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 `IntConnectRtn` with the desired routines (usually `pciIntConnect()`).

### iLevel

This parameter defines the level of the hardware interrupt.

### numRds

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

### numTds

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

### memBase

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

### memSize

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

### userFlags

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

### phyAddr

This optional parameter specifies the address on the MII (Media Independent Interface) bus of a MII-compliant PHY (Physical Layer Entity). The module that is responsible for optimally configuring the media layer will start scanning the MII bus from the address in `phyAddr`. It will retrieve the PHY’s address regardless of that, but, since the MII management interface, through which the PHY is configured, is a very slow one, providing an incorrect or invalid address may result in a particularly long boot process. If the flag `DEC_USR_MII` is not set, this parameter is ignored.

### pPhyTbl

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

**phyFlags**

This optional parameter allows the user to affect the PHY's configuration and behavior. 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 32. The number of loaner buffers must be accounted for when calculating the memory size specified by **memSize**.

**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, then the optional fields **phyAddr**, **pPhyTbl**, and **phyFlags** may be used to affect the PHY’s configuration on the network.
10Base-T Mode: when the flag DEC_USR_MII_10MB is set, then the PHY will negotiate this technology ability, if present.

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

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

Full duplex Mode: when the flag DEC_USR_MII_FD is set, then the PHY will negotiate this technology ability, if present.

Auto-negotiation: The driver’s default behavior 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 behavior is to enable the standard auto-negotiation process, as defined in "IEEE 802.3u Standard". However, the user may wish to force the PHY to negotiate its technology abilities a subset at a time, and according to a particular order. The flag DEC_USR_MII_AN_TBL in the phyFlags field may be used to tell the driver that the PHY should negotiate its abilities as dictated by the entries in the pPhyTbl of the load string. If the flag DEC_USR_MII_NO_AN is set, this parameter is ignored.

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

Transmit treshold 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

FUNC PTR _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 the necessary 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.

VOIDFUNC PTR _func_dec2114xIntAck

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

### PCI ID VALUES

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

<table>
<thead>
<tr>
<th>Chip</th>
<th>Vendor ID</th>
<th>Device ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec 21040</td>
<td>0x1011</td>
<td>0x0002</td>
</tr>
<tr>
<td>dec 21041</td>
<td>0x1011</td>
<td>0x0014</td>
</tr>
<tr>
<td>dec 21140</td>
<td>0x1011</td>
<td>0x0009</td>
</tr>
<tr>
<td>dec 21143</td>
<td>0x1011</td>
<td>0x0019</td>
</tr>
<tr>
<td>dec 21145</td>
<td>0x8086</td>
<td>0x0039</td>
</tr>
</tbody>
</table>
SEE ALSO

ei82596End

NAME  

\texttt{ei82596End} – END style Intel 82596 Ethernet network interface driver

ROUTINES  

\texttt{ei82596EndLoad( )} – initialize the driver and device

DESCRIPTION  

This module implements an Intel 82596 Ethernet network interface driver. This driver is designed to be moderately generic. It operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, this driver requires some external support routines as well as several target-specific parameters. These parameters (and the mechanisms used to communicate them to the driver) are detailed below.

This driver can run with the device configured in either big-endian or little-endian modes. Error recovery code has been added to deal with some of the known errata in the A0 version of the device. This driver supports up to four individual units per CPU.

BOARD LAYOUT  

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

EXTERNAL INTERFACE  

The driver provides one standard external interface, \texttt{ei82596EndLoad( )}. 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 \texttt{strtok_r( )}, and each parameter is converted from string to binary by a call to:

\begin{verbatim}
  strtoul(parameter, NULL, 16).
\end{verbatim}

TARGET-SPECIFIC PARAMETERS  

The format of the parameter string is:

\texttt{unit:ivec:sysbus:memBase:nTfds:nRfds:offset}

\texttt{unit}  

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

\texttt{ivec}  

This is the interrupt vector number of the hardware interrupt generated by this ethernet device. The driver uses \texttt{intConnect( )} to attach an interrupt handler to this interrupt.

\texttt{sysbus}  

This parameter tells the device about the system bus. To determine the correct value for a target, see \textit{Intel 32-bit Local Area Network (LAN) Component User’s Manual}. 
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, buffers, and other device control structures. If there is no pre-allocated memory available for the driver, this parameter should be -1 (NONE). In which case, the driver calls `cacheDmaAlloc()` to allocate cache-safe memory.

This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is zero or -1 (NULL), a default of 32 is used.

This parameter specifies the number of receive descriptor/buffers to be allocated. If this parameter is zero or -1 (NULL), a default of 32 is used.

Specifies the memory alignment offset.

This driver requires seven external support functions:

```c
void sys596IntEnable (int unit)
```

This routine provides a target-specific interface to enable Ethernet device interrupts for a given device unit.

```c
void sys596IntDisable (int unit)
```

This routine provides a target-specific interface to disable Ethernet device interrupts for a given device unit.

```c
STATUS sysEnetAddrGet (int unit, char *enetAdrs)
```

This routine provides a target-specific interface to access a device Ethernet address. This routine should provide a six-byte Ethernet address in the `enetAdrs` parameter and return OK or ERROR.

```c
STATUS sys596Init (int unit)
```

This routine performs any target-specific initialization required before the 82596 is initialized. Typically, it is empty. This routine must return OK or ERROR.

```c
void sys596Port (int unit, int cmd, UINT32 addr)
```

This routine provides access to the special port function of the 82596. It delivers the command and address arguments to the port of the specified unit. The driver calls this
routinely primarily during initialization and, under some conditions, during error recovery procedures.

sys596ChanAtn()
void sys596ChanAtn(int unit)

This routine provides the channel attention signal to the 82596 for the specified unit. The driver calls this routine frequently throughout all phases of operation.

sys596IntAck()
void sys596IntAck(int unit)

This routine must perform any required interrupt acknowledgment or clearing. Typically, this involves an operation to some interrupt control hardware.

**NOTE:** The INT signal from the 82596 behaves in an "edge-triggered" mode. Therefore, this routine typically clears a latch within the control circuitry. The driver calls this routine from the interrupt handler.

**SYSTEM RESOURCE USAGE**

The driver uses cacheDmaMalloc() to allocate memory to share with the 82596. The fixed-size pieces in this area total 160 bytes. The variable-size pieces in this area are affected by the configuration parameters specified in the eiattach() call. The size of one RFD (Receive Frame Descriptor) is 1536 bytes. The size of one TFD (Transmit Frame Descriptor) is 1534 bytes. For more on RFDs and TFDs, see the Intel 82596 User’s Manual.

The 82596 requires either that this shared memory region is non-cacheable or that the hardware implements bus snooping. The driver cannot maintain cache coherency for the device. This is because fields within the command structures are asynchronously modified by both the driver and the device, and these fields might share the same cache line.

**TUNING HINTS**

The only adjustable parameters are the number of TFDs and RFDs that are created at run-time. These parameters are given to the driver when eiattach() 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 might be a good idea. Increasing the number of TFDs provides no performance benefit after a certain point. Increasing the number of RFDs provides more buffering before packets are dropped. This can be useful if there are tasks running at a higher priority than the net task.

**SEE ALSO**

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, then input and output byte stream need not be swapped.

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

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

BOARD LAYOUT

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

EXTERNAL INTERFACE

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

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

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

TARGET-SPECIFIC PARAMETERS

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

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

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

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

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

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

**memAdrs**
This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cached 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 then a default multiplier of 2 is chosen. With a multiplier of 2 the total number of clusters allocated is 64 which is twice the cumulative number of upload and download descriptors. The device has 16 upload and 16 download descriptors. For example on choosing the buffer multiplier of 3, the total number of clusters allocated will be 96 ((16 + 16)*3). There are as many clBlks as the number of clusters. The number of mBlks allocated are twice the number of clBlks. By default there are 64 clusters, 64 clBlks and 128 mBlks allocated in the pool for the device. Depending on the load of the system increase the number of clusters allocated by incrementing the buffer multiplier.

EXTERNAL SUPPORT REQUIREMENTS

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

```c
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 \texttt{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 \texttt{sysEl3c90xIntEnable( )}.

The macro \texttt{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 \texttt{sysEl3c90xIntDisable( )}.

The macro \texttt{SYS\_DELAY} is used for a delay loop. It calls an external board level routine \texttt{sysDelay(delay)}. The granularity of \textit{delay} is one microsecond.

\section*{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.

\section*{INCLUDES}

\texttt{end.h, endLib.h, etherMultiLib.h, el3c90xEnd.h}

\section*{SEE ALSO}

\texttt{muxLib, endLib, netBufLib, VxWorks Programmer's Guide: Writing an Enhanced Network Driver}

\section*{BIBLIOGRAPHY}

3COM 3c90x and 3c90xB NICs Technical Reference

\section*{elt3c509End}

\section*{NAME}

\texttt{elt3c509End} – END network interface driver for 3COM 3C509

\section*{ROUTINES}

\texttt{elt3c509Load( )} – initialize the driver and device
\texttt{elt3c509Parse( )} – parse the init string
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.

**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:attachmentType: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 IO-mapped accesses to the device registers and the normal intConnect(), and intEnable() BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

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

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

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

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

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

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

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

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

The nRxFrames parameter can be used for tuning number of receive frames to be used for handling packet receive. More of these could help receiving more loading in case of massive reception.

INCLUDES

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

SEE ALSO

muxLib, endLib, Writing An Enhanced Network Driver

dendLib

NAME
dendLib – support library for END-based drivers

ROUTINES

mib2Init() – initialize a MIB-II structure
mib2ErrorAdd() – change a MIB-II error count
dendObjInit() – initialize an END_OBJ structure
dendObjFlagSet() – set the flags member of an END_OBJ structure
dendEtherAddressForm() – form an Ethernet address into a packet
dendEtherPacketDataGet() – return the beginning of the packet data
dendEtherPacketAddrGet() – locate the addresses in a packet

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.

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
NAME
fei82557End – END style Intel 82557 Ethernet network interface driver

ROUTINES
fei82557EndLoad() – initialize the driver and device
fei82557DumpPrint() – display statistical counters
fei82557ErrCounterDump() – dump statistical counters

DESCRIPTION
This module implements an Intel 82557 Ethernet network interface driver. This is a fast Ethernet PCI bus controller, IEEE 802.3 10Base-T and 100Base-T compatible. It also features a glueless 32-bit PCI bus master interface, fully compliant with PCI Spec version 2.1. An interface to MII compliant physical layer devices is built-in to the card. The 82557 Ethernet PCI bus controller also includes Flash support up to 1 MByte and EEPROM support, although these features are not dealt with in this driver.

The 82557 establishes a shared memory communication system with the CPU, which is divided into three parts: the Control/Status Registers (CSR), the Command Block List (CBL), and the Receive Frame Area (RFA). The CSR is on-chip and is either accessible with I/O or memory cycles, whereas the other structures reside on the host.

The CSR is the main means of communication between the device and the host, meaning that the host issues commands through these registers while the chip posts status changes in it, occurred as a result of those commands. Pointers to both the CBL and RFA are also stored in the CSR.

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

The RFA is a linked list of receive frame descriptors. Only support for the simplified memory mode is granted. In this model, the data buffer immediately follows the related frame descriptor.

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:

```c
strtol(parameter, NULL, 16)
```

The format of the parameter string is:

"memBase:memSize:nTfds:nRfds:flags:offset"

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 (like `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. 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 `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. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of Command Frame Descriptor and Receive Frame Descriptor.

`nTfds`

This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is less than two, 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 two, a default of 32 is used. In addition, four times as many loaning buffers are created. These buffers are loaned up to the network stack. When loaning buffers are exhausted, the system begins discarding incoming packets. Specifying 32 buffers results in 32 frame descriptors, 32 reserved buffers and 128 loaning buffers being created from the system heap.

`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 field is optional, the default value is zero.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

```c
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 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 `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 the net task.

ALIGNMENT

Some architectures do not support unaligned access to 32-bit data items. On these architectures (eg. PowerPC and ARM), 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.

SEE ALSO

gei82543End

NAME
gei82543End – Intel PRO/1000 F/T/XF/XT/MT network adapter END driver

ROUTINES
gei82543EndLoad() – initialize the driver and device

DESCRIPTION
The gei82543End driver supports Intel PRO1000 T/F/XF/XT/MT adaptors. These adaptors use Intel 82543GC, 82544GC/EI, or 82540/82545/82546EB 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. Although these devices also support other advanced features such as receive and transmit IP/TCP/UDP checksum offloading, jumbo frames, and provide flash support up to 512KB and EEPROM support, this driver does not support these features.

The 8254x establishes a shared memory communication system with the CPU, which is divided into two parts: the control/status registers and the receive/transmit descriptors/buffers. The control/status registers are on the 8254x chips and are only accessible with PCI or PCI-X memory cycles, whereas the other structures reside on the host. The buffer size can be programmed between 256 bytes to 16k bytes. This driver uses the receive buffer size of 2048 bytes for an MTU of 1500.

The Intel PRO/1000 F/XF adapters only implement the TBI mode of the 82543GC/82544GC 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 adaptors and TBI mode for Intel PRO1000 F/XF 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 ignored by 82544-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 occured in the copper-and-82543-based adapter, and the link partner has advertised only 1000Base-T capability.

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:

\texttt{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 \texttt{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 \texttt{OK}, or \texttt{ERROR} if it fails.

SYSTEM RESOURCE USAGE
The driver uses \texttt{cacheDmaMalloc()} to allocate memory to share with the 8254xGC. The size of this area is affected by the configuration parameters specified in the \texttt{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 achiving 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 then this will result in reduced throughput. Therefore, increasing the network stack’s number of clusters relative to the number of transmit descriptors will increase bandwidth. Of course this technique will eventually reach a point of diminishing return. There are actually several sizes of clusters available in the network pool. Increasing any or all of these cluster sizes will result in some increase in performance. However, increasing the 2048-byte cluster size will likely have the greatest impact since this size will hold an entire MTU and header.

Increasing the number of receive descriptors and clusters may also have positive impact. Increasing the buffer size of sockets can also be beneficial. This can significantly improve performance for a target system under higher transfer rates. However, it should be noted that large amounts of unread buffers idling in sockets reduces the resources available to the rest of the stack. This can, in fact, have a negative impact on bandwidth. One method to reduce this effect is to carefully adjust application tasks’ priorities and possibly increase number of receive clusters.

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

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

if_cpm

NAME
if_cpm – Motorola CPM core network interface driver

ROUTINES
cpmattach() – publish the cpm network interface and initialize the driver
cpmStartOutput() – output packet to network interface device

DESCRIPTION
This module implements the driver for the Motorola CPM core Ethernet network interface used in the M68EN360 and PPC800-series communications controllers.

The driver is designed to support the Ethernet mode of an SCC residing on the CPM processor core. It is generic in the sense that it does not care which SCC is being used, and it supports up to four individual units per board.
The 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.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance. This driver provides support for four individual device units.

This driver maintains cache coherency by allocating buffer space using the cacheDmaMalloc() routine. It is assumed that cache-safe memory is returned; this driver does not perform cache flushing and invalidating.

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

**EXTERNAL INTERFACE**
This driver presents the standard WRS network driver API: the device unit must be attached and initialized with the cpmattach() routine.

The only user-callable routine is cpmattach(), which publishes the cpm interface and initializes the driver structures.

**TARGET-SPECIFIC PARAMETERS**
These parameters are passed to the driver via cpmattach().

- **address of SCC parameter RAM**
  This parameter is the address of the parameter RAM used to control the SCC. Through this address, and the address of the SCC registers (see below), different network interface units are able to use different SCCs without conflict. This parameter points to the internal memory of the chip where the SCC physically resides, which may not necessarily be the master chip on the target board.

- **address of SCC registers**
  This parameter is the address of the registers used to control the SCC. Through this address, and the address of the SCC parameter RAM (see above), different network interface units are able to use different SCCs without conflict. This parameter points to the internal memory of the chip where the SCC physically resides, which may not necessarily be the master chip on the target board.

- **interrupt-vector offset**
  This driver configures the SCC 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 intConnect().

- **address of transmit and receive buffer descriptors**
  These parameters indicate the base locations of the transmit and receive buffer descriptor (BD) rings. Each BD takes up 8 bytes of dual-ported RAM, and it is the user’s responsibility to ensure that all specified BDs will fit within dual-ported RAM. This includes any other BDs the target board may be using, including other SCCs, SMCs, and the SPI device. There is no default for these parameters; they must be
number of transmit and receive buffer descriptors
The number of transmit and receive buffer descriptors (BDs) used is configurable by
the user upon attaching the driver. Each buffer descriptor resides in 8 bytes of the
chip’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 number of buffers, but there is a limit to how much the driver speed
increases as more buffers are added, and dual-ported RAM space is at a premium. If
this parameter is "NULL", a default value of 32 BDs is used.

base address of buffer pool
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 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) and an additional 16 receive loaner
buffers. If the number of receive BDs is less than 16, that number of loaner buffers is
used. Each buffer is 1520 bytes. If this parameter is "NONE," space for buffers is
obtained by calling cacheDmaMalloc() in cpmaattach().

EXTERNAL SUPPORT REQUIREMENTS
This driver requires seven external support functions:

STATUS sysCpmEnetEnable (int unit)
This routine is expected to perform any target-specific functions required 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. The
driver calls this routine, once per unit, from the cpmInit() routine.

void sysCpmEnetDisable (int unit)
This routine is expected to perform any target-specific functions required to disable
the Ethernet controller. This usually involves disabling the Transmit Enable (TENA)
signal. The driver calls this routine from the cpmReset() routine each time a unit is
disabled.

STATUS sysCpmEnetCommand (int unit, UINT16 command)
This routine is expected to issue a command to the Ethernet interface controller. The
driver calls this routine to perform basic commands, such as restarting the transmitter
and stopping reception.

void sysCpmEnetIntEnable (int unit)
This routine is expected to enable the interrupt for the Ethernet interface specified by
unit.

void sysCpmEnetIntDisable (int unit)
This routine is expected to disable the interrupt for the Ethernet interface specified by
unit.
void sysCpmEnetIntClear (int unit)

This routine is expected to clear the interrupt for the Ethernet interface specified by unit.

STATUS sysCpmEnetAddrGet (int unit, UINT8 * addr)

The driver expects this routine to provide the 6-byte Ethernet hardware address that will be used by unit. This routine must copy the 6-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 cpmInit() routine.

SYSTEM RESOURCE USAGE

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 may vary for other architectures. The code size (text) varies greatly between architectures, and is therefore not quoted here.

If the driver allocates the memory shared 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, 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 the shared 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 may share the same cache line. Additionally, the chip’s dual ported RAM must be declared as non-cacheable memory where applicable.

SEE ALSO

if_cs

NAME

if_cs – Crystal Semiconductor CS8900 network interface driver

ROUTINES

csAttach() – publish the cs network interface and initialize the driver.
csShow() – shows statistics for the cs network interface

DESCRIPTION

This module implements a driver for a Crystal Semiconductor CS8900 Ethernet controller chip.

The CS8900 is a single chip Ethernet controller with a direct ISA bus interface which can operate in either memory space or I/O space. It also supports a direct interface to a host DMA controller to transfer receive frames to host memory. The device has a 4K RAM which is used for transmit, and receive buffers; a serial EEPROM interface; and both 10BASE-T/AUI port support.

This driver is capable of supporting both memory mode and I/O mode operations of the chip. When configured for memory mode, the internal RAM of the chip is mapped to a contiguous 4K address block, providing the CPU direct access to the internal registers and frame buffers. When configured for I/O mode, the internal registers are accessible through eight contiguous, 16-bit I/O ports. The driver also supports an interface to an EEPROM containing device configuration.

While the DMA slave mode is supported by the device for receive frame transfers, this driver does not enable DMA.

This network interface driver does not support output hook routines, because to do so requires that an image of the transmit packet be built in memory before the image is copied to the CS8900 chip. It is much more efficient to copy the image directly from the mbuf chain to the CS8900 chip. However, this network interface driver does support input hook routines.

CONFIGURATION

The defined I/O address and IRQ in config.h must match the one stored in EEPROM by the vendor’s DOS utility program.

The I/O Address parameter is the only required csAttach() parameter. If the CS8900 chip has a EEPROM attached, then the I/O Address parameter, passed to the csAttach() routine, must match the I/O address programmed into the EEPROM. If the CS8900 chip does not have a EEPROM attached, then the I/O Address parameter must be 0x300.

The Interrupt Level parameter must have one of the following values:

0 - Get interrupt level from EEPROM
5 - IRQ 5
10 - IRQ 10
11 - IRQ 11
12 - IRQ 12
If the Interrupt Vector parameter is zero, then the network interface driver derives the interrupt vector from the interrupt level if possible. It is possible to derive the interrupt vector in an IBM PC compatible system. This parameter is present for systems which are not IBM PC compatible.

The Memory Address parameter specifies the base address of the CS8900 chip’s memory buffer (PacketPage). If the Memory Address parameter is not zero, then the CS8900 chip operates in memory mode at the specified address. If the Memory Address parameter is zero, then the CS8900 chip operates in the mode specified by the EEPROM or the Configuration Flags parameter.

The Media Type parameter must have one of the following values:
- 0 - Get media type from EEPROM
- 1 - AUI  (Thick Cable)
- 2 - BNC  10Base2 (Thin Cable)
- 3 - RJ45  10BaseT (Twisted Pair)

The Configuration Flags parameter is usually passed to the `csAttach()` routine as zero and the Configuration Flags information is retrieved from the EEPROM. The bits in the Configuration Flags parameter are usually specified by a hardware engineer and not by the end user. However, if the CS8900 chip does not have a EEPROM attached, then this information must be passed as a parameter to the `csAttach()` routine. The Configuration Flags are:

- 0x8000 - CS_CFGFLG_NOT_EEPROM    Don’t get Config. Flags from the EEPROM
- 0x0001 - CS_CFGFLG_MEM_MODE      Use memory mode to access the chip
- 0x0002 - CS_CFGFLG_USE_SA        Use system addr to qualify MEMCS16 signal
- 0x0004 - CS_CFGFLG_IOCHRDY       Use IO Channel Ready signal to slow access
- 0x0008 - CS_CFGFLG_DCDC_POL      The DC/DC conv. enable pin is active high
- 0x0010 - CS_CFGFLG_FDX           10BaseT is full duplex

If configuration flag information is passed to the `csAttach()` routine, then the `CS_CFGFLG_NOT_EEPROM` flag should be set. This ensures that the Configuration Flags parameter is not zero, even if all specified flags are zero.

If the Memory Address parameter is not zero and the Configuration Flags parameter is zero, then the CS8900 network interface driver implicitly sets the `CS_CFGFLG_MEM_MODE` flag and the CS8900 chip operates in memory mode. However, if the Configuration Flags parameter is not zero, then the CS8900 chip operates in memory mode only if the `CS_CFGFLG_MEM_MODE` flag is explicitly set. If the Configuration Flags parameter in not zero and the `CS_CFGFLG_MEM_MODE` flag is not set, then the CS8900 chip operates in I/O mode.

The Ethernet Address parameter is usually passed to the `csAttach()` routine as zero and the Ethernet address is retrieved from the EEPROM. The Ethernet address (also called hardware address and individual address) is usually supplied by the adapter manufacturer and is stored in the EEPROM. However, if the CS8900 chip does not have a EEPROM attached, then the Ethernet address must be passed as a parameter to the
csAttach() routine. The Ethernet Address parameter, passed to the csAttach() routine, contains the address of a NULL terminated string. The string consists of 6 hexadecimal numbers separated by colon characters. Each hexadecimal number is in the range 00 - FF. An example of this string is:

"00:24:20:10:FF:2A"

**BOARD LAYOUT**

This device is soft-configured. No jumpering diagram is required.

**EXTERNAL INTERFACE**

The only user-callable routines are csAttach():

```c
void csAttach()
```

Publishes the cs interface and initializes the driver and device.

The network interface driver includes a show routine, called csShow(), which displays driver configuration and statistics information. To invoke the show routine, type at the shell prompt:

```shell
-> csShow
```

To reset the statistics to zero, type at the shell prompt:

```shell
-> csShow 0, 1
```

Another routine that you may find useful is:

```shell
-> ifShow "cs0"
```

**EXTERNAL ROUTINES**

For debugging purposes, this driver calls logMsg() to print error and debugging information. This will cause the logLib library to be linked with any image containing this driver.

This driver needs the following macros defined for proper execution. Each has a default definition that assumes a PC386/PC486 system and BSP.

The macro `CS_IN_BYTE(reg, pAddr)` reads one byte from the I/O address reg, placing the result at address pAddr. There is no status result from this operation, we assume the operation completes normally, or a bus exception will occur. By default, this macro assumes there is a BSP routine sysInByte() to perform the I/O operation.

The macro `CS_IN_WORD(reg, pAddr)` reads a short word (2 bytes) from the I/O address reg, storing the result at address pAddr. We assume this completes normally, or causes a bus exception. The default declaration assumes a BSP routine sysInWord() to perform the operation.

The macro `CS_OUT_WORD(reg, data)` writes a short word value data at the I/O address reg. The default declaration assumes a BSP routine sysOutWord().

The macro `CS_INT_ENABLE(level, pResult)` is used to enable the interrupt level passed as an argument to csAttach(). The default definition call the BSP routine
sysIntEnablePIC(level). The STATUS return value from the actual routine is stored at pResult for the driver to examine.

The macro CS_INT_CONNECT(ivec, rtn, arg, pResult) macro is used to connect the driver interrupt routine to the vector provided as an argument to csAttach() (after translation by INUM_TO_IVEC). The default definition calls the CPU architecture routine intConnect().

The macro CS_IRQ0_VECTOR(pAddr) is used to fetch the base vector for the interrupt level mechanism. If the int vector argument to csAttach() is zero, then the driver will compute a vector number by adding the interrupt level to the value returned by this macro. If the user supplies a non-zero interrupt vector number, then this macro is not used. The default definition of this macro fetches the base vector number from a global value called sysVectorIRQ0.

The macro CS_MSEC_DELAY(msec) is used to delay execution for a specified number of milliseconds. The default definition uses taskDelay() to suspend task for some number of clock ticks. The resolution of the system clock is usually around 16 milliseconds (msecs), which is fairly coarse.

if_dc

NAME

if_dc – DEC 21x4x Ethernet LAN network interface driver

ROUTINES

dcattach() – publish the dc network interface
dcReadAllRom() – read entire serial rom
dcViewRom() – display lines of serial ROM for dec21140
dcCsrShow() – display dec 21040/21140 status registers 0 through 15

DESCRIPTION

This module implements an ethernet interface driver for the DEC 21x4x family, and currently supports the following variants -- 21040, 21140, and 21140A.

The DEC 21x4x PCI Ethernet controllers are inherently little-endian since they are designed for a little-endian PCI bus. While the 21040 only supports a 10Mps interface, other members of this family are dual-speed devices which support both 10 and 100Mbps.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks; and on multiple versions of the dec21x4x family. To achieve this, the driver takes several parameters, and external support routines which are detailed below. Also stated below are assumptions made by the driver of the hardware, and if any of these assumptions are not true for your hardware, the driver will probably not function correctly.

This driver supports up to 4 ethernet units per CPU, and can be configured for either big-endian or little-endian architectures. It contains error-recovery code to handle known device errata related to DMA activity.
On a dec21040, this driver configures the 10BASE-T interface by default and waits for two seconds to check the status of the link. If the link status is 'fail,' it then configures the AUI interface.

The dec21140, and dec21140A devices support both 10 and 100Mbps plus a variety of MII and non-MII PHY interfaces. This driver reads a DEC version 2.0 SROM device for PHY initialization information, and automatically configures an appropriate active PHY media.

**BOARD LAYOUT**

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

**EXTERNAL INTERFACE**

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is **NULL**.

The only user-callable routine is **dcattach()**, which publishes the **dc** interface and initializes the driver and device.

**TARGET-SPECIFIC PARAMETERS**

**bus mode**

This parameter is a global variable that can be modified at run-time.

The LAN control register #0 determines the bus mode of the device, allowing the support of big-endian and little-endian architectures. This parameter, defined as "**ULONG dcCSR0Bmr**", is the value that will be placed into device control register #0. The default is mode is little-endian.

For information about changing this parameter, see the manual **DEC Local Area Network Controller DEC21040 or DEC21140 for PCI**.

**base address of device registers**

This parameter is passed to the driver by **dcattach()**.

**interrupt vector**

This parameter is passed to the driver by **dcattach()**.

This driver configures the 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 device interrupt.

**interrupt level**

This parameter is passed to the driver by **dcattach()**.

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 **sysLanIntEnable()** is called to perform any board-specific operations required to allow the servicing of a
device interrupt. For a description of `sysLanIntEnable()`, see "External Support Requirements" below.

This parameter is passed to the external routine.

**shared memory address**

This parameter is passed to the driver by `dcattach()`.

The DEC 21x4x 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 DEC 21x4x. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

This parameter can be used to specify an explicit memory region for use by the DEC 21x4x device. This should be done on hardware that restricts the DEC 21x4x device to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

**shared memory size**

This parameter is passed to the driver by `dcattach()`.

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.

**shared memory width**

This parameter is passed to the driver by `dcattach()`.

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.

**shared memory buffer size**

This parameter is passed to the driver by `dcattach()`.

The driver and DEC 21x4x device exchange network data in buffers. This parameter permits the size of these individual buffers to be limited. A value of zero indicates that the default buffer size should be used. The default buffer size is large enough to hold a maximum-size Ethernet packet.

**pci Memory base**

This parameter is passed to the driver by `dcattach()`. This parameter gives the base address of the main memory on the PCI bus.
dcOpMode
This parameter is passed to the driver by dcattach(). This parameter gives the mode of initialization of the device. The mode flags for both the DEC21040 and DEC21140 interfaces are listed below.

- DC_PROMISCUOUS_FLAG 0x01
- DC_MULTICAST_FLAG 0x02

The mode flags specific to the DEC21140 interface are listed below.

- DC_100_MB_FLAG 0x04
- DC_21140_FLAG 0x08
- DC_SCRAMBLER_FLAG 0x10
- DC_PCS_FLAG 0x20
- DC_PS_FLAG 0x40
- DC_FULLDUPLEX_FLAG 0x10

Loopback mode flags

- DC_ILOOPB_FLAG 0x100
- DC_ELOOPB_FLAG 0x200
- DC_HBE_FLAG 0x400

Ethernet address
This is obtained by the driver by reading an ethernet ROM register or the DEC serial ROM.

EXTERNAL SUPPORT REQUIREMENTS
This driver requires one external support function:

void sysLanIntEnable (int level)
This routine provides a target-specific enable of the interrupt for the DEC 21x4x device. Typically, this involves interrupt controller hardware, either internal or external to the CPU.

This routine is called once via the macro SYS_INT_ENABLE().

SEE ALSO ifLib, DECchip 21040 or 21140 Ethernet LAN Controller for PCI
if_eex

NAME

if_eex – Intel EtherExpress 16 network interface driver

ROUTINES

eexattach() – publish the eex network interface and initialize the driver and device
eexTxStartup() – start output on the chip

DESCRIPTION

This module implements the Intel EtherExpress 16 PC network interface card driver. It is specific to that board as used in PC 386/486 hosts. This driver is written using the device’s I/O registers exclusively.

SIMPLIFYING ASSUMPTIONS

This module assumes a little-endian host (80x86); thus, no endian adjustments are needed to manipulate the 82586 data structures (little-endian).

The on-board memory is assumed to be sufficient; thus, no provision is made for additional buffering in system memory.

The “frame descriptor” and “buffer descriptor” structures can be bound into permanent pairs by pointing each FD at a “chain” of one BD of MTU size. The 82586 receive algorithm fills exactly one BD for each FD; it looks to the NEXT FD in line for the next BD.

The transmit and receive descriptor lists are permanently linked into circular queues partitioned into sublists designated by the EEX_LIST headers in the driver control structure. Empty partitions have NULL pointer fields. EL bits are set as needed to tell the 82586 where a partition ends. The lists are managed in strict FIFO fashion; thus the link fields are never modified, just ignored if a descriptor is at the end of a list partition.

BOARD LAYOUT

This device is soft-configured. No jumpering diagram is required.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine and there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the init() routine is NULL.

There is one user-callable routine, eexattach(). For details on usage, see the manual entry for this routine.

EXTERNAL SUPPORT REQUIREMENTS

None.

SYSTEM RESOURCE USAGE

- one mutual exclusion semaphore
- one interrupt vector
VxWorks Drivers API Reference, 5.5

**if_ei**

- one watchdog timer
- 8 bytes in the initialized data section (data)
- 912 bytes in the uninitialized data section (bss)

The data and bss sections are quoted for the MC68020 architecture and may vary for other architectures. The code size (text) will vary widely between architectures, and is thus not quoted here.

The device contains on-board buffer memory; no system memory is required for buffering.

**TUNING HINTS**
The only adjustable parameter is the number of TFDs to create in adapter buffer memory. The total number of TFDs and RFDs is 21, given full-frame buffering and the sizes of the auxiliary structures. `eexattach()` requires at least `MIN_NUM_RFDS` RFDs to exist. More than ten TFDs is not sensible in typical circumstances.

**SEE ALSO**
ifLib

---

**NAME**

if_ei – Intel 82596 Ethernet network interface driver

**ROUTINES**
eiattach() – publish the ei network interface and initialize the driver and device
eiTxStartup() – start output on the chip

**DESCRIPTION**
This module implements the Intel 82596 Ethernet 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, 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.

This driver can run with the device configured in either big-endian or little-endian modes. Error recovery code has been added to deal with some of the known errata in the A0 version of the device. This driver supports up to four individual units per CPU.

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

**EXTERNAL INTERFACE**
This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the
initialization routine is NULL.

The only user-callable routine is `eiattach()`, which publishes the `ei` interface and initializes the driver and device.

**TARGET-SPECIFIC PARAMETERS**

the sysbus value

This parameter is passed to the driver by `eiattach()`.

The Intel 82596 requires this parameter during initialization. This parameter tells the device about the system bus, hence the name "sysbus." To determine the correct value for a target, refer to the document *Intel 32-bit Local Area Network (LAN) Component User's Manual*.

interrupt vector

This parameter is passed to the driver by `eiattach()`.

The Intel 82596 generates hardware interrupts for various events within the device; thus it contains an interrupt handler routine. This driver calls `intConnect()` to connect its interrupt handler to the interrupt vector generated as a result of the 82596 interrupt.

shared memory address

This parameter is passed to the driver by `eiattach()`.

The Intel 82596 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 82596.

This parameter can be used to specify an explicit memory region for use by the 82596. This should be done on targets that restrict the 82596 to a particular memory region. The constant `NONE` can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

number of Receive and Transmit Frame Descriptors

These parameters are passed to the driver by `eiattach()`.

The Intel 82596 accesses frame descriptors in memory for each frame transmitted or received. The number of frame descriptors at run-time can be configured using these parameters.

Ethernet address

This parameter is obtained by a call to an external support routine.

During initialization, the driver needs to know the Ethernet address for the Intel 82596 device. The driver calls the external support routine, `sysEnetAddrGet()`, to obtain the Ethernet address. For a description of `sysEnetAddrGet()`, see "External Support Requirements" below.
EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

**STATUS sysEnetAddrGet (int unit, char *pCopy)**

This routine provides the six-byte Ethernet address used by *unit*. It must copy the six-byte address to the space provided by *pCopy*. This routine returns OK, or ERROR if it fails. The driver calls this routine, once per unit, using `eiattach()`.

**STATUS sys596Init (int unit)**

This routine performs any target-specific initialization required before the 82596 is initialized. Typically, it is empty. This routine must return OK, or ERROR if it fails. The driver calls this routine, once per unit, using `eiattach()`.

**void sys596Port (int unit, int cmd, UINT32 addr)**

This routine provides access to the special port function of the 82596. It delivers the command and address arguments to the port of the specified unit. The driver calls this routine primarily during initialization, but may also call it during error recovery procedures.

**void sys596ChanAttn (int unit)**

This routine provides the channel attention signal to the 82596, for the specified *unit*. The driver calls this routine frequently throughout all phases of operation.

**void sys596IntEnable (int unit), void sys596IntDisable (int unit)**

These routines enable or disable the interrupt from the 82596 for the specified *unit*. Typically, this involves interrupt controller hardware, either internal or external to the CPU. Since the 82596 itself has no mechanism for controlling its interrupt activity, these routines are vital to the correct operation of the driver. The driver calls these routines throughout normal operation to protect certain critical sections of code from interrupt handler intervention.

**void sys596IntAck (int unit)**

This routine must perform any required interrupt acknowledgment or clearing. Typically, this involves an operation to some interrupt control hardware.

**NOTE:** The INT signal from the 82596 behaves in an “edge-triggered” mode; therefore, this routine typically clears a latch within the control circuitry. The driver calls this routine from the interrupt handler.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- one watchdog timer.
- 8 bytes in the initialized data section (data)
- 912 bytes in the uninitialized data section (BSS)
The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The driver uses `cacheDmaMalloc()` to allocate memory to share with the 82596. The fixed-size pieces in this area total 160 bytes. The variable-size pieces in this area are affected by the configuration parameters specified in the `eiattach()` call. The size of one RFD (Receive Frame Descriptor) is 1536 bytes. The size of one TFD (Transmit Frame Descriptor) is 1534 bytes. For more information about RFDs and TFDs, see the *Intel 82596 User’s Manual*.

The 82596 can be operated only if this shared memory region is non-cacheable or if the hardware implements 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 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 `eiattach()` 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.

**CAVEAT**

This driver does not support promiscuous mode.

**SEE ALSO**


---

### if_eidve

**NAME**

`if_eidve` – Intel 82596 Ethernet network interface driver for DVE-SH7XXX

**ROUTINES**

- `eiattach()` – publish the `ei` network interface and initialize the driver and device
- `eiTxStartup()` – start output on the chip

**DESCRIPTION**

This module implements the Intel 82596 Ethernet 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, 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.
This driver can run with the device configured in either big-endian or little-endian modes. Error recovery code has been added to deal with some of the known errata in the A0 version of the device. This driver supports up to four individual units per CPU.

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

**EXTERNAL INTERFACE**
This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is `NULL`.

The only user-callable routine is `eiattach()`, which publishes the `ei` interface and initializes the driver and device.

**TARGET-SPECIFIC PARAMETERS**

- **the sysbus value**
  This parameter is passed to the driver by `eiattach()`.
  The Intel 82596 requires this parameter during initialization. It tells the device about the system bus, hence the name “sysbus.” To determine the correct value for a target, refer to the document *Intel 32-bit Local Area Network (LAN) Component User’s Manual*.

- **interrupt vector**
  This parameter is passed to the driver by `eiattach()`.
  The Intel 82596 generates hardware interrupts for various events within the device; thus it contains an interrupt handler routine. This driver calls `intConnect()` to connect its interrupt handler to the interrupt vector generated as a result of the 82596 interrupt.

- **shared memory address**
  This parameter is passed to the driver by `eiattach()`.
  The Intel 82596 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 82596.
  This parameter can be used to specify an explicit memory region for use by the 82596. This should be done on targets that restrict the 82596 to a particular memory region. The constant `NONE` can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

- **number of Receive and Transmit Frame Descriptors**
  These parameters are passed to the driver by `eiattach()`.
  The Intel 82596 accesses frame descriptors in memory for each frame transmitted or received. The number of frame descriptors at run-time can be configured using these parameters.
Ethernet address
This parameter is obtained by a call to an external support routine.

During initialization, the driver needs to know the Ethernet address for the Intel 82596 device. The driver calls the external support routine, `sysEnetAddrGet()`, to obtain the Ethernet address. For a description of `sysEnetAddrGet()`, see “External Support Requirements” below.

**EXTERNAL SUPPORT REQUIREMENTS**
This driver requires seven external support functions:

- **STATUS sysEnetAddrGet (int unit, char *pCopy)**
  This routine provides the six-byte Ethernet address used by `unit`. It must copy the six-byte address to the space provided by `pCopy`. This routine returns OK, or ERROR if it fails. The driver calls this routine, once per unit, using `eiattach()`.

- **STATUS sys596Init (int unit)**
  This routine performs any target-specific initialization required before the 82596 is initialized. Typically, it is empty. This routine must return OK, or ERROR if it fails. The driver calls this routine, once per unit, using `eiattach()`.

- **void sys596Port (int unit, int cmd, UINT32 addr)**
  This routine provides access to the special port function of the 82596. It delivers the command and address arguments to the port of the specified unit. The driver calls this routine primarily during initialization, but may also call it during error recovery procedures.

- **void sys596ChanAttn (int unit)**
  This routine provides the channel attention signal to the 82596, for the specified `unit`. The driver calls this routine frequently throughout all phases of operation.

- **void sys596IntEnable (int unit), void sys596IntDisable (int unit)**
  These routines enable or disable the interrupt from the 82596 for the specified `unit`. Typically, this involves interrupt controller hardware, either internal or external to the CPU. Since the 82596 itself has no mechanism for controlling its interrupt activity, these routines are vital to the correct operation of the driver. The driver calls these routines throughout normal operation to protect certain critical sections of code from interrupt handler intervention.

- **void sys596IntAck (int unit)**
  This routine must perform any required interrupt acknowledgment or clearing. Typically, this involves an operation to some interrupt control hardware.

**NOTE:** The INT signal from the 82596 behaves in an "edge-triggered" mode; therefore, this routine typically clears a latch within the control circuitry. The driver calls this routine from the interrupt handler.
SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- one watchdog timer
- 8 bytes in the initialized data section (data)
- 912 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The driver uses cacheDmaMalloc() to allocate memory to share with the 82596. The fixed-size pieces in this area total 160 bytes. The variable-size pieces in this area are affected by the configuration parameters specified in the eiattach() call. The size of one RFD (Receive Frame Descriptor) is 1536 bytes. The size of one TFD (Transmit Frame Descriptor) is 1534 bytes. For more information about RFDs and TFDs, see the Intel 82596 User’s Manual.

The 82596 can be operated only if this shared memory region is non-cacheable or if the hardware implements 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 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 eiattach() 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.

SEE ALSO

**if_eihk**

**NAME**

`if_eihk` – Intel 82596 Ethernet network interface driver for hkv3500

**ROUTINES**

- `eihkattach()` – publish the `ei` network interface and initialize the driver and device
- `eiTxStartup()` – start output on the chip
- `eiInt()` – entry point for handling interrupts from the 82596

**DESCRIPTION**

This module implements a hkv3500 specific Intel 82596 Ethernet network interface driver. This driver is derived from the generic `if_ei` ethernet driver to support hkv3500 target board. The receive buffer scheme has been modified from a simplified memory structure to a flexible memory structure so that receive buffers can be word-aligned, and thus support buffer loaning on a MIPS CPU architecture.

The driver requires several target-specific parameters, and some external support routines which are detailed below.

This driver can run with the device configured in either big-endian or little-endian modes. Error recovery code has been added to deal with some of the known errata in the A0 version of the device. This driver supports up to four individual units per CPU.

**BOARD LAYOUT**

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

**EXTERNAL INTERFACE**

This driver provides the standard external interface with the following exceptions. All initialization is performed within the `attach` routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is `NULL`.

The only user-callable routine is `eihkattach()`, which publishes the `ei` interface and initializes the driver and device.

**TARGET-SPECIFIC PARAMETERS**

- the `sysbus` value
  
  This parameter is passed to the driver by `eihkattach()`.

  The Intel 82596 requires this parameter during initialization. This parameter tells the device about the system bus, hence the name "sysbus." To determine the correct value for a target, refer to the document *Intel 32-bit Local Area Network (LAN) Component User’s Manual.*

- interrupt vector
  
  This parameter is passed to the driver by `eihkattach()`.

  The Intel 82596 generates hardware interrupts for various events within the device;
thus it contains an interrupt handler routine. This driver calls \texttt{intConnect()} to connect its interrupt handler to the interrupt vector generated as a result of the 82596 interrupt.

\textbf{shared memory address}
This parameter is passed to the driver by \texttt{eihkattach()}. The Intel 82596 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 82596. This parameter can be used to specify an explicit memory region for use by the 82596. This should be done on targets that restrict the 82596 to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

\textbf{number of Receive and Transmit Frame Descriptors}
These parameters are passed to the driver by \texttt{eihkattach()}. The Intel 82596 accesses frame descriptors in memory for each frame transmitted or received. The number of frame descriptors at run-time can be configured using these parameters.

\textbf{Ethernet address}
This parameter is obtained by a call to an external support routine. During initialization, the driver needs to know the Ethernet address for the Intel 82596 device. The driver calls the external support routine, \texttt{sysEnetAddrGet()}, to obtain the Ethernet address. For a description of \texttt{sysEnetAddrGet()}, see "External Support Requirements" below.

\textbf{EXTERNAL SUPPORT REQUIREMENTS}
This driver requires seven external support functions:

\textbf{STATUS sysEnetAddrGet (int unit, char *pCopy)}
This routine provides the six-byte Ethernet address used by \texttt{unit}. It must copy the six-byte address to the space provided by \texttt{pCopy}. This routine returns \texttt{OK}, or \texttt{ERROR} if it fails. The driver calls this routine, once per unit, using \texttt{eihkattach()}.  

\textbf{STATUS sys596Init (int unit, SCB *pScb)}
This routine performs any target-specific initialization required before the 82596 is initialized. Typically, it is empty. This routine must return \texttt{OK}, or \texttt{ERROR} if it fails. The driver calls this routine, once per unit, using \texttt{eihkattach()}.  

\textbf{void sys596Port (int unit, int cmd, UINT32 addr)}
This routine provides access to the special port function of the 82596. It delivers the command and address arguments to the port of the specified unit. The driver calls this routine primarily during initialization, but may also call it during error recovery procedures.
void sys596ChanAtn (int unit)
This routine provides the channel attention signal to the 82596, for the specified unit. The driver calls this routine frequently throughout all phases of operation.

void sys596IntEnable (int unit), void sys596IntDisable (int unit)
These routines enable or disable the interrupt from the 82596 for the specified unit. Typically, this involves interrupt controller hardware, either internal or external to the CPU. Since the 82596 itself has no mechanism for controlling its interrupt activity, these routines are vital to the correct operation of the driver. The driver calls these routines throughout normal operation to protect certain critical sections of code from interrupt handler intervention.

void sys596IntAck (int unit)
This routine must perform any required interrupt acknowledgment or clearing. Typically, this involves an operation to some interrupt control hardware.

NOTE: The INT signal from the 82596 behaves in an "edge-triggered" mode; therefore, this routine typically clears a latch within the control circuitry. The driver calls this routine from the interrupt handler.

SYSTEM RESOURCE USAGE
When implemented, this driver requires the following system resources:
- one mutual exclusion semaphore
- one interrupt vector
- one watchdog timer
- 8 bytes in the initialized data section (data)
- 912 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The driver uses `cacheDmaMalloc()` to allocate memory to share with the 82596. The fixed-size pieces in this area total 160 bytes. The variable-size pieces in this area are affected by the configuration parameters specified in the `eihkattach()` call. The size of one RFD (Receive Frame Descriptor) is 1536 bytes. The size of one TFD (Transmit Frame Descriptor) is 1534 bytes. For more information about RFDs and TFDs, see the Intel 82596 User’s Manual.

The 82596 can be operated only if this shared memory region is non-cacheable or if the hardware implements 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.
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 `eihkattach()` 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.

**SEE ALSO**

if_elt

**NAME**

if_elt – 3Com 3C509 Ethernet network interface driver

**ROUTINES**

eltattach( ) – publish the elt interface and initialize the driver and device
eltTxOutputStart() – start output on the board
eltShow() – display statistics for the 3C509 elt network interface

**DESCRIPTION**

This module implements the 3Com 3C509 network adapter driver.

The 3C509 (EtherLink® III) is not well-suited for use in real-time systems. Its meager on-board buffering (4K total; 2K transmit, 2K receive) forces the host processor to service the board at a high priority. 3Com makes a virtue of this necessity by adding fancy lookahead support and adding the label "Parallel Tasking" to the outside of the box. Using 3Com’s drivers, this board will look good in benchmarks that measure raw link speed.

The board is greatly simplified by using the host CPU as a DMA controller.

**BOARD LAYOUT**

This device is soft-configured by a DOS-hosted program supplied by the manufacturer. No jumpering diagram is required.

**EXTERNAL INTERFACE**

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine and there is no separate initialization routine. Thus, in the global interface structure, the function pointer to the initialization routine is NULL.

There are two user-callable routines:

eltattach( )

publishes the elt interface and initializes the driver and device.

eltShow()

displays statistics that are collected in the interrupt handler.

See the manual entries for these routines for more detail.

**SYSTEM RESOURCE USAGE**

- one mutual exclusion semaphore
- one interrupt vector
- 16 bytes in the uninitialized data section (bss)
- 180 bytes (plus overhead) of malloc’ed memory per unit
- 1530 bytes (plus overhead) of malloc’ed memory per frame buffer, minimum 5 frame buffers.
SHORTCUTS

The EISA and MCA versions of the board are not supported.

Attachment selection assumes the board is in power-on reset state; a warm restart will not clear the old attachment selection out of the hardware, and certain new selections may not clear it either. For example, if RJ45 was selected, the system is warm-booted, and AUI is selected, the RJ45 connector is still functional.

Attachment type selection is not validated against the board’s capabilities, even though there is a register that describes which connectors exist.

The loaned buffer cluster type is MC_EI; no new type is defined yet.

Although it seems possible to put the transmitter into a non-functioning state, it is not obvious either how to do this or how to detect the resulting state. There is therefore no transmit watchdog timer.

No use is made of the tuning features of the board; it is possible that proper dynamic tuning would reduce or eliminate the receive overruns that occur when receiving under task control (instead of in the ISR).

TUNING HINTS

More receive buffers (than the default 20) could help by allowing more loaning in cases of massive reception; four per receiving TCP connection plus four extras should be considered a minimum.

SEE ALSO

ifLib

if_ene

NAME

if_ene – Novell/Eagle NE2000 network interface driver

ROUTINES

eneattach() – publish the ene network interface and initialize the driver and device
enePut() – copy a packet to the interface.
eneShow() – display statistics for the NE2000 ene network interface

DESCRIPTION

This module implements the Novell/Eagle NE2000 network interface driver. There is one user-callable routine, eneattach().

BOARD LAYOUT

The diagram below shows the relevant jumpers for VxWorks configuration. Other compatible boards will be jumpered differently; many are jumperless.
EXTERNAL INTERFACE

There are two user-callable routines:

**eneattach()**

Publishes the **ene** interface and initializes the driver and device.

**eneShow()**

Displays statistics that are collected in the interrupt handler.

See the manual entries for these routines for more detail.

| W1..W8 | 1-2 | position selects AUI ("DIX") connector
|        | 2-3 | position selects BNC (10BASE2) connector
| W9..W11 | YYN | I/O address 300h, no boot ROM
|         | NYN | I/O address 320h, no boot ROM
|         | YNN | I/O address 340h, no boot ROM
|         | NNN | I/O address 360h, no boot ROM
|         | YYY | I/O address 300h, boot ROM at paragraph 0c800h
|         | NYY | I/O address 320h, boot ROM at paragraph 0cc00h
|         | YNY | I/O address 340h, boot ROM at paragraph 0d000h
|         | NNY | I/O address 360h, boot ROM at ??? (invalid configuration?)
| W12     | Y   | IRQ 2 (or 9 if you prefer)
| W13     | Y   | IRQ 3
| W14     | Y   | IRQ 4
| W15     | Y   | IRQ 5 (note that only one of W12..W15 may be installed)
| W16     | Y   | normal ISA bus timing
|         | N   | timing for COMPAQ 286 portable, PS/2 Model 30-286, C&T chipset
if_esmc

NAME

if_esmc – Ampro Ethernet2 SMC-91c9x Ethernet network interface driver

ROUTINES

esmcattach( ) – publish the esmc network interface and initialize the driver.
esmcPut( ) – copy a packet to the interface.
esmcShow( ) – display statistics for the esmc network interface

DESCRIPTION

This module implements the Ampro Ethernet2 SMC-91c9x Ethernet network interface driver.

CONFIGURATION

The W3 and W4 jumper should be set for IO address and IRQ. The defined I/O address
and IRQ in config.h must match the one stored in EEROM and the jumper setting.

BOARD LAYOUT

The diagram below shows the relevant jumpers for VxWorks configuration.

```
  * * * * 
  |      | 
  | U1   | W1  W3 
  | PROM | X   " 
  | .    | .    
  | .    | 
  | W4   |
  | .    |
  | .    |

W1: Boot PROM Size
W3: IO-address, IRQ, Media
W4: IRQ Group Selection
```
EXTERNAL INTERFACE

The only user-callable routines are `esmcattach()`:

```c
esmcattach()
```

Publishes the `esmc` interface and initializes the driver and device.

The last parameter of `esmcattach()`, `mode`, is a receive mode. If it is 0, a packet is received in the interrupt level. If it is 1, a packet is received in the task level. Receiving packets in the interrupt level requires about 10K bytes of memory, but minimize a risk of dropping packets. Receiving packets in the task level doesn’t require extra memory, but might have a risk of dropping packets.

---

### if_fei

**NAME**  
`if_fei` – Intel 82557 Ethernet network interface driver

**ROUTINES**  
`feiattach()` – publish the `fei` network interface

**DESCRIPTION**  
This module implements the Intel 82557 Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the entire range of architectures and targets supported by VxWorks. 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.

This driver supports up to four individual units.

**EXTERNAL INTERFACE**  
The user-callable routine is `feiattach()`, which publishes the `fei` interface and performs some initialization.

After calling `feiattach()` to publish the interface, an initialization routine must be called to bring the device up to an operational state. The initialization routine is not a user-callable routine; upper layers call it when the interface flag is set to UP, or when the interface’s IP address is set.

There is a global variable `feiIntConnect` which specifies the interrupt connect routine to be used depending on the BSP. This is by default set to `intConnect()` and the user can override this to use any other interrupt connect routine (like `pciIntConnect()` in `sysHwInit()` or any device specific initialization routine called in `sysHwInit()`).
TARGET-SPECIFIC PARAMETERS

shared memory address
This parameter is passed to the driver via feiattach().

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. The constant NONE can be used to indicate that there are no memory limitations, in which case the driver attempts to allocate the shared memory from the system space.

number of Command, Receive, and Loanable-Receive Frame Descriptors
These parameters are passed to the driver via feiattach().

The Intel 82557 accesses frame descriptors (and their associated buffers) in memory for each frame transmitted or received. The number of frame descriptors can be configured at run-time using these parameters.

Ethernet address
This parameter is obtained by a call to an external support routine.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires the following external support function:

```
STATUS sys557Init (int unit, 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 feiattach() call. The size of one RFD (Receive Frame Descriptor) is the same as one CFD (Command Frame Descriptor): 1536 bytes. For more information about RFDs and CFDs, see the Intel 82557 User’s Manual.

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.

Additionally, this version of the driver does not handle virtual-to-physical or physical-to-virtual memory mapping.

TUNING HINTS

The only adjustable parameters are the number of frame descriptors that will be created at run-time. These parameters are given to the driver by feiattach(). There is one CFD and
one RFD associated with each transmitted frame and each received frame, respectively. For memory-limited applications, decreasing the number of CFDs and RFDs may be desirable. Increasing CFDs 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.

SEE ALSO ifLib, Intel 82557 User’s Manual

if_fn

NAME
if_fn – Fujitsu MB86960 NICE Ethernet network interface driver

ROUTINES
fnattach( ) – publish the fn network interface and initialize the driver and device

DESCRIPTION
This module implements the Fujitsu MB86960 NICE Ethernet network interface driver.

This driver is non-generic and has only been run on the Fujitsu SPARClite Evaluation Board. It currently supports only unit number zero. The 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
This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is fnattach( ), which publishes the fn interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS
External support routines provide all parameters:

device I/O address
This parameter specifies the base address of the device’s I/O register set. This address is assumed to live in SPARClite alternate address space.

interrupt vector
This parameter specifies the interrupt vector to be used by the driver to service an interrupt from the NICE device. The driver will connect the interrupt handler to this vector by calling intConnect( ).
Ethernet address

This parameter specifies the unique, six-byte address assigned to the VxWorks target on the Ethernet.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires five external support functions:

char *sysEnetIOAddrGet (int unit)
This routine returns the base address of the NICE control registers. The driver calls this routine once, using fnattach() .

int sysEnetVectGet (int unit)
This routine returns the interrupt vector number to be used to connect the driver’s interrupt handler. The driver calls this routine once, using fnattach() .

STATUS sysEnetAddrGet (int unit, char *pCopy)
This routine provides the six-byte Ethernet address used by unit . It must copy the six-byte address to the space provided by pCopy . It returns OK , or ERROR if it fails. The driver calls this routine once, using fnattach() .

void sysEnetIntEnable (int unit), void sysEnetIntDisable (int unit)
These routines enable or disable the interrupt from the NICE for the specified unit . Typically, this involves interrupt controller hardware, either internal or external to the CPU. The driver calls these routines only during initialization, using fnattach() .

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 3944 bytes in text section (text)
– 0 bytes in the initialized data section (data)
– 3152 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the SPARClite architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The NICE device maintains a private buffer for all packets transmitted and received. Therefore, the driver does not require any system memory to share with the device. This also eliminates all data cache coherency issues.

SEE ALSO

ifLib
if_In

NAME
if_In – AMD Am7990 LANCE Ethernet network interface driver

ROUTINES
Inattach() – publish the In network interface and initialize driver structures

DESCRIPTION
This module implements the Advanced Micro Devices Am7990 LANCE Ethernet 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 parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it.

This driver supports only one LANCE unit 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.

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

EXTERNAL INTERFACE
This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is Inattach(), which publishes the In interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

bus mode
This parameter is a global variable that can be modified at run-time.

The LANCE control register #3 determines the bus mode of the device, allowing the support of big-endian and little-endian architectures. This parameter, defined as "u_short lnCSR_3B", is the value that will be placed into LANCE control register #3. The default value supports Motorola-type buses. For information about changing this parameter, see the manual Advanced Micro Devices Local Area Network Controller Am7990 (LANCE).

base address of device registers
This parameter is passed to the driver by Inattach(). It indicates to the driver where to find the RDP register.
The LANCE 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 considered the "base address."

**interrupt vector**
This parameter is passed to the driver by `lnattach()`. 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.

**interrupt level**
This parameter is passed to the driver by `lnattach()`. 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 `sysLanIntEnable()` is called to perform any board-specific operations required to allow the servicing of a LANCE interrupt. For a description of `sysLanIntEnable()`, see "External Support Requirements" below.
This parameter is passed to the external routine.

**shared memory address**
This parameter is passed to the driver by `lnattach()`. The LANCE 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 LANCE. It assumes that this shared memory is directly available to it without any arbitration or timing concerns. This parameter can be used to specify an explicit memory region for use by the LANCE. This should be done on hardware that restricts the LANCE to a particular memory region. The constant `NONE` can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

**shared memory size**
This parameter is passed to the driver by `lnattach()`. 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.

**shared memory width**
This parameter is passed to the driver by `lnattach()`.
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.

**Ethernet address**

This parameter is obtained directly from a global memory location.

During initialization, the driver needs to know the Ethernet address for the LANCE device. The driver assumes that this address is available in a global, six-byte character array, `lnEnetAddr[]`. This array is typically created and stuffed by the BSP code.

**EXTERNAL SUPPORT REQUIREMENTS**

This driver requires one external support function:

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

This routine provides a target-specific enable of the interrupt for the LANCE device. Typically, this involves interrupt controller hardware, either internal or external to the CPU. This routine is called once, from the `lnattach()` routine.

**SYSTEM RESOURCE USAGE**

When implemented, this driver requires the following system resources:

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

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

If the driver is not given a specific region of memory via the `lnattach()` routine, then it calls `cacheDmaMalloc()` to allocate the memory to be shared with the LANCE. The size requested is 80,542 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

The LANCE can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data written by the driver because fields within the shared structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

**SEE ALSO**

`ifLib`, `Advanced Micro Devices Local Area Network Controller Am7990 (LANCE)`
if_InPci

NAME

if_InPci – AMD Am79C970 PCnet-PCI Ethernet network interface driver

ROUTINES

InPciattach( ) – publish the InPci network interface and initialize the driver and device

DESCRIPTION

This module implements the Advanced Micro Devices Am79C970 PCnet-PCI Ethernet 32 bit network interface driver.

The PCnet-PCI ethernet controller is inherently little-endian because the chip is designed to operate on a PCI bus which is a little-endian bus. The software interface to the driver is divided into three parts. The first part is the PCI configuration registers and their set up. This part is done at the BSP level in the various BSFs 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 parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it.

This driver supports only one LANCE unit 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, then input and output byte stream need not be swapped.

BOARD LAYOUT

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

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is InPciattach( ), which publishes the InPci interface and initializes the driver and device.
TARGET-SPECIFIC PARAMETERS

bus mode
This parameter is a global variable that can be modified at run-time.

The LANCE control register #3 determines the bus mode of the device, allowing the support of big-endian and little-endian architectures. This parameter, defined as "u_long lnPciCSR_3B", is the value that will be placed into LANCE control register #3. The default value supports Motorola-type buses. For information about changing this parameter, see the manual Advanced Micro Devices Local Area Network Controller Am79C970 (PCnet-PCI).

base address of device registers
This parameter is passed to the driver by lnPciattach(). It indicates to the driver where to find the RDP register.

The LANCE 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 considered the "base address."

interrupt vector
This parameter is passed to the driver by lnPciattach().

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.

interrupt level
This parameter is passed to the driver by lnPciattach().

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 sysLanIntEnable() is called to perform any board-specific operations required to turn on LANCE interrupt generation. A similar routine, sysLanIntDisable(), is called by the driver before a LANCE reset to perform board-specific operations required to turn off LANCE interrupt generation. For a description of sysLanIntEnable(), and sysLanIntDisable(), see "External Support Requirements" below.

This parameter is passed to the external routine.

shared memory address
This parameter is passed to the driver by lnPciattach().

The LANCE 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 LANCE. It assumes that this shared memory
is directly available to it without any arbitration or timing concerns.

This parameter can be used to specify an explicit memory region for use by the LANCE. This should be done on hardware that restricts the LANCE to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

shared memory size
This parameter is passed to the driver by \texttt{lnPciattach()}. 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.

shared memory width
This parameter is passed to the driver by \texttt{lnPciattach()}. 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.

shared memory buffer size
This parameter is passed to the driver by \texttt{lnPciattach()}. The driver and LANCE device exchange network data in buffers. This parameter permits the size of these individual buffers to be limited. A value of zero indicates that the default buffer size should be used. The default buffer size is large enough to hold a maximum-size Ethernet packet. Use of this parameter should be rare. Network performance will be affected, since the target will no longer be able to receive all valid packet sizes.

Ethernet address
This parameter is obtained directly from a global memory location. During initialization, the driver needs to know the Ethernet address for the LANCE device. The driver assumes that this address is available in a global, six-byte character array, \texttt{lnEnetAddr[]}. This array is typically created and stuffed by the BSP code.

\textbf{EXTERNAL SUPPORT REQUIREMENTS}
This driver requires one external support function:

\begin{verbatim}
void sysLanIntEnable (int level)
\end{verbatim}
This routine provides a target-specific enable of the interrupt for the LANCE device. Typically, this involves programming an interrupt controller hardware, either internal or external to the CPU.

This routine is called during chip initialization, at startup and each LANCE device reset.

```c
void sysLanIntDisable (int level)
```

This routine provides a target-specific disable of the interrupt for the LANCE device. Typically, this involves programming an interrupt controller hardware, either internal or external to the CPU.

This routine is called before a LANCE device reset.

**SYSTEM RESOURCE USAGE**

When implemented, this driver requires the following system resources:

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

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

If the driver is not given a specific region of memory via the `inPciAttach()` routine, then it calls `cacheDmaMalloc()` to allocate the memory to be shared with the LANCE. The size requested is 80,542 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

The LANCE can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data that is written by the driver because fields within the shared structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

See Also: `ifLib`, Advanced Micro Devices PCnet-PCI Ethernet Controller for PCI.
## if_loop

**NAME**  
if_loop – software loopback network interface driver

**ROUTINES**  
loattach() – publish the lo network interface and initialize the driver and pseudo-device

**DESCRIPTION**  
This module implements the software loopback network interface driver. The only user-callable routine is loattach(), which publishes the lo interface and initializes the driver and device.

This interface is used for protocol testing and timing. By default, the loopback interface is accessible at Internet address 127.0.0.1.

To use this feature, include the following component: INCLUDE_LOOPBACK

**BOARD LAYOUT**  
This device is "software only." A jumpering diagram is not applicable.

**SEE ALSO**  
ifLib

## if_mbc

**NAME**  
if_mbc – Motorola 68EN302 network-interface driver

**ROUTINES**  
mbcattach() – publish the mbc network interface and initialize the driver  
mbcStartOutput() – output packet to network interface device  
mbcIntr() – network interface interrupt handler

**DESCRIPTION**  
This is a driver for the Ethernet controller on the 68EN302 chip. The device supports a 16-bit interface, data rates up to 10 Mbps, a dual-ported RAM, and transparent DMA. The dual-ported RAM is used for a 64-entry CAM table, and a 128-entry buffer descriptor table. The CAM table is used to set the Ethernet address of the Ethernet device or to program multicast addresses. The buffer descriptor table is partitioned into fixed-size transmit and receive tables. The DMA operation is transparent and transfers data between the internal FIFOs and external buffers pointed to by the receive- and transmit-buffer descriptors during transmits and receives.

The driver currently supports one Ethernet module controller, but it can be extended to support multiple controllers when needed. An Ethernet module is initialized by calling mbcattach().

The driver supports buffer loaning for performance and input/output hook routines. It does not support multicast addresses.
The driver requires that the memory used for transmit and receive buffers be allocated in cache-safe RAM area.

A glitch in the EN302 Rev 0.1 device causes the Ethernet transmitter to lock up from time to time. The driver uses a watchdog timer to reset the Ethernet device when the device runs out of transmit buffers and cannot recover within 20 clock ticks.

**BOARD LAYOUT**

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

**EXTERNAL INTERFACE**

This driver presents the standard WRS network driver API: first the device unit must be attached with the `mbcattach()` routine, then it must be initialized with the `mbcInit()` routine.

The only user-callable routine is `mbcattach()`, which publishes the mbc interface and initializes the driver structures.

**TARGET-SPECIFIC PARAMETERS**

- **Ethernet module base address**
  
  This parameter is passed to the driver via `mbcattach()`. This parameter is the base address of the Ethernet module. The driver addresses all other Ethernet device registers as offsets from this address.

- **interrupt vector number**
  
  This parameter is passed to the driver via `mbcattach()`. The driver configures the Ethernet device to use this parameter while generating interrupt ack cycles. The interrupt service routine `mbcIntr()` is expected to be attached to the corresponding interrupt vector externally, typically in `sysHwInit2()`.

- **number of transmit and receive buffer descriptors**
  
  These parameters are passed to the driver via `mbcattach()`. The number of transmit and receive buffer descriptors (BDs) used is configurable by the user while attaching the driver. Each BD is 8 bytes in size and resides in the chip’s dual-ported memory, while its associated buffer, 1520 bytes in size, resides in cache-safe conventional RAM. A minimum of 2 receive and 2 transmit BDs should be allocated. If this parameter is `NULL`, a default of 32 BDs will be used. The maximum number of BDs depends on how the dual-ported BD RAM is partitioned. The 128 BDs in the dual-ported BD RAM can partitioned into transmit and receive BD regions with 8, 16, 32, or 64 transmit BDs and corresponding 120, 112, 96, or 64 receive BDs.

- **Ethernet DMA parameters**
  
  This parameter is passed to the driver via `mbcattach()`. This parameter is used to specify the settings of burst limit, water-mark, and transmit early, which control the Ethernet DMA, and is used to set the EDMA register.
base address of the buffer pool
This parameter is passed to the driver via mbcattach().
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 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 will be performed. This includes all transmit and receive buffers (see above) and an additional 16 receive loaner buffers, unless the number of receive BDs is less than 16, in which case that number of loaner buffers will be used. Each buffer is 1520 bytes. If this parameter is "NONE", space for buffers will be obtained by calling cacheDmaMalloc() in cpmattach().

EXTERNAL SUPPORT REQUIREMENTS
The driver requires the following support functions:


SYSTEM RESOURCE USAGE
The driver requires the following system resource:

– one mutual exclusion semaphore
– one interrupt vector
– one watchdog timer
– 0 bytes in the initialized data section (data)
– 296 bytes in the uninitialized data section (bss)

The data and BSS sections are quoted for the CPU32 architecture.

If the driver allocates the memory shared 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, 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 only operate if the shared 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 may share the same cache line. Additionally, the chip’s dual-ported RAM must be declared as non-cacheable memory where applicable.

SEE ALSO
if_nicEvb

NAME

if_nicEvb – National Semiconductor ST-NIC Chip network interface driver

ROUTINES

nicEvbattach() – publish and initialize the nicEvb network interface driver
nicTxStartup() – the driver’s actual output routine

DESCRIPTION

This module implements the National Semiconductor 83902A ST-NIC Ethernet network interface driver.

This driver is non-generic and is for use on the IBM EVB403 board. Only unit number zero is supported. The driver must be given several target-specific parameters. 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

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is nicEvbattach(), which publishes the nicEvb interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

device I/O address

This parameter is passed to the driver by nicEvbattach(). It specifies the base address of the device’s I/O register set.

interrupt vector

This parameter is passed to the driver by nicEvbattach(). It specifies the interrupt vector to be used by the driver to service an interrupt from the ST-NIC device. The driver will connect the interrupt handler to this vector by calling intConnect().

device restart/reset delay

The global variable nicRestartDelay (UINT32), defined in this file, should be initialized in the BSP sysHwInit() routine. nicRestartDelay is used only with PowerPC platform and is equal to the number of time base increments which makes for 1.6 msec. This corresponds to the delay necessary to respect when restarting or resetting the device.

EXTERNAL SUPPORT REQUIREMENTS

The driver requires the following support functions:
STATUS sysEnetAddrGet (int unit, UINT8 * addr)
The driver expects this routine to provide the six-byte Ethernet hardware address
that will be used by 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, during device initialization, from the nicEnetAddrGet() routine.

SYSTEM RESOURCE USAGE
When implemented, this driver requires the following system resources:
- one mutual exclusion semaphore
- one interrupt vector

SEE ALSO
ifLib

NAME
if_sl – Serial Line IP (SLIP) network interface driver

ROUTINES
slipInit() – initialize a SLIP interface
slipBaudSet() – set the baud rate for a SLIP interface
slattach() – publish the sl network interface and initialize the driver and device
slipDelete() – delete a SLIP interface

DESCRIPTION
This module implements the VxWorks Serial Line IP (SLIP) network interface driver.
Support for compressed TCP/IP headers (CSLIP) is included.
The SLIP driver enables VxWorks to talk to other machines over serial connections by
encapsulating IP packets into streams of bytes suitable for serial transmission.

USER-CALLABLE ROUTINES
SLIP devices are initialized using slipInit(). Its parameters specify the Internet address
for both sides of the SLIP point-to-point link, the name of the tty device on the local host,
and options to enable CSLIP header compression. The slipInit() routine calls slattach() to
attach the SLIP interface to the network. The slipDelete() routine deletes a specified SLIP
interface.

LINK-LEVEL PROTOCOL
SLIP is a simple protocol that uses four token characters to delimit each packet:
- END (0300)
- ESC (0333)
- TRANS_END (0334)
- TRANS_ESC (0335)
The END character denotes the end of an IP packet. The ESC character is used with TRANS_END and TRANS_ESC to circumvent potential occurrences of END or ESC within a packet. If the END character is to be embedded, SLIP sends "ESC TRANS_END" to avoid confusion between a SLIP-specific END and actual data whose value is END. If the ESC character is to be embedded, then SLIP sends "ESC TRANS_ESC" to avoid confusion.

**NOTE:** 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 IP packet has been received and sends it up to the IP layer.

**TARGET-SPECIFIC PARAMETERS**

The global flag **slipLoopBack** is set to 1 by default. This flag enables the packets to be sent to the loopback interface if they are destined to to a local slip interface address. By setting this flag, any packets sent to a local slip interface address will not be seen on the actual serial link. Set this flag to 0 to turn off this facility. If this flag is not set any packets sent to the local slip interface address will actually be sent out on the link and it is the peer’s responsibility to loop the packet back.

**IMPLEMENTATION**

The write side of a SLIP connection is an independent task. Each SLIP interface has its own output task that sends SLIP packets over a particular tty device channel. Whenever a packet is ready to be sent out, the SLIP driver activates this task by giving a semaphore. When the semaphore is available, the output task performs packetization (as explained above) and writes the packet to the tty device.

The receiving side is implemented as a "hook" into the tty driver. A tty ioctl() request, **FIOPROTOHOOK**, informs the tty driver to call the SLIP interrupt routine every time a character is received from a serial port. By tracking the number of characters and watching for the END character, the number of calls to read() and context switching time have been reduced. The SLIP interrupt routine will queue a call to the SLIP read routine only when it knows that a packet is ready in the tty driver’s ring buffer. The SLIP read routine will read a whole SLIP packet at a time and process it according to the SLIP framing rules. When a full IP packet is decoded out of a SLIP packet, it is queued to IP’s input queue.

CSLIP compression is implemented to decrease the size of the TCP/IP header information, thereby improving the data to header size ratio. CSLIP manipulates header information just before a packet is sent and just after a packet is received. Only TCP/IP headers are compressed and uncompressed; other protocol types are sent and received normally. A functioning CSLIP driver is required on the peer (destination) end of the physical link in order to carry out a CSLIP "conversation."

Multiple units are supported by this driver. Each individual unit may have CSLIP support disabled or enabled, independent of the state of other units.
BOARD LAYOUT
No hardware is directly associated with this driver; therefore, a jumpering diagram is not applicable.

SEE ALSO

ACKNOWLEDGEMENT
This program is based on original work done by Rick Adams of The Center for Seismic Studies and Chris Torek of The University of Maryland. The CSLIP enhancements are based on work done by Van Jacobson of University of California, Berkeley for the "cslip-2.7" release.

if_sm

NAME
if_sm – shared memory backplane network interface driver

DESCRIPTION
This module implements the VxWorks shared memory backplane network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of hosts 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 parameters are detailed below.

There are no user-callable routines.

This driver is layered between the shared memory packet library and the network modules. The backplane driver gives CPUs residing on a common backplane the ability to communicate using IP (via shared memory).

BOARD LAYOUT
This device is "software only." There is no jumpering diagram required.

TARGET-SPECIFIC PARAMETERS
A set of target-specific parameters is used to configure shared memory and backplane networking.

local address of anchor
This parameter is the local address by which the local CPU accesses the shared memory anchor.

maximum number of input packets
This parameter specifies the maximum number of incoming shared memory packets that can be queued to this CPU at one time.
method of notification
These four parameters are used to enable a CPU to announce the method by which it is to be notified of input packets that have been queued to it.

heartbeat frequency
This parameter specifies the frequency of the shared memory backplane network’s heartbeat, which is expressed in terms of the number of CPU ticks on the local CPU corresponding to one heartbeat period.

number of buffers to loan
This parameter, when non-zero, specifies the number of shared memory packets available to be loaned out.

master CPU number
This parameter specifies the master CPU number as set during system configuration.

For detailed information refer to *VxWorks Network Programmer’s Guide: Data Link Layer Network Components*.

INCLUDE FILES
smNetLib.h

SEE ALSO
ifLib, smNetLib, *VxWorks Network Programmer’s Guide*

---

**if_sn**

**NAME**
if_sn – National Semiconductor DP83932B SONIC Ethernet network driver

**ROUTINES**
snattach() – publish the sn network interface and initialize the driver and device

**DESCRIPTION**
This module implements the National Semiconductor DP83932 SONIC Ethernet 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 parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it. This driver supports up to four individual units per CPU.

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

**EXTERNAL INTERFACE**
This driver provides the standard external interface with the following exceptions. All
initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is `NULL`.

There is one user-callable routine, `snattach()`; for details, see the manual entry for this routine.

**TARGET-SPECIFIC PARAMETERS**

- **device I/O address**
  This parameter is passed to the driver by `snattach()`. It specifies the base address of the device’s I/O register set.

- **interrupt vector**
  This parameter is passed to the driver by `snattach()`. It specifies the interrupt vector to be used by the driver to service an interrupt from the SONIC device. The driver will connect the interrupt handler to this vector by calling `intConnect()`.

- **Ethernet address**
  This parameter is obtained by calling an external support routine. It specifies the unique, six-byte address assigned to the VxWorks target on the Ethernet.

**EXTERNAL SUPPORT REQUIREMENTS**

This driver requires five external support functions:

```c
void sysEnetInit (int unit)  
This routine performs any target-specific operations that must be executed before the \nSONIC device is initialized. The driver calls this routine, once per unit, from \n`snattach()`.

STATUS sysEnetAddrGet (int unit, char *pCopy)
This routine provides the six-byte Ethernet address used by `unit`. It must copy the \nsix-byte address to the space provided by `pCopy`. This routine returns `OK`, or `ERROR` if \nit fails. The driver calls this routine, once per unit, from `snattach()`.

void sysEnetIntEnable (int unit), void sysEnetIntDisable (int unit)
These routines enable or disable the interrupt from the SONIC device for the specified `unit`. Typically, this involves interrupt controller hardware, either internal or \nexternal to the CPU. The driver calls these routines only during initialization, from \n`snattach()`.

void sysEnetIntAck (int unit)
This routine performs any interrupt acknowledgement or clearing that may be \nrequired. This typically involves an operation to some interrupt control hardware. \nThe driver calls this routine from the interrupt handler.
```

**DEVICE CONFIGURATION**

Two global variables, `snDcr` and `snDcr2`, are used to set the SONIC device configuration registers. By default, the device is programmed in 32-bit mode with zero wait states. If
these values are not suitable, the snDcr and snDcr2 variables should be modified before calling snattach(). See the SONIC manual to change these parameters.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

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

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

This driver uses cacheDmaMalloc() to allocate the memory to be shared with the SONIC device. The size requested is 117,188 bytes.

The SONIC device can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data that is written by the driver because fields within the shared structures are asynchronously modified by the driver and the device, and these fields may share the same cache line.

NOTE 1

The previous transmit descriptor does not exist until the transmitter has been asked to send at least one packet. Unfortunately the test for this condition must be done every time a new descriptor is to be added, even though the condition is only true the first time. However, it is a valuable test, since we should not use the fragment count field as an index if it is 0.

NOTE 2

The following features are not supported in this version:

- buffer loaning on receive
- output hooks
- trailer protocol
- promiscuous mode

Also, the receive setup needs work so that the number of RRA descriptors is not fixed at four. It would be a nice addition to allow all the sizes of the shared memory structures to be specified by the runtime functions that call our init routines.

SEE ALSO

ifLib
## if_ultra

**NAME**  
if_ultra – SMC Elite Ultra Ethernet network interface driver

**ROUTINES**  
* ultraattach( ) – publish ultra interface and initialize device  
* ultraPut( ) – copy a packet to the interface.  
* ultraShow( ) – display statistics for the ultra network interface

**DESCRIPTION**  
This module implements the SMC Elite Ultra Ethernet network interface driver.  
This driver supports single transmission and multiple reception. The Current register is a write pointer to the ring. The Bound register is a read pointer from the ring. This driver gets the Current register at the interrupt level and sets the Bound register at the task level. The interrupt is never masked at the task level.

**CONFIGURATION**  
The W1 jumper should be set in the position of “Software Configuration”. The defined I/O address in config.h must match the one stored in EEROM. The RAM address, the RAM size, and the IRQ level are defined in config.h. IRQ levels 2,3,5,7,10,11,15 are supported.

**EXTERNAL INTERFACE**  
The only user-callable routines are ultraattach( ) and ultraShow( ):  

```c
ultraattach( )  
    Publishes the ultra interface and initializes the driver and device.

ultraShow( )  
    Displays statistics that are collected in the interrupt handler.
```
NAME

iOlicomEnd  – END style Intel Olicom PCMCIA network interface driver

ROUTINES

iOlicomEndLoad() – initialize the driver and device
iOlicomIntHandle() – interrupt service for card interrupts

DESCRIPTION

This module implements the Olicom (Intel 82595TX) network interface driver. The physical device is a PCMCIA card. This driver also houses code to manage a Vadem PCMCIA Interface controller on the ARM PID board, which is strictly a subsystem in it’s own right.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance.

BOARD LAYOUT

The device resides on a PCMCIA card and is soft configured. No jumpering diagram is necessary.

EXTERNAL INTERFACE

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

The string contains the target specific parameters like this:

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

TARGET-SPECIFIC PARAMETERS

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

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

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

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

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

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

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

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

number of transmit and receive buffer descriptors
These are the twelfth and thirteenth parameters passed to the driver init string.

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

offset
This is the fourteenth parameter passed to the driver in the init string.
This parameter defines the offset which is used to solve alignment problem.

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

This is the sixteenth parameter passed to the driver in the init string.
The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter.

Ethernet address

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

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support function:

```c
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.
```

```c
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.
```

```c
void sysBusIntAck(void)
This routine acknowledge the interrupt if it's necessary.
```

SEE ALSO

iPIIX4

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

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

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

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

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

**User Interface**

 STATUS iPIIX4Init ()
This routine locates and initializes the PIIX4.
STATUS iPIIX4KbdInit ()
This routine does keyboard specific initialization on PIIX4.

STATUS iPIIX4FdInit ()
This routine does floppy disk specific initialization on PIIX4.

STATUS iPIIX4AtaInit ()
This routine does ATA device specific initialization on PIIX4.

STATUS iPIIX4IntrRoute
{
    int pintx, char irq
}
This routine 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 returns the interrupt level of a PCI interrupt previously set by iPIIX4IntrRoute.

INCLUDE FILES iPIIX4.h
NAME

In97xEnd – END style AMD Am79C97X PCnet-PCI Ethernet driver

ROUTINES

In97xEndLoad() – initialize the driver and device
In97xInitParse() – parse the initialization string

DESCRIPTION

This module implements the Advanced Micro Devices Am79C970A, Am79C971, Am79C972, and Am79C973 PCnet-PCI Ethernet 32-bit network interface driver.

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

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

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

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

BOARD LAYOUT

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

EXTERNAL INTERFACE

The driver provides one standard external interface, In97xEndLoad(). As input, this routine takes a string of colon-separated parameters. The parameters should be specified in hexadecimal (optionally preceded by 0x or a minus sign -). The parameter string is parsed using strtok_r().

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is:

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

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

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

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

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

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

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

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

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

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

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

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

csr3b
The PCnet-PCI Control and Status Register 3 (CSR3) controls, among other things, big-endian and little-endian modes of operation. When big-endian mode is selected, the PCnet-PCI controller will swap the order of bytes on the AD bus during a data phase on access to the FIFOs only: AD[31:24] is byte 0, AD[23:16] is byte 1, AD[15:8] is byte 2 and AD[7:0] is byte 3. In order to select the big-endian mode, set this parameter to (0x0004). Most implementations, including natively big-endian host architectures, should set this parameter to (0x0000) in order to select little-endian access to the FIFOs, as the driver is currently designed to perform byte swapping as appropriate to the host architecture.

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

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

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

**ln97xIntConnect**

```c
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**

```c
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**

```c
IMPORT STATUS (* ln97xIntEnable)
{
    int level /* interrupt level to be enabled */
};
/* default setting */
ln97xIntEnable = sysLan97xIntEnable;
```

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

In97xIntDisable

```c
IMPORT STATUS (* ln97xIntDisable)

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

In97xEnetAddrGet

```c
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.

**SEE ALSO**

**NAME**

**In7990End** – END style AMD 7990 LANCE Ethernet network interface driver

**ROUTINES**

**In7990EndLoad()** – initialize the driver and device

**DESCRIPTION**

This module implements the Advanced Micro Devices Am7990 LANCE Ethernet network interface driver. The driver can be configured to support big-endian or little-endian architectures, and it contains error recovery code to handle known device errata related to DMA activity.

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. If any of the assumptions stated below are not true for your particular hardware, this driver might not function correctly with that hardware.

**BOARD LAYOUT**

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

**EXTERNAL INTERFACE**

The only external interface is the **In7990EndLoad()** routine, which expects the **initString** parameter as input. This parameter passes in a colon-delimited string of the format:

```
```

The **In7990EndLoad()** 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.

**CSR_register_addr**

Tells the driver where to find the CSR register.

**RAP_register_addr**

Tells the driver where to find the RAP register.

**int_vector**

Configures the LANCE 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 LANCE interrupt.
int_level
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 LANCE 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.

shmem_addr
The LANCE 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 LANCE. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

This parameter can be used to specify an explicit memory region for use by the LANCE. This should be done on hardware that restricts the LANCE to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

shmem_size
Use this parameter to explicitly limit the amount of shared memory (bytes) that this driver uses. Use "NONE" to indicate that there is no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

shmem_width
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 such targets, performing an access of an invalid width causes a bus error. Use this parameter to specify the number of bytes on which data must be aligned if it is to be used by the driver during access to the shared memory. Use "NONE" to indicate that there are no restrictions.

The support for this mechanism is not robust. Thus, its current implementation might not work on all targets requiring these restrictions.

offset
Specifies the memory alignment offset.

csr3B
Specifies the value that is placed into LANCE control register #3. This value determines the bus mode of the device and thus allows the support of big-endian and little-endian architectures. The default value supports Motorola-type buses. Normally this value is 0x4. For SPARC CPUs, it is normally set to 0x7 to add the ACON and BCON control bits. For more information on this register and the bus mode of the LANCE controller, see Advanced Micro Devices Local Area Network Controller Am7990 (LANCE).
EXTERNAL SUPPORT REQUIREMENTS

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

\[
\begin{align*}
&\text{SYS\_INT\_CONNECT(pDrvCtrl, routine, arg)} \\
&\text{SYS\_INT\_DISCONNECT(pDrvCtrl, routine, arg)} \\
&\text{SYS\_INT\_ENABLE(pDrvCtrl)} \\
&\text{SYS\_OUT\_SHORT(pDrvCtrl, reg, data)} \\
&\text{SYS\_IN\_SHORT(pDrvCtrl, reg, pData)}
\end{align*}
\]

There are default values in the source code for these macros. They presume memory-mapped accesses to the device registers and the normal \text{intConnect( )}, and \text{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.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one interrupt vector
- 68 bytes in the initialized data section (data) /@HELP@/  
- 0 bytes of bss /@HELP@/

The above data and BSS requirements are for the MC68020 architecture and can vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

If the driver is not given a specific region of memory using the \text{In7990EndLoad( )} routine, then it calls \text{cacheDmaMalloc( )} to allocate the memory to be shared with the LANCE. The size requested is 80,542 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

The LANCE can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for data that is written by the driver. That is because members within the shared structures are asynchronously modified by both the driver and the device, and these members might share the same cache line.

SEE ALSO \text{muxLib, Advanced Micro Devices Local Area Network Controller Am7990 (LANC)}
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:

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

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

SEE ALSO
VxWorks Programmer's Guide: I/O System
m68302Sio

NAME m68302Sio – Motorola MC68302 bimodal tty driver

ROUTINES
m68302SioInit() – initialize a M68302_CP
m68302SioInit2() – initialize a M68302_CP (part 2)

DESCRIPTION
This is the driver for the internal communications processor (CP) of the Motorola MC68302.

USER-CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O system. Before the driver can be used, it must be initialized by calling the routines m68302SioInit() and m68302SioInit2(). Normally, they are called by sysSerialHwInit() and sysSerialHwInit2() in sysSerial.c

This driver uses 408 bytes of buffer space as follows:
128 bytes for portA tx buffer
128 bytes for portB tx buffer
128 bytes for portC tx buffer
8 bytes for portA rx buffers (8 buffers, 1 byte each)
8 bytes for portB rx buffers (8 buffers, 1 byte each)
8 bytes for portC rx buffers (8 buffers, 1 byte each)

The buffer pointer in the m68302cp structure points to the buffer area, which is usually specified as IMP_BASE_ADDR.

IOCTL FUNCTIONS
This driver responds to the same ioctl() codes as a normal tty driver; for more information, see the manual entry for tyLib. The available baud rates are 300, 600, 1200, 2400, 4800, 9600 and 19200.

SEE ALSO
ttyDrv, tyLib

INCLUDE FILES
drv/sio/m68302Sio.h, sioLib.h
m68332Sio

NAME m68332Sio – Motorola MC68332 tty driver

ROUTINES m68332DevInit() – initialize the SCC
m68332Int() – handle an SCC interrupt

DESCRIPTION This is the driver for the Motorola MC68332 on-chip UART. It has only one serial channel.

USAGE A M68332_CHAN structure is used to describe the chip. The BSP’s sysHwInit() routine typically calls sysSerialHwInit(), which initializes all the values in the M68332_CHAN structure (except the SIO_DRV_FUNCS) before calling m68332DevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2(), which connects the chips interrupt (m68332Int) via intConnect().

INCLUDE FILES drv/sio/m68332Sio.h

m68360Sio

NAME m68360Sio – Motorola MC68360 SCC UART serial driver

ROUTINES m68360DevInit() – initialize the SCC
m68360Int() – handle an SCC interrupt

DESCRIPTION This is the driver for the SCCs in the internal Communications Processor (CP) of the Motorola MC68360. This driver only supports the SCCs in asynchronous UART mode.

USAGE A M68360_CHAN structure is used to describe the chip. The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the M68360_CHAN structure (except the SIO_DRV_FUNCS) before calling m68360DevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupt (m68360Int) via intConnect().

INCLUDE FILES drv/sio/m68360Sio.h
m68562Sio

NAME  m68562Sio – MC68562 DUSCC serial driver

ROUTINES
- m68562HrdInit() – initialize the DUSCC
- m68562RxTxErrInt() – handle a receiver/transmitter error interrupt
- m68562RxInt() – handle a receiver interrupt
- m68562TxInt() – handle a transmitter interrupt

DESCRIPTION  This is the driver for the MC68562 DUSCC serial chip. It uses the DUSCC in asynchronous mode only.

USAGE  A M68562_QUSART structure is used to describe the chip. This data structure contains M68562_CHAN structures which describe the chip’s serial channels. The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the M68562_QUSART structure (except the SIO_DRV_FUNCS) before calling m68562HrdInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupts (m68562RxTxErrInt, m68562RxInt, and m68562TxInt) via intConnect().

IOCTL  This driver responds to the same ioctl() codes as a normal serial driver. See the file sioLib.h for more information.

INCLUDE FILES  drv/sio/m68562Sio.h

m68681Sio

NAME  m68681Sio – M68681 serial communications driver

ROUTINES
- m68681DevInit() – initialize a M68681_DUART
- m68681DevInit2() – initialize a M68681_DUART, part 2
- m68681ImrSetClr() – set and clear bits in the DUART interrupt-mask register
- m68681Imr() – return the current contents of the DUART interrupt-mask register
- m68681AcrSetClr() – set and clear bits in the DUART auxiliary control register
- m68681Acr() – return the contents of the DUART auxiliary control register
- m68681OprSetClr() – set and clear bits in the DUART output port register
- m68681Opr() – return the current state of the DUART output port register
- m68681OpcrSetClr() – set and clear bits in the DUART output port configuration register
- m68681Opcr() – return the state of the DUART output port configuration register
- m68681Int() – handle all DUART interrupts in one vector
DESCRIPTION
This is the driver for the M68681 DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a counter/timer device. This driver module provides control of the two serial channels and the baud-rate generator. The counter timer is controlled by a separate driver, src/drv/timer/m68681Timer.c.

A M68681_DUART structure is used to describe the chip. This data structure contains two M68681_CHAN structures which describe the chip’s two serial channels. The M68681_DUART structure is defined in m68681Sio.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. These default settings can be overridden on a channel-by-channel basis by setting the M68681_CHAN options and baudRate fields to the desired values before calling m68681DevInit(). See sioLib.h for option values. The defaults for the module can be changed by redefining the macros M68681_DEFAULT_OPTIONS and M68681_DEFAULT_BAUD and recompiling this driver.

This driver supports baud rates of 75, 110, 134.5, 150, 300, 600, 1200, 2000, 2400, 4800, 1800, 9600, 19200, and 38400.

USAGE
The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the hardware addresses in the M68681_DUART structure before calling m68681DevInit(). This enables the chip to operate in polled mode, but not in interrupt mode. Calling m68681DevInit2() from the sysSerialHwInit2() routine allows interrupts to be enabled and interrupt-mode operation to be used.

The following example shows the first part of the initialization through calling m68681DevInit():

```
#include "drv/sio/m68681Sio.h"
M68681_DUART myDuart;   /* my device structure */
#define MY_VEC  (71)    /* use single vector, #71 */
sysSerialHwInit()
{
    /* initialize the register pointers for portA */
    myDuart.portA.mr    = M68681_MRA;
    myDuart.portA.sr    = M68681_SRA;
    myDuart.portA.csr   = M68681_CSRA;
    myDuart.portA.cr    = M68681 CRA;
    myDuart.portA.rb    = M68681_RHRA;
    myDuart.portA.tb    = M68681_THRA;
    /* initialize the register pointers for portB */
    myDuart.portB.mr = M68681_MRB;
    ...
    /* initialize the register pointers/data for main duart */
    myDuart.ivr         = MY_VEC;
    myDuart.ipcr        = M68681_IPCR;
```
myDuart.acr = M68681_ACR;
myDuart.isr = M68681_ISR;
myDuart.imr = M68681_IMR;
myDuart.ip = M68681_IP;
myDuart.opcr = M68681_OPCR;
myDuart.sopbc = M68681_SOPBC;
myDuart.ropbc = M68681_ROPBC;
myDuart.ctroff = M68681_CTROFF;
myDuart.ctron = M68681_CTRON;
myDuart.ctlr = M68681_CTLR;
myDuart.ctur = M68681_CTUR;
m68681DevInit (&myDuart);
}

The BSP's `sysHwInit2()` routine typically calls `sysSerialHwInit2()` which connects the chips interrupts via `intConnect()` to the single interrupt handler `m68681Int()`. After the interrupt service routines are connected, the user then calls `m68681DevInit2()` to allow the driver to turn on interrupt enable bits, as shown in the following example:

```c
sysSerialHwInit2 ()
{
    /* connect single vector for 68681 */
    intConnect (INUM_TO_IVEC(MY_VEC), m68681Int, (int)&myDuart);
    ...  
    /* allow interrupts to be enabled */
    m68681DevInit2 (&myDuart);
}
```

**SPECIAL CONSIDERATIONS**

The CLOCAL hardware option presumes that OP0 and OP1 output bits are wired to the CTS outputs for channel 0 and channel 1 respectively. If not wired correctly, then the user must not select the CLOCAL option. CLOCAL is not one of the default options for this reason.

This driver does not manipulate the output port or its configuration register in any way. If the user selects the CLOCAL option, then the output port bit must be wired correctly or the hardware flow control will not function correctly.

**INCLUDE FILES**

drv/sio/m68681Sio.h
m68901Sio

NAME  m68901Sio – MC68901 MFP tty driver

ROUTINES  m68901DevInit() – initialize a M68901_CHAN structure

DESCRIPTION  This is the SIO driver for the Motorola MC68901 Multi-Function Peripheral (MFP) chip.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, one routine must be called directly: m68901DevInit() initializes the driver. Normally, it is called by sysSerialHwInit() in sysSerial.c

IOCTL FUNCTIONS

This driver responds to the same ioctl() codes as other tty drivers; for more information, see the manual entry for tyLib.

SEE ALSO  tyLib

mb86940Sio

NAME  mb86940Sio – MB 86940 UART tty driver

ROUTINES  mb86940DevInit() – install the driver function table

DESCRIPTION  This is the driver for the SPARClite MB86930 on-board serial ports.

USAGE

A MB86940_CHAN structure is used to describe the chip.

The BSP’s sysHwInit() routine typically calls sysSerialHwInit(), which initializes all the values in the MB86940_CHAN structure (except the SIO_DRV_FUNCS) before calling mb86940DevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2(), which connects the chips interrupts via intConnect().

IOCTL FUNCTIONS

The UARTs use timer 3 output to generate the following baud rates: 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.

NOTE: The UARTs will operate at the same baud rate.

INCLUDE FILES  drv/sio/mb86940Sio.h
mb86960End

NAME

mb86960End – END-style Fujitsu MB86960 Ethernet network interface driver

ROUTINES

mb86960EndLoad() – initialize the driver and device
mb86960InitParse() – parse the initialization string
mb86960MemInit() – initialize memory for the chip

DESCRIPTION

This module implements the Fujitsu MB86960 NICE Ethernet network interface driver. This driver is non-generic and has only been run on the Fujitsu SPARClite Evaluation Board. It currently supports only unit number zero. The 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.

The MB86960 Network Interface Controller with Encoder/Decoder (NICE) chip is a highly integrated monolithic device which incorporates both network controller, complete with buffer management and Manchester encoder/decoder.

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is:

unit:devBaseAddr:ivec

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

devBaseAddr
The base Address of the chip registers.

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 to this interrupt.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

sys86960IntEnable()

void sysEnetIntEnable (int unit)

This routine provides a target-specific interface to enable Ethernet device interrupts for a given device unit. For this driver, value of unit must be 0.
sys86960IntDisable()

```c
void sysEnetIntDisable (int unit)
```

This routine provides a target-specific interface to disable Ethernet device interrupts for a
given device unit. For this driver, value of unit must be 0.

sysEnetAddrGet()

```c
STATUS sysEnetAddrGet (int unit, char *enetAdrs)
```

This routine provides a target-specific interface to access a device Ethernet address. This
routine should provide a six-byte Ethernet address in the *enetAdrs* parameter and return
OK or ERROR.

In this driver the macros SYS_OUT_SHORT and SYS_IN_SHORT which call bsp-specific
functions to access the chip register.

**Includes**

end.h, endLib.h, etherMultiLib.h

**See also**

muxLib, endLib, Writing and Enhanced Network Driver

---

**mb87030Lib**

**Name**

mb87030Lib – Fujitsu MB87030 SCSI Protocol Controller (SPC) library

**Routines**

- mb87030CtrlCreate() – create a control structure for an MB87030 SPC
- mb87030CtrlInit() – initialize a control structure for an MB87030 SPC
- mb87030Show() – display the values of all readable MB87030 SPC registers

**Description**

This is the I/O driver for the Fujitsu MB87030 SCSI Protocol Controller (SPC) chip. It is
designed to work in conjunction with scsiLib.

**User-callable routines**

Most of the routines in this driver are accessible only through the I/O system. Two
routines, however, must be called directly: mb87030CtrlCreate() to create a controller
structure, and mb87030CtrlInit() to initialize the controller structure.

**Include Files**

mb87030.h

**See also**

scsiLib, Fujitsu Small Computer Systems Interface MB87030 Synchronous/Asynchronous
mbcEnd

NAME  
mbcEnd – Motorola 68302fads END network interface driver

ROUTINES  
mbcEndLoad() – initialize the driver and device  
mbcParse() – parse the init string  
mbcMemInit() – initialize memory for the chip  
mbcAddrFilterSet() – set the address filter for multicast addresses

DESCRIPTION  
This is a driver for the Ethernet controller on the 68EN302 chip. The device supports a 16-bit interface, data rates up to 10 Mbps, a dual-ported RAM, and transparent DMA. The dual-ported RAM is used for a 64-entry CAM table, and a 128-entry buffer descriptor table. The CAM table is used to set the Ethernet address of the Ethernet device or to program multicast addresses. The buffer descriptor table is partitioned into fixed-size transmit and receive tables. The DMA operation is transparent and transfers data between the internal FIFOs and external buffers pointed to by the receive and transmit-buffer descriptors during transmits and receives.

The driver requires that the memory used for transmit and receive buffers be allocated in cache-safe RAM area.

Up to 61 multicast addresses are supported. Multicast addresses are supported by adding the multicast ethernet addresses to the address table in the ethernet part. If more than 61 multicast addresses are desired, address hashing must be used (the address table holds 62 entries at most). However, address hashing does not appear to work in this ethernet part.

A glitch in the EN302 Rev 0.1 device causes the Ethernet transmitter to lock up from time to time. The driver uses a watchdog timer to reset the Ethernet device when the device runs out of transmit buffers and cannot recover within 20 clock ticks.

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

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

TARGET-SPECIFIC PARAMETERS  
unit  
A convenient holdover from the former model. This parameter is used only in the string name for the driver.

memAddr  
This parameter is the base address of the Ethernet module. The driver addresses all
other Ethernet device registers as offsets from this address.

**ivec**
- The interrupt vector to be used in connecting the interrupt handler.

**txBdNum**
- The number of transmit buffer descriptors to use.

**rxBdNum**
- The number of receive buffer descriptors to use.

The number of transmit and receive buffer descriptors (BDs) used is configurable by the user while attaching the driver. Each BD is 8 bytes in size and resides in the chip’s dual-ported memory, while its associated buffer, 1520 bytes in size, resides in cache-safe conventional RAM. A minimum of 2 receive and 2 transmit BDs should be allocated. If this parameter is 0, a default of 32 BDs will be used. The maximum number of BDs depends on how the dual-ported BD RAM is partitioned. The 128 BDs in the dual-ported BD RAM can partitioned into transmit and receive BD regions with 8, 16, 32, or 64 transmit BDs and corresponding 120, 112, 96, or 64 receive BDs.

**dmaParms**
- Ethernet DMA parameters.

This parameter is used to specify the settings of burst limit, water-mark, and transmit early, which control the Ethernet DMA, and is used to set the EDMA register.

**bufBase**
- Base address of the buffer pool.

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 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 will be performed. Each buffer is 1520 bytes. If this parameter is “NULL”, space for buffers will be obtained by calling `cacheDmaMalloc()` in `mbcMemInit()`.

**offset**
- Specifies the memory alignment offset.

**EXTERNAL SUPPORT REQUIREMENTS**

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

```c
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_OUT_SHORT(pDrvCtrl, reg, data)
SYS_IN_SHORT(pDrvCtrl, reg, pData)
```

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
**Driver Libraries**

`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.

**SYSTEM RESOURCE USAGE**

The driver requires the following system resources:

- one watchdog timer
- one interrupt vector
- 52 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (bss)

The above data and BSS requirements are for the MC68000 architecture and can vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

If the driver allocates the memory shared with the Ethernet device unit, it does so by calling the `cacheDmaMalloc()` routine. For the default case of 32 transmit buffers, 32 receive buffers, the total size requested is roughly 100,000 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

This driver can only operate if the shared 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 may share the same cache line. Additionally, the chip’s dual-ported RAM must be declared as non-cacheable memory where applicable.

**INCLUDES**

`end.h`, `endLib.h`, `etherMultiLib.h`

**SEE ALSO**

`muxLib`, `endLib`, *Writing and Enhanced Network Driver*
miiLib

NAME

miiLib – Media Independent Interface library

ROUTINES

miiPhyInit() – initialize and configure the PHY devices
miiPhyUnInit() – uninitialized a PHY
miiAnCheck() – check the auto-negotiation process result
miiPhyOptFuncMultiSet() – set pointers to MII optional registers handlers
miiPhyOptFuncSet() – set the pointer to the MII optional registers handler
miiLibInit() – initialize the MII library
miiLibUnInit() – uninitialized the MII library
miiShow() – show routine for MII library
miiRegsGet() – get the contents of MII registers

DESCRIPTION

This module implements a Media Independent Interface (MII) library.

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

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

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

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

miiPhyUnInit(), miiLibInit(), miiLibUnInit(), miiPhyOptFuncSet()

STATUS miiLibInit (void);
STATUS miiLibUnInit (void);
EXTERNAL SUPPORT REQUIREMENTS

**phyReadRtn()**

```c
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()**

```c
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()**

```c
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()**

```c
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`

**SEE ALSO**

IEEE 802.3.2000 Standard
motCpmEnd

NAME

motCpmEnd – END style Motorola MC68EN360/MPC800 network interface driver

ROUTINES

motCpmEndLoad() – initialize the driver and device

DESCRIPTION

This module implements the Motorola MC68EN360 QUICC as well as the MPC821 and MPC860 Power-QUICC Ethernet Enhanced network interface driver.

All the above mentioned microprocessors feature a number of Serial Communication Controllers (SCC) that support different serial protocols including IEEE 802.3 and Ethernet CSMA-CD. As a result, when the Ethernet mode of a SCC is selected, by properly programming its general Mode Register (GSMR), they can implement the full set of media access control and channel interface functions those protocol require. However, while the MC68EN360 QUICC and the MPC860 Power-QUICC support up to four SCCs per unit, the MPC821 only includes two on-chip SCCs.

This driver is designed to support the Ethernet mode of a SCC residing on the CPM processor core, no matter which among the MC68EN360 QUICC or any of the PPC800 Series. In fact, the major differences among these processors, as far as the driver is concerned, are to be found in the mapping of the internal Dual-Port RAM. The driver is generic in the sense that it does not care which SCC is being used. In addition, it poses no constraint on the number of individual units that may be used per board. However, this number should be specified in the bsp through the macro MAX_SCC_CHANNELS. The default value for this macro in the driver is 4.

To achieve these goals, the driver requires several target-specific values provided as an input string to the load routine. It also requires some external support routines. These target-specific values and the external support routines are described below.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance.

This driver maintains cache coherency by allocating buffer space using the cacheDmaMalloc() routine. This is provided for boards whose host processor use data cache space, e.g. the MPC800 Series. Although the MC68EN360 does not have cache memory, it may be used in a particular configuration: MC68EN360 in 040 companion mode where that is attached to processors that may cache memory. However, due to a lack of suitable hardware, the multiple unit support and '040 companion mode support have not been tested.

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 \texttt{motCpmEndLoad()} routine. The parameters are passed into the \texttt{motCpmEndLoad()} function as a single colon-delimited string. The \texttt{motCpmEndLoad()} function uses \texttt{strtok()} to parse the string, which it expects to be of the following format:


\textbf{TARGET-SPECIFIC PARAMETERS}

\textit{unit}

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

\textit{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.

\textit{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 \texttt{intConnect( )}.

\textit{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”).

\textit{txBdNum} and \textit{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 “\texttt{NULL}”, a default value of “32” BDs is used.

\textit{txBdBase} and \textit{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 motCpmEndLoad().

EXTERNAL SUPPORT REQUIREMENTS
This driver requires three external support functions:

sysXxxEnetEnable()
This is either sys360EnetEnable() or sysCpmEnetEnable(), 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 sysCpmEnetDisable(), 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 sysCpmEnetAddrGet(), 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 motCpmEndLoad() routine.

For the CPU32, the prototypes of the above support routines are as follows:

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

For the PPC860, the prototypes of the above support routines are as follows:

```
STATUS sysCpmEnetEnable (int unit)
void sysCpmEnetDisable (int unit)
STATUS sysCpmEnetAddrGet (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.

**motFccEnd**

**NAME**

`motFccEnd` – END style Motorola FCC Ethernet network interface driver

**ROUTINES**

`motFccEndLoad()` – initialize the driver and device

**DESCRIPTION**

This module implements a Motorola Fast Communication Controller (FCC) Ethernet network interface driver. The FCC supports several communication protocols, and when programmed to operate in Ethernet mode, it 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, the types of events that should generate an interrupt, or features like the promiscuous mode or the heartbeat control 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 stored in the internal parameter RAM. The latter also includes protocol-specific parameters, like the individual physical address of this station or the max 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 incoming or outgoing data buffers. These are located again in external memory, and the user may choose whether this is 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 must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance. In addition, no copy is performed of the outgoing packet before it is sent.

**BOARD LAYOUT**

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

**EXTERNAL INTERFACE**

The driver provides the standard external interface, `motFccEndLoad()`, 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:

"immrVal:fccNum:bdBase:bdSize:bufBase:bufSize:fifoTxBase:fifoRxBase:

**TARGET-SPECIFIC PARAMETERS**

`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` (see below), the driver is able to compute the location of the FCC parameter RAM, and, ultimately, to program the FCC for proper operations.

`fccNum`

This driver is written to support multiple individual device units. This parameter is
used to explicitly state which FCC is being used (on the vads8260 board, FCC2 is wired to the Fast Ethernet transceiver, thus this parameter equals "2").

**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 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. Therefore, the given memory space should allow for all the buffer descriptors and the 8-byte alignment factor.

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

**bdSize**

The memory size 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 the size of the provided memory region is adequate with respect to the given number of Transmit Buffer Descriptors and Receive Buffer Descriptors.

**bufBase**

This parameter tells the driver that space for data buffers 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 must be 32-byte aligned and non-cacheable. The FCC poses one more constraint in that DMA cycles may initiate even when all the incoming data have already been transferred to memory. This means that 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 `motFccEndLoad()`.

**bufSize**

The memory size 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 the size of the provided memory region is adequate with respect to the given number of Receive Buffer Descriptors and a non-user-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, as 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, he may set this parameter 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, as 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, he may set this parameter 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 the macro MOT_FCC_TBD_MIN, or if it is "NULL", a default value of 64 (see MOT_FCC_TBD_DEF_NUM) is used. This 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 1584-byte buffer again in external RAM. If this parameter is less than a minimum number specified in the macro MOT_FCC_RBD_MIN, or if it is "NULL", a default value of 32 (see MOT_FCC_RBD_DEF_NUM) 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. The default physical layer initialization routine will scan 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 will find out the PHY actual address by scanning the whole range. The one with the lowest address will be chosen.

**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.
1: Driver Libraries

**motFccEnd**

---

### pAnOrderTbl
This parameter may be set to the address of a table that specifies the order how different subsets of technology abilities should be advertised by the auto-negotiation process, if enabled. 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 behavior 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.

### mblkMult
Ratio between mBlk's and Rx BD's

### clMult
Ratio between Clusters and Rx BD's

### txJobMsgQLen
Length of the message queue from the ISR to motFccJobQueue()

---

**MOT_FCC_USR_DPRAM_ALOC**
This option allows multiple FCCs operating in the same system to share the Dual Ported RAM. It enables Dual Ported RAM allocation and freeing using the utilities `m82xxDpramFccMalloc`, `m82xxDpramFccFree`, and `m82xxDpramFccFree` via the function pointers `_func_m82xxDpramFccMalloc`, `_func_m82xxDpramFccMalloc`, and `_func_m82xxDpramFccFree` which must be loaded by the BSP if this option is used.

**MOT_FCC_USR_PHY_NO_AN**
The default physical layer initialization routine will exploit 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 will take part to the negotiation process, and the highest priority common denominator technology ability will be chosen. If the user wishes to prevent auto-negotiation from occurring, he may set this bit in the user flags.

**MOT_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 behavior 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 will look 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 will be 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 will ignore the collision detect and carrier sense signals. If the user wishes
not to negotiate full duplex mode, he should set the `MOT_FCC_USR_PHY_NO_FD` bit in the user flags.

**MOT_FCC_USR_PHY_NO_FD**

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_FCC_USR_PHY_NO_FD` bit in the user flags.

**MOT_FCC_USR_PHY_NO_HD**

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_FCC_USR_PHY_NO_HD` bit in the user flags.

**MOT_FCC_USR_PHY_NO_100**

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_FCC_USR_PHY_NO_100` bit in the user flags.

**MOT_FCC_USR_PHY_NO_10**

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 behavior on the medium layer. If the user wishes to electrically isolate all PHYs from the MII interface, he should set the `MOT_FCC_USR_PHY_ISO` bit. The default behavior is to not isolate any PHY on the board.

**MOT_FCC_USR_LOOP**

When the `MOT_FCC_USR_LOOP` bit is set, the driver will configure 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 will configure 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 will configure 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 will configure 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 will configure the FCC 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_motFccHbFail`, below). The user does not normally need to set this bit.
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.

sysFccMiiBitWr()  

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.

sysFccMiiBitRd()  

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.

```
_func_motFccPhyInit
  FUNCPTR _func_motFccPhyInit
  This driver sets the global variable _func_motFccPhyInit to the MII-compliant media initialization routine `miiPhyInit()`. 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_motFccHbFail
  FUNCPTR _func_motFccHbFail
  The FCC may be configured to perform heartbeat check following end of transmission, and to report any failure in the relevant TBD status field. If this is the case, and if the global variable _func_motFccHbFail is not NULL, the routine referenced to by _func_motFccHbFail 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_motFccHbFail is NULL.

_func_m82xxDpramFccMalloc
  FUNCPTR _func_m82xxDpramFccMalloc

_func_m82xxDpramFree
  FUNCPTR _func_m82xxDpramFree

_func_m82xxDpramFccFree
  FUNCPTR _func_m82xxDpramFccFree
  The FCC can be configured to utilize the dual ported ram located in the MPPC8260 CMP. In this case the user flag MOT_FCC_USR_DPRAM_ALOC is set and the global variables _func_m82xxDpramFccMalloc, _func_m82xxDpramFree, and _func_m82xxDpramFccFree must be populated by the BSP with the FUNCPTRs to `m82xxDpramFccMalloc()`, `m82xxDpramFree()`, and `m82xxDpramFccFree()` (respectively) from `m82xxDpramLib.h`. These functions are then used by the motFccEnd driver to allocate and free memory in the dual ported ram. If any of these FUNCPTRs are left NULL the motFccPramInit() will return an ERROR and the motFccEnd driver will not initialize.

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. For the default case of 64 transmit buffers and 32 receive buffers, the total size requested is 776 bytes, and this includes the 8-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 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. For the default case of 7 transmit clusters and 32 receive clusters, the total size requested for this memory region is 112751 bytes, and this includes the 32-byte alignment and the 32-byte pad-out area per buffer 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.

**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.

**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 in the chip.
The FEC establishes a shared memory communication system with the CPU, which is divided into two parts: Control/Status Registers (CSR) and 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:

```c
strtol (parameter, NULL, 16)
```

The format of the parameter string is:


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.
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 an 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 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. 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 exploits the auto-negotiation mechanism as described in the IEEE Std 802.3, to bring a valid link-up. All the link partners on the media take part to the negotiation, and the highest-priority common denominator technology ability is chosen. If you wish to prevent auto-negotiation, 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 behavior 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 behavior 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 behavior 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_motFecHbFail
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 heartbeat 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.

SEE ALSO

`ifLib`, `MPC860T Fast Ethernet Controller (Supplement to the MPC860 User’s Manual)` Motorola `MPC860 User’s Manual`
n72001Sio

**NAME**

n72001Sio – NEC PD72001 MPSC (Multiprotocol Serial Communications Controller)

**ROUTINES**

- n72001DevInit() – initialize a N72001_MPSC
- n72001IntWr() – handle a transmitter interrupt
- n72001IntRd() – handle a receiver interrupt
- n72001Int() – interrupt level processing

**DESCRIPTION**

This is a driver for the NEC PD72001 MPSC (Multiprotocol Serial Communications Controller). It uses the MPSC in asynchronous mode only.

**USAGE**

A N72001_MPSC structure is used to describe the chip. This data structure contains two N72001_CHAN structures which describe the chip’s two serial channels. The BSP’s sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the N72001_MPSC structure (except the SIO_DRV_FUNCS) before calling n72001DevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupts via intConnect() (either the single interrupt n72001Int or the three interrupts n72001IntWr, n72001IntRd, and n72001IntEx).

**INCLUDE FILES**

drv/sio/n72001Sio.h

ncr710CommLib

**NAME**

ncr710CommLib – common library for ncr710Lib.c and ncr710Lib2.c

**ROUTINES**

- ncr710SingleStep() – perform a single-step
- ncr710StepEnable() – enable/disable script single-step

**DESCRIPTION**

Contains ncr710Lib and ncr710Lib2 common driver interfaces which can be called from user code.

**SEE ALSO**

ncr710Lib

NAME  ncr710Lib – NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-1)

ROUTINES  ncr710CtrlCreate( ) – create a control structure for an NCR 53C710 SIOP
ncr710CtrlInit( ) – initialize a control structure for an NCR 53C710 SIOP
ncr710SetHwRegister( ) – set hardware-dependent registers for the NCR 53C710 SIOP
ncr710Show( ) – display the values of all readable NCR 53C710 SIOP registers

DESCRIPTION  This is the I/O driver for the NCR 53C710 SCSI I/O Processor (SIOP). It is designed to work with scsiLib. It also runs in conjunction with a script program for the NCR 53C710 chip. This script uses the NCR 53C710 DMA function for data transfers. 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: ncr710CtrlCreate( ) to create a controller structure, and ncr710CtrlInit( ) to initialize it. The NCR 53C710 hardware registers need to be configured according to the hardware implementation. If the default configuration is not proper, the routine ncr710SetHwRegister( ) should be used to properly configure the registers.

INCLUDE FILES  ncr710.h, ncr710_1.h, ncr710Script.h, ncr710Script1.h


ncr710Lib2

NAME  ncr710Lib2 – NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-2)

ROUTINES  ncr710CtrlCreateScsi2( ) – create a control structure for the NCR 53C710 SIOP
ncr710CtrlInitScsi2( ) – initialize a control structure for the NCR 53C710 SIOP
ncr710SetHwRegisterScsi2( ) – set hardware-dependent registers for the NCR 53C710
ncr710ShowScsi2( ) – display the values of all readable NCR 53C710 SIOP registers

DESCRIPTION  This is the I/O driver for the NCR 53C710 SCSI I/O Processor (SIOP). It is designed to work with scsiLib. This driver runs in conjunction with a script program for the NCR 53C710 chip. The script uses the NCR 53C710 DMA function for data transfers. 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. `ncr710CtrlCreate( )` creates a controller structure and `ncr710CtrlInit( )` initializes it. The NCR 53C710 hardware registers need to be configured according to the hardware implementation. If the default configuration is not correct, the routine `ncr710SetHwRegister( )` must be used to properly configure the registers.

INCLUDE FILES

`ncr710.h`, `ncr710_2.h`, `ncr710Script.h`, `ncr710Script2.h`

SEE ALSO

`scsiLib`, `scsi2Lib`, `cacheLib`, VxWorks Programmer’s Guide: I/O System

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.

NAME
crc5390Lib – NCR5390 SCSI-Bus Interface Controller library (SBIC)

ROUTINES
ncr5390CtrlInit() – initialize the user-specified fields in an ASC structure
crc5390Show() – display the values of all readable NCR5390 chip registers

DESCRIPTION This library contains the main interface routines to the SCSI-Bus Interface Controllers (SBIC). These routines simply switch the calls to the SCSI-1 or SCSI-2 drivers, implemented in ncr5390Lib1.c or ncr5390Lib2.c as configured by the Board Support Package (BSP).

In order to configure the SCSI-1 driver, which depends upon scsi1Lib, the ncr5390CtrlCreate() routine, defined in ncr5390Lib1, must be invoked. Similarly ncr5390CtrlCreateScsi2(), defined in ncr5390Lib2 and dependent on scsi2Lib, must be called to configure and initialize the SCSI-2 driver.

INCLUDE FILES
ncr5390.h, ncr5390_1.h, ncr5390_2.h
ncr5390Lib1

NAME   ncr5390Lib1 – NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-1)

ROUTINES ncr5390CtrlCreate() – create a control structure for an NCR 53C90 ASC

DESCRIPTION This is the I/O driver for the NCR 53C90 Advanced SCSI Controller (ASC). It is designed to work in conjunction with scsiLib.

USER-CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is the ncr5390CtrlCreate() which creates a controller structure.

INCLUDE FILES ncr5390.h


ncr5390Lib2

NAME   ncr5390Lib2 – NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-2)

ROUTINES ncr5390CtrlCreateScsi2() – create a control structure for an NCR 53C90 ASC

DESCRIPTION This is the I/O driver for the NCR 53C90 Advanced SCSI Controller (ASC). It is designed to work in conjunction with scsiLib.

USER-CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is the ncr5390CtrlCreateScsi2() which creates a controller structure.

INCLUDE FILES ncr5390.h

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:


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)
- SYS_INT_ENABLE(pDrvCtrl)
- SYS_IN_CHAR(pDrvCtrl, reg, pData)
- SYS_OUT_CHAR(pDrvCtrl, reg, pData)
- SYS_IN_WORD_STRING(pDrvCtrl, reg, pData)
- SYS_OUT_WORD_STRING(pDrvCtrl, reg, pData)

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

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

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

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

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

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

INCLUDES end.h, endLib.h, etherMultiLib.h

SEE ALSO muxLib, endLib, 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.

**SEE ALSO**

VxWorks Programmer’s Guide: I/O System

---

**nicEvbEnd**

**NAME**

nicEvbEnd – National Semiconductor ST-NIC Chip network interface driver

**ROUTINES**

- nicEndLoad() – initialize the driver and device
- nicEvbInitParse() – parse the initialization string

**DESCRIPTION**

This module implements the National Semiconductor 83902A ST-NIC Ethernet network interface driver.

This driver is non-generic and is for use on the IBM EVB403 board. The driver must be given several target-specific parameters. 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 only external interface is the `nicEvbEndLoad()` routine, which expects the `initString` parameter as input. This parameter passes in a colon-delimited string of the format:

```
unit:nic_addr:int_vector:int_level
```

The `nicEvbEndLoad()` 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.

- **nic_addr**
  Base address for NIC chip

- **int_vector**
  Configures the NIC 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.

- **int_level**
  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 NIC 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.

- **device restart/reset delay**
  The global variable `nicRestartDelay` (UINT32), defined in this file, should be initialized in the BSP `sysHwInit()` routine. `nicRestartDelay` is used only with PowerPC platform and is equal to the number of time base increments which makes for 1.6 msec. This corresponds to the delay necessary to respect when restarting or resetting the device.

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)
```

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.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector

SEE ALSO muxLib

---

ns16550Sio

NAME

ns16550Sio – NS 16550 UART tty driver

ROUTINES

ns16550DevInit() – initialize 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).

The driver sets 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 to TRUE and remain set until a HUPCL is performed.

INCLUDE FILES

drv/sio/ns16552Sio.h
NAME
ns83902End – National Semiconductor DP83902A ST-NIC

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

DESCRIPTION
This module implements the National Semiconductor dp83902A ST-NIC Ethernet network interface driver.

This driver is moderately generic. The driver must be given several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

The driver supports big-endian or little-endian architectures.

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:dmPort: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

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

nvr4101DSIUSio

NAME
nvr4101DSIUSio – NEC VR4101 DSIU UART tty driver

ROUTINES
nvr4101DSIUDevInit() – initialization of the NVR4101DSIU DSIU.
nvr4101DSIUInt() – interrupt level processing
nvr4101DSIUIntMask() – mask interrupts from the DSIU.
nvr4101DSIUIntUnmask() – unmask interrupts from the DSIU.

DESCRIPTION
This is the device driver for the nvr4101 DSIU UART.

USAGE
An NVR4101_DSIU_CHAN data structure is used to describe the DSIU.
nvr4101SIUSio

NAME  nvr4101SIUSio – NEC VR4101 SIU UART tty driver

ROUTINES  
nvr4101SIUDevInit() – initialization of the NVR4101SIU SIU.
nvr4101SIUInt() – interrupt level processing
nvr4101SIUIntMask() – mask interrupts from the SIU.
nvr4101SIUIntUnmask() – unmask interrupts from the SIU.
nvr4101SIUCharToTxWord() – translate character to output word format.

DESCRIPTION  This is the device driver for the nvr4101 UART.

USAGE  A NVR4101_CHAN data structure is used to describe the SIU.

The BSP’s sysHwInit() routine typically calls sysSerialHwInit(), which should pass a pointer to an uninitialized NVR4101_CHAN structure to nvr4101SIUDevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2(), which connects the chip’s interrupts via intConnect().

INCLUDE FILES  drv/sio/nvr4101SIUSio.h

nvr4102DSIUSio

NAME  nvr4102DSIUSio – NEC VR4102 DSIU UART tty driver

ROUTINES  
nvr4102DSIUDevInit() – initialization of the NVR4102DSIU DSIU.
nvr4102DSIUInt() – interrupt level processing
nvr4102DSIUIntMask() – mask interrupts from the DSIU.
nvr4102DSIUIntUnmask() – unmask interrupts from the DSIU

DESCRIPTION  This is the device driver for the nvr4102 DSIU UART.
1: Driver Libraries

nvr4102DSIUso

**USAGE**

An NVR4102_DSIU_CHAN data structure is used to describe the DSII.

The BSP’s sysHwInit() routine typically calls sysSerialHwInit(), which should pass a pointer to an uninitialized NVR4102_DSIU_CHAN structure to nvr4102DSIUDevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2(), which connects the chip’s interrupts via intConnect().

**INCLUDE FILES**

drv/sio/nvr4102DSIUso.h
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.

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 non-excluded 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
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, auto configuration takes place in two phases. In the first phase, all devices and subordinate busses in a given system are scanned and each device that is found causes an entry to be created in the Probelist or list of devices found during the probe/configuration process.

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

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

HOST BRIDGE DETECTION/CONFIGURATION

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

PCI-PCI BRIDGE DETECTION/CONFIGURATION

Busses are scanned by first writing the primary, secondary, and subordinate bus information into the bridge that implements the bus. Specifically, the primary and secondary bus numbers are set to their corresponding value, and the subordinate bus number is set to 0xFF, because the final number of sub-busses is not known. The subordinate bus number is later updated to indicate the highest numbered sub-bus that was scanned once the scan is complete.

GENERIC DEVICE DETECTION/CONFIGURATION

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

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

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

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

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

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

Lastly, the functions are enabled with what resources were able to be provided from the applicable resource pools.
RESOURCES 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 documentation for pciAutoCfgCtl() for more information.

PCI RESOURCE POOLS

Resources used by pciAutoConfigLib can be divided into two groups.

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

pciMem32
Specifies the 32-bit prefetchable memory pool base address. Normally, this is given by the BSP constant PCI_MEM_ADRS. It can be set with the pciAutoCfgCtl() command PCI_MEM32_LOC_SET.

pciMem32Size
Specifies the 32-bit prefetchable memory pool size. Normally, this is given by the BSP constant PCI_MEM_SIZE. It can be set with the pciAutoCfgCtl() command PCI_MEM32_SIZE_SET.

pciMemIo32
Specifies the 32-bit non-prefetchable memory pool base address. Normally, this is given by the BSP constant PCI_MEMIO_ADRS. It can be set with the pciAutoCfgCtl() command PCI_MEMIO32_LOC_SET.

pciMemIo32Size
Specifies the 32-bit non-prefetchable memory pool size. Normally, this is given by the BSP constant PCI_MEMIO_SIZE. It can be set with the pciAutoCfgCtl() command PCI_MEMIO32_SIZE_SET.

pciIo32
Specifies the 32-bit I/O pool base address. Normally, this is given by the BSP constant PCI_IO_ADRS. It can be set with the pciAutoCfgCtl() command PCI_IO32_LOC_SET.
pciIo32Size
Specifies the 32-bit I/O pool size. Normally, this is given by the BSP constant PCI_IO_SIZE. It can be set with the pciAutoCfgCtl() command PCI_IO32_SIZE_SET.

pciIo16
Specifies the 16-bit I/O pool base address. Normally, this is given by the BSP constant PCI_ISA_IO_ADDR. It can be set with the pciAutoCfgCtl() command PCI_IO16_LOC_SET.

pciIo16Size
Specifies the 16-bit I/O pool size. Normally, this is given by the BSP constant PCI_ISA_IO_SIZE. It can be set with the pciAutoCfgCtl() command PCI_IO16_SIZE_SET.

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

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

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

DEVICE INCLUSION
Specific devices other than bridges can be excluded from auto configuration and either not used or manually configured later. For 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 when each device or bridge is configured. For information, see the PCI_INT_ASSIGN_FUNC_SET section in the entry for pciAutoCfgCtl().

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

**TRUE**

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

**FALSE**

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

**BRIDGE CONFIGURATION**

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

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

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

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

**HOST BRIDGE CONFIGURATION**

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

**LIBRARY CONFIGURATION MACROS**

The following four macros can be defined by the BSP Developer in config.h to govern the operation of the autoconfig library.
PCI_AUTO_MAX_FUNCTIONS
Defines the maximum number of functions that can be stored in the probe list during the autoconfiguration pass. The default value for this define is 32, but this may be overridden by defining PCI_AUTO_MAX_FUNCTIONS in config.h.

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

PCI_AUTO_RECLAIM_LIST
This define may only be used if PCI_AUTO_STATIC_LIST is not defined. If defined, this allows the autoconfig routine to perform a free() operation on a dynamically allocated probe list.

NOTE: 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: The example BSP-specific routines are merely stubs. The code in each routine varies by BSP and application.

```c
#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 */
}


def

UCCHAR sysPciAutoconfigIntrAssign

void sysPciAutoconfigPreEnumBridgeInit

void sysPciAutoconfigPostEnumBridgeInit

CONFIGURATION SPACE PARAMETERS

NOTE: 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 S10 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.
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.

**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

---

**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.
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, these functions should not be called from interrupt level, (except pciInt()) because configuration space access, which is slow, should be limited to initialization only.

The functions addressed here include:

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

PCI BUS CONCEPTS

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

To accommodate additional devices, the PCI standard allows multiple PCI buses to be interconnected via PCI-to-PCI bridge (PPB) devices to form one large bus. Each constituent bus is referred to as a bus segment and is subject to the above limitations.
The bus segment accessible from the host bus adapter is designated the primary bus segment (see figure). Progressing outward from the primary bus (designated segment number zero from the PCI architecture point of view) are the secondary and tertiary buses, numbered as segments one and two, respectively. Due to clock skew concerns and propagation delays, practical PCI bus architectures do not implement bus segments beyond the tertiary level.

For further details, refer to the PCI to PCI Bridge Architecture Specification.
I/O MACROS AND CPU ENDIANESS

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

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

- **PCI_IN_BYTE**
  Read a byte from PCI I/O Space.

- **PCI_IN_WORD**
  Read a word from PCI I/O Space.

- **PCI_IN_LONG**
  Read a longword from PCI I/O Space.

- **PCI_OUT_BYTE**
  Write a byte from PCI I/O Space.

- **PCI_OUT_WORD**
  Write a word from PCI I/O Space.

- **PCI_OUT_LONG**
  Write a longword from PCI I/O Space.

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

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

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

Any PCI device can be uniquely addressed within Configuration Space by the geographic specification of a Bus segment number, Device number, and a Function number (BDF). The configuration registers of a PCI device are arranged by the PCI standard according to a Configuration Header structure. The BDF triplet specifies the location of the header structure of one device. To access a configuration register, its location in the header must be given. The location of a configuration register of interest is simply the structure member offset defined for the register. For further details, refer to the PCI Local Bus Specification, Revision 2.1. Refer to the header file `pciConfigLib.h` for the defined standard configuration register offsets.
The maximum number of Type-1 Configuration Space buses supported in the 2.1 Specifications is 256 (0x00 - 0xFF), far greater than most systems currently support. Most buses are numbered sequentially from 0. An optional define called PCI_MAX_BUS may be declared in config.h to override the default definition of 256. Similarly, the default number of devices and functions may be overridden by defining PCI_MAX_DEV and/or PCI_MAX_FUNC.

**NOTE:** The number of devices applies only to bus zero, all others being restricted to 16 by the 2.1 spec.

**ACCESS MECHANISM 1**

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

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

```c
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:

```c
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

Do not use pciConfigOutWord(), pciConfigOutByte(), pciConfigModifyWord(), or
pciConfigModifyByte() to modify 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 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.

```c
/*
 * 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:

\textit{pBusNo} \\
Where to return bus segment number containing the device.

\textit{pDeviceNo} \\
Where to return the device ID of the device.

\textit{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 PCI bus 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**
  - Is the PCI bus of interest.

- **Device Number**
  - Is set to all 1’s (01Fh).

- **Function Number**
  - Is set to all 1’s (07d).

- **Configuration Register Number**
  - Is zeroed.

The \texttt{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 \texttt{pciSpecialCycle()} are:

\textit{busNo} \\
Bus on which Special Cycle is to be initiated.

\textit{message} \\
Data driven onto AD[31:0] during the Special Cycle.

**PCI DEVICE CONFIGURATION SPACE ACCESS**

The routines \texttt{pciConfigInByte()}, \texttt{pciConfigInWord()}, \texttt{pciConfigInLong()}, \texttt{pciConfigOutByte()}, \texttt{pciConfigOutWord()}, and \texttt{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 behavior 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 behavior. If user code behavior changes, the user should investigate why and fix the code that is calling into this library with invalid offset values.

**PCI DEVICE CONFIGURATION**

The function `pciDevConfig()` is used to configure PCI devices that require no more than one Memory Space and one I/O Space. According to the PCI standard, a device may have up to six 32-bit Base Address Registers (BARs) each of which can have either a Memory Space or I/O Space base address. In 64-bit PCI devices, the registers double up to give a maximum of three 64-bit BARs. The 64-bit BARs are not supported by this function nor are more than one 32-bit BAR of each type, Memory or I/O.

The `pciDevConfig()` function sets up one PCI Memory Space and/or one I/O Space BAR and issues a specified command to the device to enable it. It takes the following parameters:

- `pciBusNo`  
  PCI bus segment number.

- `pciDevNo`  
  PCI device number.

- `pciFuncNo`  
  PCI function number.

- `devIoBaseAdrs`  
  Base address of one IO-mapped resource.

- `devMemBaseAdrs`  
  Base address of one memory-mapped resource.

- `command`  
  Command to issue to device after configuration.

**UNIFORM DEVICE ACCESS**

The function `pciConfigForeachFunc()` is used to perform some action on every device on the bus. This does a depth-first recursive search of the bus and calls a specified routine for each function it finds. It takes the following parameters:

- `bus`  
  The bus segment to start with. This allows configuration on and below a specific place in the bus hierarchy.

- `recurse`  
  A boolean argument specifying whether to do a recursive search or to do just the specified bus.
**funcCheckRtn**

A user supplied function which will be called for each PCI function found. It must return STATUS. It takes four arguments: "bus", "device", "function", and a user-supplied arg "pArg". The typedef PCI_FOREACH_FUNC is defined in pciConfigLib.h for these routines.

**NOTE:** 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:** This function disables all access to the PCI bus except for the use of PCI config space. If there are devices on the PCI bus which are required to reboot, then those devices must be re-enabled after the call to pciConfigReset() or the system will not be able to reboot.

**USAGE**
The following code sample illustrates the usage of this library. Initialization of the library is performed first, then a sample device is found and initialized.

```c
#include "drvpci/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_AUTOBED); 
}

/*
 * 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_ADDRS, 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_ADDRS,
        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

184
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().

pciConfigShow

NAME  pciConfigShow – show routines of PCI bus (IO mapped) library

ROUTINES  pciDeviceShow() – print information about PCI devices
          pciHeaderShow() – print a header of the specified PCI device
          pciFindDeviceShow() – find a device by device Id, then print an 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 did not 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.

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.

---

### pcmciaShow

<table>
<thead>
<tr>
<th><strong>NAME</strong></th>
<th><code>pcmciaShow</code> – PCMCIA show library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROUTINES</strong></td>
<td><code>pcmciaShowInit()</code> – initialize all show routines for PCMCIA drivers</td>
</tr>
<tr>
<td></td>
<td><code>pcmciaShow()</code> – show all configurations of the PCMCIA chip</td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
<td>This library provides a show routine that shows the status of the PCMCIA chip and the PC card.</td>
</tr>
</tbody>
</table>
ppc403Sio

NAME  ppc403Sio – ppc403GA serial driver

ROUTINES  ppc403DummyCallback() – dummy callback routine
ppc403DevInit() – initialize the serial port unit
ppc403IntWr() – handle a transmitter interrupt
ppc403IntRd() – handle a receiver interrupt
ppc403IntEx() – handle error interrupts

DESCRIPTION  This is the driver for PPC403GA serial port on the on-chip peripheral bus. The SPU (serial port unit) consists of three main elements: receiver, transmitter, and baud-rate generator. For details, refer to the 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

ppc555SciSio

NAME  ppc555SciSio – MPC555 SCI serial driver

ROUTINES  ppc555SciDevInit() – initialize a PPC555SCI channel
ppc555SciDevInit2() – initialize a PPC555SCI, part 2
ppc555SciInt() – handle a channel’s interrupt

DESCRIPTION  This is the driver for SCIs of the QSMC the Motorola PPC555. The SMC has two SCI channels. Both channels are compatible with earlier SCI devices from Motorola (eg. MC68332), with enhancements to allow external baud clock source and queued operation for the first SCI channel.

DATA STRUCTURES  An PPC555SCI_CHAN data structure is used to describe each channel, this structure is described in h/drv/sio/ppc555SciSio.h. Based on the “options” field of this structure, the
The page contains text about the `ppc860Sio` 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.

**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 `PPC800SMC_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`
sa1100Sio

NAME

sa1100Sio – Digital Semiconductor SA-1100 UART tty driver

ROUTINES

sa1100DevInit() – initialize an SA1100 channel
sa1100Int() – handle an interrupt

DESCRIPTION

This is the device driver for the Digital Semiconductor SA-1100 UARTs. This chip contains 5 serial ports, but only ports 1 and 3 are usable as UARTs, the others support Universal Serial Bus (USB), SDLC, IrDA Infrared Communications Port (ICP) and Multimedia Communications Port (MCP)/Synchronous Serial Port (SSP).

The UARTs are identical in design. They contain a universal asynchronous receiver/transmitter, and a baud-rate generator. The UARTs contain an 8-entry, 8-bit FIFO to buffer outgoing data and a 12-entry 11-bit FIFO to buffer incoming data. 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 only mode of operation supported is with the FIFOs enabled.

The UART design does not support modem control input or output signals e.g. DTR, RI, RTS, DCD, CTS and DSR.

An interrupt is generated when a framing, parity or receiver overrun error is present within the bottom four entries of the receive FIFO, when the transmit FIFO is half-empty or receive FIFO is one- to two-thirds full, when a begin and end of break is detected on the receiver, and when the receive FIFO is partially full and the receiver is idle for three or more frame periods.

Only asynchronous serial operation is supported by the UARTs which supports 7 or 8 bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity. The default baud rate is determined by the BSP by filling in the SA1100_CHAN structure before calling sa1100DevInit().

The UART supports baud rates from 56.24 to 230.4 kbps.

DATA STRUCTURES

An SA1100_CHAN data structure is used to describe each channel, this structure is described in h/drv/sio/sa1100Sio.h.

CALLBACKS

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

The driver is typically only called by the BSP. The directly callable routines in this modules are `sa1100DevInit()` and `sa1100Int()`.

The BSP’s `sysHwInit()` routine typically calls `sysSerialHwInit()`, which initializes the hardware-specific fields in the `SA1100_CHAN` structure (e.g. register I/O addresses, etc.) before calling `sa1100DevInit()` 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:

```c
#include "drv/sio/sa1100Sio.h"
LOCAL SA1100_CHAN sa1100Chan[N_SA1100_UART_CHANNELS];
void sysSerialHwInit (void)
{
    int i;
    for (i = 0; i < N_SA1100_UART_CHANNELS; i++)
    {
        sa1100Chan[i].regs = devParas[i].baseAdrs;
        sa1100Chan[i].baudRate = CONSOLE_BAUD_RATE;
        sa1100Chan[i].xtal = UART_XTAL_FREQ;
        sa1100Chan[i].level = devParas[i].intLevel;
        /* set up GPIO pins and UART pin reassignment */
        ...
        /* Initialise driver functions, getTxChar, putRcvChar
         * and channelMode and initialise UART
         */
        sa1100DevInit(&sa1100Chan[i]);
    }
}
```

The BSP’s `sysHwInit2()` routine typically calls `sysSerialHwInit2()`, which connects and enables the chips interrupts via `intConnect()`, as shown in the following example:

```c
void sysSerialHwInit2 (void)
{
    int i;
    for (i = 0; i < N_SA1100_UART_CHANNELS; i++)
    {
        /* connect and enable interrupts */
        (void)intConnect (INUM_TO_Ivec(devParas[i].vector),
            sa1100Int, (int) &sa1100Chan[i]);
        intEnable (devParas[i].intLevel);
    }
}
```
BSP

By convention, all the BSP-specific serial initialization is performed in a file called
sysSerial.c, which is #included 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 SA1100_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/sa1100Sio.h, sioLib.h

SEE ALSO
Digital StrongARM SA-1100 Portable Communications Microcontroller, Data Sheet,

sab82532

NAME

sab82532 – Siemens SAB 82532 UART tty driver

ROUTINES

sab82532DevInit() – initialize an SAB82532 channel
sab82532Int() – interrupt level processing

DESCRIPTION

This is the device driver for the sab82532 (D)UART.

USAGE

A SAB82532_CHAN data structure is used to describe each channel on the chip.
The BSP’s sysHwInit() routine typically calls sysSerialHwInit(), which initializes all the
values in the SAB82532_CHAN structure (except the SIO_DRV_FUNCS) before calling
sab82532DevInit(). The BSP’s sysHwInit2() routine typically calls sysSerialHwInit2(),
which connects the chips interrupts via intConnect().

INCLUDE FILES
drv/sio/ns16552Sio.h
sh7615End

NAME  
sh7615End – sh7615End END network interface driver

ROUTINES  
sh7615EndLoad() – initialize the driver and device

DESCRIPTION  
This module implements an network interface driver for the Hitachi SH7615 on-chip Ethernet controller (EtherC) and EtherNet CONtroller Direct Memory Access Controller (E-DMAC). The EtherC is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. Hardware support of the Media Independent Interface (MII) is off-chip.

The Ethernet controller is connected to dedicated transmit and receive Ethernet DMACs (E-DMACs) in the SH7615, and carries out high-speed data transfer to and from memory. The operation of the E-DMAC is controlled with the transmit and receive descriptor rings. The start address of the descriptors is set in the RDLAR and TDLAR registers, so they can reside anywhere. The descriptors must reside on boundaries that are multiple of their size (16, 32, or 64 bytes).

EXTERNAL INTERFACE  
The driver provides the standard external interface, sh7615EndLoad(), 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:

```c
strtol (parameter, NULL, 16)
```

The format of the parameter string is:

"ivec:ilevel:numRds:numTds:phyDefMode:userFlags"

TARGET-SPECIFIC PARAMETERS  

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.

ilevel  
This parameter defines the level of the hardware interrupt.

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

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

**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. Bit [0-3] reserved for receive FIFO depth and bit [4-7] reserved for transmit FIFO depth. The actual FIFO size is 256 times + 256 the set value for each FIFO. The max FIFO depth is 512 bytes for SH7615 (i.e. 0x1 for receive FIFO, 0x10 for transmit FIFO) and 2048 bytes for SH7616 (i.e. 0x7 for receive FIFO, 0x70 for transmit FIFO).

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. 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 **SH7615END_DRV_CTRL** pointer. By default this routine copies the ethernet address stored in the global variable **sysTemplateEnetAddr** into the **SH7615END_DRV_CTRL** structure.

**SEE ALSO**
**muxLib**, **endLib**, *Writing An Enhanced Network Driver SH7615 Hardware Manual*
shScifSio

NAME

shScifSio – Hitachi 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 channel’s transmitter-ready interrupt
shScifIntErr() – handle a channel’s error interrupt
dummyCallback() – dummy callback routine

DESCRIPTION

This is the driver for the Hitachi 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, sioLib.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 channel’s transmitter-ready interrupt.
shSciIntErr() – handle a channel’s error interrupt.
dummyCallback() – dummy callback routine.

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
NAME

smcFdc37b78x – a super IO (fdc37b78x) initialization source module

ROUTINES

smcFdc37b78xDevCreate() – set correct IO port addresses for Super I/O chip
smcFdc37b78xInit() – initializes Super I/O chip Library
smcFdc37b78xKbdInit() – initializes the keyboard controller

DESCRIPTION

The FDC37B78x with advanced Consumer IR and IrDA v1.0 support incorporates a keyboard interface, real-time clock, SMSC’s true CMOS 765B floppy disk controller, advanced digital data separator, 16 byte data FIFO, two 16C550 compatible UARTs, one Multi-Mode parallel port which includes ChiProtect circuitry plus EPP and ECP support, on-chip 12 mA AT bus drivers, and two floppy direct drive support, soft power management and SMI support and Intelligent Power Management including PME and SCI/ACPI support. The true CMOS 765B core provides 100% compatibility with IBM PC/XT and PC/AT architectures in addition to providing data overflow and underflow protection. The SMSC advanced digital data separator incorporates SMSC’s patented data separator technology, allowing for ease of testing and use. Both on-chip UARTs are compatible with the NS16C550. The parallel port, the IDE interface, and the game port select logic are compatible with IBM PC/AT architecture, as well as EPP and ECP.

The FDC37B78x incorporates sophisticated power control circuitry (PCC) which includes support for keyboard, mouse, modem ring, power button support and consumer infrared wake-up events. The PCC supports multiple low power down modes.

The FDC37B78x provides features for compliance with the "Advanced Configuration and Power Interface Specification" (ACPI). These features include support of both legacy and ACPI power management models through the selection of SMI or SCI. It implements a power button override event (4 second button hold to turn off the system) and either edge triggered interrupts.

The FDC37B78x provides support for the ISA Plug-and-Play Standard (Version 1.0a) and provides for the recommended functionality to support Windows95, PC97 and PC98.

Through internal configuration registers, each of the FDC37B78x’s logical device’s I/O address, DMA channel and IRQ channel may be programmed. There are 480 I/O address location options, 12 IRQ options or Serial IRQ option, and four DMA channel options for each logical device.

USAGE

This library provides routines to initialize various logical devices on super IO 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 the facility can selectively initialize a set of devices on super IO chip.
--- Initializing keyboard by sending commands to its controller embedded in super IO chip.

**INTERNAL DATABASES**

This library provides its user to changes super IO'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_IOCPORTS` 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 super IO chip.

- `SMCFDC37B78X_INDEX_PORT` 
  Defines the index I/O port for SMC-FDC37B78X super IO chip.

- `SMCFDC37B78X_DATA_PORT` 
  Defines the data I/O port for SMC-FDC37B78X super IO chip.

**USER INTERFACE**

`VOID smcFdc37b78xDevCreate` 

```c
SMCFDC37B78X_IOCPORTS *smcFdc37b78x_iop
)
```

This is a very first routine that should be called by the user of this library. This routine sets up IO port address that will subsequently be used later on. The IO PORT setting could either be overridden by redefining `SMCFDC37B78X_CONFIG_PORT`, `SMCFDC37B78X_INDEX_PORT`, and `SMCFDC37B78X_DATA_PORT` or on-the-fly by passing in a pointer of type `SMCFDC37B78X_IOCPORTS`.

`VOID smcFdc37b78xInit` 

```c
int devInitMask
)
```

This is routine intakes device initialization mask and initializes only those devices that are requested by user. Device initialization mask holds bitwise ORed values of all devices that are requested by user to enable on super IO device.

The mnemonics that are supported in current version of this facility are:

- `SMCFDC37B78X_COM1_EN` 
  Use this mnemonic to enable COM1 only.

- `SMCFDC37B78X_COM2_EN` 
  Use this mnemonic to enable COM2 only.

- `SMCFDC37B78X_LPT1_EN` 
  Use this mnemonic to enable LPT1 only.
SMCFDC37B78X_KBD_EN

Use this mnemonic to enable KBD only.

SMCFDC37B78X_FDD_EN

Use this mnemonic to enable FDD only.

The above mentioned can be bitwise ORed to enable more than one device at a time. e.g. if you want COM1 and COM2 to be enable on super IO chip call:

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

The prerequisites for above mentioned call, super IOchip library should be initialized using `smcFdc37b78xDevCreate()` with parameter as per user’s need.

```c
STATUS smcFdc37b78xKbdInit
{
    VOID
}
```

This routine sends some keyboard commands to keyboard controller embedded in super IO chip. Call to this function is required for proper functioning of keyboard driver.

**INCLUDE FILES**

`smcFdc37b78x.h`

---

**smNetLib**

**NAME**

`smNetLib` – VxWorks interface to shared memory network (backplane) driver

**DESCRIPTION**

This library implements the VxWorks-specific portions of the shared memory network interface driver. It provides the interface between VxWorks and the network driver modules (e.g., how the OS initializes and attaches the driver, interrupt handling, etc.), as well as VxWorks-dependent system calls.

There are no user-callable routines.

The backplane master initializes the backplane shared memory and network structures by first calling `smNetInit()`. Once the backplane has been initialized, all processors can be attached to the shared memory network via the `smNetAttach()` routine. Both `smNetInit()` and `smNetAttach()` are called automatically during system initialization when backplane parameters are specified in the boot line.

For detailed information refer to *VxWorks Network Programmer’s Guide: Data Link Layer Network Components*.

**INCLUDE FILES**

`smNetLib.h`, `smPktLib.h`, `smUtilLib.h`

**SEE ALSO**

`ifLib`, `if_sm`, *VxWorks Network Programmer’s Guide*


**smNetShow**

**NAME**
smNetShow – shared memory network driver show routines

**ROUTINES**
smNetShow( ) – show information about a shared memory network

**DESCRIPTION**
This library provides show routines for the shared memory network interface driver. The smNetShow( ) routine is provided as a diagnostic aid to show current shared memory network status.

**INCLUDE FILES**
smNetLib.h, smPktLib.h

**SEE ALSO**
if_sm, smNetLib, smPktLib, VxWorks Network Programmer’s Guide

---

**sn83932End**

**NAME**
sn83932End – Nat. Semi DP83932B SONIC Ethernet driver

**ROUTINES**
sn83932EndLoad( ) – initialize the driver and device

**DESCRIPTION**
This module implements the National Semiconductor DP83932 SONIC 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 several target-specific parameters. The driver also depends on a few external support routines. These parameters and support routines are described below. If any of the assumptions stated below are not true for your particular hardware, this driver probably cannot function correctly with that hardware. This driver supports up to four individual units per CPU.

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

**EXTERNAL INTERFACE**
This driver provides the END external interface. Thus, the only normal external interface is the sn83932EndLoad( ) routine, although snEndClkEnable( ) and snEndClkDisable( ) are provided for the use (optional) of the internal clock. All required parameters are passed into the load function by means of a single colon-delimited string. The sn83932Load( ) function uses strtok( ) to parse the string, which it expects to be of the following format:
The entry point for `sn83932EndLoad()` is defined within the `endDevTbl` in `configNet.h`.

**TARGET-SPECIFIC PARAMETERS**

- **unit_ID**: A convenient holdover from the former model, this is only used in the string name for the driver.
- **devIO_addr**: Denotes the base address of the device’s I/O register set.
- **ivec**: Denotes the interrupt vector to be used by the driver to service an interrupt from the SONIC device. The driver connects the interrupt handler to this vector by calling `intConnect()`.
- **e_addr**: This parameter is obtained by calling `sysEnetAddrGet()`, an external support routine. It specifies the unique six-byte address assigned to the VxWorks target on the Ethernet.

**EXTERNAL SUPPORT REQUIREMENTS**

This driver requires the following external support routines:

- **sysEnetInit()**
  ```c
  void sysEnetInit (int unit)
  ```
  This routine performs any target-specific operations that must be executed before the SONIC device is initialized. The driver calls this routine, once per unit, during the unit start-up phase.

- **sysEnetAddrGet()**
  ```c
  STATUS sysEnetAddrGet (int unit, char *pCopy)
  ```
  This routine provides the six-byte Ethernet address used by `unit`. It must copy the six-byte address to the space provided by `pCopy`. This routine returns OK, or ERROR if it fails. The driver calls this routine, once per unit, during the unit start-up phase.

- **sysEnetIntEnable()**
  ```c
  void sysEnetIntEnable (int unit), void sysEnetIntDisable (int unit)
  ```
  These routines enable or disable the interrupt from the SONIC device for the specified `unit`. Typically, this involves interrupt controller hardware, either internal or external to the CPU. The driver calls these routines only during initialization, during the unit start-up phase.

- **sysEnetIntAck()**
  ```c
  void sysEnetIntAck (int unit)
  ```
  This routine performs any interrupt acknowledgment or clearing that may be required. This typically involves an operation to some interrupt control hardware. The driver calls this routine from the interrupt handler.
DEVICE CONFIGURATION

Two global variables, \texttt{snEndDcr} and \texttt{snEndDcr2}, are used to set the SONIC device configuration registers. By default, the device is programmed in 32-bit mode with zero-wait states. If these values are not suitable, the \texttt{snEndDcr} and \texttt{snEndDcr2} variables should be modified before loading the driver. See the SONIC manual for information on appropriate values for these parameters.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

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

The above data and BSS requirements are for the MC68020 architecture and can vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

This driver uses \texttt{cacheDmaMalloc()} to allocate the memory to be shared with the SONIC device. The size requested is 117,188 bytes.

The SONIC device can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data that is written by the driver because fields within the shared structures are asynchronously modified by the driver and the device, and these fields may share the same cache line.

SEE ALSO

ifLib

\texttt{sramDrv}

NAME

\texttt{sramDrv} – PCMCIA SRAM device driver

ROUTINES

\texttt{sramDrv()} – install a PCMCIA SRAM memory driver  
\texttt{sramMap()} – map PCMCIA memory onto a specified ISA address space  
\texttt{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: \texttt{sramDrv()} to initialize the driver, and \texttt{sramDevCreate()} to create block devices. Additionally, the \texttt{sramMap()} routine is called directly to map the PCMCIA memory onto the ISA address space.

\textbf{NOTE:} 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 \texttt{sramDrv()}. This routine should be called only once, before any reads, writes, or calls to \texttt{sramDevCreate()} or \texttt{sramMap()}. It can be called from \texttt{usrRoot()} in \texttt{usrConfig.c} or at some later point.

\textbf{SEE ALSO} \hspace{1em} \textit{VxWorks Programmer's Guide: I/O System}

---

\textbf{st16552Sio}

\textbf{NAME} \hspace{1em} \texttt{st16552Sio} – ST 16C552 DUART tty driver

\textbf{ROUTINES}

\begin{itemize}
  \item \texttt{st16552DevInit()} – initialize an ST16552 channel
  \item \texttt{st16552IntWr()} – handle a transmitter interrupt
  \item \texttt{st16552IntRd()} – handle a receiver interrupt
  \item \texttt{st16552IntEx()} – miscellaneous interrupt processing
  \item \texttt{st16552Int()} – interrupt level processing
  \item \texttt{st16552MuxInt()} – multiplexed interrupt level processing
\end{itemize}

\textbf{DESCRIPTION} \hspace{1em} This is the device driver for the Startech ST16C552 DUART, similar, but not quite identical to the National Semiconductor 16550 UART.

The chip is a dual universal asynchronous receiver/transmitter with 16 byte transmit and receive FIFOs and a programmable baud-rate generator. Full modem control capability is included and control over the four interrupts that can be generated: Tx, Rx, Line status, and modem status. Only the Rx and Tx interrupts are used by this driver. The FIFOs are enabled for both Tx and Rx by this driver.

Only asynchronous serial operation is supported by the UART which supports 5 to 8 bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity. The default baud rate is determined by the BSP by filling in the \texttt{ST16552_CHAN} structure before calling \texttt{st16552DevInit()}. 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 50 to about 115200 are possible.
DATA STRUCTURES

An ST16552_CHAN data structure is used to describe the two channels of the chip and, if necessary, an ST16552_MUX structure is used to describe the multiplexing of the interrupts for the two channels of the DUART. These structures are described in h/drv/sio/st16552Sio.h.

CALLBACKS

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

MODES

This driver supports both polled and interrupt modes.

USAGE

The driver is typically only called by the BSP. The directly callable routines in this module are st16552DevInit(), st16552Int(), st16552IntRd(), st16552IntWr(), and st16552MuxInt().

The BSP's sysHwInit() routine typically calls sysSerialHwInit(), which initializes all the hardware-specific values in the ST16552_CHAN structure before calling st16552DevInit() 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/st16552Sio.h"
LOCAL ST16552_CHAN st16552Chan[N_16552_CHANNELS];
void sysSerialHwInit (void)
{
    int i;
    for (i = 0; i < N_16552_CHANNELS; i++)
    {
        st16552Chan[i].regDelta = devParas[i].regSpace;
        st16552Chan[i].regs = devParas[i].baseAdrs;
        st16552Chan[i].baudRate = CONSOLE_BAUD_RATE;
        st16552Chan[i].xtal = UART_XTAL_FREQ;
        st16552Chan[i].level = devParas[i].intLevel;
        /*
        * Initialise driver functions, getTxChar, putRcvChar and
        * channelMode and init UART.
        */
        st16552DevInit(&st16552Chan[i]);
    }
}
```
The BSP’s `sysHwInit2()` routine typically calls `sysSerialHwInit2()`, which connects the chips interrupts via `intConnect()` (either the single interrupt `st16552Int`, the three interrupts `st16552IntWr`, `st16552IntRd`, and `st16552IntEx`, or the multiplexed interrupt handler `st16552MuxInt` which will cope with both channels of a DUART producing the same interrupt). It then enables those interrupts in the interrupt controller as shown in the following example:

```c
void sysSerialHwInit2 (void)
{
    /* Connect the multiplexed interrupt handler */
    (void) intConnect (INUM_TO_IVEC(devParas[0].vector),
                   st16552MuxInt, (int) &st16552Mux);
    intEnable (devParas[0].intLevel);
}
```

**BSP**

By convention all the BSP-specific serial initialization is performed in a file called `sysSerial.c`, which is #included 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 `ST16552_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/st16552Sio.h, sioLib.h
```

**SEE ALSO**

- Startech ST16C552 Data Sheet

---

**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
- `sym895HwInit()` – hardware initialization for the 895 Chip
- `sym895SetHwOptions()` – set the Sym895 chip Options
- `sym895Intr()` – interrupt service routine for the SCSI Controller
- `sym895Show()` – display values of all readable SYM 53C8xx SIOP registers
- `sym895GPIOConfig()` – configure general purpose pins GPIO 0-4
sym895GPIOCtrl() – control general purpose pins GPIO 0-4
sym895Loopback() – perform loopback diagnostics on 895 chip

DESCRIPTION

The SYM53C895 PCI-SCSI I/O Processor (SIOP) brings Ultra2 SCSI performance to Host adapter, making it easy to add a high performance SCSI Bus to any PCI System. It supports Ultra-2 SCSI rates and allows increased SCSI connectivity and cable length Low Voltage Differential (LVD) signaling for SCSI. This driver runs in conjunction with SCRIPTS Assembly program for the Symbios SCSI controllers. These scripts use DMA transfers for all data, messages, and status transfers.

For each controller device a manager task is created to manage SCSI threads on each bus. A SCSI thread represents each unit of SCSI work.

This driver supports multiple initiators, disconnect/reconnect, tagged command queuing and synchronous data transfer protocol. In general, the SCSI system and this driver will automatically choose the best combination of these features to suit the target devices used. However, the default choices may be over-ridden by using the function ’scsiTargetOptionsSet()’ (see scsiLib).

Scatter/Gather memory support: Scatter-Gather transfers are used when data scattered across memory must be transferred across the SCSI bus together with out CPU intervention. This is achieved by a chain of block move script instructions together with the support from the driver. The driver is expected to provide a set of addresses and byte counts for the SCRIPTS code. However there is no support as such from vxworks SCSI Manager for this kind of data transfers. So the implementation, as of today, is not completely integrated with vxworks, and assumes support from SCSI manager in the form of array of pointers. The macro SCATTER_GATHER in sym895.h is thus not defined to avoid compilation errors.

Loopback mode allows 895 chip to control all SCSI signals, regardless of whether it is in initiator or target role. This mode insures proper SCRIPTS instructions fetches and data paths. SYM895 executes initiator instructions through the SCRIPTS, and the target role is implemented in sym895Loopback by asserting and polling the appropriate SCSI signals in the SOCL, SODL, SBCL, and SBDL registers.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly sym895CtrlCreate() to create a controller structure, and sym895CtrlInit() to initialize it. If the default configuration is not correct, the routine sym895SetHwRegister() must be used to properly configure the registers.

Critical events, which are to be logged anyway irrespective of whether debugging is being done or not, can be logged by using the SCSI_MSG macro.

PCI MEMORY ADDRESSING

The global variable sym895PciMemOffset was created to provide the BSP with a means of changing the VIRT_TO_PHYS mapping without changing the functions in the cacheFuncs structures. In generating physical addresses for DMA on the PCI bus, local
addresses are passed through the function CACHE_DMA_VIRT_TO_PHYS and then the value of sym895PciMemOffset is added. For backward compatibility, the initial value of sym895PciMemOffset comes from the macro PCI_TO_MEM_OFFSET.

INTERFACE

The BSP must connect the interrupt service routine for the controller device to the appropriate interrupt system. The routine to be called is sym895Intr(), and the argument is the pointer to the controller device pSiop, i.e.

```c
pSiop = sym895CtrlCreate (...);
intConnect (XXXX, sym895Intr, pSiop);
sym895CtrlInit (pSiop, ...);
```

HARDWARE ACCESS

All hardware access is to be done through macros. The default definition of the SYM895_REGx_READ() and SYM895_REGx_WRITE() macros (where x stands for the width of the register being accessed) assumes an I/O mapped model. Register access mode can be set to either IO/memory using SYM895_IO_MAPPED macro in sym895.h. The macros can be redefined as necessary to accommodate other models, and situations where timing and write pipe considerations need to be addressed. In IO mapped mode, BSP routines sysInByte() and 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, sym895Script.c

SEE ALSO

**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.

**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().
ultraEnd

NAME
ultraEnd – SMC Ultra Elite END network interface driver

ROUTINES
ultraLoad() – initialize the driver and device

DESCRIPTION
This module implements the SMC Elite Ultra Ethernet network interface driver.
This driver supports single transmission and multiple reception. The Current register is a
write pointer to the ring. The Bound register is a read pointer from the ring. This driver
gets the Current register at the interrupt level and sets the Bound register at the task level.
The interrupt is only masked during configuration or in polled mode.

CONFIGURATION
The W1 jumper should be set in the position of "Software Configuration". The defined I/O
address in config.h must match the one stored in EEROM. The RAM address, the RAM
size, and the IRQ level are defined in config.h. IRQ levels 2,3,5,7,10,11,15 are supported.

EXTERNAL SUPPORT REQUIREMENTS
This driver requires several external support functions, defined as macros. These macros
allow the driver to be customized for BSPs that use special versions of these routines:

SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_IN_BYTE(pDrvCtrl, reg, pData)
SYS_OUT_BYTE(pDrvCtrl, reg, pData)

SYS_INT_CONNECT
connects the interrupt handler to the appropriate vector. By default it is the routine
intConnect().

SYS_INT_DISCONNECT
disconnects the interrupt handler prior to unloading the module. By default this is a
dummy routine that returns OK.

SYS_INT_ENABLE
enables the interrupt level for the end device. It is called once during initialization. It
calls an external board level routine sysUltraIntEnable().

SYS_INT_DISABLE
disables the interrupt level for the end device. It is called once during shutdown. It
calls an external board level routine sysUltraIntDisable().

SYS_IN_BYTE and SYS_OUT_BYTE
access the ultra device. The default macros map these operations onto sysInByte() and
sysOutByte().
INCLUDES
end.h, endLib.h, etherMultiLib.h

SEE ALSO
muxLib, endLib, Writing an Enhanced Network Driver
NAME

vgaInit – a VGA 3+ mode initialization source module

ROUTINES

vgaInit() – initialize the VGA chip and loads font in memory

DESCRIPTION

This library provides initialization routines to configure VGA in 3+ alphanumeric mode.

The functions addressed here include:

Initialization of the VGA specific register set.

USER INTERFACE

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

**wd33c93Lib**

**NAME**
wd33c93Lib – WD33C93 SCSI-Bus Interface Controller (SBIC) library

**ROUTINES**
wd33c93CtrlInit() – initialize the user-specified fields in an SBIC structure
wd33c93Show() – display the values of all readable WD33C93 chip registers

**DESCRIPTION**
This library contains the main interface routines to the Western Digital WD33C93 and WD33C93A SCSI-Bus Interface Controllers (SBIC). However, these routines simply switch the calls to either the SCSI-1 or SCSI-2 drivers, implemented in wd33c93Lib1 and wd33c93Lib2 respectively, as configured by the Board Support Package (BSP).

In order to configure the SCSI-1 driver, which depends upon scsi1Lib, the wd33c93CtrlCreate() routine, defined in wd33c93Lib1, must be invoked. Similarly, wd33c93CtrlCreateScsi2(), defined in wd33c93Lib2 and dependent on scsi2Lib, must be called to configure and initialize the SCSI-2 driver.

**INCLUDE FILES**
wd33c93.h, wd33c93_1.h, wd33c93_2.h

**SEE ALSO**
scsiLib, scsi1Lib, scsi2Lib, wd33c93Lib1, wd33c93Lib2, Western Digital WD33C92/93 SCSI-Bus Interface Controller, Western Digital WD33C92A/93A SCSI-Bus Interface Controller, VxWorks Programmer’s Guide: I/O System

**wd33c93Lib1**

**NAME**
wd33c93Lib1 – WD33C93 SCSI-Bus Interface Controller library (SCSI-1)

**ROUTINES**
wd33c93CtrlCreate() – create and partially initialize a WD33C93 SBIC structure

**DESCRIPTION**
This library contains part of the I/O driver for the Western Digital WD33C93 and WD33C93A SCSI-Bus Interface Controllers (SBIC). The driver routines in this library depend on the SCSI-1 version of the SCSI standard; for driver routines that do not depend on SCSI-1 or SCSI-2, and for overall SBIC driver documentation. See wd33c93Lib.

**USER-CALLABLE ROUTINES**
Most routines in this driver are accessible only through the I/O system. The one exception in this portion of the driver is wd33c93CtrlCreate(), which creates a controller structure.

**INCLUDE FILES**
wd33c93.h, wd33c93_1.h

**SEE ALSO**
scsiLib, scsi1Lib, wd33c93Lib
wd33c93Lib2

NAME  wd33c93Lib2 – WD33C93 SCSI-Bus Interface Controller library (SCSI-2)

ROUTINES  wd33c93CtrlCreateScsi2( ) – create and partially initialize an SBIC structure

DESCRIPTION  This library contains part of the I/O driver for the Western Digital WD33C93 family of SCSI-2 Bus Interface Controllers (SBIC). It is designed to work with scsi2Lib. The driver routines in this library depend on the SCSI-2 ANSI specification; for general driver routines and for overall SBIC documentation, see wd33c93Lib.

USER-CALLABLE ROUTINES  Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is wd33c93CtrlCreateScsi2( ), which creates a controller structure.

INCLUDE FILES  wd33c93.h, wd33c93_2.h

SEE ALSO  scsiLib, scsi2Lib, wd33c93Lib, VxWorks Programmer’s Guide: I/O System

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 config file `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.

---

**wdbPipePktDrv**

**NAME**

`wdbPipePktDrv` – pipe packet driver for lightweight UDP/IP

**ROUTINES**

`wdbPipePktDevInit()` – initialize a pipe packet device

**DESCRIPTION**

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 amount of target memory because the lightweight UDP-IP interpreter is much smaller than the VxWorks network stack (about 800 bytes total). Second, they provide a communication path which is independent of the OS, and thus can be used to support an external mode (e.g., monitor style) debug agent.

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 "In" 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:

```c
  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 beginning of usrWdb.c to insure that this is the case by copying the way it is done for the other drivers. The routine wdbCommIfInIt() also needs to be modified so that if your driver is selected as the WDB_COMM_TYPE, then your 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 ultimately makes calls to the routines `wdbMbufAlloc()` and `wdbMbufFree()`, which are provided in source code in `usrWdb.c`.

### wdbSlipPktDrv

**NAME**

`wdbSlipPktDrv` – serial line pocket-size 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, then SLIP sends "ESC TRANS_ESC" to avoid confusion.

**NOTE:** The SLIP ESC is not the same as the ASCII ESC.

On the receiving side of the connection, SLIP uses the opposite actions to decode the SLIP packets. Whenever an END character is received, SLIP assumes a full packet has been received and sends on.

This driver has an MTU of 1006 bytes. If the host is using a real SLIP driver with a smaller MTU, then you will need to lower the definition of `WDB_MTU` in `configAll.h` so that the host and target MTU match. If you are not using a SLIP driver on the host, but instead are using the target server’s wdbserial backend to connect to the agent, then you do not need to worry about incompatibilities between the host and target MTUs.
VxWorks Drivers API Reference, 5.5
wdbTsfsDrv

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. 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

216


read-only results in an error, with `errno` set to `EISDIR`. Calling `read()` on a directory returns `ERROR` with `errno` set to `EISDIR`.

**OPEN FLAGS**

When the target server that is providing the TSFS is running on a Windows host, the default file-translation mode is binary translation. If text translation is required, then `WDB_TSFS_O_TEXT` can be included in the mode argument to `open()`. For example:

```c
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:

```c
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:

```c
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/go` produces the same outcome as the path `/tgtsvr/foo/go`. The path is not modified to account for the current working
directory, and therefore must be an absolute path.

```c
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**

**FIOPUTFL**

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:

```c
int arg = 1;  
status = ioctl (fd, SO_SETDEBUG, arg);
```

To determine whether recording of debugging information for the TSFS-socket file descriptor is enabled or disabled, call:

```c
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.

ERROR CODES
   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.

SEE ALSO

---

wdbUlipPktDrv

NAME
wdbUlipPktDrv – WDB communication interface for the ULIP driver

ROUTINES
wdbUlipPktDevInit() – initialize the communication functions for ULIP

DESCRIPTION
   This is a lightweight ULIP driver that interfaces with the WDB agent’s UDP/IP
   interpreter. It is the lightweight equivalent of the ULIP netif driver. This module provides
   a communication path which supports both a task mode and an external mode WDB
   agent.
**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:

<table>
<thead>
<tr>
<th>Channel</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Virtual console</td>
</tr>
<tr>
<td>1-0xffffffff</td>
<td>Dynamically created on the host</td>
</tr>
<tr>
<td>&gt;= 0x1000000</td>
<td>User defined</td>
</tr>
</tbody>
</table>

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.

**SEE ALSO**
Tornado User’s Guide
NAME
z8530Sio – Z8530 SCC Serial Communications Controller driver

ROUTINES
z8530DevInit() – initialize a Z8530_DUSART
z8530IntWr() – handle a transmitter interrupt
z8530IntRd() – handle a receiver interrupt
z8530IntExt() – handle error interrupts
z8530Int() – handle all interrupts in one vector

DESCRIPTION
This is the driver for the Z8530 SCC (Serial Communications Controller). It uses the SCCs in asynchronous mode only.

USAGE
A Z8530_DUSART structure is used to describe the chip. This data structure contains two Z8530_CHAN structures which describe the chip's two serial channels. Supported baud rates range from 50 to 38400. The default baud rate is Z8530_DEFAULT_BAUD (9600). The BSP may redefine this.

The BSP's sysHwInit() routine typically calls sysSerialHwInit() which initializes all the values in the Z8530_DUSART structure (except the SIO_DRV_FUNCS) before calling z8530DevInit().

The BSP's sysHwInit2() routine typically calls sysSerialHwInit2() which connects the chips interrupts via intConnect() (either the single interrupt z8530Int or the three interrupts z8530IntWr, z8530IntRd, and z8530IntEx).

This driver handles setting of hardware options such as parity (odd, even) and number of data bits (5, 6, 7, 8). Hardware flow control is provided with the signals CTS on transmit and DSR on read. Refer to the target documentation for the RS232 port configuration. The function HUPCL (hang up on last close) is supported. Default hardware options are defined by Z8530_DEFAULT_OPTIONS. The BSP may redefine them.

All device registers are accessed via BSP-defined macros so that memory-mapped as well as I/O space accesses can be supported. The BSP may re-define the REG_8530_READ and REG_8530_WRITE macros as needed. By default, they are defined as simple memory-mapped accesses.

The BSP may define DATA_REG_8530_DIRECT to cause direct access to the Z8530 data register, where hardware permits it. By default, it is not defined.

The BSP may redefine the macro for the channel reset delay Z8530_RESET_DELAY as well as the channel reset delay counter value Z8530_RESET_DELAY_COUNT as required. The delay is defined as the minimum time between successive chip accesses (6 PCLKs + 200 nSec for a Z8530, 4 PCLKs for a Z85C30 or Z85230) plus an additional 4 PCLKs. At a typical PCLK frequency of 10 MHz, each PCLK is 100 nSec, giving a minimum reset delay as follows:
Z8530: 10 PCLKs + 200 nSec = 1200 nSec = 1.2 uSec
Z85x30: 8 PCLKs = 800 nSec = 0.8 uSec

INCLUDE FILES
drv/sio/z8530Sio.h
aicc7880CtrlCreate() – create a control structure for the AIC 7880................................................... 231
aicc7880dFifoThresholdSet() – set the data FIFO threshold ............................................................. 231
aicc7880EnableFast20() – enable double speed SCSI data transfers.................................................. 232
aicc7880GetNumOfBuses() – perform a PCI bus scan ......................................................................... 232
aicc7880ReadConfig() – read from PCI config space.................................................................... 233
aicc7880ScbCompleted() – successfully completed execution of a client thread................................... 233
aicc7880WriteConfig() – read to PCI config space ......................................................................... 234
ambaDevInit() – initialize an AMBA channel.................................................................................. 234
ambaIntRx() – handle a receiver interrupt ........................................................................................ 235
ambaIntTx() – handle a transmitter interrupt.................................................................................... 235
ataDevCreate() – create a device for a ATA/IDE disk...................................................................... 236
ataDrvInit() – initialize ATA driver................................................................................................. 237
ataDrv() – initialize the ATA driver ................................................................................................. 237
ataRawio() – do raw I/O access ......................................................................................................... 238
ataShow() – show the ATA/IDE disk parameters............................................................................... 238
ataShowInit() – initialize the ATA/IDE disk driver show routine.................................................... 239
aucDump() – display device status...................................................................................................... 239
aucEndLoad() – initialize the driver and device .................................................................................. 239
aucInitParse() – parse the initialization string................................................................................ 240
cd2400HrdInit() – initialize the chip................................................................................................. 242
cd2400Int() – handle special status interrupts .................................................................................. 242
cd2400IntRx() – handle receiver interrupts...................................................................................... 242
cd2400IntTx() – handle transmitter interrupts.................................................................................. 242
cisConfigregGet() – get the PCMCIA configuration register............................................................ 243
cisConfigregSet() – set the PCMCIA configuration register............................................................ 243
cisFree() – free tuples from the linked list ......................................................................................... 244
cisGet() – get information from a PC card’s CIS................................................................................ 244
cisShow() – show CIS information..................................................................................................... 245
coldfireAcr() – return aux control register contents......................................................................... 245
coldfireAcrSetClr() – set and clear bits in the UART’s aux control register ........................................ 246
coldfireDevInit() – initialize a COLDFIRE_CHAN ................................................................. 246
coldfireDevInit2() – initialize a COLDFIRE_CHAN, part 2........................................... 247
coldfireImr() – return current interrupt mask register contents .................................. 247
coldfireImrSetClr() – set and clear bits in the UART’s interrupt mask register.............. 248
coldfireOpr() – return the current state of the output register ...................................... 249
coldfireOprSetClr() – set and clear bits in the output port register ............................... 249
cpmAttach() – publish the cpm network interface and initialize the driver .................. 250
cpmStartOutput() – output packet to network interface device ................................... 251
csAttach() – publish the cs network interface and initialize the driver ....................... 252
csShow() – shows statistics for the cs network interface ........................................... 253
cb69000VgaInit() – initialize the B69000 chip and loads font in memory ................ 253
dcAttach() – publish the dc network interface ............................................................. 254
dcCsrShow() – display dec 21040/21140 status registers 0 thru 15 .......................... 255
dcReadAllRom() – read entire serial rom ................................................................ 255
dcViewRom() – display lines of serial ROM for dec21140 ......................................... 256
dec21x4xEndLoad() – initialize the driver and device ................................................ 256
dec21x40EndLoad() – initialize the driver and device ................................................ 257
dec21x40PhyFind() – find the first PHY connected to DEC MII port ......................... 257
dec21x405rKsrWordRead() – read two bytes from the serial ROM ........................ 258
dec21x405rIreadBack() – read all PHY registers out .................................................. 258
dummyCallback() – dummy callback routine ............................................................. 259
eexAttach() – publish the eex network interface and initialize the driver and device ... 260
eexTxStartUp() – start output on the chip ................................................................ 260
ei82596EndLoad() – initialize the driver and device .................................................... 261
eiAttach() – publish the ei network interface and initialize the driver and device .... 262
eihAttach() – publish the ei network interface and initialize the driver and device .... 263
eiInt() – entry point for handling interrupts from the 82596 ..................................... 264
eiTxStartUp() – start output on the chip ................................................................ 265
ei3c90xEndLoad() – initialize the driver and device .................................................... 266
ei3c90xInitParse() – parse the initialization string ..................................................... 266
elcAttach() – publish the elc network interface and initialize the driver and device ... 267
elePut() – copy a packet to the interface .................................................................. 268
eleShow() – display statistics for the SMC 8013WC elc network interface ............. 269
ei3c509xLoad() – initialize the driver and device ....................................................... 269
ei3c509xParse() – parse the initial string ................................................................ 270
eiAttach() – publish the elt interface and initialize the driver and device ............... 271
eiShow() – display statistics for the 3C509 elt network interface ............................. 271
eiTxOutputStart() – start output on the board ............................................................ 272
dEtherAddressForm() – form an Ethernet address into a packet .................................. 272
dEtherPacketAddrGet() – locate the addresses in a packet ........................................ 273
dEtherPacketDataGet() – return the beginning of the packet data ........................... 273
dObjFlagSet() – set the flags member of an END_OBJ structure ............................. 274
dObjInit() – initialize an END_OBJ structure .......................................................... 274
dTok_r() – get a token string (modified version) ......................................................... 275
2: Routines

eveattach() – publish the ene network interface and initialize the driver and device .. 276
evePut() – copy a packet to the interface.......................................................... 276
everShow() – display statistics for the NE2000 ene network interface............. 277
esmattach() – publish the esmc network interface and initialize the driver ...... 277
esmcPut() – copy a packet to the interface......................................................... 278
esmcShow() – display statistics for the esmc network interface........................ 278
evbs16550HrdInit() – initialize the NS 16550 chip ........................................ 279
evbs16550Int() – handle a receiver/transmitter interrupt for the NS 16550 chip 279
fdDevCreate() – create a device for a floppy disk ........................................ 280
fdDrv() – initialize the floppy disk driver ..................................................... 281
fdRawio() – provide raw I/O access ............................................................... 281
fei82557DumpPrint() – Display statistical counters ...................................... 282
fei82557EndLoad() – initialize the driver and device ..................................... 282
fei82557ErrCounterDump() – dump statistical counters............................... 283
feiattach() – publish the fei network interface ............................................ 284
fnattach() – publish the fn network interface and initialize the driver and device .. 285
gei82543EndLoad() – initialize the driver and device ..................................... 286
i8250HrdInit() – initialize the chip............................................................... 287
i8250Int() – handle a receiver/transmitter interrupt .................................... 287
iOlicomEndLoad() – initialize the driver and device ..................................... 288
iOlicomIntHandle() – interrupt service for card interrupts ......................... 288
iPIIX4AtaInit() – low level initialization of ATA device ............................... 289
iPIIX4FdInit() – initialize the floppy disk device ........................................ 289
iPIIX4GetIntr() – give device an interrupt level to use .............................. 290
iPIIX4Init() – initialize PIIX4 ..................................................................... 290
iPIIX4IntrRoute() – route PIRQ[A:D]............................................................ 291
iPIIX4KbdInit() – initialize the PCI-ISA/IDE bridge ...................................... 291
ln79xEndLoad() – initialize the driver and device ....................................... 292
ln79xInitParse() – parse the initialization string .......................................... 292
ln97xEndLoad() – initialize the driver and device ....................................... 294
lnattach() – publish the ln network interface and initialize driver structures ... 294
lnPciattach() – publish the lnPci network interface and initialize the driver and device .......................................................... 295
loattach() – publish the lo network interface and initialize driver and pseudo-device .......................................................... 296
lptDevCreate() – create a device for an LPT port ...................................... 297
lptDrv() – initialize the LPT driver ............................................................... 297
lptShow() – show LPT statistics..................................................................... 298
m68302SioInit() – initialize a M68302_CP .................................................... 299
m68302SioInit2() – initialize a M68302_CP (part 2) ..................................... 299
m68332DevInit() – initialize the SCC............................................................ 300
m68332Int() – handle an SCC interrupt ....................................................... 300
m87360DevInit() – initialize the SCC ............................................................ 301
m87360Int() – handle an SCC interrupt ....................................................... 301
m87362HrdInit() – initialize the DUSCC....................................................... 302
m87362RxInt() – handle a receiver interrupt ................................................ 302
m87362RxTxErrInt() – handle a receiver/transmitter error interrupt .......... 303
m68681Opr() – parse the init string ................................................................. 308
m68681Opcr() – set and clear bits in the DUART output port configuration register .... 309
m68681OpcrSetClr() – set and clear bits in the DUART output port configuration register .... 309
m68681Int() – handle a transmitter interrupt .................................................... 303
m68681Acr() – return the contents of the DUART auxiliary control register ............. 304
m68681AcrSetClr() – set and clear bits in the DUART auxiliary control register ............ 304
m68681DevInit() – initialize a M68681_DUART................................................. 305
m68681DevInit2() – initialize a M68681_DUART, part 2 ......................................... 305
m68681MemInit() – initialize memory for the chip .............................................. 311
m68681Imr() – set and clear bits in the DUART interrupt-mask register ............... 306
m68681Imrs() – set pointers to MII optional registers handlers ............................. 309
m68681DevInit() – initialize the driver and device .............................................. 308
m68681DevInit2() – initialize the driver and device .............................................. 305
m68681Int() – handle all DUART interrupts in one vector.................................... 307
m68681Opf() – return the state of the DUART output port configuration register ....... 308
m68681OpfSetClr() – return the current state of the DUART output port configuration register ...... 308
m68681Opf() – return the current contents of the DUART interrupt-mask register ....... 306
m68681OpfSetClr() – return the current contents of the DUART interrupt-mask register ....... 306
m68681OpfSetClr() – return the current state of the DUART output port configuration register ...... 308
m68681OpfSetClr() – set and clear bits in the DUART output port register ............... 309
m68901DevInit() – initialize a M68901_CHAN structure ........................................ 309
mb86940DevInit() – install the driver function table ............................................ 310
mb86960EndLoad() – initialize the driver and device ........................................... 310
mb86960InitParse() – parse the initialization string ............................................ 311
mb86960MemInit() – initialize memory for the chip ............................................ 311
mb87030CtrlCreate() – create a control structure for an MB87030 SPC .................... 312
mb87030CtrlInit() – initialize a control structure for an MB87030 SPC .................... 312
mb87030Show() – display the values of all readable MB87030 SPC registers .............. 314
mbcAddrFilterSet() – set the address filter for multicast addresses ....................... 315
mbcAttach() – publish the mbc network interface and initialize the driver .............. 315
mbcEndLoad() – initialize the driver and device ................................................. 316
mbcIntr() – network interface interrupt handler .................................................. 317
mbcMemInit() – initialize memory for the chip .................................................. 317
mbcParse() – parse the init string ...................................................................... 318
mbcStartOutput() – output packet to network interface device ................................ 319
mbi2ErrorAdd() – change a MIB-II error count .................................................. 320
mbi2Init() – initialize a MIB-II structure .......................................................... 320
miiAnCheck() – check the auto-negotiation process result ..................................... 321
miiLibInit() – initialize the MII library ............................................................. 321
miiLibUnInit() – uninitialize the MII library ....................................................... 322
miiPhyInit() – initialize and configure the PHY devices ....................................... 322
miiPhyOptFuncMultiSet() – set pointers to MII optional registers handlers ............. 325
miiPhyOptFuncSet() – set the pointer to the MII optional registers handler ............. 325
miiPhyUnInit() – uninitialize a PHY ................................................................. 326
miiRegsGet() – get the contents of MII registers .................................................. 326
miiShow() – show routine for MII library .......................................................... 327
motCpmEndLoad() – initialize the driver and device ............................................. 327
motFccEndLoad() – initialize the driver and device .............................................. 328
motFccEndLoad() – initialize the driver and device .............................................. 329
n72001DevInit() – initialize a N72001_MPSC...................................................... 331
n72001Int() – interrupt level processing ........................................................... 331
n72001IntRd() – handle a receiver interrupt ...................................................... 332
n72001IntWr() – handle a transmitter interrupt .................................................. 332
ncr710CtrlCreate() – create a control structure for an NCR 53C710 SIOP ......................... 333
ncr710CtrlCreateScsi2() – create a control structure for the NCR 53C710 SIOP .................. 334
ncr710CtrlInit() – initialize a control structure for an NCR 53C710 SIOP ......................... 335
ncr710CtrlInitScsi2() – initialize a control structure for the NCR 53C710 SIOP ................ 336
ncr710SetHwRegister() – set hardware-dependent registers for the NCR 53C710 SIOP ........ 337
ncr710SetHwRegisterScsi2() – set hardware-dependent registers for the NCR 53C710 .......... 338
ncr710Show() – display the values of all readable NCR 53C710 SIOP registers .................... 339
ncr710ShowScsi2() – display the values of all readable NCR 53C710 SIOP registers .............. 340
ncr710SingleStep() – perform a single-step ........................................................................ 341
ncr710StepEnable() – enable/disable script single-step..................................................... 342
ncr810CtrlCreate() – create a control structure for the NCR 53C8xx SIOP ......................... 342
ncr810CtrlInit() – initialize a control structure for the NCR 53C8xx SIOP ......................... 343
ncr810SetHwRegister() – set hardware-dependent registers for the NCR 53C8xx SIOP ....... 344
ncr810SetHwRegisterScsi2() – set hardware-dependent registers for the NCR 53C710 SIOP .... 345
ncr5390CtrlCreate() – create a control structure for an NCR 53C90 ASC .......................... 346
ncr5390CtrlCreateScsi2() – create a control structure for an NCR 53C90 ASC .................. 347
ncr5390CtrlInit() – initialize the user-specified fields in an ASC structure .......................... 349
ncr5390Show() – display the values of all readable NCR5390 chip registers ....................... 350
ne2000EndLoad() – initialize the driver and device ......................................................... 351
ns16550DevInit() – initialize an NS16550 channel............................................................ 353
ns16550Int() – interrupt level processing ......................................................................... 354
ns16550IntEx() – miscellaneous interrupt processing .................................................... 354
ns16550IntRd() – handle a receiver interrupt .................................................................... 355
ns16550IntWrt() – handle a transmitter interrupt ............................................................ 355
ns83902EndLoad() – initialize the driver and device ......................................................... 356
ns83902RegShow() – prints the current value of the NIC registers ..................................... 356
nsr4101DSIUDevInit() – initialize the NVR4101DSIU DSIU. ............................................ 357
nsr4101DSIUInt() – interrupt level processing ................................................................... 357
nsr4101DSIUIntMask() – mask interrupts from the DSIU. ................................................ 358
nsr4101DSIUIntUnmask() – unmask interrupts from the DSIU .......................................... 358
nsr4101SIUCharToTxWord() – translate character to output word format ......................... 359
nsr4101SIUDevInit() – initialization of the NVR4101SIU SIU ........................................... 359
nsr4101SIUInt() – interrupt level processing ..................................................................... 360
nsr4101SIUIntMask() – mask interrupts from the SIU ....................................................... 360
nsr4101SIUIntUnmask() – unmask interrupts from the SIU ................................................ 361
nsr4102DSIUDevInit() – initialize the NVR4102DSIU DSIU. ............................................ 361
nsr4102DSIUInt() – interrupt level processing .................................................................. 362
nsr4102DSIUIntMask() – mask interrupts from the DSIU ................................................ 362
nsr4102DSIUIntUnmask() – unmask interrupts from the DSIU .......................................... 363
pccardAtaEnabler() – enable the PCMCIA-ATA device ..................................................... 364
pccardEtEnabler() – enable the PCMCIA Etherlink III card ................................................ 364
pccardMkfs() – initialize a device and mount a DOS file system ........................................ 365
pccardMount() – mount a DOS file system........................................................................ 365
pccardSramEnabler() – enable the PCMCIA-SRAM driver.................................................... 366
pccardTffsEnabler() – enable the PCMCIA-TFFS driver ....................................................... 366
pciAutoAddrAlign() – align a PCI address and check boundary conditions...................... 367
pciAutoBusNumberSet() – set the primary, secondary, and subordinate bus number ......... 367
pciAutoCfg() – automatically configure all nonexcluded PCI headers................................ 368
pciAutoCfgCtl() – set or get pciAutoConfigLib options .................................................... 369
pciAutoConfigLibInit() – initialize PCI autoconfig library ............................................... 377
pciAutoDevReset() – quiesce a PCI device and reset all writeable status bits..................... 377
pciAutoFuncDisable() – disable a specific PCI function.................................................... 378
pciAutoFuncEnable() – perform final configuration and enable a function....................... 378
pciAutoGetNextClass() – find the next device of specific type from probe list.................. 379
pciAutoRegConfig() – assign PCI space to a single PCI base address register.................... 379
pciInit() – initialize the PCIC chip..................................................................................... 380
pciConfigBdfPack() – pack parameters for the Configuration Address Register.................. 380
pciConfigCmdWordShow() – show the decoded value of the command word...................... 381
pciConfigExtCapFind() – find extended capability in ECP linked list.................................. 381
pciConfigForeachFunc() – check condition on specified bus ............................................. 382
pciConfigFuncShow() – show configuration details about a function................................ 382
pciConfigInByte() – read one byte from the PCI configuration space................................ 383
pciConfigInLong() – read one longword from the PCI configuration space....................... 383
pciConfigInWord() – read one word from the PCI configuration space............................. 384
pciConfigLibInit() – initialize the configuration access-method and addresses.................. 384
pciConfigModifyByte() – perform a masked longword register update.............................. 385
pciConfigModifyLong() – perform a masked longword register update.............................. 386
pciConfigModifyWord() – perform a masked longword register update.............................. 387
pciConfigOutByte() – write one byte to the PCI configuration space.................................. 388
pciConfigOutLong() – write one longword to the PCI configuration space......................... 388
pciConfigOutWord() – write one 16-bit word to the PCI configuration space....................... 389
pciConfigReset() – disable cards for warm boot................................................................. 389
pciConfigStatusWordShow() – show the decoded value of the status word......................... 390
pciConfigTopoShow() – show PCI topology......................................................................... 390
pciShow() – show all configurations of the PCIC chip......................................................... 391
pciDevConfig() – configure a device on a PCI bus.............................................................. 391
pciDeviceShow() – print information about PCI devices.................................................... 392
pciFindClass() – find the nth occurrence of a device by PCI class code............................ 392
pciFindClassShow() – find a device by 24-bit class code.................................................... 393
pciFindDevice() – find the nth device with the given device & vendor ID............................ 393
pciFindDeviceShow() – find a device by deviceld, then print an information...................... 394
pciHeaderShow() – print a header of the specified PCI device............................................ 394
pciInt() – interrupt handler for shared PCI interrupt......................................................... 395
pciIntConnect() – connect the interrupt handler to the PCI interrupt............................... 395
pciIntDisconnect() – disconnect the interrupt handler (obsolete)..................................... 396
pciInitLibInit() – initialize the pcilnLib module .......................................................... 397
pciSpecialCycle() – generate a special cycle with a message .................................... 397
pciScaI() – handle task-level PCMCIA events ...................................................... 398
pcmciaInit() – initialize the PCMCIA event-handling package .......................... 398
pcmciaShow() – show all configurations of the PCMCIA chip ....................... 399
pcmciaShowInit() – initialize all show routines for PCMCIA drivers .......... 399
ppc403DevInit() – initialize the serial port unit ................................................. 400
ppc403DummyCallback() – dummy callback routine ....................................... 400
ppc403IntExt() – handle error interrupts ............................................................ 401
ppc403IntRd() – handle a receiver interrupt ....................................................... 401
ppc403IntWr() – handle a transmitter interrupt ................................................. 402
ppc555SciDevInit() – initialize a PPC555SCI channel .................................... 402
ppc555SciDevInit2() – initialize a PPC555SCI, part 2 ........................................ 403
ppc555SciInt() – handle a channel’s interrupt .................................................... 403
ppc860DevInit() – initialize the SMC ................................................................. 404
ppc860Int() – handle an SMC interrupt .............................................................. 404
sa1100DevInit() – initialize an SA1100 channel .............................................. 405
sa1100Int() – handle an interrupt .................................................................... 405
sab82532DevInit() – initialize an SAB82532 channel ...................................... 406
sab82532Int() – interrupt level processing ....................................................... 406
sh7615EndLoad() – initialize the driver and device .......................................... 407
shSciDevInit() – initialize a on-chip serial communication interface .......... 407
shSciDevInit2() – initialize a on-chip serial communication interface .......... 408
shSciIntErr() – handle a channel’s error interrupt ........................................ 408
shSciIntRcv() – handle a channel’s receive-character interrupt .................... 409
shSciIntTx() – handle a channel’s transmitter-ready interrupt ....................... 409
shSciIntErr() – handle a channel’s error interrupt ........................................ 410
shSciIntRcv() – handle a channel’s receive-character interrupt .................... 410
shSciIntTx() – handle a channel’s transmitter-ready interrupt ....................... 411
slattach() – publish the sl network interface and initialize the driver and device .... 411
slipBaudSet() – set the baud rate for a SLIP interface ...................................... 412
slipDelete() – delete a SLIP interface ................................................................ 412
slipInit() – initialize a SLIP interface ............................................................. 413
smcFdc37b78xDevCreate() – set correct IO port addresses for super I/O chip ... 414
smcFdc37b78xInit() – initialize Super I/O chip Library ..................................... 414
smcFdc37b78xKbdInit() – initialize the keyboard controller ............................. 415
smNetShow() – show information about a shared memory network ................ 415
sn83932EndLoad() – initialize the driver and device ......................................... 416
snattach() – publish the sn network interface and initialize the driver and device .... 416
sramDevCreate() – create a PCMCIA memory disk device ............................ 417
sramDrv() – install a PCMCIA SRAM memory driver ...................................... 417
sramMap() – map PCMCIA memory onto a specified ISA address space ........ 418
st16552DevInit() – initialize an ST16552 channel ............................................ 418
st16552Int() – interrupt level processing .......................................................... 419
st16552IntEx( ) – miscellaneous interrupt processing ................................................. 419
st16552IntRd( ) – handle a receiver interrupt................................................................. 420
st16552IntWr( ) – handle a transmitter interrupt ......................................................... 420
st16552MuxInt( ) – multiplexed interrupt level processing ........................................... 421
sym895CtrlCreate( ) – create a structure for a SYM895 device................................... 421
sym895CtrlInit( ) – initialize a SCSI Controller Structure............................................. 423
sym895GPIOConfig( ) – configure general purpose pins GPIO 0-4................................. 423
sym895GPIOCtrl( ) – controls general purpose pins GPIO 0-4 ........................................ 424
sym895HwInit( ) – hardware initialization for the 895 Chip .......................................... 425
sym895Intr( ) – interrupt service routine for the SCSI Controller ............................... 425
sym895Loopback( ) – this routine performs loopback diagnostics on 895 chip .......... 426
sym895Show( ) – display values of all readable SYM 53C8xx SIOP registers ................ 428
tcicInit( ) – initialize the TCIC chip ............................................................................. 430
tcicShow( ) – show all configurations of the TCIC chip .............................................. 430
ultraAttach( ) – publish ultra interface and initialize device ........................................ 431
ultraLoad( ) – initialize the driver and device ............................................................... 431
ultraPut( ) – copy a packet to the interface .................................................................... 432
ultraShow( ) – display statistics for the ultra network interface .................................... 432
vgaInit( ) – initialize the VGA chip and loads font in memory ..................................... 433
wd33c93CtrlCreate( ) – create and partially initialize a WD33C93 SBIC structure ...... 434
wd33c93CtrlCreateScsi2( ) – create and partially initialize an SBIC structure .............. 435
wd33c93CtrlInit( ) – initialize the user-specified fields in an SBIC structure ................. 437
wd33c93Show( ) – display the values of all readable WD33C93 chip registers .............. 438
wdbEndPktDevInit( ) – initialize an END packet device ............................................... 439
wdbNetromPktDevInit( ) – initialize a NETROM packet device for the WDB agent .. 439
wdbPipePktDevInit( ) – initialize a pipe packet device .................................................... 440
wdbSlipPktDevInit( ) – initialize a SLIP packet device for a WDB agent ...................... 440
wdbTsfsDrv( ) – initialize the TSFS device driver for a WDB agent .............................. 441
wdbUlippktDevInit( ) – initialize the communication functions for ULIP ................... 441
wdbVioDrv( ) – initialize the tty driver for a WDB agent .............................................. 442
z8530DevInit( ) – initialize a Z8530_DUSART ............................................................. 443
z8530Int( ) – handle all interrupts in one vector ......................................................... 443
z8530IntEx( ) – handle error interrupts ....................................................................... 444
z8530IntRd( ) – handle a receiver interrupt ................................................................... 444
z8530IntWr( ) – handle a transmitter interrupt ............................................................. 445
aic7880CtrlCreate()

NAME    aic7880CtrlCreate() – create a control structure for the AIC 7880

SYNOPSIS AIC_7880_SCSI_CTRL * aic7880CtrlCreate
             (int busNo,                /* PCI bus Number */
              int devNo,                /* PCI device Number */
              int scsiBusId             /* SCSI Host Adapter Bus Id */
             )

DESCRIPTION This routine creates an AIC_7880_SCSI_CTRL structure and must be called before using the
SCSI Host Adapter chip. It must be called exactly once for a specified Host Adapter.

RETURNS A pointer to the AIC_7880_SCSI_CTRL structure, or NULL if memory is unavailable or there
are invalid parameters.

SEE ALSO aic7880Lib

aic7880dFifoThresholdSet()

NAME    aic7880dFifoThresholdSet() – set the data FIFO threshold

SYNOPSIS STATUS aic7880dFifoThresholdSet
             (SCSI_CTRL * pScsiCtrl,    /* ptr to SCSI controller */
              UBYTE       threshHold    /* data FIFO threshold value */
             )

DESCRIPTION This routine specifies to the AIC-7880 host adapter how to manage its data FIFO. Below is
a description of the threshold values for SCSI reads and writes.

SCSI READS
– 0 Xfer data from FIFO as soon as it is available.
– 1 Xfer data from FIFO as soon as the FIFO is half full.
– 2 Xfer data from FIFO as soon as the FIFO is 75% full.
– 3 Xfer data from FIFO as soon as the FIFO is 100% full.

SCSI WRITES
– 0 Xfer data as soon as there is room in the FIFO.
VxWorks Drivers API Reference, 5.5

aic7880EnableFast20()

NAME
aic7880EnableFast20() – enable double speed SCSI data transfers

SYNOPSIS
VOID aic7880EnableFast20

(SCSI_CTRL * pScsiCtrl,  /* ptr to SCSI controller */
 BOOL enable        /* enable = 1 / disable = 0 */
)

DESCRIPTION
This routine enables double speed SCSI data transfers for the SCSI host adapter. This allows the host adapter to transfer data up to 20 MB/s for an 8 bit device and up to 40 MB/s for a 16 bit device.

RETURNS
N/A

SEE ALSO
aic7880Lib

aic7880GetNumOfBuses()

NAME
aic7880GetNumOfBuses() – perform a PCI bus scan

SYNOPSIS
DWORD aic7880GetNumOfBuses ()

DESCRIPTION
This routine provides a callback mechanism from the HIM to the OSM. It allows the OSM to scan the PCI bus, before the HIM is allowed to perform the bus scan.

RETURNS
0x55555555 if the OSM is not able to conduct its own bus scan.

SEE ALSO
aic7880Lib
aic7880ReadConfig()

NAME  
aic7880ReadConfig() – read from PCI config space

SYNOPSIS

DWORD aic7880ReadConfig
{
    cfp_struct * configPtr, /* ptr to cf_struct */
    UBYTE busNo, /* PCI bus number */
    UBYTE devNo, /* PCI device number */
    UBYTE regNo /* register */
}

DESCRIPTION

This routine provides a callback mechanism from the HIM to the OSM. The purpose of this routine is to allow the OSM to do its own Read access of the PCI configuration space. If the OSM cannot successfully complete the Read access, the OSM returns 0x55555555. If this happens the HIM attempts to conduct the configuration space Read access.

RETURNS

Value read or 0x55555555, if the OSM is not able to conduct read access to the PCI configuration space.

SEE ALSO

aic7880Lib

aic7880ScbCompleted()

NAME  
aic7880ScbCompleted() – successfully completed execution of a client thread

SYNOPSIS

VOID aic7880ScbCompleted
{
    sp_struct * pScb /* ptr to completed SCSI Command Block */
}

DESCRIPTION

This routine is called from within the context of the ISR. The HIM calls this routine passing in the pointer of the of the completed SCB. This routine sets the thread status, handles the completed SCB and returns program control back to the HIM which then returns from the PH_IntHandler() routine.

This routine could be called more than once from the same PH_IntHandler() call. Each call to this routine indicates the completion of an SCB. For each SCB completed, this routine sets the event type and calls the appropriate AIC–7880 event handler routines which sets the SCSI Controller, SCSI Physical Device and SCSI Thread, state variables
appropriately. This routine also handles synchronization with the SCSI Manager so that
the next runnable thread can be scheduled for execution.

RETURNS
N/A

SEE ALSO
aic7880Lib

aic7880WriteConfig()

NAME
daic7880WriteConfig() – read to PCI config space

SYNOPSIS
DWORD aic7880WriteConfig
{
    cfp struct * config_ptr, /* ptr to cf struct */
    UBYTE busNo, /* PCI bus number */
    UBYTE devNo, /* PCI device number */
    UBYTE regNo, /* register */
    DWORD regVal /* register value */
}

DESCRIPTION
This routine provides a callback mechanism from the HIM to the OSM. The purpose of
this routine is to allow the OSM to do its own write access of the PCI configuration space.
If the OSM cannot successfully complete the write access, the OSM returns 0x55555555. If
this happens the HIM attempts to conduct the configuration space write access.

RETURNS
OK or 0x55555555, if the OSM is not able to conduct write access to the PCI configuration
space.

SEE ALSO
aic7880Lib

ambaDevInit()

NAME
ambaDevInit() – initialize an AMBA channel

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


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

SEE ALSO
ambaSio

### ambaIntRx()

**NAME**
ambaIntRx() – handle a receiver interrupt

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

**DESCRIPTION**
This routine handles read interrupts from the UART.

**RETURNS**
N/A

**SEE ALSO**
ambaSio

### ambaIntTx()

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

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

**DESCRIPTION**
This routine handles write interrupts from the UART.

**RETURNS**
N/A

**SEE ALSO**
ambaSio
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 */
 int nBlocks, /* number of blocks on device, 0 = use entire disk */
 int 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.

SEE ALSO
ataDrv, dosFsMkfs(), dosFsDevInit(), rt11FsDevInit(), rt11FsMkfs(), rawFsDevInit()
ataDriveInit()

NAME
ataDriveInit() – initialize ATA drive

SYNOPSIS
STATUS ataDriveInit
   (int ctrl,
    int drive)

DESCRIPTION
This routine checks the drive presents, identifies its type, initializes the drive controller
and driver control structures.

RETURNS
OK, if drive was initialized successfully, or ERROR.

SEE ALSO
ataDrv

ataDrv()

NAME
ataDrv() – initialize the ATA driver

SYNOPSIS
STATUS ataDrv
   (int ctrl,                 /* controller no. */
    int drives,               /* number of drives */
    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/IDE/ATAPI CDROM driver, sets up interrupt vectors,
and performs hardware initialization of the ATA/IDE chip.

This routine must be called exactly once, before any reads, writes, or calls to
ataDevCreate(). Normally, it is called by usrRoot() in usrConfig.c.

RETURNS
OK, or ERROR if initialization fails.

SEE ALSO
ataDrv, ataDevCreate()
 ataRawio()

NAME
ataRawio() – do raw I/O access

SYNOPSIS
 STATUS ataRawio
   ( 
      int ctrl,
      int drive,
      ATA_RAW * pAtaRaw
   )

DESCRIPTION
This routine is called to perform raw I/O access.
drive is a drive number for the hard drive: it must be 0 or 1.
The pAtaRaw is a pointer to the structure ATA_RAW which is defined in ataDrv.h.

RETURNS
OK, or ERROR if the parameters are not valid.

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.

SEE ALSO
ataShow
### ataShowInit()

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

**SYNOPSIS**
```c
void 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

**SEE ALSO**
ataShow

### auDump()

**NAME**
auDump() – display device status

**SYNOPSIS**
```c
void auDump
    (    int unit
    )
```

**SEE ALSO**
auEnd
auEndLoad()

NAME
auEndLoad() – initialize the driver and device

SYNOPSIS
END_OBJ * auEndLoad
(char * initString /* string to be parse by the driver */)  

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


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.

SEE ALSO
auEnd

auInitParse()

NAME
auInitParse() – parse the initialization string

SYNOPSIS
STATUS auInitParse
(AU1000_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


240
2: Routines

auInitParse( )

**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.

**SEE ALSO**
auEnd
cd2400HrdInit()

NAME

cd2400HrdInit() – initialize the chip

SYNOPSIS

void cd2400HrdInit

(CD2400_QUSART * pQusart /* chip to reset */)

DESCRIPTION

This routine initializes the chip and the four channels.

SEE ALSO

cd2400Sio

---

cd2400Int()

NAME

cd2400Int() – handle special status interrupts

SYNOPSIS

void cd2400Int

(CD2400_CHAN * pChan)

DESCRIPTION

This routine handles special status interrupts from the MPCC.

SEE ALSO

cd2400Sio

---

cd2400IntRx()

NAME

cd2400IntRx() – handle receiver interrupts

SYNOPSIS

void cd2400IntRx

(CD2400_CHAN * pChan)

DESCRIPTION

This routine handles the interrupts for all channels for a Receive Data Interrupt.

SEE ALSO

cd2400Sio
cd2400IntTx( )

NAME

cd2400IntTx() – handle transmitter interrupts

SYNOPSIS

void cd2400IntTx
    (    
        CD2400_CHAN * pChan
    )

DESCRIPTION

This routine handles transmitter interrupts from the MPCC.

SEE ALSO

cd2400Sio

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.

SEE ALSO

cisLib

cisConfigregSet( )

NAME

cisConfigregSet() – set the PCMCIA configuration register

SYNOPSIS

STATUS cisConfigregSet
    (    
        int    sock,               /* socket no. */
        int    reg,                /* register no. */
    )
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

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.

SEE ALSO  cisLib
cisShow()

NAME  
cisShow() – show CIS information

SYNOPSIS  
void cisShow
           (  
             int sock             /* sock 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

SEE ALSO  
cisShow

coldfireAcr()

NAME  
coldfireAcr() – return aux control register contents

SYNOPSIS  
UCHAR coldfireAcr
           (  
             COLDFIRE_CHAN * pChan  
           )

DESCRIPTION  
This routine returns the auxiliary control register contents. The acr is not directly readable,
a copy of the last value written is kept in the UART data structure.

RETURNS  
Returns auxiliary control register (acr) contents.

SEE ALSO  
coldfireSio
coldfireAcrSetClr()

NAME  
coldfireAcrSetClr() – set and clear bits in the UART’s aux control register

SYNOPSIS  
void coldfireAcrSetClr  
  (  
    COLDIRE_CHAN * pChan,  
    UCHAR setBits, /* which bits to set in the ACR */  
    UCHAR clearBits /* which bits to clear in the ACR */  
  )

DESCRIPTION  
This routine sets and clears bits in the UART’s ACR. This routine sets and clears bits in a local copy of the ACR, then writes that local copy to the UART. This means that all changes to the ACR must go through this routine. Otherwise, any direct changes to the ACR would be lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit-fields by specifying the field mask as the clear bits.

RETURNS  
N/A

SEE ALSO  
coldfireSio

coldfireDevInit()

NAME  
coldfireDevInit() – initialize a COLDIRE_CHAN

SYNOPSIS  
void coldfireDevInit  
  (  
    COLDIRE_CHAN * pChan  
  )

DESCRIPTION  
The BSP must have already initialized all the device addresses, etc. in COLDIRE_CHAN structure. This routine initializes some transmitter & receiver status values to be used in the interrupt mask register and then resets the chip to a quiescent state.

RETURNS  
N/A

SEE ALSO  
coldfireSio
coldfireDevInit2()

NAME
coldfireDevInit2() – initialize a COLDFIRE_CHAN, part 2

SYNOPSIS
void coldfireDevInit2
    (COLDFIRE_CHAN * pChan)

DESCRIPTION
This routine is called as part of sysSerialHwInit2() and tells the driver that interrupt
vectors are connected and that it is safe to allow interrupts to be enabled.

RETURNS
N/A

SEE ALSO
coldfireSio

coldfireImr()

NAME
coldfireImr() – return current interrupt mask register contents

SYNOPSIS
UCHAR coldfireImr
    (COLDFIRE_CHAN * pChan)

DESCRIPTION
This routine returns the interrupt mask register contents. The imr is not directly readable,
a copy of the last value written is kept in the UART data structure.

RETURNS
Returns interrupt mask register contents.

SEE ALSO
coldfireSio
coldfireImrSetClr()

NAME
coldfireImrSetClr() – set and clear bits in the UART’s interrupt mask register

SYNOPSIS
void coldfireImrSetClr
    (COLDFIRE_CHAN * pChan,
     UCHAR setBits, /* which bits to set in the IMR */
     UCHAR clearBits /* which bits to clear in the IMR */
    )

DESCRIPTION
This routine sets and clears bits in the UART’s IMR.
This routine sets and clears bits in a local copy of the IMR, then writes that local copy to
the UART. This means that all changes to the IMR must go through this routine.
Otherwise, any direct changes to the IMR would be lost the next time this routine is
called.
Set has priority over clear. Thus you can use this routine to update multiple bit-fields by
specifying the field mask as the clear bits.

RETURNS
N/A

SEE ALSO
coldfireSio

coldfireInt()

NAME
coldfireInt() – handle all interrupts in one vector

SYNOPSIS
void coldfireInt
    (COLDFIRE_CHAN * pChan
    )

DESCRIPTION
All interrupts share a single interrupt vector. We identify each interrupting source and
service it. We must service all interrupt sources for those systems with edge-sensitive
interrupt controllers.

RETURNS
N/A

SEE ALSO
coldfireSio
coldfireOpr()

NAME
coldfireOpr() – return the current state of the output register

SYNOPSIS
UCHAR coldfireOpr
  (  
    COLDFIRE_CHAN * pChan  
  )

DESCRIPTION
This routine returns the current state of the output register from the saved copy in the UART data structure. The actual opr contents are not directly readable.

RETURNS
Returns the current output register state.

SEE ALSO
coldfireSio

coldfireOprSetClr()

NAME
coldfireOprSetClr() – set and clear bits in the output port register

SYNOPSIS
void coldfireOprSetClr
  (  
    COLDFIRE_CHAN * pChan,  
    UCHAR setBits, /* which bits to set in the OPR */  
    UCHAR clearBits /* which bits to clear in the OPR */  
  )

DESCRIPTION
This routine sets and clears bits in the UART’s OPR.
A copy of the current opr contents is kept in the UART data structure.

RETURNS
N/A

SEE ALSO
coldfireSio
cpmattach()

NAME
cpmattach() – publish the cpm network interface and initialize the driver

SYNOPSIS
STATUS cpmattach
(int unit, /* unit number */
SCC * pScc, /* address of SCC parameter RAM */
SCC_REG * pSccReg, /* address of SCC registers */
VOIDFUNCPT * ivec, /* interrupt vector offset */
SCC_BUF * txBdBase, /* transmit buffer descriptor base address */
SCC_BUF * rxBdBase, /* receive buffer descriptor base address */
int txBdNum, /* number of transmit buffer descriptors */
int rxBdNum, /* number of receive buffer descriptors */
UINT8 * bufBase /* address of memory pool; NONE = malloc it */
)

DESCRIPTION
The routine publishes the cpm interface by filling in a network Interface Data Record (IDR) and adding this record to the system’s interface list.

The SCC shares a region of memory with the driver. The caller of this routine can specify the address of a shared, non-cacheable memory region with bufBase. If this parameter is NONE, the driver obtains this memory region by calling cacheDmaMalloc().

Non-cacheable memory space is important for cases where the SCC is operating with a processor that has a data cache.

Once 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. Additional buffers are reserved as receive loaner buffers. The number of loaner buffers is the lesser of rxBdNum and a default value of 16.

The user must specify the location of the transmit and receive BDs in the CPU’s dual-ported RAM. txBdBase and rxBdBase give the base address 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.

Up to four individual device units are supported by this driver. Device units may reside on different processor chips, or may be on different SCCs within a single CPU.

Before this routine returns, it calls cpmReset() and cpmInit() to configure the Ethernet controller, and connects the interrupt vector ivec.

RETURNS
OK or ERROR.

SEE ALSO
if_cpm, ifLib, Motorola MC68360 User’s Manual, Motorola MPC821 and MPC860 User’s Manual
cpmStartOutput( )

NAME
cpmStartOutput( ) – output packet to network interface device

SYNOPSIS
#define BSD43_DRIVER LOCAL void cpmStartOutput
    (int unit /* unit number */)

DESCRIPTION
cpmStartOutput( ) takes a packet from the network interface output queue, copies the mbuf chain into an interface buffer, and sends the packet over the interface. etherOutputHookRtns are supported.

Collision stats are collected in this routine from previously sent BDs. These BDs will not be examined until after the transmitter has cycled the ring, coming upon the BD after it has been sent. Thus, collision stat collection will be delayed a full cycle through the Tx ring.

This routine is called from several possible threads. Each one will be described below.

The first, and most common thread, is when a user task requests the transmission of data. Under BSD 4.3, this will cause cpmOutput( ) to be called, which calls ether_output( ), which usually calls this routine. This routine will not be called if ether_output( ) finds that our interface output queue is full. In this very rare case, the outgoing data will be thrown out. BSD 4.4 uses a slightly different model in which the generic ether_output( ) routine is called directly, followed by a call to this routine.

The second thread is when a transmitter error occurs that causes a TXE event interrupt. This happens for the following errors: transmitter under run, retry limit reached, late collision, and heartbeat error. The ISR sets the txStop flag to stop the transmitter until the errors are serviced. These events require a RESTART command of the transmitter, which occurs in the cpmTxRestart( ) routine. After the transmitter is restarted, cpmTxRestart( ) does a netJobAdd of cpmStartOutput( ) to send any packets left in the interface output queue. Thus, the second thread executes in the context of netTask( ).

The third, and most unlikely, thread occurs when this routine is executing and it runs out of free Tx BDs. In this case, this routine turns on transmit interrupt and exits. When the next BD is actually sent, an interrupt occurs. The ISR does a netJobAdd of cpmStartOutput( ) to continue sending packets left in the interface output queue. Once again, we find ourselves executing in the context of netTask( ).

RETURNS
N/A

SEE ALSO
if_cpm
csAttach( )

NAME  

csAttach( ) – publish the cs network interface and initialize the driver

SYNOPSIS

```c
STATUS csAttach
     (  
         int    unit,              /* unit number */
         int    ioAddr,            /* base IO address */
         int    intVector,         /* interrupt vector, or zero */
         int    intLevel,          /* interrupt level */
         int    memAddr,           /* base memory address */
         int    mediaType,         /* 0: Autodetect 1: AUI 2: BNC 3: RJ45 */
         int    configFlags,       /* configuration flag */
         char * pEnetAddr          /* ethernet address */
     )
```

DESCRIPTION

This routine is a major entry point to this network interface driver and is called only once per operating system reboot by the operating system startup code. This routine is called before the csInit( ) routine.

This routine takes passed-in configuration parameters and parameters from the EEPROM and fills in the instance global variables in the cs_softc structure. These variables are later used by the csChipInit( ) routine.

This routine connects the interrupt handler, csIntr( ), to the specified interrupt vector, initializes the 8259 PIC and resets the CS8900 chip.

Finally, this routine calls the ether_attach( ) routine, to fill in the ifnet structure and attach this network interface driver to the system. The driver’s main entry points (csInit( ), csLocl(), csOutput( ), csReset( )) are made visible to the protocol stack.

Refer to "man if_cs" for detailed description of the configuration flags.

RETURNS

OK or ERROR.

SEE ALSO

if_cs
csShow( )

NAME  csShow( ) – shows statistics for the cs network interface

SYNOPSIS  void csShow
       (  
           int  unit,               /* interface unit */
           BOOL  zap                 /* zero totals */
       )

DESCRIPTION  This routine displays statistics about the cs Ethernet network interface. It has two parameters:

unit
       Interface unit; should be 0.

zap
       If 1, all collected statistics are cleared to zero.

RETURNS  N/A

SEE ALSO  if_cs

ctB69000VgaInit( )

NAME  ctB69000VgaInit( ) – initialize 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

SEE ALSO  ctB69000Vga
NAME
dcattach() – publish the dc network interface.

SYNOPSIS

```
STATUS dcattach
    (
        int    unit,    /* unit number */
        ULONG  devAdrs,  /* device I/O address */
        int    ivec,    /* interrupt vector */
        int    ilevel,  /* interrupt level */
        char * memAdrs,  /* address of memory pool (-1 = malloc it) */
        ULONG  memSize,  /* only used if memory pool is NOT malloc() */
        int    memWidth, /* byte-width of data (-1 = any width) */
        ULONG  pciMemBase,  /* main memory base as seen from PCI bus */
        int    dcOpMode   /* mode of operation */
    )
```

DESCRIPTION

This routine publishes the dc interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

The `unit` parameter is used to specify the device unit to initialize.

The `devAdrs` is used to specify the I/O address base of the device.

The `ivec` parameter is used to specify the interrupt vector associated with the device interrupt.

The `ilevel` parameter is used to specify the level of the interrupt which the device would use.

The `memAdrs` parameter can be used to specify the location of the memory that will be shared between the driver and the device. The value NONE is used to indicate that the driver should obtain the memory.

The `memSize` parameter is valid only if the `memAdrs` parameter is not set to NONE, in which case `memSize` indicates the size of the provided memory region.

The `memWidth` parameter sets the memory pool’s data port width (in bytes); if it is NONE, any data width is used.

The `pciMemBase` parameter defines the main memory base as seen from PCI bus.

The `dcOpMode` parameter defines the mode in which the device should be operational.

BUGS

To zero out DEC 21x4x data structures, this routine uses `bzero()`, which ignores the `memWidth` specification and uses any size data access to write to memory.
dcReadAllRom()

NAME
dcReadAllRom() – read entire serial rom

SYNOPSIS
void dcReadAllRom(
    ULONG   devAdrs,          /* device base I/O address */
    UCHAR * buffer,           /* destination buffer */
    int     cnt               /* Amount to extract in bytes */
)

DESCRIPTION
Function to read all of serial rom and store the data in the data structure passed to the
function. The count value will indicate how much of the serial rom to read. The routine
with also swap the bytes as the come in.

RETURNS
N/A.

SEE ALSO
if_dc
dcViewRom()

NAME
dcViewRom() – display lines of serial ROM for dec21140

SYNOPSIS
int dcViewRom
(    
    ULONG devAdrs,            /* device base I/O address */
    UCHAR lineCnt,            /* Serial ROM line Number */
    int   cnt                 /* Amount to display */
)

RETURNS
Number of bytes displayed.

SEE ALSO
if_dc

dec21x4xEndLoad()

NAME
dec21x4xEndLoad() – initialize the driver and device

SYNOPSIS
END_OBJ * dec21x4xEndLoad
(    
    char * initStr        /* 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 the initString.

This routine can be called in two modes. If it is called with an empty, but allocated string
then it places the name of this device (i.e. dc) into the initString and returns 0.

If the string is allocated then 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.

SEE ALSO
dec21x4xEnd
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.

SEE ALSO
dec21x40End

dec21x40PhyFind()

NAME

dec21x40PhyFind() – find the first PHY connected to DEC MII port

SYNOPSIS

UINT8 dec21x40PhyFind

(DRV_CTRL * pDrvCtrl)

RETURNS

Address of PHY or 0xFF if not found.

SEE ALSO
dec21x40End
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.

SEE ALSO
dec21x40End

dec21145SPIReadBack()

NAME
dec21145SPIReadBack() – read all PHY registers out

SYNOPSIS
void dec21145SPIReadBack
        (  
            DRV_CTRL * pDrvCtrl /* pointer to DRV_CTRL structure */
        )

RETURNS
Nothing.

SEE ALSO
dec21x40End
dummyCallback()

NAME       dummyCallback() – dummy callback routine
SYNOPSIS   STATUS dummyCallback (void)
RETURNS    ERROR.
SEE ALSO   shSciSio
**eexattach( )**

**NAME**

eexattach() – publish the eex network interface and initialize the driver and device

**SYNOPSIS**

```c
STATUS eexattach
(  int unit,                 /* unit number */
   int port,                 /* base I/O address */
   int ivec,                 /* interrupt vector number */
   int ilevel,               /* interrupt level */
   int nTfds,
   int attachment            /* 0=default, 1=AUI, 2=BNC, 3=TPE */
 )
```

**DESCRIPTION**

The routine publishes the eex interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

**RETURNS**

OK or ERROR.

**SEE ALSO**

if_eex, ifLib

---

**eexTxStartup( )**

**NAME**

eexTxStartup() – start output on the chip

**SYNOPSIS**

```c
#ifdef BSD43_DRIVER static void eexTxStartup
(  int unit
 )
#endif
```

**DESCRIPTION**

Looks for any action on the queue, and begins output if there is anything there. This routine is called from several possible threads. Each will be described below.

The first, and most common thread, is when a user task requests the transmission of data. Under BSD 4.3, this will cause eexOutput() to be called, which will cause ether_output() to be called, which will cause this routine to be called (usually). This routine will not be called if ether_output() finds that our interface output queue is full. In this case, the outgoing data will be thrown out. BSD 4.4 uses a slightly different model in which the generic ether_output() routine is called directly, followed by a call to this routine.
The second, and most obscure thread, is when the reception of certain packets causes an immediate (attempted) response. For example, ICMP echo packets (ping), and ICMP "no listener on that port" notifications. All functions in this driver that handle the reception side are executed in the context of netTask(). Always. So, in the case being discussed, netTask() will receive these certain packets, cause IP to be stimulated, and cause the generation of a response to be sent. We then find ourselves following the thread explained in the second example, with the important distinction that the context is that of netTask().

The third thread occurs when this routine runs out of TFDs and returns. If this occurs when our output queue is not empty, this routine would typically not get called again until new output was requested. Even worse, if the output queue was also full, this routine would never get called again and we would have a lock state. It DOES happen. To guard against this, the transmit clean-up handler detects the out-of-TFDs state and calls this function. The clean-up handler also runs from netTask.

NOTE: This function is ALWAYS called between an splnet() and an splx(). This is true because netTask(), and ether_output() take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO

if_eex

ei82596EndLoad()

NAME
ei82596EndLoad() – initialize the driver and device

SYNOPSIS

END_OBJ *ei82596EndLoad
(
    char * initString /* parameter string */
)

DESCRIPTION

This routine initializes both driver and device to an operational state using the device-specific values specified by initString. The initString parameter expects an ordered list of colon-separated values.

The format of the initString is:

unit:ivectorbus:memBase:nTfds:nRfds

unit

    Specifies the unit number for this device.

ivector

    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.
eiattach( )

sysbus
Passes in values as described in the Intel manual for the 82596. 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.

memBase
Inform the driver about the shared memory region. The 82596 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. If this parameter is set to the constant "NONE", this routine tries 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.

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 non-cacheable memory. The attributes of this memory are checked, and, if the memory is not both read- and write-coherent, this routine aborts.

RETURNS
An END object pointer or NULL.

SEE ALSO
ei82596End, ifLib, Intel 82596 User’s Manual
The 82596 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.

The *sysbus* parameter accepts values as described in the Intel manual for the 82596. 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," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use *cacheDmaMalloc()* to obtain some non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, this routine will abort and return ERROR.

**RETURNS**

OK or ERROR.

**SEE ALSO**

if_eidve, ifLib, Intel 82596 User’s Manual

eihkattach()

**NAME**

eihkattach() – publish the ei network interface and initialize the driver and device

**SYNOPSIS**

```c
STATUS eihkattach
    ( int    unit,              /* unit number */
      int    ivec,              /* interrupt vector number */
      UINT8  sysbus,            /* sysbus field of SCP */
      char * memBase,           /* address of memory pool or NONE */
      int    nTfds,             /* no. of transmit frames (0 = default) */
      int    nRfds              /* no. of receive frames (0 = default) */
    )
```

**DESCRIPTION**

This routine publishes the ei interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.
eiInt()

The 82596 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.

The `sysbus` parameter accepts values as described in the Intel manual for the 82596. 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," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use `cacheDmaMalloc()` to obtain some non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, this routine will abort and return `ERROR`.

RETURNS OK or `ERROR`.

SEE ALSO `if_eihk`, `ifLib`, Intel 82596 User's Manual
eiTxStartup()

NAME

eiTxStartup() – start output on the chip

SYNOPSIS

void eiTxStartup

(  
   DRV_CTRL * pDrvCtrl
)

DESCRIPTION

Looks for any action on the queue, and begins output if there is anything there. This routine is called from several possible threads. Each will be described below.

The first, and most common thread, is when a user task requests the transmission of data. This will cause eiOutput() to be called, which will cause ether_output() to be called, which will cause this routine to be called (usually). This routine will not be called if ether_output() finds that our interface output queue is full. In this case, the outgoing data will be thrown out.

The second, and most obscure thread, is when the reception of certain packets causes an immediate (attempted) response. For example, ICMP echo packets (ping), and ICMP "no listener on that port" notifications. All functions in this driver that handle the reception side are executed in the context of netTask(). Always. So, in the case being discussed, netTask() will receive these certain packets, cause IP to be stimulated, and cause the generation of a response to be sent. We then find ourselves following the thread explained in the second example, with the important distinction that the context is that of netTask().

The third thread occurs when this routine runs out of TFDs and returns. If this occurs when our output queue is not empty, this routine would typically not get called again until new output was requested. Even worse, if the output queue was also full, this routine would never get called again and we would have a lock state. It DOES happen. To guard against this, the transmit clean-up handler detects the out-of-TFDs state and calls this function. The clean-up handler also runs from netTask.

Note that this function is ALWAYS called between an splnet() and an splx(). This is true because netTask(), and ether_output() take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO

if_ei, if_eidve
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:


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.

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:


buffMultiplier

unit
    Device unit number, a small integer.

devMemAddr
    Device register base memory address

devIoAddr
    Device register base IO address

pciMemBase
    Base address of PCI memory space

vecNum
    Interrupt vector number.

intLvl
    Interrupt level.

memAdrs
    Memory pool address or NONE.

memSize
    Memory pool size or zero.

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

flags
    Device specific flags, for future use.

buffMultiplier
    Buffer Multiplier or NONE. If NONE is specified, it defaults to 2

RETURNS
OK, or ERROR if any arguments are invalid.

SEE ALSO
e13c90xEnd
elcattach()

NAME
elcattach() – publish the elc network interface and initialize the driver and device

SYNOPSIS
STATUS elcattach
(  
    int unit,              /* unit number */
    int ioAddr,            /* address of elc\xd5 s shared memory */
    int ivec,              /* interrupt vector to connect to */
    int ilevel,            /* interrupt level */
    int memAddr,           /* address of elc\xd5 s shared memory */
    int memSize,           /* size of elc\xd5 s shared memory */
    int config             /* 0: RJ45 + AUI(Thick) 1: RJ45 + BNC(Thin) */
)

DESCRIPTION
This routine attaches an elc Ethernet interface to the network if the device exists. It makes
the interface available by filling in the network interface record. The system will initialize
the interface when it is ready to accept packets.

RETURNS
OK or ERROR.

SEE ALSO
if_elc, ifLib, netShow

elcPut()

NAME
elcPut() – copy a packet to the interface.

SYNOPSIS
#ifdef BSD43_DRIVER LOCAL void elcPut
(  
    int unit
)
#endif

DESCRIPTION
Copy from mbuf chain to transmitter buffer in shared memory.

SEE ALSO
if_elc
### elcShow()

**NAME**  
elcShow() – display statistics for the SMC 8013WC elc network interface

**SYNOPSIS**  
```c
void elcShow
(  
    int  unit,                /* interface unit */
    BOOL zap                  /* 1 = zero totals */
);
```

**DESCRIPTION**  
This routine displays statistics about the elc Ethernet network interface. It has two parameters:

- **unit**  
  Interface unit; should be 0.

- **zap**  
  If 1, all collected statistics are cleared to zero.

**RETURNS**  
N/A.

**SEE ALSO**  
if_elc

---

### elt3c509Load()

**NAME**  
elt3c509Load() – initialize the driver and device

**SYNOPSIS**  
```c
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:attachmentType: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.
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:attachmentType:nRxFrames

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  
Number of Rx Frames in integer format.

RETURNS OK or ERROR for invalid arguments.

SEE ALSO elt3c509End
eltattach()

NAME
eltattach() – publish the elt interface and initialize the driver and device

SYNOPSIS
STATUS eltattach
(
    int    unit,              /* unit number */
    int    port,              /* base I/O address */
    int    ivec,              /* interrupt vector number */
    int    intLevel,          /* interrupt level */
    int    nRxFrames,         /* # of receive frames (0=default) */
    int    attachment,        /* Ethernet connector to use */
    char * ifName             /* interface name */
)

DESCRIPTION
The routine publishes the elt interface by filling in a network interface record and adding
this record to the system list. This routine also initializes the driver and the device to the
operational state.

RETURNS
OK or ERROR.

SEE ALSO
if_elt, ifLib

eltShow()

NAME
eltShow() – display statistics for the 3C509 elt network interface

SYNOPSIS
void eltShow
(
    int  unit,                /* interface unit */
    BOOL zap                  /* 1 = zero totals */
)

DESCRIPTION
This routine displays statistics about the elt Ethernet network interface.

unit  Interface unit; should be 0.
zap   If 1, all collected statistics are cleared to zero.

RETURNS
N/A

SEE ALSO
if_elt
**eltTxOutputStart()**

**NAME**
eltTxOutputStart() – start output on the board

**SYNOPSIS**
```c
#ifdef BSD43_DRIVER
static void eltTxOutputStart(int unit)
#endif
```

**DESCRIPTION**
This routine is called from ether_output() when a new packet is enqueued in the interface mbuf queue. Note that this function is always called between splnet() and splx(). This is true because netTask() and ether_output() take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

**SEE ALSO**
if_elt

---

**endEtherAddressForm()**

**NAME**
endEtherAddressForm() – form an Ethernet address into a packet

**SYNOPSIS**
```c
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-cBlk-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 is 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 or ERROR.

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

274
endTok_r()

**NAME**
endTok_r() – get a token string (modified version)

**SYNOPSIS**

```c
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.

```c
/* 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);
```

**SEE ALSO**
dec21x40End
eneattach()  

NAME  
eneattach() – publish the ene network interface and initialize the driver and device  

SYNOPSIS  
STATUS eneattach  
(  
  int unit,               /* unit number */  
  int ioAddr,             /* address of ene's shared memory */  
  int ivec,               /* interrupt vector to connect to */  
  int ilevel               /* interrupt level */  
)  

DESCRIPTION  
This routine attaches an ene Ethernet interface to the network if the device exists. It makes  
the interface available by filling in the network interface record. The system will initialize  
the interface when it is ready to accept packets.  

RETURNS  
OK or ERROR.  

SEE ALSO  
if_ene, ifLib, netShow  

enePut()  

NAME  
eneput() – copy a packet to the interface.  

SYNOPSIS  
#ifdef BSD43_DRIVER static void enePut  
(  
  int unit  
)  

DESCRIPTION  
Copy from mbuf chain to transmitter buffer in shared memory.  

SEE ALSO  
if_ene
eneShow()

NAME
eneShow() – display statistics for the NE2000 ene network interface

SYNOPSIS
void eneShow
    (             /* interface unit */
    int  unit,                  
    BOOL zap                  /* 1 = zero totals */
    )

DESCRIPTION
This routine displays statistics about the ene Ethernet network interface.

unit  Interface unit; should be 0.
zap    If 1, all collected statistics are cleared to zero.

RETURNS
N/A.

SEE ALSO
if_ene

esmcattach()

NAME
esmcattach() – publish the esmc network interface and initialize the driver

SYNOPSIS
STATUS esmcattach
    (             /* unit number */
    int unit,                 /* address of esmc\xd5 s shared memory */
    int ioAddr,               /* interrupt vector to connect to */
    int intVec,               /* interrupt level */
    int config,               /* 0: Autodetect 1: AUI 2: BNC 3: RJ45 */
    int mode                  /* 0: rx in interrupt 1: rx in task(netTask) */
    )

DESCRIPTION
This routine attaches an esmc Ethernet interface to the network if the device exists. It makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURNS
OK or ERROR.

SEE ALSO
if_esmc, ifLib, netShow
esmcPut()

NAME  esmcPut() – copy a packet to the interface

SYNOPSIS  

```c
#ifdef BSD43_DRIVER LOCAL void esmcPut
    (int unit
     )
#endif
```

DESCRIPTION  Copy from mbuf chain to transmitter buffer in shared memory.

RETURNS  N/A.

SEE ALSO  if_esmc

esmcShow()

NAME  esmcShow() – display statistics for the esmc network interface

SYNOPSIS  

```c
void esmcShow
    (int unit,                /* interface unit */
     BOOL zap                  /* zero totals */
    )
```

DESCRIPTION  This routine displays statistics about the esmc Ethernet network interface. It has two parameters:

- `unit`  Interface unit; should be 0.
- `zap`  If 1, all collected statistics are cleared to zero.

RETURNS  N/A.

SEE ALSO  if_esmc
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.

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.

SEE ALSO evbNs16550Sio
**fdDevCreate()**

**NAME**

`fdDevCreate()` – create a device for a floppy disk

**SYNOPSIS**

```c
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:

```c
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.
A pointer to a block device structure (BLK_DEV) or NULL if memory cannot be allocated for the device structure.

SEE ALSO
nec765Fd, fdDrv(), fdRawio(), dosFsMkfs(), dosFsDevInit(), rt11FsDevInit(), rt11FsMkfs(), 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.

**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.
fei82557DumpPrint()

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.

SEE ALSO nec765Fd, fdDrv(), fdDevCreate()

-----------

fei82557DumpPrint()

NAME fei82557DumpPrint() – Display statistical counters

SYNOPSIS STATUS fei82557DumpPrint
(      DRV_CTRL * pDrvCtrl       /* pointer to DRV_CTRL structure */
   )

DESCRIPTION This routine displays i82557 statistical counters

RETURNS OK, or ERROR if the DUMP command failed.

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.

282
The parameter string, initString, is an ordered list of parameters each separated by a colon. The format of initString is, "unit:memBase:memSize:nCFDs:nRFDs:flags:deviceId"

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 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," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The memSize parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive frames.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use cacheDmaMalloc() to obtain some non-cacheable memory. The attributes of this memory will be checked, and if the memory is not write coherent, this routine will abort and return ERROR.

RETURNS
An END object pointer, or NULL on error.

SEE ALSO
fei82557End, ifLib, Intel 82557 User’s Manual

---

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 \textit{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 \textbf{ERROR}.

**SEE ALSO**

\texttt{fei82557End}

---

### \texttt{feiattach( )}

**NAME**

\texttt{feiattach( )} – publish the \texttt{fei} network interface

**SYNOPSIS**

\begin{verbatim}
STATUS feiattach
(     int    unit,              /* unit number */
    char * memBase,           /* address of shared memory (NONE = malloc) */
    int    nCFD,              /* command frames (0 = default) */
    int    nRFD,              /* receive frames (0 = default) */
    int    nRFDLoan           /* loanable rx frames (0 = default, -1 = 0) */
)
\end{verbatim}

**DESCRIPTION**

This routine publishes the \texttt{fei} interface by filling in a network interface record and adding the record to the system list.

The 82557 shares a region of main memory with the CPU. The caller of this routine can specify the address of this shared memory region through the \textit{memBase} parameter; if \textit{memBase} is set to the constant \texttt{NONE}, the driver will allocate the shared memory region.

If the caller provides the shared memory region, the driver assumes that this region does not require cache coherency operations.

If the caller indicates that \texttt{feiattach( )} must allocate the shared memory region, \texttt{feiattach( )} will use \texttt{cacheDmaMalloc( )} to obtain a block of non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, \texttt{feiattach( )} will abort and return \textbf{ERROR}.

A default number of 32 command (transmit) and 32 receive frames can be selected by passing zero in the parameters \textit{nCFD} and \textit{nRFD}, respectively. If \textit{nCFD} or \textit{nRFD} is used to select the number of frames, the values should be greater than two.

A default number of 8 loanable receive frames can be selected by passing zero in the parameters \textit{nRFDLoan}, else set \textit{nRFDLoan} to the desired number of loanable receive frames. If \textit{nRFDLoan} is set to -1, no loanable receive frames will be allocated/used.
fnattach( )

NAME
fnattach( ) – publish the fn network interface and initialize the driver and device

SYNOPSIS
STATUS fnattach
(   int unit                  /* unit number */
 )

DESCRIPTION
The routine publishes the fn interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

RETURNS
OK or ERROR.

SEE ALSO
if_fn
### gei82543EndLoad()

**NAME**
gei82543EndLoad() – initialize the driver and device

**SYNOPSIS**
```c
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:

**RETURNS**
An END object pointer, NULL if error, or zero.

**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.

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.

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 then it places the name of this device (i.e. oli) into the initString and returns 0.

If the string is allocated then 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.

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.

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() – initialize 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
**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

---

**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( ) – initialize 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
In97xEndLoad()

NAME
ln97xEndLoad() – initialize the driver and device

SYNOPSIS
END_OBJ * ln97xEndLoad
  (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.

SEE ALSO
ln97xEnd

In97xInitParse()

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
ln97xInitParse()

: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.

SEE ALSO
ln97xEnd
In7990EndLoad()

NAME

In7990EndLoad() – initialize the driver and device

SYNOPSIS

END_OBJ* ln7990EndLoad
(  
  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:


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.

SEE ALSO

ln7990End

Inattach()

NAME

Inattach() – publish the ln network interface and initialize driver structures

SYNOPSIS

STATUS lnattach
(  
  int    unit,        /* unit number */
  char * devAdrs,    /* LANCE I/O address */
  int    ivec,       /* interrupt vector */
  int    ilevel,     /* interrupt level */
  char * memAdrs,    /* address of memory pool (-1 = malloc it) */
  ULONG  memSize,    /* only used if memory pool is NOT malloc()ed */
  int    memWidth,   /* byte-width of data (-1 = any width) */
  int    spare,      /* not used */
  int    spare2      /* not used */
)
This routine publishes the In interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

The *memAdrs* parameter can be used to specify the location of the memory that will be shared between the driver and the device. The value NONE is used to indicate that the driver should obtain the memory.

The *memSize* parameter is valid only if the *memAdrs* parameter is not set to NONE, in which case *memSize* indicates the size of the provided memory region.

The *memWidth* parameter sets the memory pool’s data port width (in bytes); if it is NONE, any data width is used.

To zero out LANCE data structures, this routine uses *bzero( )*, which ignores the *memWidth* specification and uses any size data access to write to memory.

**RETURNS**

OK or ERROR.

**SEE ALSO**

if_In

### InPciattach( )

**NAME**

**InPciattach( )** – publish the InPci network interface and initialize the driver and device

**SYNOPSIS**

```c
STATUS InPciattach(
    int    unit,              /* unit number */
    char * devAdrs,           /* LANCE I/O address */
    int    ivec,              /* interrupt vector */
    int    ilevel,            /* interrupt level */
    char * memAdrs,           /* address of memory pool (-1 = malloc it) */
    ULONG  memSize,           /* used if memory pool is NOT malloc()ed */
    int    memWidth,          /* byte-width of data (-1 = any width) */
    ULONG  pciMemBase,        /* memory base as seen from PCI*/
    int    spare2             /* not used */
)
```

**DESCRIPTION**

This routine publishes the In interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.
The `memAdrs` parameter can be used to specify the location of the memory that will be shared between the driver and the device. The value NONE is used to indicate that the driver should obtain the memory.

The `memSize` parameter is valid only if the `memAdrs` parameter is not set to NONE, in which case `memSize` indicates the size of the provided memory region.

The `memWidth` parameter sets the memory pool’s data port width (in bytes); if it is NONE, any data width is used.

**BUGS**
To zero out LANCE data structures, this routine uses `bzero()` which ignores the `memWidth` specification and uses any size data access to write to memory.

**RETURNS**
OK or ERROR.

**SEE ALSO**
`if_lnPci`

---

### loattach()

**NAME**
loattach() – publish the lo network interface and initialize the driver and pseudo-device

**SYNOPSIS**
```c
STATUS loattach (void)
```

**DESCRIPTION**
This routine attaches an Io Ethernet interface to the network, if the interface exists. It makes the interface available by filling in the network interface record. The system initializes the interface when it is ready to accept packets.

**RETURNS**
OK.

**SEE ALSO**
`if_loop`
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.

SEE ALSO
lptDrv, lptDevCreate()

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.

SEE ALSO
lptDrv, lptDevCreate()
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.

SEE ALSO  lptDrv
m68302SioInit( )

NAME m68302SioInit( ) – initialize a M68302_CP

SYNOPSIS void m68302SioInit

(M68302_CP * pCp

DESCRIPTION This routine initializes the driver function pointers and then resets the chip to a quiescent state. The BSP must already have initialized all the device addresses and the baudFreq fields in the M68302_CP structure before passing it to this routine. The routine resets the device and initializes everything to support polled mode (if possible), but does not enable interrupts.

RETURNS N/A

SEE ALSO m68302Sio

m68302SioInit2( )

NAME m68302SioInit2( ) – initialize a M68302_CP (part 2)

SYNOPSIS void m68302SioInit2

(M68302_CP * pCp

DESCRIPTION Enables interrupt mode of operation.

RETURNS N/A

SEE ALSO m68302Sio
m68332DevInit()

NAME
m68332DevInit() – initialize the SCC

SYNOPSIS
void m68332DevInit

M68332_CHAN * pChan

DESCRIPTION
This initializes the chip to a quiescent state.

RETURNS
N/A

SEE ALSO
m68332Sio

m68332Int()

NAME
m68332Int() – handle an SCC interrupt

SYNOPSIS
void m68332Int

M68332_CHAN * pChan

DESCRIPTION
This routine handles SCC interrupts.

RETURNS
N/A

SEE ALSO
m68332Sio
m68360DevInit()

NAME m68360DevInit() – initialize the SCC

SYNOPSIS
void m68360DevInit

     (M68360_CHAN * pChan

DESCRIPTION
This routine is called to initialize the chip to a quiescent state.

SEE ALSO m68360Sio

m68360Int()

NAME m68360Int() – handle an SCC interrupt

SYNOPSIS
void m68360Int

     (M68360_CHAN * pChan

DESCRIPTION
This routine gets called to handle SCC interrupts.

SEE ALSO m68360Sio
m68562HrdInit()

NAME

m68562HrdInit() – initialize the DUSCC

SYNOPSIS

void m68562HrdInit

(M68562_QUSART * pQusart

)

DESCRIPTION

The BSP must have already initialized all the device addresses, etc in M68562_DUSART structure. This routine resets the chip in a quiescent state.

SEE ALSO

m68562Sio

m68562RxInt()

NAME

m68562RxInt() – handle a receiver interrupt

SYNOPSIS

void m68562RxInt

(M68562_CHAN * pChan

)

RETURNS

N/A

SEE ALSO

m68562Sio
m68562RxTxErrInt()

NAME  m68562RxTxErrInt() – handle a receiver/transmitter error interrupt

SYNOPSIS  void m68562RxTxErrInt
            (  
                M68562_CHAN * pChan
            )

DESCRIPTION  Only the receive overrun condition is handled.

RETURNS  N/A

SEE ALSO  m68562Sio

m68562TxInt()

NAME  m68562TxInt() – handle a transmitter interrupt

SYNOPSIS  void m68562TxInt
            (  
                M68562_CHAN * pChan
            )

DESCRIPTION  If there is another character to be transmitted, it sends it. If not, or if a device has never been created for this channel, disable the interrupt.

RETURNS  N/A

SEE ALSO  m68562Sio
m68681Acr()

NAME
m68681Acr() – return the contents of the DUART auxiliary control register

SYNOPSIS
UCHAR m68681Acr
{
    M68681_DUART * pDuart
}

DESCRIPTION
This routine returns the contents of the auxiliary control register (ACR). The ACR is not
directly readable; a copy of the last value written is kept in the DUART data structure.

RETURNS
The contents of the auxiliary control register.

SEE ALSO
m68681Sio

m68681AcrSetClr()

NAME
m68681AcrSetClr() – set and clear bits in the DUART auxiliary control register

SYNOPSIS
void m68681AcrSetClr
{
    M68681_DUART * pDuart,
    UCHAR    setBits, /* which bits to set in the ACR */
    UCHAR    clearBits /* which bits to clear in the ACR */
}

DESCRIPTION
This routine sets and clears bits in the DUART auxiliary control register (ACR). It sets and
clears bits in a local copy of the ACR, then writes that local copy to the DUART. This
means that all changes to the ACR must be performed by this routine. Any direct changes
to the ACR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by
specifying the field mask as the clear bits.

RETURNS
N/A

SEE ALSO
m68681Sio
m68681DevInit( )

NAME
m68681DevInit() – initialize a M68681_DUART

SYNOPSIS
void m68681DevInit
{
    M68681_DUART * pDuart
}

DESCRIPTION
The BSP must already have initialized all the device addresses and register pointers in the
M68681_DUART structure as described in m68681Sio. This routine initializes some
transmitter and receiver status values to be used in the interrupt mask register and then
resets the chip to a quiescent state.

RETURNS
N/A

SEE ALSO
m68681Sio

m68681DevInit2( )

NAME
m68681DevInit2() – initialize a M68681_DUART, part 2

SYNOPSIS
void m68681DevInit2
{
    M68681_DUART * pDuart
}

DESCRIPTION
This routine is called as part of sysSerialHwInit2(). It tells the driver that interrupt
vectors are connected and that it is safe to allow interrupts to be enabled.

RETURNS
N/A

SEE ALSO
m68681Sio
m68681Imr()

NAME
m68681Imr() – return the current contents of the DUART interrupt-mask register

SYNOPSIS
UCHAR m68681Imr
    (M68681_DUART * pDuart
    )

DESCRIPTION
This routine returns the contents of the interrupt-mask register (IMR). The IMR is not directly readable; a copy of the last value written is kept in the DUART data structure.

RETURNS
The contents of the interrupt-mask register.

SEE ALSO
m68681Sio

m68681ImrSetClr()

NAME
m68681ImrSetClr() – set and clear bits in the DUART interrupt-mask register

SYNOPSIS
void m68681ImrSetClr
    (M68681_DUART * pDuart,
     UCHAR setBits, /* which bits to set in the IMR */
     UCHAR clearBits /* which bits to clear in the IMR */
    )

DESCRIPTION
This routine sets and clears bits in the DUART interrupt-mask register (IMR). It sets and clears bits in a local copy of the IMR, then writes that local copy to the DUART. This means that all changes to the IMR must be performed by this routine. Any direct changes to the IMR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS
N/A

SEE ALSO
m68681Sio
m68681Int()

NAME  m68681Int() – handle all DUART interrupts in one vector

SYNOPSIS  void m68681Int
          (  
              M68681_DUART * pDuart
          )

DESCRIPTION  This routine handles all interrupts in a single interrupt vector. It identifies and services each interrupting source in turn, using edge-sensitive interrupt controllers.

RETURNS  N/A

SEE ALSO  m68681Sio

m68681Opcr()

NAME  m68681Opcr() – return the state of the DUART output port configuration register

SYNOPSIS  UCHAR m68681Opcr
           (  
               M68681_DUART * pDuart
           )

DESCRIPTION  This routine returns the state of the output port configuration register (OPCR) from the saved copy in the DUART data structure. The actual OPCR contents are not directly readable.

RETURNS  The state of the output port configuration register.

SEE ALSO  m68681Sio
m68681OpcrSetClr()

NAME
m68681OpcrSetClr() – set and clear bits in the DUART output port configuration register

SYNOPSIS
void m68681OpcrSetClr
(void)
      (M68681_DUART * pDuart,
       UCHAR         setBits,   /* which bits to set in the OPCR */
       UCHAR         clearBits  /* which bits to clear in the OPCR */
      );

DESCRIPTION
This routine sets and clears bits in the DUART output port configuration register (OPCR). It sets and clears bits in a local copy of the OPCR, then writes that local copy to the DUART. This means that all changes to the OPCR must be performed by this routine. Any direct changes to the OPCR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS
N/A

SEE ALSO
m68681Sio

m68681Opr()

NAME
m68681Opr() – return the current state of the DUART output port register

SYNOPSIS
UCHAR m68681Opr
(void)
      (M68681_DUART * pDuart
      );

DESCRIPTION
This routine returns the current state of the output port register (OPR) from the saved copy in the DUART data structure. The actual OPR contents are not directly readable.

RETURNS
The current state of the output port register.

SEE ALSO
m68681Sio
m68681OprSetClr()

NAME
m68681OprSetClr() – set and clear bits in the DUART output port register

SYNOPSIS
void m68681OprSetClr
(M68681_DUART * pDuart,
 UCHAR setBits, /* which bits to set in the OPR */
 UCHAR clearBits /* which bits to clear in the OPR */
);

DESCRIPTION
This routine sets and clears bits in the DUART output port register (OPR). It sets and
clears bits in a local copy of the OPR, then writes that local copy to the DUART. This
means that all changes to the OPR must be performed by this routine. Any direct changes
to the OPR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by
specifying the field mask as the clear bits.

RETURNS
N/A

SEE ALSO
m68681Sio

m68901DevInit()

NAME
m68901DevInit() – initialize a M68901_CHAN structure

SYNOPSIS
void m68901DevInit
(M68901_CHAN * pChan
);

DESCRIPTION
This routine initializes the driver function pointers and then resets the chip to a quiescent
state. The BSP must have already initialized all the device addresses and the baudFreq
fields in the M68901_CHAN structure before passing it to this routine.

RETURNS
N/A

SEE ALSO
m68901Sio
mb86940DevInit()

NAME  
mb86940DevInit() – install the driver function table

SYNOPSIS  
void mb86940DevInit

(  
    MB86940_CHAN * pChan

)

DESCRIPTION  
This routine installs the driver function table. It also prevents the serial channel from functioning by disabling the interrupt.

RETURNS  
N/A

SEE ALSO  
mb86940Sio

mb86960EndLoad()

NAME  
mb86960EndLoad() – initialize the driver and device

SYNOPSIS  
END_OBJ * mb86960EndLoad

(  
    char * pInitString    /* String to be parsed by the driver. */

)

DESCRIPTION  
This routine initializes the driver and puts the device to an operational state. All of the device specific parameters are passed in via the initString, which expects a string of the following format:

unit:base_addr:int_vector:int_level

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, "fn") 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.

SEE ALSO  
mb86960End
mb86960InitParse( )

NAME
mb86960InitParse( ) – parse the initialization string

SYNOPSIS
STATUS mb86960InitParse
{
    MB86960_END_CTRL * pDrvCtrl, /* device pointer */
    char * pInitString /* information string */

DESCRIPTION
This routine parses the input string, filling in values in the driver control structure.
The initialization string format is: unit:baseAddr:ivec

    unit
    Device unit number, a small integer. Must always be 0.

    devBaseAddr
    Base address of the device register set.

    ivec
    Interrupt vector number, used with sysIntConnect( ).

RETURNS
OK or ERROR for invalid arguments.

SEE ALSO
mb86960End

mb86960MemInit( )

NAME
mb86960MemInit( ) – initialize memory for the chip

SYNOPSIS
STATUS mb86960MemInit
{
    MB86960_END_CTRL * pDrvCtrl /* device to be initialized */

DESCRIPTION
This routine is highly specific to the device.

RETURNS
OK or ERROR.

SEE ALSO
mb86960End
**mb87030CtrlCreate()**

**NAME**
mb87030CtrlCreate() – create a control structure for an MB87030 SPC

**SYNOPSIS**

```c
MB_87030_SCSI_CTRL *mb87030CtrlCreate
(
    UINT8 * spcBaseAdrs,      /* base address of SPC */
    int regOffset,        /* addr offset between consecutive regs. */
    UINT clkPeriod,        /* period of controller clock (nsec) */
    int spcDataParity,    /* type of input to SPC DP (data parity) */
    FUNCPTIR spcDMABytesIn,    /* SCSI DMA input function */
    FUNCPTIR spcDMABytesOut    /* SCSI DMA output function */
)
```

**DESCRIPTION**
This routine creates a data structure that must exist before the SPC chip can be used. This routine should be called once and only once for a specified SPC. It should be the first routine called, since it allocates memory for a structure needed by all other routines in the library.

After calling this routine, at least one call to mb87030CtrlInit() should be made before any SCSI transaction is initiated using the SPC chip.

A detailed description of the input parameters follows:

- **spcBaseAdrs**
  the address at which the CPU would access the lowest register of the SPC.

- **regOffset**
  the address offset (bytes) to access consecutive registers. (This must be a power of 2, for example, 1, 2, 4, etc.)

- **clkPeriod**
  the period in nanoseconds of the signal to the SPC clock input (only used for select command timeouts).

- **spcDataParity**
  the parity bit must be defined by one of the following constants, according to whether the input to SPC DP is GND, +5V, or a valid parity signal, respectively:

  - SPC_DATA_PARITY_LOW
  - SPC_DATA_PARITY_HIGH
  - SPC_DATA_PARITY_VALID

- **spcDmaBytesIn** and **spcDmaBytesOut**
  pointers to board-specific routines to handle DMA input and output. If these are NULL (0), SPC program transfer mode is used. DMA is possible only during SCSI data in/out phases. The interface to these DMA routines must be of the form:
mb87030CtrlInit()

NAME

mb87030CtrlInit() – initialize a control structure for an MB87030 SPC

SYNOPSIS

STATUS mb87030CtrlInit

(  MB_87030_SCSI_CTRL * pSpc,              /* ptr to SPC struct */  MB_87030_SCSI_CTRL * pSpc,              /* ptr to SPC struct */
   int                  scsiCtrlBusId,     /* SCSI bus ID of this SPC */
   UINT                 defaultSelTimeOut, /* default dev sel timeout */
                        /* (microsec) */
   int                  scsiPriority       /* priority of task doing */
                        /* SCSI I/O */
)

DESCRIPTION

This routine initializes an SPC control structure created by mb87030CtrlCreate(). It must be called before the SPC is used. This routine can be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pSpc

a pointer to the MB_87030_SCSI_CTRL structure created with mb87030CtrlCreate().

scsiCtrlBusId

the SCSI bus ID of the SIOP, in the range 0 - 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.

defaultSelTimeOut

the timeout, in microseconds, for selecting a SCSI device attached to this controller. The recommended value 0 specifies SCSI_DEF_SELECT_TIMEOUT (250 milliseconds).
The maximum timeout possible is approximately 3 seconds. Values exceeding this revert to the maximum.

\textit{scsiPriority}

the priority to which a task is set when performing a SCSI transaction. Valid priorities range from 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

\textbf{RETURNS}

\texttt{OK}, or \texttt{ERROR} if parameters are out of range.

\textbf{SEE ALSO}

\texttt{mb87030Lib}
**mbcAddrFilterSet()**

**NAME**
mbcAddrFilterSet() – set the address filter for multicast addresses

**SYNOPSIS**
```c
void mbcAddrFilterSet
(    
    MBC_DEVICE * pDrvCtrl     /* device to be updated */
(    
```

**DESCRIPTION**
This routine goes through all of the multicast addresses on the list of addresses (added with the `endAddrAdd()` routine) and sets the device’s filter correctly.

**RETURNS**
N/A.

**SEE ALSO**
mbcEnd

---

**mbcattach()**

**NAME**
mbcattach() – publish the mbc network interface and initialize the driver

**SYNOPSIS**
```c
STATUS mbcattach
(    
    int unit,             /* unit number */
    void * pEmBase,          /* ethernet module base address */
    int inum,             /* interrupt vector number */
    int txBdNum,          /* number of transmit buffer descriptors */
    int rxBdNum,          /* number of receive buffer descriptors */
    int dmaParms,         /* DMA parameters */
    UINT8 * bufBase           /* address of memory pool; NONE = malloc it */
(    
```

**DESCRIPTION**
The routine publishes the mbc interface by adding an mbc Interface Data Record (IDR) to the global network interface list.

The Ethernet controller uses buffer descriptors from an on-chip dual-ported RAM region, while the buffers are allocated in RAM external to the controller. The buffer memory pool can be allocated in a non-cacheable RAM region and passed as parameter `bufBase`. Otherwise `bufBase` is NULL and the buffer memory pool is allocated by the routine using `cacheDmaMalloc()`. The driver uses this buffer pool to allocate the specified number of 1518-byte buffers for transmit, receive, and loaner pools.
The parameters `txBdNum` and `rxBdNum` specify the number of buffers to allocate for transmit and receive. If either of these parameters is NULL, the default value of 2 is used. The number of loaner buffers allocated is the lesser of `rxBdNum` and 16.

The on-chip dual ported RAM can only be partitioned so that the maximum receive and maximum transmit BDs are:

- Transmit BDs: 8, Receive BDs: 120
- Transmit BDs: 16, Receive BDs: 112
- Transmit BDs: 32, Receive BDs: 96
- Transmit BDs: 64, Receive BDs: 64

RETURNS
ERROR, if `unit` is out of range or non-cacheable memory cannot be allocated; otherwise TRUE.

SEE ALSO
`if_mbc`, `ifLib`, Motorola MC68EN302 User’s Manual

---

**mbcEndLoad()**

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

**SYNOPSIS**
```c
END_OBJ* mbcEndLoad
    (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:
```
```

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

**SEE ALSO**
mbcEnd
### mbcIntr()

**NAME**
mbcIntr() – network interface interrupt handler

**SYNOPSIS**
```c
void mbcIntr
    (int unit                  /* unit number */)
```

**DESCRIPTION**
This routine is called at interrupt level. It handles work that requires minimal processing. Interrupt processing that is more extensive gets handled at task level. The network task, netTask(), is provided for this function. Routines get added to the netTask() work queue via the netJobAdd() command.

**RETURNS**
N/A

**SEE ALSO**
if_mbc

### mbcMemInit()

**NAME**
mbcMemInit() – initialize memory for the chip

**SYNOPSIS**
```c
STATUS mbcMemInit
    (MBC_DEVICE * pDrvCtrl     /* device to be initialized */)
```

**DESCRIPTION**
Allocates and initializes the memory pools for the mbc device.

**RETURNS**
OK or ERROR.

**SEE ALSO**
mbcEnd
mbcParse()

NAME
mbcParse() – parse the init string

SYNOPSIS
STATUS mbcParse

(MBC_DEVICE * pDrvCtrl, /* device pointer */
 char * initString /* information string */

DESCRIPTION
Parse the input string. Fill in values in the driver control structure.
The initialization string format is:


unit
Device unit number, a small integer.

memAddr
ethernet module base address.

ivec
Interrupt vector number (used with sysIntConnect)

txBdNum
transmit buffer descriptor

rxBdNum
receive buffer descriptor

dmaParms
dma parameters

bufBase
address of memory pool

offset
packet data offset

RETURNS
OK or ERROR for invalid arguments.

SEE ALSO
mbcEnd
mbcStartOutput()

NAME

mbcStartOutput() — output packet to network interface device

SYNOPSIS

#ifdef BSD43_DRIVER
LOCAL void mbcStartOutput

   (int unit /* unit number */)

#else

DESCRIPTION

mbcStartOutput() takes a packet from the network interface output queue, copies the
mbuf chain into an interface buffer, and sends the packet over the interface.
etherOutputHookRtns are supported.

Collision stats are collected in this routine from previously sent BDs. These BDs will not
be examined until after the transmitter has cycled the ring, coming upon the BD after it
has been sent. Thus, collision stat collection will be delayed a full cycle through the Tx
ring.

This routine is called under several possible scenarios. Each one will be described below.

The first, and most common, is when a user task requests the transmission of data. Under
BSD 4.3, this results in a call to mbcOutput(), which in turn calls ether_output(). The
routine, ether_output(), will make a call to mbcStartOutput() if our interface output
queue is not full, otherwise, the outgoing data is discarded. BSD 4.4 uses a slightly
different model, in which the generic ether_output() routine is called directly, followed
by a call to this routine.

The second scenario is when this routine, while executing runs out of free Tx BDs, turns
on transmit interrupts and exits. When the next BD is transmitted, an interrupt occurs and
the ISR does a netJobAdd of the routine which executes in the context of netTask() and
continues sending packets from the interface output queue.

The third scenario is when the device is reset, typically when the promiscuous mode is
altered; which results in a call to mbcInit(). This resets the device, does a netJobAdd() of
this routine to enable transmitting queued packets.

RETURNS

N/A

SEE ALSO

if_mbc
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 or ERROR.

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

320
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. User should call this routine to check link status and update phyFlags. pPhyInfo should include a valid PHY bus number (phyAddr), and include the phyFlags that was used last time to configure auto-negotiation process.

RETURNS
OK or ERROR.

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.

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.

SEE ALSO miiLib

miiPhyInit()

NAME miiPhyInit() – initialize and configure the PHY devices

SYNOPSIS STATUS miiPhyInit

SYNOPSIS

DESCRIPTION This routine scans, initializes and configures the PHY device described in phyInfo. Space for phyInfo is to be provided by the calling task.

This routine is called from the driver’s Start routine to perform media initialization and configuration. To access the PHY device through the MII-management interface, it uses the read and write routines which are provided by the driver itself in the fields phyReadRtn(), phyWriteRtn() of the phyInfo structure. Before it attempts to use this routine, the driver has to properly initialize some of the fields in the phyInfo structure, and optionally fill in others, as below:

/* fill in mandatory fields in phyInfo */
 pDrvCtrl->phyInfo->pDrvCtrl = (void *) pDrvCtrl;
 pDrvCtrl->phyInfo->phyWriteRtn = (FUNCPTR) xxxMiiWrite;
 pDrvCtrl->phyInfo->phyReadRtn = (FUNCPTR) xxxMiiRead;
/* fill in some optional fields in phyInfo */
 pDrvCtrl->phyInfo->phyFlags = 0;
 pDrvCtrl->phyInfo->phyAddr = (UINT8) MII_PHY_DEF_ADDR;
 pDrvCtrl->phyInfo->phyDefMode = (UINT8) PHY_10BASE_T;
 pDrvCtrl->phyInfo->phyAnOrderTbl = (MII_AN_ORDER_TBL *)
 &xxxPhyAnOrderTbl;
/*
 * fill in some more optional fields in phyInfo: the delay stuff
 * we want this routine to use our xxxDelay() routine, with
 * the constant one as an argument, and the max delay we may
 * tolerate is the constant MII_PHY_DEF_DELAY, in xxxDelay units
 */
pDrvCtrl->phyInfo->phyDelayRtn = (FUNCPTR) xxxDelay;
pDrvCtrl->phyInfo->phyMaxDelay = MII_PHY_DEF_DELAY;
pDrvCtrl->phyInfo->phyDelayParm = 1;
/*
 * fill in some more optional fields in phyInfo: the PHY's callback
 * to handle "link down" events. This routine is invoked whenever
 * the link status in the PHY being used is detected to be low.
 */
pDrvCtrl->phyInfo->phyStatChngRtn = (FUNCPTR) xxxRestart;

Some of the above fields may be overwritten by this routine, since for instance, the logical
address of the PHY actually used may differ from the user's initial setting. Likewise, the
specific PHY being initialized, may not support all the technology abilities the user has
allowed for its operations.

This routine first scans for all possible PHY addresses in the range 0-31, checking for an
MII-compliant PHY, and attempts at running some diagnostics on it. If none is found,
ERROR is returned.

Typically PHYs are scanned from address 0, but if the user specifies an alternative start
PHY address via the parameter phyAddr in the phyInfo structure, PHYs are scanned in
order starting with the specified PHY address. In addition, if the flag MII_ALL_BUS_SCAN
is set, this routine will scan the whole bus even if a valid PHY has already been found,
and stores bus topology information. If the flags MII_PHY_ISO, MII_PHY_PWR_DOWN are
set, all of the PHYs found but the first will be respectively electrically isolated from the
MII interface and/or put in low-power mode. These two flags are meaningless in a
configuration where only one PHY is present.

The phyAddr parameter is very important from a performance point of view. Since the
MII management interface, through which the PHY is configured, is a very slow one,
providing an incorrect or invalid address in this field may result in a particularly long
boot process.

If the flag MII_ALL_BUS_SCAN is not set, this routine will assume that the first PHY found
is the only one.

This routine then attempts to bring the link up. This routine offers two strategies to select
a PHY and establish a valid link. The default strategy is to use the standard 802.3 style
auto-negotiation, where both link partners negotiate all their technology abilities at the
same time, and the highest common denominator ability is chosen. Before the
auto-negotiation is started, the next-page exchange mechanism is disabled.

If GMII interface is used, users can specify it through user flags — MII_PHY_GMII_TYPE.
The user can prevent the PHY from negotiating certain abilities via user flags — MII_PHY_FD, MII_PHY_100, MII_PHY_HD, and MII_PHY_10. as well as MII_PHY_1000T_HD and MII_PHY_1000T_FD if GMII is used. When MII_PHY_FD is not specified, full duplex will not be negotiated; when MII_PHY_HD is not specified half duplex will not be negotiated, when MII_PHY_100 is not specified, 100Mbps ability will not be negotiated; when MII_PHY_10 is not specified, 10Mbps ability will not be negotiated. Also, if GMII is used, when MII_PHY_1000T_HD is not specified, 1000T with half duplex mode will not be negotiated. Same thing applied to 1000T with full duplex mode via MII_PHY_1000T_FD.

Flow control ability can also be negotiated via user flags — MII_PHY_TX_FLOW_CTRL and MII_PHY_RX_FLOW_CTRL. For symmetric PAUSE ability (MII), user can set/clean both flags together. For asymmetric PAUSE ability (GMII), user can separate transmit and receive flow control ability. However, user should be aware that flow control ability is meaningful only if full duplex mode is used.

When MII_PHY_TBL is set in the user flags, the BSP specific table whose address may be provided in the phyAnOrderTbl field of the phyInfo structure, is used to obtain the list, and the order of technology abilities to be negotiated. The entries in this table are ordered such that entry 0 is the highest priority, entry 1 in next and so on. Entries in this table may be repeated, and multiple technology abilities can be OR’d to create a single entry. If a PHY cannot support a ability in an entry, that entry is ignored.

If no PHY provides a valid link, and if MII_PHY_DEF_SET is set in the phyFlags field of the PHY_INFO structure, the first PHY that supports the default abilities defined in the phyDefMode of the phyInfo structure will be selected, regardless of the link status.

In addition, this routine adds an entry in a linked list of PHY devices for each active PHY it found. If the flag MII_PHY_MONITOR is set, the link status for the relevant PHY is continually monitored for a link down event. If such event is detected, and if the phyLinkDownRtn in the PHY_INFO structure is a valid function pointer, then the routine it points at is executed in the context of the netTask(). The parameter MII_MONITOR_DELAY may be used to define the period in seconds with which the link status is checked. Its default value is 5.

**RETURNS**

OK or ERROR if the PHY could not be initialized,

**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 employs more than one type of network device requiring a PHY-specific registers handler use this routine instead of miiPhyOptFuncSet() to ensure device specific handlers and to avoid overwriting one’s with the other’s.

PROTECTION DOMAINS

(VxAE) This function can only be called from within the kernel protection domain. The argument optRegsFunc must be a pointer to function in the kernel protection domain.

RETURNS

N/A.

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 is 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.
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, or ERROR in case of fatal errors.

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.

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.

SEE ALSO
miiLib

motCpmEndLoad()

NAME
motCpmEndLoad() – initialize the driver and device

SYNOPSIS
END_OBJ *motCpmEndLoad
  (  
      char * initString /* parameter string */ 
  )

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:


The parameters of this string are individually described in the reference entry for
motCpmEnd.

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. \texttt{txBdBase} and \texttt{rxBdBase} give the offsets from \texttt{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 \texttt{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 \textit{ivec}.

**RETURNS**

An END object pointer or NULL on error.

**SEE ALSO**

\texttt{motFccEndLoad}(), \texttt{Motorola MC68EN360 User’s Manual}, \texttt{Motorola MPC860 User’s Manual}, \texttt{Motorola MPC821 User’s Manual}

---

**motFccEndLoad()**

**NAME**

\texttt{motFccEndLoad}() – initialize the driver and device

**SYNOPSIS**

\begin{verbatim}
END_OBJ* motFccEndLoad
    (char * initString)

DESCRIPTION**

This routine initializes both driver and device to an operational state using device specific parameters specified by \textit{initString}.

The parameter string, \textit{initString}, is an ordered list of parameters each separated by a colon. The format of \textit{initString} is:

\begin{verbatim}
"immrVal:ivec:bufBase:bufSize:fifoTxBase:fifoRxBase:
tbdNum:rbdNum:phyAddr:phyDefMode:pAnOrderTbl:userFlags"
\end{verbatim}

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 \texttt{tbdNum} and \texttt{rbdNum}. In other cases, the number of buffers selected should be greater than two.
The `bufBase` parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant `NONE`, then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The `bufSize` parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive buffer descriptors.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use `cacheDmaMalloc()` to obtain some cache-safe memory. The attributes of this memory will be checked, and if the memory is not write coherent, this routine will abort and return `NULL`.

**RETURNS**

An END object pointer, or `NULL` on error.

**SEE ALSO**

`motFccEnd`, `ifLib`, *MPC8260 Power QUICC II User’s Manual*

---

**motFecEndLoad()**

**NAME**

`motFecEndLoad()` – initialize the driver and device

**SYNOPSIS**

```c
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:

```
```

The FEC shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.

A default number of transmit/receive buffer descriptors of 32 can be selected by passing zero in the parameters `tbdNum` and `rbdNum`. In other cases, the number of buffers selected should be greater than two.
The `bufBase` parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used. The `bufSize` parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive buffer descriptors.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use `cacheDmaMalloc()` to obtain some cache-safe memory. The attributes of this memory will be checked, and if the memory is not write coherent, this routine will abort and return `NULL`.

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

**SEE ALSO**
motFecEnd, ifLib, MPC860T Fast Ethernet Controller (Supplement to MPC860 User’s Manual)
n72001DevInit( )

NAME  n72001DevInit( ) – initialize a N72001_MPS

SYNOPSIS  void n72001DevInit
           (  
               N72001_MPS * pMpsc /* serial device descriptor */
           )

DESCRIPTION  The BSP must have already initialized all the device addresses, etc. in N72001_MPS structure. This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state.

SEE ALSO  n72001Sio

n72001Int( )

NAME  n72001Int( ) – interrupt level processing

SYNOPSIS  void n72001Int
           (  
               N72001_MPS * pMpsc /* serial device descriptor */
           )

DESCRIPTION  This routine handles interrupts from MPSC channels.

 On some boards, all MPSC interrupts for both ports share a single interrupt vector. This is the ISR for such boards. We determine from the parameter which MPSC interrupted, then look at the code to find out which channel and what kind of interrupt.

RETURNS  N/A

SEE ALSO  n72001Sio
n72001IntRd()

NAME
n72001IntRd() – handle a receiver interrupt

SYNOPSIS
void n72001IntRd
(  
    N72001_CHAN * pChan /* MPSC channel descriptor */
)

DESCRIPTION
This routine handles read interrupts from the MPSC.

RETURNS
N/A

SEE ALSO
n72001Sio

n72001IntWr()

NAME
n72001IntWr() – handle a transmitter interrupt

SYNOPSIS
void n72001IntWr
(  
    N72001_CHAN * pChan /* MPSC channel descriptor */
)

DESCRIPTION
This routine handles write interrupts from the MPSC.

RETURNS
N/A

SEE ALSO
n72001Sio
ncr710CtrlCreate()  

**NAME**  
ncr710CtrlCreate() – create a control structure for an NCR 53C710 SIOP

**SYNOPSIS**  
NCR_710_SCSI_CTRL *ncr710CtrlCreate
  (  
    UINT8 * baseAdrs,         /* base address of the SIOP */  
    UINT    freqValue         /* clock controller period (nsec* 100) */  
  )

**DESCRIPTION**  
This routine creates an SIOP data structure and must be called before using an SIOP chip. It should be called once and only once for a specified SIOP. Since it allocates memory for a structure needed by all routines in ncr710Lib, it must be called before any other routines in the library. After calling this routine, ncr710CtrlInit() should 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 register of the SIOP.

**freqValue**  
The value at the SIOP SCSI clock input. This is used to determine the clock period for the SCSI core of the chip and the synchronous divider value for synchronous transfer. It is important to have the right timing on the SCSI bus. The `freqValue` parameter is defined as the SCSI clock input value, in nanoseconds, multiplied by 100. Several `freqValue` constants are defined in `ncr710.h` as follows:

- NCR710_1667MHZ 5998 /* 16.67Mhz chip */
- NCR710_20MHZ 5000 /* 20Mhz chip */
- NCR710_25MHZ 4000 /* 25Mhz chip */
- NCR710_3750MHZ 2666 /* 37.50Mhz chip */
- NCR710_40MHZ 2500 /* 40Mhz chip */
- NCR710_50MHZ 2000 /* 50Mhz chip */
- NCR710_66MHZ 1515 /* 66Mhz chip */
- NCR710_6666MHZ 1500 /* 66.66Mhz chip */

**RETURNS**  
A pointer to the NCR_710_SCSI_CTRL structure, or NULL if memory is insufficient or parameters are invalid.

**SEE ALSO**  
ncr710Lib
ncr710CtrlCreateScsi2()

NAME
ncr710CtrlCreateScsi2() – create a control structure for the NCR 53C710 SIOP

SYNOPSIS
NCR_710_SCSI_CTRL *ncr710CtrlCreateScsi2

(  
  UINT8 * baseAdrs,         /* base address of the SIOP */
  UINT    clkPeriod         /* clock controller period (nsec* 100) */
)

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 ncr710Lib, it must be called before any other routines in the library. After calling this routine, ncr710CtrlInitScsi2() 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 ncr710.h as follows:

NCR710_1667MHZ 6000 /* 16.67Mhz chip */
NCR710_20MHZ 5000 /* 20Mhz chip */
NCR710_25MHZ 4000 /* 25Mhz chip */
NCR710_3750MHZ 2667 /* 37.50Mhz chip */
NCR710_40MHZ 2500 /* 40Mhz chip */
NCR710_50MHZ 2000 /* 50Mhz chip */
NCR710_66MHZ 1515 /* 66Mhz chip */
NCR710_6666MHZ 1500 /* 66.66Mhz chip */

RETURNS
A pointer to the NCR_710_SCSI_CTRL structure, or NULL if memory is unavailable or there are invalid parameters.

SEE ALSO
ncr710Lib2

334
**ncr710CtrlInit()**

**NAME**

ncr710CtrlInit() – initialize a control structure for an NCR 53C710 SIOP

**SYNOPSIS**

```c
STATUS ncr710CtrlInit
    ( NCR_710_SCSI_CTRL * pSiop,        /* ptr to SIOP struct */
      int              scsiCtrlBusId,    /* SCSI bus ID of this SIOP */
      int              scsiPriority     /* priority of task when doing */
                            /* SCSI I/O */
    )
```

**DESCRIPTION**

This routine initializes an SIOP structure, after the structure is created with ncr710CtrlCreate(). This structure must be initialized before the SIOP can be used. It may be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

* `pSiop` - A pointer to the NCR_710_SCSI_CTRL structure created with ncr710CtrlCreate().
* `scsiCtrlBusId` - The SCSI bus ID of the SIOP, in the range 0 - 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.
* `scsiPriority` - The priority to which a task is set when performing a SCSI transaction. Valid priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

**RETURNS**

OK, or ERROR if parameters are out of range.

**SEE ALSO**

ncr710Lib
ncr710CtrlInitScsi2()

NAME
ncr710CtrlInitScsi2( ) – initialize a control structure for the NCR 53C710 SIOP

SYNOPSIS
STATUS ncr710CtrlInitScsi2
( 
  NCR_710_SCSI_CTRL * pSiop, /* ptr to SIOP struct */
  int scsiCtrlBusId, /* SCSI bus ID of this SIOP */
  int scsiPriority /* task priority when doing SCSI I/O */
)

DESCRIPTION
This routine initializes an SIOP structure after the structure is created with ncr710CtrlCreateScsi2( ). 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_710_SCSI_CTRL structure created with ncr710CtrlCreateScsi2( ).

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.

scsiPriority
This parameter is ignored. All SCSI I/O is now done in the context of the SCSI manager task; if necessary, the priority of the manager task may be changed using taskPrioritySet( ) or by setting the value of the global variable ncr710ScsiTaskPriority before calling ncr710CtrlCreateScsi2( ).

RETURNS
OK, or ERROR if the parameters are out of range.

SEE ALSO
ncr710Lib2, ncr710CtrlCreateScsi2( )
ncr710SetHwRegister( )

NAME
ncr710SetHwRegister( ) – set hardware-dependent registers for the NCR 53C710 SIOP

SYNOPSIS
STATUS ncr710SetHwRegister

(   SIOP * pSiop,  /* pointer to SIOP info */
    NCR710_HW_REGS * pHwRegs  /* pointer to NCR710_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 board support package. The input parameters are as follows:

pSiop
Pointer to the NCR_710_SCSI_CTRL structure created with ncr710CtrlCreate().

pHwRegs
Pointer to a NCR710_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 ncr710CtlrCreate() and ncr710CrtlInit() is [0,0,0,0,1,0,0,0,0,0,0,0,0,1,0].

typedef struct

{   int ctest4Bit7;    /* host bus multiplex mode */
    int ctest7Bit7;    /* disable/enable burst cache capability */
    int ctest7Bit6;    /* snoo control bit1 */
    int ctest7Bit5;    /* snoo control bit0 */
    int ctest7Bit1;    /* invert ttl pin (sync bus host mode only) */
    int ctest7Bit0;    /* enable differential SCSI bus capability */
    int ctest8Bit0;    /* set snoo pins mode */
    int dmodeBit7;     /* burst length transfer bit 1 */
    int dmodeBit6;     /* burst length transfer bit 0 */
    int dmodeBit5;     /* function code bit FC2 */
    int dmodeBit4;     /* function code bit FC1 */
    int dmodeBit3;     /* program data bit (FC0) */
    int dmodeBit1;     /* user-programmable transfer type */
    int dcntlBit5;     /* enable ACK pin */
    int dcntlBit1;     /* enable fast arbitration on host port */
 } NCR710_HW_REGS;

For a more detailed description of the register bits, see the NCR 53C710 SCSI I/O Processor Programming Guide.
NOTE: Because this routine writes to the NCR 53C710 chip registers, it cannot be used when there is any SCSI bus activity.

RETURNS
OK, or ERROR if an input parameter is NULL.

SEE ALSO
ncr710Lib, ncr710CtlrCreate(), NCR 53C710 SCSI I/O Processor Programming Guide

ncr710SetHwRegisterScsi2()

NAME
ncr710SetHwRegisterScsi2() – set hardware-dependent registers for the NCR 53C710

SYNOPSIS
STATUS ncr710SetHwRegisterScsi2
(   SIOP * pSiop,   /* pointer to SIOP info */
   NCR710_HW_REGS * pHwRegs  /* pointer to a NCR710_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
Pointer to the NCR_710_SCSI_CTRL structure created with ncr710CtlCreateScsi2().

pHwRegs
Pointer to a NCR710_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 ncr710CtlrCreateScsi2() and ncr710CtlrInitScsi2() is {0,0,0,0,1,0,0,0,0,0,0,0,0,1,0}.

typedef struct
{
   int ctest4Bit7;   /* Host bus multiplex mode */
   int ctest7Bit7;   /* Disable/enable burst cache capability */
   int ctest7Bit6;   /* Snoop control bit1 */
   int ctest7Bit5;   /* Snoop control bit0 */
   int ctest7Bit1;   /* invert tt1 pin (sync bus host mode only)*/
   int ctest7Bit0;   /* enable differential scsi bus capability*/
   int ctest8Bit0;   /* Set snoop pins mode */
   int dmodeBit7;   /* Burst Length transfer bit 1 */
   int dmodeBit6;   /* Burst Length transfer bit 0 */
}
int dmodeBit5;    /* Function code bit FC2 */
int dmodeBit4;    /* Function code bit FC1 */
int dmodeBit3;    /* Program data bit (FC0) */
int dmodeBit1;    /* user programmable transfer type */
int dcntlBit5;    /* Enable Ack pin */
int dcntlBit1;    /* Enable fast arbitration on host port */
} NCR710_HW_REGS;

For a more detailed explanation of the register bits, refer to the NCR 53C710 SCSI I/O Processor Programming Guide.

**NOTE:** Because this routine writes to the chip registers you cannot use it if there is any SCSI bus activity.

**RETURNS**
OK, or ERROR if any input parameter is NULL.

**SEE ALSO**
cr710Lib2, ncr710CtrlCreateScsi2(), NCR 53C710 SCSI I/O Processor Programming Guide

---

### ncr710Show()

**NAME**
ncr710Show() – display the values of all readable NCR 53C710 SIOP registers

**SYNOPSIS**

```c
STATUS ncr710Show
(  
    SCSI_CTRL * pScsiCtrl     /* ptr to SCSI controller info */
)
```

**DESCRIPTION**

This routine displays the state of the NCR 53C710 SIOP registers in a user-friendly manner. It is useful primarily for debugging. The input parameter is the pointer to the SIOP information structure returned by the ncr710CtrlCreate() call.

**NOTE:** The only readable register during a script execution is the Istat register. If this routine is used during the execution of a SCSI command, the result could be unpredictable.

**EXAMPLE**

```c
-> ncr710Show
NCR710 Registers
----------------
0xff47000: Sien = 0xa5 Sdid = 0x00 Scntl1 = 0x00 Scntl0 = 0x04
0xff47004: Socl = 0x00 Sodl = 0x00 Sxfer = 0x80 Scid = 0x80
0xff47008: Sbc1 = 0x00 Sbd1 = 0x00 Sidl = 0x00 Sfbr = 0x00
0xff4700c: Sstat2 = 0x00 Sstat1 = 0x00 Sstat0 = 0x00 Dstat = 0x80
0xff47010: Dsa = 0x00000000
```
ncr710ShowScsi2() – display the values of all readable NCR 53C710 SIOP registers

Synopsis

```
STATUS ncr710ShowScsi2
    (  
      SCSI_CTRL * pScsiCtrl  /* ptr to SCSI controller info */
    )
```

Description

This routine displays the state of the NCR 53C710 SIOP registers in a user-friendly way. It is primarily used for debugging. The input parameter is the pointer to the SIOP information structure returned by the ncr710CtrlCreateScsi2() 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

```c
-> ncr710Show
NCR710 Registers
--------------
0xfff47000: Sien = 0xa5 Sdid = 0x00 Scntl1 = 0x00 Scntl0 = 0x04
0xfff47004: Socl = 0x00 Socl = 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
```
ncr710SingleStep()

NAME
ncr710SingleStep() – perform a single-step

SYNOPSIS
void ncr710SingleStep
    (SIOP * pSiop,             /* pointer to SIOP info */
     BOOL   verbose            /* show all registers */
    )

DESCRIPTION
This routine performs a single-step by writing the STD bit in the DCNTL register. The
pSiop parameter is a pointer to the SIOP information. Before executing, enable the
single-step facility by calling ncr710StepEnable().

RETURNS
N/A

SEE ALSO
ncr710CommLib, ncr710StepEnable()
ncr710StepEnable()

NAME  ncr710StepEnable() – enable/disable script single-step

SYNOPSIS  void ncr710StepEnable
            (  
                SIOP * pSiop,             /* pointer to SIOP info */
                BOOL   boolValue          /* TRUE/FALSE to enable/disable single step */
            )

DESCRIPTION  This routine enables/disables the single-step facility on the chip. It also unmasks/masks the single-step interrupt in the Dien register. Before executing any SCSI routines, enable the single-step facility by calling ncr710StepEnable() with boolValue set to TRUE. To disable, call it with boolValue set to FALSE.

RETURNS  N/A.

SEE ALSO  ncr710CommLib, ncr710SingleStep()

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.

342
Routines

ncr810CtrlInit()

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.

SEE ALSO
ncr810Lib

ncr810CtrlInit()

NAME
ncr810CtrlInit() – initialize a control structure for the NCR 53C8xx SIOP

SYNOPSIS

STATUS ncr810CtrlInit

(    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
Pointer to the NCR_810_SCSI_CTRL structure created with ncr810CtrlCreate().
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.

SEE ALSO
ncr810Lib

ncr810SetHwRegister( )

NAME
ncr810SetHwRegister( ) – set hardware-dependent registers for the NCR 53C8xx SIOP

SYNOPSIS
STATUS ncr810SetHwRegister
   (   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
   Pointer to the NCR_810_SCSI_CTRL structure created with ncr810CrtlCreate().

pHwRegs
   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 ncr810CrtlCreate() 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.

SEE ALSO
ncr810Lib, ncr810.h, ncr810CtlrCreate()

ncr810Show()

NAME
ncr810Show() – display values of all readable NCR 53C8xx SIOP registers

SYNOPSIS
STATUS ncr810Show
  (    
        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 ncr810CtlrCreate() call.

NOTE: The only readable register during a script execution is the Istat register. If you use this routine during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE
-> ncr810Show
NCR810 Registers
-----------------           
0xffff47000: Sien       = 0xa5 Sdid  = 0x00 Scntl1 = 0x00 Scntl0 = 0x04
0xffff47004: Soc1      = 0x00 Sod1  = 0x00 Sxfer = 0x80 Scid   = 0x80
0xffff47008: Sbc1      = 0x00 Sbd1  = 0x00 Sid1  = 0x00 Sfbr   = 0x00
0xffff4700c: Sstat2    = 0x00 Sstat1 = 0x00 Sstat0 = 0x00 Dstat  = 0x80
0xffff47010: Dsa       = 0x00000000
0xffff47014: Ctest3    = ???? Ctest2 = 0x21 Ctest1 = 0xf0 Ctest0 = 0x00
0xffff47018: Ctest7    = 0x32 Ctest6 = ???? Ctest5  = 0x00 Ctest4 = 0x00
0xffff4701c: Temp      = 0x00000000
0xffff47020: Lcrc      = 0x00 Ctest8 = 0x00 Istat  = 0x00 Dfifo  = 0x00
0xffff47024: Dcmd/Ddc=  0x50000000

345
VxWorks Drivers API Reference, 5.5
ncr5390CtrlCreate( )

ncr5390CtrlCreate( )
ncr5390CtrlCreate( ) – create a control structure for an NCR 53C90 ASC

SYNOPSIS

NCR_5390_SCSI_CTRL *ncr5390CtrlCreate
(  
    UINT8 * baseAdrs,         /* base address of ASC */
    int regOffset,        /* addr offset between consecutive regs. */
    UINT clkPeriod,        /* period of controller clock (nsec) */
    FUNCPTR ascDmaBytesIn,    /* SCSI DMA input function */
    FUNCPTR ascDmaBytesOut    /* SCSI DMA output function */
)

DESCRIPTION

This routine creates a data structure that must exist before the ASC chip can be used. This
routine must be called exactly once for a specified ASC, and must be the first routine
called, since it allocates a structure needed by all other routines in the library.

The input parameters are as follows:

baseAdrs
    The address at which the CPU would access the lowest register of the ASC.

regOffset
    The address offset (bytes) to access consecutive registers. (This must be a power of 2,
    for example, 1, 2, 4, etc.)

clkPeriod
    The period, in nanoseconds, of the signal to the ASC clock input (used only for select
    command timeouts).

ascDmaBytesIn and ascDmaBytesOut
    Board-specific parameters to handle DMA input and output. If these are NULL (0),
    ASC program transfer mode is used. DMA is possible only during SCSI data in/out

RETURNS

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

SEE ALSO

ncr810Lib, ncr810CtrlCreate( )
phases. The interface to these DMA routines must be of the form:

```c
STATUS xxDmaBytes(In, Out)
{
    SCSI_PHYS_DEV *pScsiPhysDev, /* ptr to phys dev info */
    UINT8          *pBuffer,      /* ptr to the data buffer */
    int            bufLength      /* number of bytes to xfer */
}
```

**RETURNS**

A pointer to an NCR_5390_SCSI_CTRL structure, or NULL if memory is insufficient or the parameters are invalid.

**SEE ALSO**

ncr5390Lib1

---

### ncr5390CtrlCreateScsi2()

**NAME**

ncr5390CtrlCreateScsi2() – create a control structure for an NCR 53C90 ASC

**SYNOPSIS**

```c
NCR_5390_SCSI_CTRL *ncr5390CtrlCreateScsi2
(
    UINT8*  baseAdrs,           /* base address of ASC */
    int     regOffset,          /* offset between consecutive regs. */
    UINT    clkPeriod,          /* period of controller clock (nsec) */
    UINT    sysScsiDmaMaxBytes, /* maximum byte count using DMA */
    FUNCPTR sysScsiDmaStart,    /* function to start SCSI DMA xfer */
    FUNCPTR sysScsiDmaAbort,    /* function to abort SCSI DMA xfer */
    int     sysScsiDmaArg       /* argument to pass to above funcs. */
)
```

**DESCRIPTION**

This routine creates a data structure that must exist before the ASC chip can be used. This routine must be called exactly once for a specified ASC, and must be the first routine called, since it calloc’s a structure needed by all other routines in the library.

The input parameters are as follows:

* `baseAdrs`:
  The address at which the CPU would access the lowest register of the ASC.

* `regOffset`:
  The address offset (bytes) to access consecutive registers.

* `clkPeriod`:
  The period, in nanoseconds, of the signal to the ASC clock input.
Board-specific routines to handle DMA transfers to and from the ASC; if the maximum DMA byte count is zero, programmed I/O is used. Otherwise, non-NULL function pointers to DMA start and abort routines must be provided. The specified argument is passed to these routines when they are called; it may be used to identify the DMA channel to use, for example. The interface to these DMA routines must be of the form:

```c
STATUS xxDmaStart (arg, pBuffer, bufLength, direction)  
  int arg;                     /* call-back argument */  
  UINT8 *pBuffer;              /* ptr to the data buffer */  
  UINT bufLength;              /* number of bytes to xfer */  
  int direction;  
  /* 0 = SCSI->mem, 1 = mem->SCSI */
```

```c
STATUS xxDmaAbort (arg)  
  int arg;                    /* call-back argument */
```

Implementation details for the DMA routines can be found in the specific DMA driver for that board.

**NOTE:** If there is no DMA interface, synchronous transfers are not supported. This is a limitation of the NCR5390 hardware.

**RETURNS**
A pointer to an `NCR_5390_SCSI_CTRL` structure, or `NULL` if memory is insufficient or the parameters are invalid.

**SEE ALSO**
ncr5390Lib2
ncr5390CtrlInit( )

NAME
ncr5390CtrlInit( ) – initialize the user-specified fields in an ASC structure

SYNOPSIS
STATUS ncr5390CtrlInit
    (int * pAsc,               /* ptr to ASC info */
     int scsiCtrlBusId,       /* SCSI bus ID of this ASC */
     UINT defaultSelTimeOut, /* default dev. select timeout (microsec) */
     int scsiPriority         /* priority of task when doing SCSI I/O */
    )

DESCRIPTION
This routine initializes an ASC structure, after the structure is created with
ncr5390CtrlCreate(). This structure must be initialized before the ASC can be used. It may
be called more than once; however, it should be called only while there is no activity on
the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all
attached devices.

The input parameters are as follows:

pAsc
    Pointer to the NCR5390_SCSI_CTRL structure created with ncr5390CtrlCreate().

scsiCtrlBusId
    The SCSI bus ID of the ASC, in the range 0 - 7. The ID is somewhat arbitrary; the
    value 7, or highest priority, is conventional.

defaultSelTimeOut
    The timeout, in microseconds, for selecting a SCSI device attached to this controller.
    This value is used as a default if no timeout is specified in scsiPhysDevCreate(). The
    recommended value zero (0) specifies SCSI_DEF_SELECT_TIMEOUT (250 millisec).
    The maximum timeout possible is approximately 2 seconds. Values exceeding this
    revert to the maximum.

scsiPriority
    The priority to which a task is set when performing a SCSI transaction. Valid
    priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not
    be altered during SCSI transactions.

RETURNS
OK, or ERROR if a parameter is out of range.

SEE ALSO
ncr5390Lib, scsiPhysDevCreate()
ncr5390Show()

NAME
ncr5390Show() – display the values of all readable NCR5390 chip registers

SYNOPSIS
int ncr5390Show
                   (  
           int * pScsiCtrl          /* ptr to SCSI controller info */ 
                   )

DESCRIPTION
This routine displays the state of the ASC registers in a user-friendly manner. It is useful primarily for debugging. It should not be invoked while another running process is accessing the SCSI controller.

EXAMPLE
-> ncr5390Show
REG #00 (Own ID ) = 0x07
REG #01 (Control ) = 0x00
REG #02 (Timeout Period ) = 0x20
REG #03 (Sectors ) = 0x00
REG #04 (Heads ) = 0x00
REG #05 (Cylinders MSB ) = 0x00
REG #06 (Cylinders LSB ) = 0x00
REG #07 (Log. Addr. MSB ) = 0x00
REG #08 (Log. Addr. 2SB ) = 0x00
REG #09 (Log. Addr. 3SB ) = 0x00
REG #0a (Log. Addr. LSB ) = 0x00
REG #0b (Sector Number ) = 0x00
REG #0c (Head Number ) = 0x00
REG #0d (Cyl. Number MSB) = 0x00
REG #0e (Cyl. Number LSB) = 0x00
REG #0f (Target LUN ) = 0x00
REG #10 (Command Phase ) = 0x00
REG #11 (Synch. Transfer) = 0x00
REG #12 (Xfer Count MSB ) = 0x00
REG #13 (Xfer Count 2SB ) = 0x00
REG #14 (Xfer Count LSB ) = 0x00
REG #15 (Destination ID ) = 0x03
REG #16 (Source ID ) = 0x00
REG #17 (SCSI Status ) = 0x42
REG #18 (Command ) = 0x07

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

SEE ALSO
ncr5390Lib
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 device to the operational state. All of the device specific parameters are passed in the initString. The string contains the target specific parameters in the following format:

"unit:register addr:int vector:int level:shmem addr:shmem size:shmem width"

RETURNS An END object pointer or NULL on error.

SEE ALSO ne2000End

nicEndLoad()

NAME nicEndLoad() – initialize the driver and device

SYNOPSIS END_OBJ* nicEvbEndLoad

(char* initString /* string to be parse by the driver */
)

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

unit:base_addr:int_vector:int_level

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.

SEE ALSO nicEvbEnd
nicEvbattach()

NAME

nicEvbattach() – publish and initialize the nicEvb network interface driver

SYNOPSIS

STATUS nicEvbattach

( int          unit,        /* unit number */
  NIC_DEVICE * pNic,        /* address of NIC chip */
  int          ivec         /* interrupt vector to use */
)

DESCRIPTION

This routine publishes the nicEvb interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

RETURNS

OK, or ERROR if the receive buffer memory could not be allocated.

SEE ALSO

if_nicEvb

nicEvbInitParse()

NAME

nicEvbInitParse() – parse the initialization string

SYNOPSIS

STATUS nicEvbInitParse

( NICEVB_END_DEVICE * pDrvCtrl,
  char *              initString
)

DESCRIPTION

Parse the input string. Fill in values in the driver control structure. The initialization string format is:

unit:base_adrs:vecnum:intLvl

unit

Device unit number, a small integer.

base_adrs

Base address for NIC device.

vecNum

Interrupt vector number (used with sysIntConnect()).
**intLvl**  
Interrupt level.

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

**SEE ALSO**  
nicEvbEnd

---

### nicTxStartup()

**NAME**  
nicTxStartup() – the driver’s actual output routine

**SYNOPSIS**  
```c
#ifdef BSD43_DRIVER LOCAL STATUS nicTxStartup
    (int unit)
#endif
```

**DESCRIPTION**  
This routine accepts outgoing packets from the if_snd queue, and then gains exclusive access to the DMA (through a mutex semaphore), then calls nicTransmit() to send the packet out onto the interface.

**RETURNS**  
OK, or ERROR if the packet could not be transmitted.

**SEE ALSO**  
if_nicEvb

---

### ns16550DevInit()

**NAME**  
ns16550DevInit() – initialize an NS16550 channel

**SYNOPSIS**  
```c
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.

**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.

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.

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.

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.

SEE ALSO
ns16550Sio
ns83902EndLoad()

NAME  ns83902EndLoad() – initialize the driver and device

SYNOPSIS  

```c
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, "In") 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.

SEE ALSO  ns83902End

ns83902RegShow()

NAME  ns83902RegShow() – prints the current value of the NIC registers

SYNOPSIS  

```c
void ns83902RegShow
    (NS3902_END_DEVICE* pDrvCtrl)
```

DESCRIPTION  This routine reads and displays the register values of the NIC registers.

RETURNS  N/A.

SEE ALSO  ns83902End
### nvr4101DSIUDevInit()

**NAME**

*nvr4101DSIUDevInit() – initialize the NVR4101DSIU DSIU.*

**SYNOPSIS**

```c
void nvr4101DSIUDevInit
    (NVR4101_DSIU_CHAN * pChan)
```

**DESCRIPTION**

This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state. The caller needs to initialize the channel structure with the requested word length and parity.

**RETURNS**

N/A.

**SEE ALSO**

*nvr4101DSIUSio*

---

### nvr4101DSIUInt()

**NAME**

*nvr4101DSIUInt() – interrupt level processing*

**SYNOPSIS**

```c
void nvr4101DSIUInt
    (NVR4101_DSIU_CHAN * pChan)
```

**DESCRIPTION**

This routine handles interrupts from the DSIU.

**RETURNS**

N/A.

**SEE ALSO**

*nvr4101DSIUSio*
nvr4101DSIUIntMask( )

NAME  nvr4101DSIUIntMask( ) – mask interrupts from the DSIU.
SYNOPSIS  void nvr4101DSIUIntMask ()
DESCRIPTION  This function masks all possible interrupts from the DSIU subsystem.
RETURNS  N/A.
SEE ALSO  nvr4101DSIUSio

nvr4101DSIUIntUnmask( )

NAME  nvr4101DSIUIntUnmask( ) – unmask interrupts from the DSIU.
SYNOPSIS  void nvr4101DSIUIntUnmask ()
DESCRIPTION  This function unmasks all desired interrupts from the DSIU subsystem.
RETURNS  N/A.
SEE ALSO  nvr4101DSIUSio
nvr4101SIUCharToTxWord()

NAME
nvr4101SIUCharToTxWord() – translate character to output word format.

SYNOPSIS
UINT16 nvr4101SIUCharToTxWord
    (char outChar)

DESCRIPTION
This routine performs the bit manipulations required to convert a character into the
output word format required by the SIU. This routine only supports 8-bit word lengths,
two stop bits and no parity.

RETURNS
The translated output character.

SEE ALSO
nvr4101SIUSio

nvr4101SIUDevInit()

NAME
nvr4101SIUDevInit() – initialization of the NVR4101SIU SIU.

SYNOPSIS
void nvr4101SIUDevInit
    (NVR4101_SIU_CHAN * pChan)

DESCRIPTION
This routine initializes some SIO_CHAN function pointers and then resets the chip in a
quiescent state. No initialization of the NVR4101_SIU_CHAN structure is required before
this routine is called.

RETURNS
N/A

SEE ALSO
nvr4101SIUSio
nvr4101SIUInt( )

NAME  nvr4101SIUInt( ) – interrupt level processing

SYNOPSIS  void nvr4101SIUInt
            (  
             NVR4101_SIU_CHAN * pChan
            )

DESCRIPTION  This routine handles interrupts from the SIU.

RETURNS  N/A.

SEE ALSO  nvr4101SIUSio

nvr4101SIUIntMask( )

NAME  nvr4101SIUIntMask( ) – mask interrupts from the SIU.

SYNOPSIS  void nvr4101SIUIntMask()

DESCRIPTION  This function masks all possible interrupts from the SIU subsystem.

RETURNS  N/A.

SEE ALSO  nvr4101SIUSio
nvr4101SIUIntUnmask()

NAME
nvr4101SIUIntUnmask() – unmask interrupts from the SIU.

SYNOPSIS
void nvr4101SIUIntUnmask()

DESCRIPTION
This function unmask interrupts from the SIU subsystem.

RETURNS
N/A.

SEE ALSO
nvr4101SIUSio

nvr4102DSIUDevInit()

NAME
nvr4102DSIUDevInit() – initialize the NVR4102DSIU DSIU.

SYNOPSIS
void nvr4102DSIUDevInit
(NVR4102_DSIU_CHAN * pChan)

DESCRIPTION
This routine initializes some SIO_CHAN function pointers and then resets the chip in a quiescent state. The caller needs to initialize the channel structure with the requested word length and parity.

RETURNS
N/A.

SEE ALSO
nvr4102DSIUSio
nvr4102DSIUInt()

NAME
nvr4102DSIUInt() – interrupt level processing

SYNOPSIS
void nvr4102DSIUInt
(NVR4102_DSIU_CHAN * pChan)

DESCRIPTION
This routine handles interrupts from the DSIU.

RETURNS
N/A.

SEE ALSO
nvr4102DSIUSio

nvr4102DSIUIntMask()

NAME
nvr4102DSIUIntMask() – mask interrupts from the DSIU.

SYNOPSIS
void nvr4102DSIUIntMask()

DESCRIPTION
This function masks all possible interrupts from the DSIU subsystem.

RETURNS
N/A.

SEE ALSO
nvr4102DSIUSio
### nvr4102DSIUIntUnmask()

**NAME**

nvr4102DSIUIntUnmask() – unmask interrupts from the DSIU.

**SYNOPSIS**

```c
void nvr4102DSIUIntUnmask();
```

**DESCRIPTION**

This function unmasks all desired interrupts from the DSIU subsystem.

**RETURNS**

N/A.

**SEE ALSO**

nvr4102DSIUSio
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.

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.

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.

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.

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.

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.

SEE ALSO
pccardLib
**pciAutoAddrAlign()**

**NAME**

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

**SYNOPSIS**

```c
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 */
    )
```

**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**

```c
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
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 non-excluded 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:

   Status register.
   Command register.
   Latency timer.
   Cache line size.
   Memory and/or I/O base address and limit registers.
   Primary, secondary, subordinate bus number (for PCI-PCI bridges).
   Expansion ROM disable.
   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
pciAutoCfgCtl() can be considered analogous to ioctl() calls: the call takes arguments of (1) a pCookie, returned by pciAutoConfigLibInit(); (2) A command, macros for which are defined in pciAutoConfigLib.h; and, (3) an argument, the type of which depends on the specific command, but will always fit in a pointer variable. Currently, only globally effective commands are implemented.

The commands available are:

**PCI_FBB_ENABLE** - BOOL *pArg
**PCI_FBB_DISABLE** - void
**PCI_FBB_UPDATE** - BOOL *pArg
**PCI_FBB_STATUS_GET** - BOOL *pArg

Enable and disable the functions which check Fast Back-To-Back functionality. **PCI_FBB_UPDATE** is for use with dynamic/HA applications. It 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: In the current implementation, FBB will be enabled or disabled on the entire bus. If any device anywhere on the bus cannot support FBB, then it is not enabled, even if specific sub-busses could support it.

PCI_MAX_LATENCY_FUNC_SET - FUNC PTR * 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 - FUNC PTR * 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: 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 auto configuration 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:

```c
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:** 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:

```c
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 - FUNCPTER * 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:

```c
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:

```c
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 - FUNCPTER * 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:

```c
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:

```c
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_SYSTEM_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:

```c
void *pCookie;
pCookie = pciAutoConfigLibInit(NULL);
pciAutoCfgCtl(pCookie, PCI_SYSTEM_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.

ERRNO
   EINVAL if pCookie is not NULL or cmd is not recognized

RETURNS
   OK, or ERROR if the command or argument is invalid.

SEE ALSO
   pciAutoConfigLib
**NAME**

pciAutoConfig() – automatically configure all nonexcluded PCI headers; obsolete

**SYNOPSIS**

```c
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:

- Status register.
- Command register.
- Latency timer.
- Cache line size.
- Memory and/or I/O base address and limit registers.
- Primary, secondary, subordinate bus number (for PCI-PCI bridges).
- Expansion ROM disable.
- 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**
```c
void * pciAutoConfigLibInit

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

**DESCRIPTION**
pciAutoConfigLib initialization function.

**ERRNO**
Not set.

**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**
```c
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 in the 64-bit address space. Returns (0) otherwise.

**SEE ALSO**
pciAutoConfigLib
pcicInit()

NAME  pcicInit() – initialize the PCIC chip

SYNOPSIS  STATUS pcicInit
            (
                int     ioBase,           /* IO base address */
                int     intVec,           /* interrupt vector */
                int     intLevel,         /* interrupt level */
                FUNCPTR showRtn           /* show routine */
            )

DESCRIPTION  This routine initializes the PCIC chip.

RETURNS  OK, or ERROR if the PCIC chip cannot be found.

SEE ALSO  pcic

--------------------------------------------------------

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 prints the value in a human-readable format.

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.

ERRNO
Not set.

RETURNS
OK normally, or ERROR if funcCheckRtn( ) does not return OK.

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 prints the information in a human-readable format.

RETURNS
OK, always.

SEE ALSO
pciConfigShow
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

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
**NAME**  
pciConfigInWord() – read one word from the PCI configuration space

**SYNOPSIS**  
```c
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

---

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

**SYNOPSIS**  
```c
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 IO ports located at addresses 0x0cf8 and 0x0cfc. These two ports are:

- 32-bit configuration address port, at 0x0cf8.
- 32-bit configuration data port, at 0x0cfc.

Accessing a PCI function’s configuration port is two step process.

- Write the bus number, physical device number, function number and register number to the configuration address port.
– Perform an IO read from or an write to the configuration data port.

Configuration mechanism two uses following two single-byte IO ports.
– Configuration space enable, or CSE, register, at 0x0cf8.
– Forward register, at 0x0cfa.

To generate a PCI configuration transaction, the following actions are performed.
– Write the target bus number into the forward register.
– Write a one byte value to the CSE register at 0x0cf8. The bit pattern written to this register has three effects: disables the generation of special cycles; enables the generation of configuration transactions; specifies the target PCI functional device.
– Perform a one, two or four byte IO read or write transaction within the IO range 0xc000 through 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 are not 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

```c
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 does not 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 are not 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.

ERRNO
Not set.

RETURNS
OK, always.

SEE ALSO
pciConfigLib
pciConfigStatusWordShow()

**NAME**
pciConfigStatusWordShow() – show the decoded value of the status word

**SYNOPSIS**

```c
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 prints the value in a human-readable format.

**RETURNS**
OK, always.

**SEE ALSO**
pciConfigShow

pciConfigTopoShow()

**NAME**
pciConfigTopoShow() – show PCI topology

**SYNOPSIS**

```c
void pciConfigTopoShow ()
```

**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
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.

SEE ALSO  pcicShow

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 IO base address */
              UINT32 devMemBaseAdrs,  /* device memory base address */
              UINT32 command  /* command to issue */
          }

DESCRIPTION  This routine configures a device 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 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**

```c
STATUS pciDeviceShow
    ( int busNo                 /* bus number */
    )
```

**DESCRIPTION**

This routine prints information about PCI devices. 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 the Master Abort bit of the host/PCI bridge does not change.

**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**

```c
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 program).

The class code arg of must be carefully constructed from class and sub-class macros.

Example: To find an ethernet class device, construct the class code arg as follows:

```c
((PCI_CLASS_NETWORK_CTLR << 16 | PCI_SUBCLASS_NET_ETHERNET << 8))
```
PCI Routines

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 did not 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

pciConfigLib

393
pciFindDeviceShow()

NAME      pciFindDeviceShow() – find a device by deviceId, then print an 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 print an 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
**pcmciad()**

**NAME**

pcmciad() – handle task-level PCMCIA events

**SYNOPSIS**

```c
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.

**SEE ALSO**

pcmciaLib, pcmciaInit()

---

**pcmciaInit()**

**NAME**

pcmciaInit() – initialize the PCMCIA event-handling package

**SYNOPSIS**

```c
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.

**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

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.

SEE ALSO pcmciaShow
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.

SEE ALSO  ppc403Sio

ppc403DummyCallback()

NAME  ppc403DummyCallback() – dummy callback routine

SYNOPSIS  STATUS ppc403DummyCallback (void)

RETURNS  ERROR (always).

SEE ALSO  ppc403Sio
ppc403IntEx()

NAME
ppc403IntEx() – handle error interrupts

SYNOPSIS
void ppc403IntEx
(   PPC403_CHAN * pChan
)

DESCRIPTION
This routine handles miscellaneous interrupts on the serial communication controller.

RETURNS
N/A.

SEE ALSO
ppc403Sio

ppc403IntRd()

NAME
ppc403IntRd() – handle a receiver interrupt

SYNOPSIS
void ppc403IntRd
(   PPC403_CHAN * pChan
)

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

RETURNS
N/A.

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.

SEE ALSO
ppc403Sio

ppc555SciDevInit()

NAME
ppc555SciDevInit() – initialize a PPC555SCI channel

SYNOPSIS
void ppc555SciDevInit
    (PPC555SCI_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 PPC555SCI_CHAN structure before passing it to this routine.

RETURNS
N/A.

SEE ALSO
ppc555SciSio
**ppc555SciDevInit2()**

**NAME**
ppc555SciDevInit2() – initialize a PPC555SCI, part 2

**SYNOPSIS**
```c
void ppc555SciDevInit2
   (PPC555SCI_CHAN * pChan    /* device to initialize */
    )
```

**DESCRIPTION**
This routine is called by the BSP after interrupts have been connected. The driver can now operate in interrupt mode. Before this routine is called only polled mode operations should be allowed.

**RETURNS**
N/A.

**SEE ALSO**
ppc555SciSio

---

**ppc555SciInt()**

**NAME**
ppc555SciInt() – handle a channel’s interrupt

**SYNOPSIS**
```c
void ppc555SciInt
   (PPC555SCI_CHAN * pChan    /* channel generating the interrupt */
    )
```

**RETURNS**
N/A.

**SEE ALSO**
ppc555SciSio
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.  
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.  
SEE ALSO  ppc860Sio
sa1100DevInit()

NAME sa1100DevInit() – initialize an SA1100 channel

SYNOPSIS void sa1100DevInit

            (SA1100_CHAN * pChan /* ptr to SA1100_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 SA1100_CHAN structure.

RETURNS N/A.

SEE ALSO sa1100Sio

sa1100Int()

NAME sa1100Int() – handle an interrupt

SYNOPSIS void sa1100Int

            (SA1100_CHAN * pChan /* ptr to SA1100_CHAN describing this channel */)

DESCRIPTION This routine handles interrupts from the UART.

RETURNS N/A.

SEE ALSO sa1100Sio
sab82532DevInit( )

NAME sab82532DevInit( ) – initialize an SAB82532 channel

SYNOPSIS void sab82532DevInit

          ( 
              SAB82532_DUART * pDuart
          )

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 SAB82532_CHAN structure.

RETURNS N/A.

SEE ALSO sab82532

sab82532Int( )

NAME sab82532Int( ) – interrupt level processing

SYNOPSIS void sab82532Int

          ( 
              SAB82532_DUART * pDuart
          )

DESCRIPTION This routine handles interrupts from the UART.

RETURNS N/A.

SEE ALSO sab82532
sh7615EndLoad()

NAME  sh7615EndLoad() – initialize the driver and device

SYNOPSIS  END_OBJ* sh7615EndLoad

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:

"ivec:ilevel:numRds:numTds:phyDefMode:userFlags"

RETURNS  An END object pointer or NULL on error.

SEE ALSO  sh7615End

shSciDevInit()

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

SYNOPSIS  void shSciDevInit

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.

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.

SEE ALSO
shScifSio

shScifIntErr()

NAME
shScifIntErr() – handle a channel’s error interrupt

SYNOPSIS
void shScifIntErr
    (  
       SCIF_CHAN * pChan /* channel generating the interrupt */
    )

RETURNS
N/A.

SEE ALSO
shScifSio
### shScifIntRcv()

**NAME**

shScifIntRcv() – handle a channel’s receive-character interrupt

**SYNOPSIS**

```c
void shScifIntRcv
  (  
    SCIF_CHAN * pChan    /* channel generating the interrupt */
  )
```

**RETURNS**

N/A.

**SEE ALSO**

shScifSio

### shScifIntTx()

**NAME**

shScifIntTx() – handle a channel’s transmitter-ready interrupt

**SYNOPSIS**

```c
void shScifIntTx
  (  
    SCIF_CHAN * pChan    /* channel generating the interrupt */
  )
```

**RETURNS**

N/A.

**SEE ALSO**

shScifSio
shSciIntErr()

NAME
shSciIntErr() – handle a channel’s error interrupt

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

RETURNS
N/A.

SEE ALSO
shSciSio

shSciIntRcv()

NAME
shSciIntRcv() – handle a channel’s receive-character interrupt

SYNOPSIS
void shSciIntRcv
        (  
            SCI_CHAN * pChan  /* channel generating the interrupt */
        )

RETURNS
N/A.

SEE ALSO
shSciSio
shSciIntTx()

NAME
shSciIntTx() – handle a channel’s transmitter-ready interrupt

SYNOPSIS
void shSciIntTx
(SCI_CHAN * pChan          /* channel generating the interrupt */
);

RETURNS
N/A.

SEE ALSO
shSciSio

slattach()

NAME
slattach() – publish the sl network interface and initialize the driver and device

SYNOPSIS
STATUS slattach
(int  unit,                /* SLIP device unit number */
int  fd,                  /* fd of tty device for SLIP interface */
BOOL compressEnable,      /* explicitly enable CSLIP compression */
BOOL compressAllow,       /* enable CSLIP compression on Rx */
int  mtu                  /* user setable MTU */
);

DESCRIPTION
This routine publishes the sl interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

This routine is usually called by slipInit().

RETURNS
OK or ERROR.

SEE ALSO
if_sl
slipBaudSet()

NAME
slipBaudSet() – set the baud rate for a SLIP interface

SYNOPSIS
STATUS slipBaudSet

(int unit,                /* SLIP device unit number */
 int baud                 /* baud rate */
)

DESCRIPTION
This routine adjusts the baud rate of a tty device attached to a SLIP interface. It provides a way to modify the baud rate of a tty device being used as a SLIP interface.

RETURNS
OK, or ERROR if the unit number is invalid or uninitialized.

SEE ALSO
if_sl

slipDelete()

NAME
slipDelete() – delete a SLIP interface

SYNOPSIS
STATUS slipDelete

(int unit                   /* SLIP unit number */
)

DESCRIPTION
This routine resets a specified SLIP interface. It detaches the tty from the sl unit and deletes the specified SLIP interface from the list of network interfaces. For example, the following call will delete the first SLIP interface from the list of network interfaces:

    slipDelete (0);

RETURNS
OK, or ERROR if the unit number is invalid or uninitialized.

SEE ALSO
if_sl
slipInit( )

NAME

slipInit( ) – initialize a SLIP interface

SYNOPSIS

STATUS slipInit

( int unit, /* SLIP device unit number (0 - 19) */
  char * devName, /* name of the tty device to be initialized */
  char * myAddr, /* address of the SLIP interface */
  char * peerAddr, /* address of the remote peer SLIP interface */
  int baud, /* baud rate of SLIP device: 0=don’t set rate */
  BOOL compressEnable, /* explicitly enable CSLIP compression */
  BOOL compressAllow, /* enable CSLIP compression on Rx */
  int mtu /* user set-able MTU */
)

DESCRIPTION

This routine initializes a SLIP device. Its parameters specify the name of the tty device, the Internet addresses of both sides of the SLIP point-to-point link (i.e., the local and remote sides of the serial line connection), and CSLIP options.

The Internet address of the local side of the connection is specified in myAddr and the name of its tty device is specified in devName. The Internet address of the remote side is specified in peerAddr. If baud is not zero, the baud rate will be the specified value; otherwise, the default baud rate will be the rate set by the tty driver. The unit parameter specifies the SLIP device unit number. Up to twenty units may be created.

The CLSIP options parameters compressEnable and compressAllow determine support for TCP/IP header compression. If compressAllow is TRUE (1), then CSLIP will be enabled only if a CSLIP type packet is received by this device. If compressEnable is TRUE (1), then CSLIP compression will be enabled explicitly for all transmitted packets, and compressed packets can be received.

The MTU option parameter allows the setting of the MTU for the link.

For example, the following call initializes a SLIP device, using the console’s second port, where the Internet address of the local host is 192.10.1.1 and the address of the remote host is 192.10.1.2. The baud rate will be the default rate for /tyCo/1. CSLIP is enabled if a CSLIP type packet is received. The MTU of the link is 1006.

    slipInit (0, "/tyCo/1", "192.10.1.1", "192.10.1.2", 0, 0, 1, 1006);

RETURNS

OK, or ERROR if the device cannot be opened, memory is insufficient, or the route is invalid.

SEE ALSO

if_sl
**smcFdc37b78xDevCreate()**

**NAME**

smcFdc37b78xDevCreate() – set correct IO port addresses for super I/O chip

**SYNOPSIS**

VOID smcFdc37b78xDevCreate

  (  
    SMCFDC37B78X_IOPORTS * smcFdc37b78x_iop  
  )

**DESCRIPTION**

This routine will initialize smcFdc37b78xIoPorts data structure. These ioports can either be changed on-the-fly or overriding SMCFDC37B78X_CONFIG_PORT, SMCFDC37B78X_INDEX_PORT and SMCFDC37B78X_DATA_PORT. This is a necessary step in initialization of super IO chip and logical devices embedded in it.

**RETURNS**

NONE.

**SEE ALSO**

smcFdc37b78x

---

**smcFdc37b78xInit()**

**NAME**

smcFdc37b78xInit() – initialize 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 initialization

**RETURNS**

NONE.

**SEE ALSO**

smcFdc37b78x
smcFdc37b78xKbdInit()

NAME
smcFdc37b78xKbdInit() – initialize the keyboard controller

SYNOPSIS
STATUS smcFdc37b78xKbdInit
{
    VOID
}

DESCRIPTION
This routine will initialize keyboard controller.

RETURNS
OK/ERROR.

SEE ALSO
smcFdc37b78x

smNetShow()

NAME
smNetShow() – show information about a shared memory network

SYNOPSIS
STATUS smNetShow
{
    char * ifName,            /* backplane interface name (NULL == "sm0") */
    BOOL   zero               /* TRUE = zap totals */
}

DESCRIPTION
This routine displays information about the different CPUs configured in a shared
memory network specified by ifName. It prints error statistics and zeros these fields if zero
is set to TRUE.

EXAMPLE
-> smNetShow
Anchor at 0x800000
heartbeat = 705, header at 0x800010, free pkts = 237.
cpu int type arg1 arg2 arg3 queued pkts
--- -------- ---------- ---------- ---------- ------------
 0 poll            0x0        0x0        0x0       0
 1 poll            0x0        0x0        0x0       0
 2 bus-int         0x3       0xc9        0x0       0
 3 mbox-2          0x2d     0x8000        0x0       0
input packets = 192 output packets = 164
output errors = 0 collisions = 0
sn83932EndLoad()

NAME

sn83932EndLoad() – initialize the driver and device

SYNOPSIS

END_OBJ * sn83932EndLoad

(  
   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 the initString parameter. This string must be of the
format:

unit_number:device_reg_addr:ivec

These parameters are all individually described in the sn83932End man page.

RETURNS

An END object pointer or NULL on error.

SEE ALSO

sn83932End

snattach()

NAME

snattach() – publish the sn network interface and initialize the driver and device

SYNOPSIS

STATUS snattach

(  
   int unit,       /* unit number */
   char * pDevRegs,  /* addr of device’s regs */
   int  ivec        /* vector number */
)

DESCRIPTION

This routine publishes the sn interface by filling in a network interface record and adding
this record to the system list. It also initializes the driver and the device to the operational
state.
2: Routines
sramDrv( )

RETURNS
OK or ERROR.

SEE ALSO
if_sn

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.

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.

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.

**SEE ALSO**
sramDrv

---

**st16552DevInit()**

**NAME**
st16552DevInit() – initialize an ST16552 channel

**SYNOPSIS**
```
void st16552DevInit
(  
    ST16552_CHAN * pChan
)
```

**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 ST16552_CHAN structure.

**RETURNS**
N/A.

**SEE ALSO**
st16552Sio
### st16552Int()

**NAME**

st16552Int() – interrupt level processing

**SYNOPSIS**

```c
void st16552Int
    (ST16552_CHAN * pChan    /* ptr to struct describing channel */)
```

**DESCRIPTION**

This routine handles interrupts from the UART.

**RETURNS**

N/A.

**SEE ALSO**

st16552Sio

---

### st16552IntEx()

**NAME**

st16552IntEx() – miscellaneous interrupt processing

**SYNOPSIS**

```c
void st16552IntEx
    (ST16552_CHAN * pChan    /* ptr to struct describing channel */)
```

**DESCRIPTION**

This routine handles miscellaneous interrupts on the UART.

**RETURNS**

N/A.

**SEE ALSO**

st16552Sio
st16552IntRd()

NAME  st16552IntRd() – handle a receiver interrupt

SYNOPSIS  void st16552IntRd
              (  
               ST16552_CHAN * pChan  /* ptr to struct describing channel */  
              )

DESCRIPTION  This routine handles read interrupts from the UART.

RETURNS  N/A.

SEE ALSO  st16552Sio

st16552IntWr()

NAME  st16552IntWr() – handle a transmitter interrupt

SYNOPSIS  void st16552IntWr
              (  
               ST16552_CHAN * pChan  /* ptr to struct describing channel */  
              )

DESCRIPTION  This routine handles write interrupts from the UART.

RETURNS  N/A.

SEE ALSO  st16552Sio
### st16552MuxInt()

**NAME**

st16552MuxInt() – multiplexed interrupt level processing

**SYNOPSIS**

```c
void st16552MuxInt
    (ST16552_MUX * pMux /* ptr to struct describing multiplexed chans */)
```

**DESCRIPTION**

This routine handles multiplexed interrupts from the DUART. It assumes that channels 0 and 1 are connected so that they produce the same interrupt.

**RETURNS**

N/A.

**SEE ALSO**

st16552Sio

---

### sym895CtrlCreate()

**NAME**

sym895CtrlCreate() – create a structure for a SYM895 device

**SYNOPSIS**

```c
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.
sym895CtrlCreate()

**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** 66.66Mhz 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

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

DESCRIPTION

This routine initializes a 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.

ERRORS

N/A.

SEE ALSO

sym895Lib

sym895GPIOConfig()

NAME

sym895GPIOConfig() – configure 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 */
    )
sym895GPIOCtrl()

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 corresponding bit set in this parameter.

SEE ALSO
sym895Lib

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 corresponding bit set in this parameter.

SEE ALSO
sym895Lib
sym895HwInit( )

NAME  sym895HwInit( ) – hardware initialization for the 895 Chip

SYNOPSIS  

```c
STATUS sym895HwInit
    (  
        SIOP * pSiop              /* pointer to the SIOP structure */  
    )
```

DESCRIPTION  This function puts the SIOP in a known quiescent state. Also, if copying of SCSI scripts is enabled, this routine copies entire SCRIPTS code from host memory to On-Chip SCRIPTS RAM. This routine does not modify any of the registers that are set by sym895SetHwOptions().

For details of the register bits initialized here, refer to SYM53C895 data manual Version 3.0.

`pSiop`

Pointer to the SIOP structure.

RETURNS  OK, or ERROR if parameters are out of range.

ERRORS  N/A.

SEE ALSO  sym895Lib

sym895Intr( )

NAME  sym895Intr( ) – interrupt service routine for the SCSI Controller

SYNOPSIS  

```c
void sym895Intr
    (  
        SIOP * pSiop                /* pointer to the SIOP structure */  
    )
```

DESCRIPTION  The first thing to determine is whether the device is generating an interrupt. If not, then this routine must exit as quickly as possible.

Find the event type corresponding to this interrupt, and carry out any actions which must be done before the SCSI Controller is re-started. Determine whether or not the SCSI Controller is connected to the bus (depending on the event type, see note below). If not,
start a client script if possible or else just make the SCSI Controller wait for something else to happen.

The "connected" variable, at the end of switch statement, reflects the status of the currently executing thread. If it is TRUE it means that the thread is suspended and must be processed at the task level. Set the state of SIOP to IDLE and leave the control to the SCSI Manager. The SCSI Manager, in turn invokes the driver through a "resume" call.

Notify the SCSI manager of a controller event.

**RETURNS**

N/A.

**SEE ALSO**

sym895Lib

---

**sym895Loopback()**

**NAME**

sym895Loopback() — this routine performs loopback diagnostics on 895 chip

**SYNOPSIS**

```c
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:

- 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.
- Bit 4 of DCNTL should be set to turn on single step mode. This allows the target program (this routine) to monitor when an initiator SCRIPTS instruction has completed.
- In this routine, SELECTION, MSG_OUT, and DATA_OUT phases are checked. This will ensure that data and control paths are proper.

**SEE ALSO**

sym895Lib
**NAME**

`sym895SetHwOptions()` – set the Sym895 chip options

**SYNOPSIS**

```c
STATUS sym895SetHwOptions
(
    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.

```c
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*/
    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;     /* STST2:b1, Extend SREQ/SACK filtering */
    int TE : 1;      /* STST3:b7, Tolerant Enable */
    int BL : 3;      /* DMODE:b7,b6, CTEST5:b2 : Burst length */
    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.
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
SYN895 Registers
-------------
Scntl0  = 0xd0 Scntl1  = 0x00 Scntl2  = 0x00 Scntl3  = 0x00
Scid   = 0x67 Sxfer   = 0x00 Sdid   = 0x00 Gpreg   = 0x0f
Sfbr   = 0x0f Socl    = 0x00 Ssid   = 0x00 Scntl   = 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
S1par  = 0x4c Swide   = 0x00 Macntl  = 0xd0 Gpcntl  = 0x0f

428
2: Routines

sym895Show( )

\[\begin{align*}
\text{Stime0} &= 0x00 \\
\text{Stime1} &= 0x00 \\
\text{Respid0} &= 0x80 \\
\text{Respid1} &= 0x00 \\
\text{Stest0} &= 0x07 \\
\text{Stest1} &= 0x00 \\
\text{Stest2} &= 0x00 \\
\text{Stest3} &= 0x80 \\
\text{Sid1} &= 0x0000 \\
\text{Sod1} &= 0x0000 \\
\text{Sbdl} &= 0x0000 \\
\text{Scratchb} &= 0x00000200 \\
\text{value} &= 0 \times 00
\end{align*}\]

**RETURNS**

OK, or ERROR if \textit{pScsiCtrl} and \textit{pSysScsiCtrl} are both NULL.

**SEE ALSO**

sym895Lib, sym895CtrlCreate( )
### tcicInit()

**NAME**

tcicInit() – initialize the TCIC chip

**SYNOPSIS**

```c
STATUS tcicInit
    (int     ioBase,           /* IO 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.

**SEE ALSO**

tcic

### tcicShow()

**NAME**

tcicShow() – show all configurations of the TCIC chip

**SYNOPSIS**

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

**DESCRIPTION**

This routine shows all configurations of the TCIC chip.

**RETURNS**

N/A.

**SEE ALSO**

tcicShow
**ultraattach( )**

**NAME**

ultraattach( ) – publish ultra interface and initialize device

**SYNOPSIS**

```c
STATUS ultraattach
  (  
    int unit,             /* unit number */
    int ioAddr,           /* address of ultra's shared memory */
    int ivec,             /* interrupt vector to connect to */
    int ilevel,           /* interrupt level */
    int memAddr,          /* address of ultra's shared memory */
    int memSize,          /* size of ultra's shared memory */
    int config            /* 0: RJ45 + AUI(Thick) 1: RJ45 + BNC(Thin) */
  )
```

**DESCRIPTION**

This routine attaches an ultra Ethernet interface to the network if the device exists. It makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

**RETURNS**

OK or ERROR.

**SEE ALSO**

if_ultra, ifLib, netShow

---

**ultraLoad( )**

**NAME**

ultraLoad( ) – initialize the driver and device

**SYNOPSIS**

```c
END_OBJ* ultraLoad
  (  
    char * initString  /* String to be parsed by the driver. */
  )
```

**DESCRIPTION**

This routine initializes the driver and device to the operational state. All device-specific parameters are passed in `initString`, which expects a string of the following format:

- If the routine is called with an empty, but allocated string, it places the name of this device (that is, "ultra") 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.
ultraPut()

NAME   ultraPut() – copy a packet to the interface

SYNOPSIS  #ifdef BSD43_DRIVER LOCAL void ultraPut
    (int unit                  /* device unit number */
    )

DESCRIPTION Copy from mbuf chain to transmitter buffer in shared memory.

RETURNS  N/A.

SEE ALSO  if_ultra

ultraShow()

NAME   ultraShow() – display statistics for the ultra network interface

SYNOPSIS  void ultraShow
             (int  unit,              /* interface unit */
              BOOL zap              /* zero totals */
             )

DESCRIPTION This routine displays statistics about the elc Ethernet network interface. It has two parameters:

unit    Interface unit; should be 0.
zap     If 1, all collected statistics are cleared to zero.

RETURNS  N/A.

SEE ALSO  if_ultra
**vgaInit()**

**NAME**

vgaInit() – initialize 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.

**SEE ALSO**

vgaInit
**NAME**

wd33c93CtrlCreate() – create and partially initialize a WD33C93 SBIC structure

**SYNOPSIS**

```c
WD_33C93_SCSI_CTRL *wd33c93CtrlCreate
    (    
    UINT8 * sbicBaseAdrs,     /* base address of SBIC */
    int     regOffset,        /* addr offset between consecutive regs. */
    UINT    clkPeriod,        /* period of controller clock (nsec) */
    int     devType,          /* SBIC device type */
    FUNCPTR sbicScsiReset,    /* SCSI bus reset function */
    FUNCPTR sbicDmaBytesIn,   /* SCSI DMA input function */
    FUNCPTR sbicDmaBytesOut   /* SCSI DMA output function */
    )
```

**DESCRIPTION**

This routine creates an SBIC data structure and must be called before using an SBIC chip. It should be called once and only once for a specified SBIC. Since it allocates memory for a structure needed by all routines in `wd33c93Lib`, it must be called before any other routines in the library. After calling this routine, at least one call to `wd33c93CtrlInit()` should be made before any SCSI transaction is initiated using the SBIC.

**NOTE:** Note that only the non-multiplexed processor interface is supported.

The input parameters are as follows:

- `sbicBaseAdrs`
  The address where the CPU accesses the lowest register of the SBIC.

- `regOffset`
  The address offset (in bytes) to access consecutive registers. (This must be a power of 2; for example, 1, 2, 4, etc.)

- `clkPeriod`
  The period, in nanoseconds, of the signal-to-SBIC clock input used only for select command timeouts.

- `devType`
  A constant corresponding to the type (part number) of this controller; possible options are enumerated in `wd33c93.h` under the heading “SBIC device type.”

- `sbicScsiReset`
  A board-specific routine to assert the RST line on the SCSI bus, which causes all connected devices to return to a known quiescent state.

- `sbicDmaBytesIn` and `sbicDmaBytesOut`
  Board-specific routines to handle DMA input and output. If these are NULL (0), SBIC program transfer mode is used. DMA is implemented only during SCSI data in/out
phases. The interface to these DMA routines must be of the form:

```c
STATUS xxDmaBytes{In, Out}
{
    SCSI_PHYS_DEV *pScsiPhysDev, /* ptr to phys dev info */
    UINT8 *pBuffer,           /* ptr to the data buffer */
    int bufLength            /* number of bytes to xfer */
}
```

**RETURNS**

A pointer to the SBIC control structure, or NULL if memory is insufficient or parameters are invalid.

**SEE ALSO**

wd33c93Lib1, wd33c93.h

---

**wd33c93CtrlCreateScsi2**

**NAME**

wd33c93CtrlCreateScsi2() – create and partially initialize an SBIC structure

**SYNOPSIS**

```c
WD_33C93_SCSI_CTRL *wd33c93CtrlCreateScsi2
(
    UINT8 * sbicBaseAdrs,       /* base address of the SBIC */
    int regOffset,             /* address offset between SBIC registers */
    UINT clkPeriod,           /* period of the SBIC clock (nsec) */
    FUNCPTR sysScsiBusReset,   /* function to reset SCSI bus */
    int sysScsiResetArg,       /* argument to pass to above function */
    UINT sysScsiDmaMaxBytes,   /* maximum byte count using DMA */
    FUNCPTR sysScsiDmaStart,   /* function to start SCSI DMA transfer */
    FUNCPTR sysScsiDmaAbort,   /* function to abort SCSI DMA transfer */
    int sysScsiDmaArg          /* argument to pass to above functions */
)
```

**DESCRIPTION**

This routine creates an SBIC data structure and must be called before using an SBIC chip. It must be called exactly once for a specified SBIC. Since it allocates memory for a structure needed by all routines in wd33c93Lib2, it must be called before any other routines in the library. After calling this routine, at least one call to wd33c93CtrlInit() must be made before any SCSI transaction is initiated using the SBIC.

**NOTE:** Only the non-multiplexed processor interface is supported.

A detailed description of the input parameters follows:

- **sbicBaseAdrs**
  The address at which the CPU would access the lowest (AUX STATUS) register of the
SBIC.

regOffset
The address offset (bytes) to access consecutive registers. (This must be a power of 2, for example, 1, 2, 4, etc.)

clockPeriod
The period in nanoseconds of the signal to SBIC CLK input.

sysScsiBusReset and sysScsiResetArg
The board-specific routine to pulse the SCSI bus RST signal. The specified argument is passed to this routine when it is called. It may be used to identify the SCSI bus to be reset, if there is a choice. The interface to this routine is of the form:

```c
void xxBusReset(
    int arg;         /* call-back argument */
)
```

sysScsiDmaMaxBytes, sysScsiDmaStart, sysScsiDmaAbort, and sysScsiDmaArg
Board-specific routines to handle DMA transfers to and from the SBIC; if the maximum DMA byte count is zero, programmed I/O is used. Otherwise, non-NULL function pointers to DMA start and abort routines must be provided. The specified argument is passed to these routines when they are called; it may be used to identify the DMA channel to use, for example. Note that DMA is implemented only during SCSI data in/out phases. The interface to these DMA routines must be of the form:

```c
STATUS xxDmaStart(
    int arg;         /* call-back argument */
    UINT8 *pBuffer; /* ptr to the data buffer */
    UINT buffsize;  /* number of bytes to xfer */
    int direction;  /* 0 = SCSI->mem, 1 = mem->SCSI */
)

STATUS xxDmaAbort(
    int arg;         /* call-back argument */
)
```

RETURNS A pointer to the SBIC structure, or NULL if memory is insufficient or the parameters are invalid.

SEE ALSO wd33c93Lib2
NAME

wd33c93CtrlInit() – initialize the user-specified fields in an SBIC structure

SYNOPSIS

STATUS wd33c93CtrlInit
    (int * pSbic,              /* ptr to SBIC info */
     int scsiCtrlBusId,        /* SCSI bus ID of this SBIC */
     UINT defaultSelTimeOut,  /* default dev. select timeout (microsec) */
     int scsiPriority         /* priority of task when doing SCSI I/O */
    )

DESCRIPTION

This routine initializes an SBIC structure, after the structure is created with either
wd33c93CtrlCreate() or wd33c93CtrlCreateScsi2(). This structure must be initialized
before the SBIC can be used. It may be called more than once; however, it should be called
only while there is no activity on the SCSI interface. Before returning, this routine pulses
RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pSbic
A pointer to the WD_33C93_SCSI_CTRL structure created with wd33c93CtrlCreate() or
wd33c93CtrlCreateScsi2().

scsiCtrlBusId
The SCSI bus ID of the SBIC, in the range 0 - 7. The ID is somewhat arbitrary; the
value 7, or highest priority, is conventional.

defaultSelTimeOut
The timeout, in microseconds, for selecting a SCSI device attached to this controller.
This value is used as a default if no timeout is specified in scsiPhysDevCreate(). The
recommended value zero (0) specifies SCSI_DEF_SELECT_TIMEOUT (250 millisec).
The maximum timeout possible is approximately 2 seconds. Values exceeding this
revert to the maximum. For more information about chip timeouts, see the manuals
Western Digital WD33C92/93 SCSI-Bus Interface Controller, Western Digital
WD33C92A/93A SCSI-Bus Interface Controller.

scsiPriority
The priority to which a task is set when performing a SCSI transaction. Valid
priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not
be altered during SCSI transactions.

RETURNS

OK, or ERROR if a parameter is out of range.

SEE ALSO

wd33c93Lib, scsiPhysDevCreate(), Western Digital WD33C92/93 SCSI-Bus Interface
Controller, Western Digital WD33C92A/93A SCSI-Bus Interface Controller
wd33c93Show() – display the values of all readable WD33C93 chip registers

**SYNOPSIS**

```c
int wd33c93Show
    (  
        int * pScsiCtrl               /* ptr to SCSI controller info */  
    )
```

**DESCRIPTION**

This routine displays the state of the SBIC registers in a user-friendly manner. It is useful primarily for debugging. It should not be invoked while another running process is accessing the SCSI controller.

**EXAMPLE**

```plaintext
   -> wd33c93Show
   REG #00 (Own ID   ) = 0x07  
   REG #01 (Control  ) = 0x00  
   REG #02 (Timeout Period) = 0x20  
   REG #03 (Sectors   ) = 0x00  
   REG #04 (Heads     ) = 0x00  
   REG #05 (Cylinders MSB) = 0x00  
   REG #06 (Cylinders LSB) = 0x00  
   REG #07 (Log. Addr. MSB) = 0x00  
   REG #08 (Log. Addr. 2SB) = 0x00  
   REG #09 (Log. Addr. 3SB) = 0x00  
   REG #0a (Log. Addr. LSB) = 0x00  
   REG #0b (Sector Number) = 0x00  
   REG #0c (Head Number ) = 0x00  
   REG #0d (Cyl. Number MSB)= 0x00  
   REG #0e (Cyl. Number LSB) = 0x00  
   REG #0f (Target LUN  ) = 0x00  
   REG #10 (Command Phase) = 0x00  
   REG #11 (Synch. Transfer) = 0x00  
   REG #12 (Xfer Count MSB) = 0x00  
   REG #13 (Xfer Count 2SB) = 0x00  
   REG #14 (Xfer Count LSB) = 0x00  
   REG #15 (Destination ID) = 0x03  
   REG #16 (Source ID    ) = 0x00  
   REG #17 (SCSI Status  ) = 0x42  
   REG #18 (Command      ) = 0x07
```

**RETURNS**

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

**SEE ALSO**

wd33c93Lib
**wdbEndPktDevInit()**

**NAME**  
wdbEndPktDevInit() – initialize an END packet device

**SYNOPSIS**  
```c
void wdbEndPktDevInit(
    STATUS wdbEndPktDevInit
    
    WDB_END_PKT_DEV * pPktDev, /* device structure to init */
    void (* stackRcv) (),      /* receive packet callback (udpRcv) */
    char *            pDevice, /* Device (ln, ie, etc.) that we wish to */
                     /* bind to. */
    int               unit     /* unit number (0, 1, etc.) */
);
```

**DESCRIPTION**  
This routine initializes an END packet device. It is typically called from configlette wdbEnd.c when the WDB agent’s lightweight END communication path (INCLUDE_WDB_COMM_END) is selected.

**RETURNS**  
OK or ERROR.

**SEE ALSO**  
wdbEndPktDrv

**wdbNetromPktDevInit()**

**NAME**  
wdbNetromPktDevInit() – initialize a NETROM packet device for the WDB agent

**SYNOPSIS**  
```c
void wdbNetromPktDevInit(
    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\xd5 s index in a ROM word */
    int                  numAccess, /* to pod zero per byte read */
    void (* stackRcv)(),            /* callback when packet arrives */
    int                  pollDelay  /* poll task delay */
);
```

**DESCRIPTION**  
This routine initializes a NETROM packet device. It is typically called from usrWdb.c when the WDB agents NETROM communication path is selected. The dpBase parameter is the address of NetROM’s dualport RAM. The width parameter is the width of a word in ROM space, and can be 1, 2, or 4 to select 8-bit, 16-bit, or 32-bit width respectively (use the macro WDB_NETROM_WIDTH in configAll.h for this parameter). The index parameter
wdbPipePktDevInit()

NAME
wdbPipePktDevInit() – initialize a pipe packet device

SYNOPSIS

```c
STATUS wdbPipePktDevInit
{
    WDB_PIPE_PKT_DEV * pPktDev, /* pipe device structure to init */
    void (* stackRcv)()         /* receive packet callback (udpRcv) */
}
```

SEE ALSO
wdbPipePktDrv

wdbSlipPktDevInit()

NAME
wdbSlipPktDevInit() – initialize a SLIP packet device for a WDB agent

SYNOPSIS

```c
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.

SEE ALSO
wdbSlipPktDrv

wdbPipePktDevInit() refers to which byte of the ROM contains pod zero. The numAccess parameter should be set to the number of accesses to POD zero that are required to read a byte. It is typically one, but some boards actually read a word at a time. This routine spawns a task which polls the NetROM for incoming packets every pollDelay clock ticks.

RETURNS
N/A.

SEE ALSO
wdbNetromPktDrv
## wdbTsfsDrv()

**NAME**  
`wdbTsfsDrv()` – initialize the TSFS device driver for a WDB agent

**SYNOPSIS**  
```c
STATUS wdbTsfsDrv  
  (  
    char * name               /* root name in i/o system */  
  )
```

**DESCRIPTION**  
This routine initializes the 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 called by `usrRoot()` in `usrConfig.c`, 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:

```c
fd = open ("/tgtsvr/etc/passwd", O_RDWR, 0)
```

**RETURNS**  
OK, or ERROR if the driver can not be installed.

**SEE ALSO**  
wdbTsfsDrv

## wdbUlipPktDevInit()

**NAME**  
`wdbUlipPktDevInit()` – initialize the communication functions for ULIP

**SYNOPSIS**  
```c
void wdbUlipPktDevInit  
  (  
    WDB_ULIP_PKT_DEV * pDev,    /* ULIP packet device to initialize */  
    char *         ulipDev, /* name of UNIX device to use */  
    void (* stackRcv)()       /* routine to call when a packet arrives */  
  )
```

**DESCRIPTION**  
This routine initializes a ULIP device for use by the WDB debug agent. It provides a communication path to the debug agent which can be used with both a task and an external mode agent. It is typically called by `usrWdb.c` when the WDB agent’s lightweight ULIP communication path is selected.

**RETURNS**  
N/A.

**SEE ALSO**  
wdbUlipPktDrv
### wdbVioDrv()

**NAME**  
wdbVioDrv() – initialize the tty driver for a WDB agent

**SYNOPSIS**  
STATUS wdbVioDrv  
\(  
\)  
char * 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 called by usrRoot() in usrConfig.c, and the device name created is "vio".

After this routine has been called, individual virtual I/O channels can be open 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.

**SEE ALSO**  
wdbVioDrv
z8530DevInit()

NAME  z8530DevInit() – initialize 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.

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.

SEE ALSO  z8530Sio
**z8530IntEx()**

**NAME**

z8530IntEx() – handle error interrupts

**SYNOPSIS**

```c
void z8530IntEx
```

```c
(Z8530_CHAN * pChan)
```

**DESCRIPTION**

This routine handles miscellaneous interrupts on the SCC.

**RETURNS**

N/A.

**SEE ALSO**

z8530Sio

---

**z8530IntRd()**

**NAME**

z8530IntRd() – handle a receiver interrupt

**SYNOPSIS**

```c
void z8530IntRd
```

```c
(Z8530_CHAN * pChan)
```

**DESCRIPTION**

This routine handles read interrupts from the SCC.

**RETURNS**

N/A.

**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.

SEE ALSO  z8530Sio
z8530IntWr()
## Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface driver for 3COM 3C509. END network</td>
<td>elt3c509End</td>
<td>42</td>
</tr>
<tr>
<td>display statistics for 3C509 elt network interface.</td>
<td>eltShow()</td>
<td>271</td>
</tr>
<tr>
<td>interface driver. 3Com 3C509 Ethernet network</td>
<td>if_elt</td>
<td>77</td>
</tr>
<tr>
<td>network interface driver for 3COM 3C509. END</td>
<td>elt3c509End</td>
<td>42</td>
</tr>
<tr>
<td>interface driver. 3Com 3C509 Ethernet network</td>
<td>if_elt</td>
<td>77</td>
</tr>
<tr>
<td>network interface driver for 3COM 3C90xB XL. END</td>
<td>elt3c90xEnd</td>
<td>38</td>
</tr>
<tr>
<td>registers for NCR 53C710. /hardware-dependent</td>
<td>ncr710SetHwRegisterScsi2()</td>
<td>338</td>
</tr>
<tr>
<td>(SIOP) library (SCSI-1). NCR 53C710 SCSI I/O Processor</td>
<td>ncr710Lib</td>
<td>151</td>
</tr>
<tr>
<td>(SIOP) library (SCSI-2). NCR 53C710 SCSI I/O Processor</td>
<td>ncr710Lib2</td>
<td>151</td>
</tr>
<tr>
<td>control structure for NCR 53C710 SIOP. create</td>
<td>ncr710CtrlCreate()</td>
<td>333</td>
</tr>
<tr>
<td>control structure for NCR 53C710 SIOP. create</td>
<td>ncr710CtrlCreateScsi2()</td>
<td>334</td>
</tr>
<tr>
<td>control structure for NCR 53C710 SIOP. initialize</td>
<td>ncr710CtrlInit()</td>
<td>335</td>
</tr>
<tr>
<td>control structure for NCR 53C710 SIOP. initialize</td>
<td>ncr710CtrlInitScsi2()</td>
<td>336</td>
</tr>
<tr>
<td>/registers for NCR 53C710 SIOP</td>
<td>ncr710SetHwRegister()</td>
<td>337</td>
</tr>
<tr>
<td>/values of all readable NCR 53C710 SIOP registers</td>
<td>ncr710Show()</td>
<td>339</td>
</tr>
<tr>
<td>/values of all readable NCR 53C710 SIOP registers</td>
<td>ncr710ShowScsi2()</td>
<td>340</td>
</tr>
<tr>
<td>(SIOP) library (SCSI-2). NCR 53C8xx PCI SCSI I/O Processor</td>
<td>ncr810Lib</td>
<td>152</td>
</tr>
<tr>
<td>control structure for NCR 53C8xx SIOP. create</td>
<td>ncr810CtrlCreate()</td>
<td>342</td>
</tr>
<tr>
<td>control structure for NCR 53C8xx SIOP. initialize</td>
<td>ncr810CtrlInit()</td>
<td>343</td>
</tr>
<tr>
<td>/registers for NCR 53C8xx SIOP</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>/values of all readable NCR 53C8xx SIOP registers</td>
<td>ncr810Show()</td>
<td>345</td>
</tr>
<tr>
<td>/values of all readable SYM 53C8xx SIOP registers</td>
<td>sym895Show()</td>
<td>428</td>
</tr>
<tr>
<td>(ASC) library (SCSI-1). NCR 53C90 Advanced SCSI Controller</td>
<td>ncr5390Lib1</td>
<td>154</td>
</tr>
<tr>
<td>(ASC) library (SCSI-2). NCR 53C90 Advanced SCSI Controller</td>
<td>ncr5390Lib2</td>
<td>154</td>
</tr>
<tr>
<td>control structure for NCR 53C90 ASC. create</td>
<td>ncr5390CtrlCreate()</td>
<td>346</td>
</tr>
<tr>
<td>control structure for NCR 53C90 ASC. create</td>
<td>ncr5390CtrlCreateScsi2()</td>
<td>347</td>
</tr>
<tr>
<td>driver. Motorola 68EN302 network-interface</td>
<td>if_mbc</td>
<td>92</td>
</tr>
<tr>
<td>display statistics for SMC 8013WC elc network interface.</td>
<td>elcShow()</td>
<td>269</td>
</tr>
<tr>
<td>interface driver. SMC 8013WC Ethernet network</td>
<td>if_elc</td>
<td>76</td>
</tr>
<tr>
<td>Siemens SAB 82532 UART tty driver.</td>
<td>sab82532</td>
<td>192</td>
</tr>
<tr>
<td>for handling interrupts from 82596. entry point</td>
<td>eint()</td>
<td>264</td>
</tr>
</tbody>
</table>

447
interface/ END style Intel 82596 Ethernet network ...................... e82596End 35
interface driver. Intel 82596 Ethernet network .......................... if_el 66
interface driver for / Intel 82596 Ethernet network ...................... if_eidve 69
interface driver for / Intel 82596 Ethernet network ...................... if_eihk 73
Library File. Adaptec 7880 SCSI Host Adapter .......................... aic7880Lib 5
interface driver. AMD Am7990 LANCE Ethernet ......................... ln97xEnd 110
network interface driver. AMD Am79C970 PCnet-PCI Ethernet ......... if_InPci 88
driver. END style AMD Am79C97X PCnet-PCI Ethernet ................. In97xEnd 110
initialize AMBA channel. ................................................... ambaDevInit() 234
ARM AMBA UART tty driver. ............................................... ambaSio 8
network interface/ END style AMD 7990 LANCE Ethernet ............... ln7990End 115
network interface driver. AMD Am7990 LANCE Ethernet ............... if_In 85
Ethernet network interface/ AMD Am79C970 PCnet-PCI .................. if_InPci 88
Ethernet driver. END style AMD Am79C97X PCnet-PCI ................ ln97xEnd 110
ARM AMBA UART tty driver. ............................................... ambaSio 8
structure for NCR 53C90 ASC create control ............................ ncr5390CtrlCreate() 346
structure for NCR 53C90 ASC create control ............................ ncr5390CtrlCreateScsi2() 347
53C90 Advanced SCSI Controller (ASC) library (SCSI-1). NCR ncr5390Lib1 154
53C90 Advanced SCSI Controller (ASC) library (SCSI-2). NCR ncr5390Lib2 154
user-specified fields in ASC structure. initialize ....................... ncr5390CtrlInit() 349
low level initialization of ATA device...................................... iPIIX4AtaInit() 289
initialize ATA drive. ......................................................... ataDriveInit() 237
initialize ATA drive. ......................................................... ataDrv() 237
and PCMCIA disk device/ create device for ATA/IDE and ATAPI CDROM (LOCAL) ataDrv 11
create device for ATA/IDE disk. .......................................... ataDevCreate() 236
routine. initialize ATA/IDE disk driver show ......................... ataShowInit() 239
show ATA/IDE disk parameters. .......................................... ataShow() 238
disk device driver show/ disk device/ ATA/IDE and ATAPI CDROM (LOCAL and PCMCIA) ataShow 14
return contents of DUART auxiliary control register .................... m68681Acr( ) 304
set and clear bits in DUART auxiliary control register ................ m68681AcrSetCrl( ) 304
memory. initialize module. CHIPS B69000 chip and loads font in ... ctB69000VgaInit() 253
driver. shared memory backplane network interface ................... if_sm 98
set baud rate for SLIP interface ........................................... slipBaudSet() 412
disable cards for warm boot. .............................................. pciConfigReset() 389
check condition on specified bus. ........................................... pciConfigForeachFunc() 382
configure device on PCI bus ................................................. pciDevConfig() 391
Databook TCIC/2 PCMCIA host bus adaptor chip driver ................. tcic 207
Intel 82365SL PCMCIA host bus adaptor chip library ..................... pciCle 173
Intel 82365SL PCMCIA host bus adaptor chip show library ............ pciShow 185
Databook TCIC/2 PCMCIA host bus adaptor chip show library ............ tcicShow 207
show routines of PCI bus (IO mapped) library .......................... pciConfigShow 185
secondary, and subordinate bus number. set primary ................... pciAutoBusNumberSet() 367
perform PCI character to output word .................................... nvr4101SIUCharToTxWord() 359
allocation facility. PCI device/ ATA/IDE and ATAPI CDRom (LOCAL and PCMCIA) ataDrv 11
driver. ColdFire Serial Communications .................................. coldfireSpi 19
format.. translate CL-CD2400 MPCC serial driver ..................... cd2400Spi 18

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>and addresses, initialize configuration access-method</td>
<td>pciConfigLibInit()</td>
<td>384</td>
</tr>
<tr>
<td>Register, pack parameters for Configuration Address</td>
<td>pciConfigBdfPack()</td>
<td>380</td>
</tr>
<tr>
<td>function, perform final configuration and enable pciAutoFuncEnable()</td>
<td>378</td>
<td></td>
</tr>
<tr>
<td>function, show configuration details about pciConfigFuncShow()</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>get PCMCIA configuration register</td>
<td>cisConfigregGet()</td>
<td>243</td>
</tr>
<tr>
<td>set PCMCIA configuration register</td>
<td>cisConfigregSet()</td>
<td>243</td>
</tr>
<tr>
<td>/state of DUART output port register</td>
<td>m68681Opcr()</td>
<td>307</td>
</tr>
<tr>
<td>bits in DUART output port configuration register</td>
<td>m68681OpcrSetClr()</td>
<td>308</td>
</tr>
<tr>
<td>read one byte from PCI configuration space</td>
<td>pciConfigInByte()</td>
<td>383</td>
</tr>
<tr>
<td>read one longword from PCI configuration space</td>
<td>pciConfigInLong()</td>
<td>383</td>
</tr>
<tr>
<td>read one word from PCI configuration space</td>
<td>pciConfigInWord()</td>
<td>384</td>
</tr>
<tr>
<td>write one byte to PCI configuration space</td>
<td>pciConfigOutByte()</td>
<td>388</td>
</tr>
<tr>
<td>write one longword to PCI configuration space</td>
<td>pciConfigOutLong()</td>
<td>388</td>
</tr>
<tr>
<td>write one 16-bit word to PCI configuration space</td>
<td>pciConfigOutWord()</td>
<td>389</td>
</tr>
<tr>
<td>support for PCI drivers, PCI interface headers, automatically configure PCMCIA</td>
<td>pciConfigLib</td>
<td>174</td>
</tr>
<tr>
<td>configure all nonexcluded PCI configuration</td>
<td>pciAutoCfg()</td>
<td>368</td>
</tr>
<tr>
<td>configure all nonexcluded PCI configuration</td>
<td>pciAutoConfig()</td>
<td>376</td>
</tr>
<tr>
<td>configure device on PCI bus</td>
<td>pciDevConfig()</td>
<td>391</td>
</tr>
<tr>
<td>initialize and configure general purpose pins sym895GpioConfig()</td>
<td>423</td>
<td></td>
</tr>
<tr>
<td>configure PHY devices</td>
<td>miPhyInit()</td>
<td>322</td>
</tr>
<tr>
<td>copy packet to interface</td>
<td>elcPut()</td>
<td>268</td>
</tr>
<tr>
<td>copy packet to interface</td>
<td>enePut()</td>
<td>276</td>
</tr>
<tr>
<td>copy packet to interface</td>
<td>esmcPut()</td>
<td>278</td>
</tr>
<tr>
<td>copy packet to interface</td>
<td>ultraPut()</td>
<td>432</td>
</tr>
<tr>
<td>driver, Motorola CPM core network interface</td>
<td>if_cpm</td>
<td>54</td>
</tr>
<tr>
<td>initialize driver, publish CPM network interface and</td>
<td>cpmattach()</td>
<td>250</td>
</tr>
<tr>
<td>driver, Crystal Semiconductor</td>
<td>CS8990 network interface</td>
<td>if_cs</td>
</tr>
<tr>
<td>registers 0 thru 15, display interface driver, END-style</td>
<td>dec21040/21140 status</td>
<td>decCsrShow()</td>
</tr>
<tr>
<td>interface driver, END-style DEC 21x40 PCI Ethernet network</td>
<td>dec21x40End()</td>
<td>28</td>
</tr>
<tr>
<td>interface driver, END style DEC 21x4 Ethernet LAN network</td>
<td>if_dc</td>
<td>61</td>
</tr>
<tr>
<td>find first PHY connected to DEC 21x4x PCI Ethernet network</td>
<td>dec21x4xEnd()</td>
<td>24</td>
</tr>
<tr>
<td>find nth device with given device &amp; vendor ID</td>
<td>dec21x4xPhyFind()</td>
<td>257</td>
</tr>
<tr>
<td>initialize driver and device</td>
<td>pciFindDevice()</td>
<td>393</td>
</tr>
<tr>
<td>initialize driver and device</td>
<td>auEndLoad()</td>
<td>240</td>
</tr>
<tr>
<td>device</td>
<td>cpmStartOutput()</td>
<td>251</td>
</tr>
<tr>
<td>device</td>
<td>dec21x4EndLoad()</td>
<td>256</td>
</tr>
<tr>
<td>device</td>
<td>dec21x40EndLoad()</td>
<td>257</td>
</tr>
<tr>
<td>device, /eex network interface</td>
<td>eexattach()</td>
<td>260</td>
</tr>
<tr>
<td>device, /eix network interface</td>
<td>ei82596EndLoad()</td>
<td>261</td>
</tr>
<tr>
<td>device, /ei network interface</td>
<td>ei82596EndLoad()</td>
<td>261</td>
</tr>
<tr>
<td>device, /ei network interface</td>
<td>ei82543EndLoad()</td>
<td>266</td>
</tr>
<tr>
<td>device, /eic network interface</td>
<td>ei82543EndLoad()</td>
<td>266</td>
</tr>
<tr>
<td>device, /eic network interface</td>
<td>ei82543EndLoad()</td>
<td>266</td>
</tr>
<tr>
<td>device, /eic network interface</td>
<td>ei82543EndLoad()</td>
<td>266</td>
</tr>
<tr>
<td>device, publish elt interface</td>
<td>eltattach()</td>
<td>268</td>
</tr>
<tr>
<td>device, publish elt interface</td>
<td>eltattach()</td>
<td>268</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>eNateattach()</td>
<td>276</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>eNateattach()</td>
<td>276</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>eNateattach()</td>
<td>276</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>eNateattach()</td>
<td>276</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>fOnicate()</td>
<td>285</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>fOnicate()</td>
<td>285</td>
</tr>
<tr>
<td>device, /ene network interface</td>
<td>fOnicate()</td>
<td>285</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>level initialization of ATA device. low</td>
<td>iPIIX4AtaInit()</td>
<td>289</td>
</tr>
<tr>
<td>initialize floppy disk device.</td>
<td>iPIIX4FdInit()</td>
<td>289</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ln97xEndLoad()</td>
<td>292</td>
</tr>
<tr>
<td>initialize and initialize driver and device.</td>
<td>ln7990EndLoad()</td>
<td>294</td>
</tr>
<tr>
<td>packet to network interface device. output</td>
<td>mbcStartOutput()</td>
<td>319</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>motCpmEndLoad()</td>
<td>327</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>motFecEndLoad()</td>
<td>328</td>
</tr>
<tr>
<td>initialize and initialize driver and device.</td>
<td>ne2000EndLoad()</td>
<td>351</td>
</tr>
<tr>
<td>create PCMCIA memory disk device.</td>
<td>ns83902EndLoad()</td>
<td>356</td>
</tr>
<tr>
<td>create structure for SYM895 device.</td>
<td>sym895CtrlCreate()</td>
<td>421</td>
</tr>
<tr>
<td>ultra interface and initialize device. publish</td>
<td>ultraAttach()</td>
<td>431</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ultraLoad()</td>
<td>431</td>
</tr>
<tr>
<td>initialize END packet device.</td>
<td>wdbEndPktDevInit()</td>
<td>439</td>
</tr>
<tr>
<td>initialize pipe packet device.</td>
<td>wdbPipePktDevInit()</td>
<td>440</td>
</tr>
<tr>
<td>system. initialize device and mount DOS file</td>
<td>pccardMkfs()</td>
<td>365</td>
</tr>
<tr>
<td>status bits. quiesce PCI device and reset all writeable</td>
<td>pciAutoDevReset()</td>
<td>377</td>
</tr>
<tr>
<td>find information. find device by deviceld, then print</td>
<td>pciFindDeviceShow()</td>
<td>394</td>
</tr>
<tr>
<td>find nth occurrence of CDROM (LOCAL and PCMCIA) diskdevice driver. / and ATAPI</td>
<td>ataDrv</td>
<td>11</td>
</tr>
<tr>
<td>CDROM (LOCAL and PCMCIA) disk</td>
<td>device driver.</td>
<td>17</td>
</tr>
<tr>
<td>NEC 765 floppy disk device.</td>
<td>nec765Fd</td>
<td>201</td>
</tr>
<tr>
<td>PCMCIA SRAM parallel chip device driver for IBM-PC LPT.</td>
<td>iPIIX4AtaInit()</td>
<td>289</td>
</tr>
<tr>
<td>initialize TSFS device driver for WDB agent.</td>
<td>wdbTsfsDrv()</td>
<td>441</td>
</tr>
<tr>
<td>/(LOCAL and PCMCIA) disk device driver show routine</td>
<td>ataShow</td>
<td>14</td>
</tr>
<tr>
<td>create device for ATA/IDE device.</td>
<td>ataDevCreate()</td>
<td>236</td>
</tr>
<tr>
<td>create device for floppy disk.</td>
<td>fdDevCreate()</td>
<td>280</td>
</tr>
<tr>
<td>create device for LPT port.</td>
<td>lptDevCreate()</td>
<td>297</td>
</tr>
<tr>
<td>initialize NETROM packet device for WDB agent.</td>
<td>wdbNetromPktDevInit()</td>
<td>439</td>
</tr>
<tr>
<td>initialize SLIP packet device for WDB agent.</td>
<td>wdbSlipPktDevInit()</td>
<td>440</td>
</tr>
<tr>
<td>give device interrupt level to use.</td>
<td>iPIIX4GetInt()</td>
<td>290</td>
</tr>
<tr>
<td>probe list. find next device of specific type from</td>
<td>pciAutoGetNextClass()</td>
<td>379</td>
</tr>
<tr>
<td>configure device on PCI bus.</td>
<td>pciDevConfig()</td>
<td>391</td>
</tr>
<tr>
<td>display device status.</td>
<td>auDump()</td>
<td>239</td>
</tr>
<tr>
<td>vendor ID. find nth device with given device &amp;</td>
<td>pciFindDevice()</td>
<td>393</td>
</tr>
<tr>
<td>initialize and configure PHY devices.</td>
<td>miiPhyInit()</td>
<td>322</td>
</tr>
<tr>
<td>print information about PCI devices.</td>
<td>pciDeviceShow()</td>
<td>392</td>
</tr>
<tr>
<td>create device for ATA/IDE disk</td>
<td>ataDevCreate()</td>
<td>236</td>
</tr>
</tbody>
</table>
### Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>create device for floppy disk</td>
<td>fdDevCreate()</td>
<td>280</td>
</tr>
<tr>
<td>initialize floppy</td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>create PCMCIA memory</td>
<td></td>
<td>417</td>
</tr>
<tr>
<td>ATAPI CDROM (LOCAL and PCMCIA) disk device driver</td>
<td>ataShow</td>
<td>14</td>
</tr>
<tr>
<td>NE/IDE (LOCAL and PCMCIA) disk device driver</td>
<td>ataShowInit()</td>
<td>239</td>
</tr>
<tr>
<td>ATA/IDE (LOCAL and PCMCIA) disk device show</td>
<td>ataDrv</td>
<td>11</td>
</tr>
<tr>
<td>National Semiconductor DP83932B SONIC Ethernet</td>
<td></td>
<td>199</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>if_sn</td>
<td>99</td>
</tr>
<tr>
<td>PCMCIA network interface</td>
<td>iOlicomEnd</td>
<td>102</td>
</tr>
<tr>
<td>Am79C97X PCl/Ethernet</td>
<td>lnt97xEnd</td>
<td>103</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>lnt990End</td>
<td>104</td>
</tr>
<tr>
<td>Motorola MC68302 bimodal tty</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>(LOCAL and PCMCIA) disk device</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>MC68360 SCC UART serial</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>M68681 serial communications</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>M68901 MFP tty</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>MB 86940 UART tty</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>END network interface</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>network interface</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>FCC Ethernet network interface</td>
<td>motCpmEnd</td>
<td>114</td>
</tr>
<tr>
<td>FDC Ethernet network interface</td>
<td>motFcEnd</td>
<td>115</td>
</tr>
<tr>
<td>END style Au MAC Ethernet</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>NE2000 END network interface</td>
<td></td>
<td>117</td>
</tr>
<tr>
<td>NEC 765 floppy disk device</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>ST-NIC Chip network interface</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>NS 16350 UART tty</td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>NEC VR1401 DSIU UART tyy</td>
<td>nvr4101DSIUEnd</td>
<td>121</td>
</tr>
<tr>
<td>NEC VR4101 SIU UART tyy</td>
<td>nvr4101SIUEnd</td>
<td>122</td>
</tr>
<tr>
<td>NEC VR4102 DSIU UART tyy</td>
<td>nvr4102DSIUEnd</td>
<td>123</td>
</tr>
<tr>
<td>ppc403GA serial</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>MPC555 SCI serial</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>MPC800 SMC UART serial</td>
<td></td>
<td>126</td>
</tr>
<tr>
<td>CL-CD2400 MPC serial</td>
<td></td>
<td>127</td>
</tr>
<tr>
<td>Semiconductor SA-1100 UART tyy</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>Siemens SAB 82532 UART tyy</td>
<td></td>
<td>129</td>
</tr>
<tr>
<td>END network interface</td>
<td></td>
<td>130</td>
</tr>
<tr>
<td>Communications Interface</td>
<td></td>
<td>131</td>
</tr>
<tr>
<td>Communications Interface</td>
<td></td>
<td>132</td>
</tr>
<tr>
<td>memory network (backplane)</td>
<td></td>
<td>133</td>
</tr>
<tr>
<td>Semi DP83932B SONIC Ethernet</td>
<td>sramDrv</td>
<td>134</td>
</tr>
<tr>
<td>ColdFire Serial Communications</td>
<td></td>
<td>135</td>
</tr>
<tr>
<td>PCMCIA SRAM device</td>
<td></td>
<td>136</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>ST 16C552 DUART tty</td>
<td>driver.</td>
<td>st16552Sio</td>
</tr>
<tr>
<td>PCMCIA host bus adaptor chip</td>
<td>driver. Databook TC1C/2</td>
<td>ifc</td>
</tr>
<tr>
<td>Elite END network interface</td>
<td>driver. SMC Ultra</td>
<td>ultaEnd</td>
</tr>
<tr>
<td>interface for ULIP</td>
<td>driver. WDB communication</td>
<td>wdbUlipPkDrv</td>
</tr>
<tr>
<td>Communications Controller</td>
<td>driver. Z8530 SCC Serial</td>
<td>z8530Sio</td>
</tr>
<tr>
<td>initialize ATA</td>
<td>driver.</td>
<td>ataDrv</td>
</tr>
<tr>
<td>PCI Ethernet network interface</td>
<td>driver. END style DEC 21x4x</td>
<td>dec21x4xEnd</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver. publish cmap network</td>
<td>cpattach</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver. publish cs network</td>
<td>csAttach</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver. publish esmc network</td>
<td>esmattach</td>
</tr>
<tr>
<td>initialize floppy disk</td>
<td>driver.</td>
<td>fdDrv</td>
</tr>
<tr>
<td>PCI Ethernet network interface</td>
<td>driver. END-style DEC 21x40</td>
<td>dec21x40End</td>
</tr>
<tr>
<td>initialize LPT</td>
<td>driver.</td>
<td>lptDrv</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver. publish mbc network</td>
<td>mbcattach</td>
</tr>
<tr>
<td>nicEvb network interface</td>
<td>driver. /and initialize</td>
<td>nicEvbattach</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. END style Intel 82596</td>
<td>e82596End</td>
</tr>
<tr>
<td>enable PCMCIA-SRAM</td>
<td>driver.</td>
<td>pccard5ramEnabled</td>
</tr>
<tr>
<td>enable PCMCIA-TFFS</td>
<td>driver.</td>
<td>pccardTffsEnabled</td>
</tr>
<tr>
<td>install PCMCIA SRAM memory</td>
<td>driver.</td>
<td>sramDrv</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. END style Intel 82557</td>
<td>fei82557End</td>
</tr>
<tr>
<td>network adapter END</td>
<td>driver. /PRO/1000 F/T/XF/XT/MT</td>
<td>gei82543End</td>
</tr>
<tr>
<td>16/8250 serial</td>
<td>driver.</td>
<td>if_cpm</td>
</tr>
<tr>
<td>CPM core network interface</td>
<td>driver. Motorola</td>
<td>if_cs</td>
</tr>
<tr>
<td>CS8900 network interface</td>
<td>driver. Crystal Semiconductor</td>
<td>if_cs</td>
</tr>
<tr>
<td>Ethernet LAN network interface</td>
<td>driver. DEC 21x4x</td>
<td>if_de</td>
</tr>
<tr>
<td>16 network interface</td>
<td>driver. Intel EtherExpress</td>
<td>if_eex</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. Intel 82596</td>
<td>if_ei</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. SMC 8013WC</td>
<td>if_elc</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. 3Com 3C509</td>
<td>if_elt</td>
</tr>
<tr>
<td>NE2000 network interface</td>
<td>driver. Novell/Eagle</td>
<td>if_ene</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. /Ethernet2 SMC-91c9x</td>
<td>if_esmc</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. Intel 82557</td>
<td>if_fei</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. Fujitsu MB86960 NICE</td>
<td>if_fn</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. AMD Am7990 LANCE</td>
<td>if_ln</td>
</tr>
<tr>
<td>Ethernet network interface</td>
<td>driver. /Am79C970 PCnet-PCI</td>
<td>if_InPci</td>
</tr>
<tr>
<td>ARM AMBA UART tty</td>
<td>driver.</td>
<td>ambaSio</td>
</tr>
<tr>
<td>loopback network interface</td>
<td>driver. software</td>
<td>if_loop</td>
</tr>
<tr>
<td>68EN302 network interface</td>
<td>driver. Motorola</td>
<td>if_mbc</td>
</tr>
<tr>
<td>ST-NIC Chip network interface</td>
<td>driver. /Semiconductor</td>
<td>if_nicEvb</td>
</tr>
<tr>
<td>IP (SLIP) network interface</td>
<td>driver. Serial Line</td>
<td>if_sl</td>
</tr>
<tr>
<td>backplane network interface</td>
<td>driver. shared memory</td>
<td>if_sm</td>
</tr>
<tr>
<td>SONIC Ethernet network</td>
<td>driver. /DP3932B</td>
<td>if_sn</td>
</tr>
<tr>
<td>initialize</td>
<td>driver and device.</td>
<td>auEndLoad</td>
</tr>
<tr>
<td>initialize</td>
<td>driver and device.</td>
<td>dec21x4xEndLoad</td>
</tr>
<tr>
<td>initialize</td>
<td>driver and device.</td>
<td>dec21x40EndLoad</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>eexattach</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>ei82596EndLoad</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>eiattach</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>eiattach</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>el3c90xEndLoad</td>
</tr>
<tr>
<td>interface and initialize</td>
<td>driver and device.</td>
<td>el3c90xEndLoad</td>
</tr>
</tbody>
</table>
Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface and initialize driver and device.</td>
<td>elcattach()</td>
<td>268</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>elt3c509Load()</td>
<td>269</td>
</tr>
<tr>
<td>elt interface and initialize driver and device.</td>
<td>eltattach()</td>
<td>271</td>
</tr>
<tr>
<td>interface and initialize driver and device.</td>
<td>elt82557EndLoad()</td>
<td>276</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>gei82543EndLoad()</td>
<td>282</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>iOlicomEndLoad()</td>
<td>286</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>In97xEndLoad()</td>
<td>288</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ln97xEndLoad()</td>
<td>292</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ln7990EndLoad()</td>
<td>294</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ln7990EndLoad()</td>
<td>294</td>
</tr>
<tr>
<td>interface and initialize driver and device.</td>
<td>lnPciattach()</td>
<td>295</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>mb86960EndLoad()</td>
<td>310</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>mbcEndLoad()</td>
<td>316</td>
</tr>
<tr>
<td>interface and initialize driver and device.</td>
<td>motCpmEndLoad()</td>
<td>327</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>motFccEndLoad()</td>
<td>328</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>motFccEndLoad()</td>
<td>329</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ne2000EndLoad()</td>
<td>351</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>nicEndLoad()</td>
<td>351</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ns83902EndLoad()</td>
<td>356</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>sh7615EndLoad()</td>
<td>407</td>
</tr>
<tr>
<td>interface and initialize driver and device.</td>
<td>slattach()</td>
<td>411</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>sn83932EndLoad()</td>
<td>416</td>
</tr>
<tr>
<td>interface and initialize driver and device.</td>
<td>snattach()</td>
<td>416</td>
</tr>
<tr>
<td>initialize driver and device.</td>
<td>ultraLoad()</td>
<td>431</td>
</tr>
<tr>
<td>END network interface driver for 3COM 3C509</td>
<td>elt3c509End</td>
<td>42</td>
</tr>
<tr>
<td>END network interface driver for 3COM 3C90xXL</td>
<td>elt3c90xEnd</td>
<td>38</td>
</tr>
<tr>
<td>Ethernet network interface driver for DVE-SH7XXX</td>
<td>if_eidve</td>
<td>69</td>
</tr>
<tr>
<td>Ethernet network interface driver for hkv3500</td>
<td>if_eihk</td>
<td>73</td>
</tr>
<tr>
<td>evaluation. NS16550 serial parallel chip device</td>
<td>evbNs16550Sio</td>
<td>45</td>
</tr>
<tr>
<td>END based packet driver for IBM PPC403GA</td>
<td>lptDrv</td>
<td>118</td>
</tr>
<tr>
<td>pipe packet driver for lightweight UDP/IP</td>
<td>wdbEndPktDrv</td>
<td>212</td>
</tr>
<tr>
<td>virtual generic file I/O</td>
<td>wdbNetromPktDrv</td>
<td>213</td>
</tr>
<tr>
<td>virtual tty I/O</td>
<td>wdbTtsfDrv</td>
<td>216</td>
</tr>
<tr>
<td>initialize TSFS device</td>
<td>wdbVioDrv</td>
<td>220</td>
</tr>
<tr>
<td>initialize tty</td>
<td>wdbVioDrv</td>
<td>441</td>
</tr>
<tr>
<td>install</td>
<td>mb86940DevInit()</td>
<td>442</td>
</tr>
<tr>
<td>(LOCAL and PCMCIA) disk device</td>
<td>atasShow</td>
<td>310</td>
</tr>
<tr>
<td>initialize ATA/IDE disk driver show routine.</td>
<td>atasShowInit()</td>
<td>14</td>
</tr>
<tr>
<td>shared memory network interface and initialize</td>
<td>smNetShow</td>
<td>239</td>
</tr>
<tr>
<td>space access support for PCI</td>
<td>lnattach()</td>
<td>199</td>
</tr>
<tr>
<td>all show routines for PCMCIA</td>
<td>pccConfigLib</td>
<td>294</td>
</tr>
<tr>
<td>support library for END-based register.</td>
<td>pccmciaShowInit()</td>
<td>174</td>
</tr>
<tr>
<td>return contents of set and clear bits in return</td>
<td>DUART auxiliary control</td>
<td>399</td>
</tr>
<tr>
<td>current contents of</td>
<td>DUART auxiliary control/</td>
<td>45</td>
</tr>
<tr>
<td>DUART interrupt-mask register.</td>
<td>DUART interrupt-mask register.</td>
<td>404</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>set and clear bits in DUART interrupt-mask register.</td>
<td>m68681ImrSetClr()</td>
<td>306</td>
</tr>
<tr>
<td>vector. handle all DUART interrupts in one</td>
<td>m68681Int()</td>
<td>307</td>
</tr>
<tr>
<td>configuration / return state of DUART output port</td>
<td>m68681Opcr()</td>
<td>307</td>
</tr>
<tr>
<td>return current state of DUART output port</td>
<td>m68681OpSetClr()</td>
<td>308</td>
</tr>
<tr>
<td>set and clear bits in DUART output port register.</td>
<td>m68681Opr()</td>
<td>308</td>
</tr>
<tr>
<td>ST 16C552 DUART tty driver.</td>
<td>st16552sio()</td>
<td>202</td>
</tr>
<tr>
<td>initialize driver and / publish eex network interface and</td>
<td>exattach()</td>
<td>260</td>
</tr>
<tr>
<td>initialize driver and / publish ei network interface and</td>
<td>eattach()</td>
<td>262</td>
</tr>
<tr>
<td>initialize driver and / publish /statistics for SMC 8013WC elc network interface.</td>
<td>ecshow()</td>
<td>269</td>
</tr>
<tr>
<td>initialize driver and / publish driver. SMC Ultra interface driver. SMC Elite Ultra Ethernet network</td>
<td>if_ultra</td>
<td>102</td>
</tr>
<tr>
<td>driver and device. publish elt interface and initialize elt network interface.</td>
<td>elattach()</td>
<td>271</td>
</tr>
<tr>
<td>display statistics for 3C509 eiec network interface and</td>
<td>elshow()</td>
<td>271</td>
</tr>
<tr>
<td>lightweight UDP/IP: END based packet driver for</td>
<td>wdbEndPktDrv()</td>
<td>212</td>
</tr>
<tr>
<td>F/T/XF/XT/MT network adapter END style AMD PRO/1000</td>
<td>giel97x()</td>
<td>50</td>
</tr>
<tr>
<td>Motorola 68302fads END network interface driver.</td>
<td>mbcEnd()</td>
<td>127</td>
</tr>
<tr>
<td>NE2000 END network interface driver.</td>
<td>ne2000End()</td>
<td>155</td>
</tr>
<tr>
<td>sh7615End END network interface driver.</td>
<td>sh7615End()</td>
<td>193</td>
</tr>
<tr>
<td>SMC Ultra Elite END network interface driver.</td>
<td>ultraEnd()</td>
<td>208</td>
</tr>
<tr>
<td>for 3COM 3C509. END network interface driver</td>
<td>if_3c509()</td>
<td>42</td>
</tr>
<tr>
<td>for 3COM 3C90xB XL. END network interface driver</td>
<td>if_3c90x()</td>
<td>38</td>
</tr>
<tr>
<td>initialize END packet device.</td>
<td>wdbEndPktDevInit()</td>
<td>439</td>
</tr>
<tr>
<td>Ethernet network interface/ END style AMD 7990 LANCE</td>
<td>ln7990End()</td>
<td>115</td>
</tr>
<tr>
<td>PCnet-PCI Ethernet driver. END style AMD Am79C97X</td>
<td>ln97xEnd()</td>
<td>110</td>
</tr>
<tr>
<td>Ethernet network interface/ END style Au MAC Ethernet</td>
<td>auEnd()</td>
<td>14</td>
</tr>
<tr>
<td>Ethernet network interface/ END style DEC 21x4x PCI</td>
<td>dec21x4x()</td>
<td>24</td>
</tr>
<tr>
<td>network interface driver. END style Intel 82557 Ethernet</td>
<td>fei82557End()</td>
<td>47</td>
</tr>
<tr>
<td>network interface driver. END style Intel 82596 Ethernet</td>
<td>ei82596End()</td>
<td>35</td>
</tr>
<tr>
<td>Ethernet network interface/ END style Intel Olcim PCMCIA</td>
<td>iOlcamEnd()</td>
<td>103</td>
</tr>
<tr>
<td>Ethernet network interface/ END style Motorola FCC</td>
<td>motFccEnd()</td>
<td>135</td>
</tr>
<tr>
<td>Ethernet network interface/ END style Motorola FEC</td>
<td>motFcEnd()</td>
<td>143</td>
</tr>
<tr>
<td>MC68EN360/MPC800 network/ display statistics for NE2000 ene network interface.</td>
<td>enesh()</td>
<td>277</td>
</tr>
<tr>
<td>initialize driver and / publish ene network interface and</td>
<td>enattach()</td>
<td>276</td>
</tr>
<tr>
<td>change MIB-II error count.</td>
<td>mib2ErrorAdd()</td>
<td>320</td>
</tr>
<tr>
<td>handle receiver/transmitter error interrupt.</td>
<td>m68562RxTxErrInt()</td>
<td>303</td>
</tr>
<tr>
<td>handle channel's error interrupt.</td>
<td>shScifIntErr()</td>
<td>408</td>
</tr>
<tr>
<td>handle channel's error interrupt.</td>
<td>shScifIntErr()</td>
<td>410</td>
</tr>
<tr>
<td>handle error interrupts.</td>
<td>ppc403IntEx()</td>
<td>401</td>
</tr>
<tr>
<td>handle error interrupts.</td>
<td>z8530IntEx()</td>
<td>444</td>
</tr>
<tr>
<td>interface driver. Intel EtherExpress 16 network</td>
<td>if_eex</td>
<td>65</td>
</tr>
<tr>
<td>form Ethernet address into packet.</td>
<td>endEtherAddressForm()</td>
<td>272</td>
</tr>
<tr>
<td>style AMD Am79C97X PCnet-PCI Ethernet driver. END</td>
<td>ln97xEnd()</td>
<td>110</td>
</tr>
<tr>
<td>END style Au MAC Ethernet driver.</td>
<td>auEnd()</td>
<td>14</td>
</tr>
<tr>
<td>Nat. Semi DP83932B SONIC Ethernet LAN network interface</td>
<td>if_dc</td>
<td>61</td>
</tr>
<tr>
<td>driver. DEC 21x4x Ethernet network driver.</td>
<td>if_sn</td>
<td>99</td>
</tr>
</tbody>
</table>
Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver. SMC Elite Ultra Ethernet network interface</td>
<td>if_ultra</td>
<td>102</td>
</tr>
<tr>
<td>END style AMD 7990 LANCE Ethernet network interface</td>
<td>ln7990End</td>
<td>115</td>
</tr>
<tr>
<td>END-style Fujitsu MB86960 Ethernet network interface</td>
<td>mb86960End</td>
<td>125</td>
</tr>
<tr>
<td>END style Motorola FEC Ethernet network interface</td>
<td>motFecEnd</td>
<td>135</td>
</tr>
<tr>
<td>END style DEC 21x4x PCI Ethernet network interface</td>
<td>dec21x4xEnd</td>
<td>24</td>
</tr>
<tr>
<td>END-style DEC 21x40 PCI Ethernet network interface</td>
<td>dec21x40End</td>
<td>28</td>
</tr>
<tr>
<td>driver. END style Intel 82596 Ethernet network interface</td>
<td>ei82596End</td>
<td>35</td>
</tr>
<tr>
<td>driver. END style Intel 82557 Ethernet network interface</td>
<td>fei82557End</td>
<td>47</td>
</tr>
<tr>
<td>driver. Intel 82596 Ethernet network interface</td>
<td>if_ei</td>
<td>66</td>
</tr>
<tr>
<td>driver. Intel 82557 Ethernet network interface</td>
<td>if_elc</td>
<td>76</td>
</tr>
<tr>
<td>Ampro Ethernet2 SMC-91c9x Ethernet network interface</td>
<td>if_esmc</td>
<td>80</td>
</tr>
<tr>
<td>driver. Intel 82557 Ethernet network interface</td>
<td>if_fei</td>
<td>81</td>
</tr>
<tr>
<td>driver. Fujitsu MB86960 NICE Ethernet network interface</td>
<td>if_fn</td>
<td>83</td>
</tr>
<tr>
<td>driver. AMD Am7990 LANCE Ethernet network interface</td>
<td>if_fn</td>
<td>83</td>
</tr>
<tr>
<td>AMD Am79C970 PCNet-PCI Ethernet network interface</td>
<td>if_InPci</td>
<td>88</td>
</tr>
<tr>
<td>driver for/ Intel 82596 Ethernet network interface</td>
<td>if_eidve</td>
<td>69</td>
</tr>
<tr>
<td>driver for/ Intel 82596 Ethernet network interface</td>
<td>if_elhk</td>
<td>73</td>
</tr>
<tr>
<td>7880 SCSI Host Adapter Library File. Adaptec virtual generic driver. SMC 8013WC Ethernet network interface</td>
<td>aic7880Lib</td>
<td>5</td>
</tr>
<tr>
<td>device and mount DOS file system, initialize</td>
<td>pccardMks()</td>
<td>365</td>
</tr>
<tr>
<td>mount DOS file system.</td>
<td>pccardMount()</td>
<td>365</td>
</tr>
<tr>
<td>initialize driver and/ publish fn network interface and</td>
<td>fnattach()</td>
<td>285</td>
</tr>
<tr>
<td>character to output word format. translate</td>
<td>nvr4101SIUCharToTxWord()</td>
<td>359</td>
</tr>
<tr>
<td>network interface/ END-style Fujitsu MB86960 Ethernet network interface</td>
<td>mb86960End</td>
<td>125</td>
</tr>
<tr>
<td>network interface driver. Fujitsu MB86960 NICE Ethernet network interface</td>
<td>if_fn</td>
<td>83</td>
</tr>
<tr>
<td>Controller (SCC) library. Fujitsu MB86960 SCSI Protocol</td>
<td>mb87030Lib</td>
<td>126</td>
</tr>
<tr>
<td>895 Chip. hardware initialization for</td>
<td>sym895HwInit()</td>
<td>425</td>
</tr>
<tr>
<td>show CIS information.</td>
<td>cisFree()</td>
<td>244</td>
</tr>
<tr>
<td>device by deviceld, then print information. find</td>
<td>pciFindDeviceShow()</td>
<td>394</td>
</tr>
<tr>
<td>print information about PCI devices.</td>
<td>pciDeviceShow()</td>
<td>392</td>
</tr>
<tr>
<td>memory network, show information about shared</td>
<td>smNetShow()</td>
<td>415</td>
</tr>
<tr>
<td>CIS, get information from PC card’s</td>
<td>cisGet()</td>
<td>244</td>
</tr>
<tr>
<td>hardware initialization for 895 Chip.</td>
<td>sym895HwInit()</td>
<td>425</td>
</tr>
<tr>
<td>low level initialization of ATA device.</td>
<td>iPIIX4AtaInit()</td>
<td>289</td>
</tr>
<tr>
<td>SIU.. initialization of NVR4101SIU</td>
<td>nvr4101SIUDevInit()</td>
<td>359</td>
</tr>
<tr>
<td>super IO (fcf37b78x) initialization source module.</td>
<td>smcFdc37b78x</td>
<td>196</td>
</tr>
<tr>
<td>VGA 3+ mode initialization source module.</td>
<td>vgaInit</td>
<td>210</td>
</tr>
<tr>
<td>CHIPS B69000 initialization source module.</td>
<td>ctB69000Vga</td>
<td>21</td>
</tr>
<tr>
<td>parse initialization string.</td>
<td>auInitParse()</td>
<td>240</td>
</tr>
<tr>
<td>parse initialization string.</td>
<td>e3c90xInitParse()</td>
<td>266</td>
</tr>
<tr>
<td>parse initialization string.</td>
<td>ln97xInitParse()</td>
<td>292</td>
</tr>
<tr>
<td>parse initialization string.</td>
<td>mb86960InitParse()</td>
<td>311</td>
</tr>
<tr>
<td>parse initialization string.</td>
<td>nicEvbInitParse()</td>
<td>352</td>
</tr>
<tr>
<td>adaptor chip library. Intel 82365SL PCMCIA host bus</td>
<td>pcie</td>
<td>173</td>
</tr>
<tr>
<td>adaptor chip show library. Intel 82365SL PCMCIA host bus</td>
<td>pcieShow</td>
<td>185</td>
</tr>
<tr>
<td>interface driver. END style Intel 82557 Ethernet network interface</td>
<td>fei82557End</td>
<td>47</td>
</tr>
<tr>
<td>interface driver. Intel 82557 Ethernet network interface</td>
<td>if_fei</td>
<td>81</td>
</tr>
</tbody>
</table>
interface driver. END style Intel 82596 Ethernet network .................. ei82596End 35
interface driver. Intel 82596 Ethernet network if_el 66
interface driver for/ Intel 82596 Ethernet network if_eidve 69
interface driver for hkv3500. Intel 82596 Ethernet network if_eihk 73
interface driver. Intel EtherExpress 16 network if_eex 65
interface driver. END style Intel Olicom PCMCIA network iOlicomEnd 72
network adapter END driver. Intel PRO/1000 F/T/XF/XT/MT gei82543End 50
handle receiver interrupt. i8250Int() 235
handle transmitter interrupt. amiIntRx() 235
handle receiver/transmitter interrupt. m68332Int() 287
handle SCC interrupt. m68360Int() 300
handle receiver interrupt. m68562RxInt() 301
receiver/transmitter error interrupt. m68562RtxErrInt() 302
handle transmitter interrupt. m68562TxInt() 303
handle receiver interrupt. n72001IntRd() 332
handle transmitter interrupt. n72001IntWr() 332
handle receiver interrupt. ns16550IntRd() 355
handle transmitter interrupt. ns16550IntWr() 355
handler for shared PCI interrupt. pciInt() 395
interrupt handler to PCI interrupt. pciIntConnect() 395
interrupt handler from PCI Interrupt disconnect. pciIntDisconnect2() 396
handle receiver interrupt. ppc403IntRd() 401
handle transmitter interrupt. ppc403IntWr() 402
handle channel’s interrupt. ppc555SciInt() 403
handle SMC interrupt. ppc860Int() 404
handle channel’s error interrupt. sa1100Int() 405
channel’s receive-character interrupt. shSciIntErr() 408
channel’s transmitter-ready interrupt. shSciIntRcv() 409
channel’s receive-character interrupt. shSciIntRcv() 410
channel’s transmitter-ready interrupt. st16552IntRd() 420
channel’s receive-character interrupt. st16552IntWr() 420
handle receiver interrupt. z8530IntRd() 444
handle transmitter interrupt. z8530IntWr() 445
handle receiver/transmitter Interrupt disconnect for NS 16550 chip. evbNs16550Int() 279
network interface interrupt handler. mbcInt() 317
PCI interrupt. interrupt handler for shared pciInt() 395
interrupt disconnect. pciIntDisconnect2() 396
interrupt handler from PCI pciIntDisconnect() 396
interrupt connect. pciIntConnect() 395
interrupt level processing. n72001Int() 331
interrupt level processing. ns16550Int() 354
interrupt level processing. nvr4101DSIU() 357
interrupt level processing. nvr4101DSIU() 360
interrupt level processing. nvr4102DSIU() 362
interrupt level processing. sab82532Int() 406
interrupt level processing. st16552Int() 419

456
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplexed interrupt level processing</td>
<td>st16552MuxInt()</td>
<td>421</td>
</tr>
<tr>
<td>give device interrupt level to use</td>
<td>fPiIX4GetIntr()</td>
<td>290</td>
</tr>
<tr>
<td>set and clear bits in UART's contents, return current interrupt mask register</td>
<td>coldfireImrSetClr()</td>
<td>248</td>
</tr>
<tr>
<td>miscellaneous interrupt processing</td>
<td>coldfireImr()</td>
<td>247</td>
</tr>
<tr>
<td>miscellaneous interrupt processing</td>
<td>ns16550IntEx()</td>
<td>354</td>
</tr>
<tr>
<td>miscellaneous interrupt processing</td>
<td>st16552IntEx()</td>
<td>419</td>
</tr>
<tr>
<td>interrupts</td>
<td>iOlicomIntHandle()</td>
<td>288</td>
</tr>
<tr>
<td>SCSI Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCI Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle special status interrupts</td>
<td>cd2400In()</td>
<td>242</td>
</tr>
<tr>
<td>handle receiver interrupts</td>
<td>cd2400InRx()</td>
<td>242</td>
</tr>
<tr>
<td>handle transmitter interrupts</td>
<td>cd2400InTx()</td>
<td>243</td>
</tr>
<tr>
<td>interrupt service for card</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle error interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle error interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>entry point for handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from 82596</td>
<td>etin()</td>
<td>264</td>
</tr>
<tr>
<td>mask interrupts from DSIIU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from SIU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from SIU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from SIU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from SIU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>entry point for handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do raw I/O access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>provide raw I/O access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO port addresses for super</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual generic file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual tty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(SCSI-1), NCR 53C710 SCSI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(SCSI-2), NCR 53C710 SCSI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(SCSI-2), NCR 53C8xx PCI SCSI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interface/ END style AMD Am7990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interface driver. AMD Am7990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>free tuples from linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended capability in ECP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>free tuples from linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of specific type from probe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>capability in ECP linked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>initialize driver / publish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>initialize driver and / publish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chip. this routine performs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>driver. software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chip device for IBM-PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>initialize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>create device for show</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask interrupts from DSIIU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Keyword Index**
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel PRO/1000 F/T/XF/XT/MT network adapter</td>
<td>sn83932End</td>
<td>50</td>
</tr>
<tr>
<td>NCR 53C710 (SIOP) library (SCSI-1)</td>
<td>ncr710SetHwRegisterScsi2()</td>
<td>151</td>
</tr>
<tr>
<td>NCR 53C710 (SIOP) library (SCSI-2)</td>
<td>ncr710Lib</td>
<td>151</td>
</tr>
<tr>
<td>display values of all readable Processor (SIOP) library</td>
<td>ncr710Show()</td>
<td>39</td>
</tr>
<tr>
<td>create control structure for NCR 53C710 SIOP</td>
<td>ncr710CreateScsi2()</td>
<td>338</td>
</tr>
<tr>
<td>display values of all readable Controller (ASC) library</td>
<td>ncr810Show()</td>
<td>340</td>
</tr>
<tr>
<td>create control structure for NCR 53C8xx SIOP</td>
<td>ncr810Create()</td>
<td>342</td>
</tr>
<tr>
<td>create control structure for NCR 53C8xx SIOP</td>
<td>ncr810CreateScsi2()</td>
<td>345</td>
</tr>
<tr>
<td>display values of all readable Controller (ASC) library</td>
<td>ncr810Show()</td>
<td>342</td>
</tr>
<tr>
<td>controller for NCR 53C90 ASC</td>
<td>ncr5390CreateScsi2()</td>
<td>347</td>
</tr>
<tr>
<td>controller for NCR 53C90 ASC</td>
<td>ncr5390Create()</td>
<td>349</td>
</tr>
<tr>
<td>display statistics for NE2000 ene network interface</td>
<td>eneShow()</td>
<td>277</td>
</tr>
<tr>
<td>driver. Novell/Eagle</td>
<td>if_ene</td>
<td>78</td>
</tr>
<tr>
<td>driver. NEC 765 floppy disk device</td>
<td>ncr710SetHwRegister()</td>
<td>333</td>
</tr>
<tr>
<td>(Multiprotocol Serial/)</td>
<td>ncr710SetHwRegister()</td>
<td>337</td>
</tr>
<tr>
<td>driver. NEC PD72001 MPSC</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>driver. NEC VR4101 DSU UART tty</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>driver. NEC VR4101 SIU UART tty</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>driver. NEC VR4102 DSU UART tty</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>agent. initialize NETROM packet device for WDB</td>
<td>wdbNetromPktDevInit()</td>
<td>439</td>
</tr>
<tr>
<td>agent. NETROM packet driver for WDB</td>
<td>wdbNetromPktDrv</td>
<td>213</td>
</tr>
<tr>
<td>about shared memory network. show information</td>
<td>smNetShow()</td>
<td>415</td>
</tr>
<tr>
<td>Intel PRO/1000 F/T/XF/XT/MT network adapter END driver</td>
<td>gei82543End</td>
<td>50</td>
</tr>
<tr>
<td>network (backplane) driver</td>
<td>smNetLib</td>
<td>198</td>
</tr>
<tr>
<td>shared memory network driver. show routines</td>
<td>smNetShow()</td>
<td>199</td>
</tr>
<tr>
<td>shows statistics for cs network interface</td>
<td>csShow()</td>
<td>253</td>
</tr>
<tr>
<td>publish dc network interface</td>
<td>dcattach()</td>
<td>254</td>
</tr>
<tr>
<td>statistics for SMC 8013WC etc</td>
<td>ncr710SetHwRegister()</td>
<td>333</td>
</tr>
<tr>
<td>statistics for 3C509 elt</td>
<td>ncr710SetHwRegister()</td>
<td>337</td>
</tr>
<tr>
<td>display statistics for esmc network interface</td>
<td>esmcShow()</td>
<td>278</td>
</tr>
<tr>
<td>publish fei network interface</td>
<td>feattach()</td>
<td>284</td>
</tr>
<tr>
<td>display statistics for ultra network interface</td>
<td>ultraShow()</td>
<td>432</td>
</tr>
<tr>
<td>initialize driver. publish cpm network interface</td>
<td>cpmattach()</td>
<td>250</td>
</tr>
<tr>
<td>initialize driver. publish cs network interface</td>
<td>csAttach()</td>
<td>252</td>
</tr>
<tr>
<td>initialize driver. publish esmc network interface</td>
<td>esmcattach()</td>
<td>277</td>
</tr>
<tr>
<td>initialize driver. publish mbc network interface</td>
<td>mbattach()</td>
<td>315</td>
</tr>
<tr>
<td>network interface and</td>
<td>exattach()</td>
<td>260</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>initialize driver / publish ei</td>
<td>network interface and eiattach()</td>
<td>262</td>
</tr>
<tr>
<td>initialize driver / publish ei</td>
<td>network interface and eihkattach()</td>
<td>263</td>
</tr>
<tr>
<td>initialize driver / publish eic</td>
<td>network interface and elcattach()</td>
<td>268</td>
</tr>
<tr>
<td>initialize driver / publish fn</td>
<td>network interface and fnattach()</td>
<td>285</td>
</tr>
<tr>
<td>initialize / publish InPci</td>
<td>network interface and InPciattach()</td>
<td>295</td>
</tr>
<tr>
<td>initialize driver / publish sl</td>
<td>network interface and slattach()</td>
<td>411</td>
</tr>
<tr>
<td>initialize driver / publish sn</td>
<td>network interface and snattach()</td>
<td>416</td>
</tr>
<tr>
<td>initialize driver / publish lo</td>
<td>network interface and loattach()</td>
<td>296</td>
</tr>
<tr>
<td>initialize driver / publish ln</td>
<td>network interface and lnattach()</td>
<td>294</td>
</tr>
<tr>
<td>output packet to network interface device</td>
<td>cpmStartOutput()</td>
<td>319</td>
</tr>
<tr>
<td>/Motorola 68302fads END</td>
<td>network interface driver</td>
<td>102</td>
</tr>
<tr>
<td>/Motorola MC68EN360/MPC800</td>
<td>network interface driver END</td>
<td>103</td>
</tr>
<tr>
<td>style AMD 7990 LANCE Ethernet</td>
<td>network interface driver</td>
<td>115</td>
</tr>
<tr>
<td>AMD Olicom PCMCIA</td>
<td>network interface driver iOlicomEnd</td>
<td>125</td>
</tr>
<tr>
<td>AMD 7990 LANCE Ethernet</td>
<td>network interface driver</td>
<td>127</td>
</tr>
<tr>
<td>Fujitsu MB86960 Ethernet</td>
<td>network interface driver</td>
<td>132</td>
</tr>
<tr>
<td>Fujitsu MB86960 Ethernet END</td>
<td>network interface driver</td>
<td>135</td>
</tr>
<tr>
<td>DEC 21x4x PCI Ethernet</td>
<td>network interface driver END</td>
<td>137</td>
</tr>
<tr>
<td>DEC 21x40 PCI Ethernet</td>
<td>network interface driver END</td>
<td>24</td>
</tr>
<tr>
<td>DEC 21x4x Ethernet LAN</td>
<td>network interface driver dec21x4xEnd</td>
<td>28</td>
</tr>
<tr>
<td>Intel EtherExpress 16</td>
<td>network interface driver</td>
<td>61</td>
</tr>
<tr>
<td>Intel 82596 Ethernet</td>
<td>network interface driver if_eex</td>
<td>65</td>
</tr>
<tr>
<td>Intel 82596 Ethernet</td>
<td>network interface driver if_ei</td>
<td>66</td>
</tr>
<tr>
<td>SMC 8013WC Ethernet</td>
<td>network interface driver if_e1c</td>
<td>76</td>
</tr>
<tr>
<td>3Com 3C509 Ethernet</td>
<td>network interface driver if_el</td>
<td>77</td>
</tr>
<tr>
<td>Novell/Eagle NE2000 END</td>
<td>network interface driver if_ene</td>
<td>78</td>
</tr>
<tr>
<td>/Ethernet2 SMC-919x Ethernet</td>
<td>network interface driver if_esmc</td>
<td>80</td>
</tr>
<tr>
<td>Intel 82557 Ethernet</td>
<td>network interface driver if_fei</td>
<td>81</td>
</tr>
<tr>
<td>AMD Am7990 LANCE Ethernet</td>
<td>network interface driver if_fn</td>
<td>83</td>
</tr>
<tr>
<td>AMD Am7990 LANCE Ethernet END</td>
<td>network interface driver</td>
<td>85</td>
</tr>
<tr>
<td>3Com 3C509. END</td>
<td>network interface driver el3c509End</td>
<td>42</td>
</tr>
<tr>
<td>3COM 3C90B XL. END</td>
<td>network interface driver el3c90xEnd</td>
<td>38</td>
</tr>
<tr>
<td>Intel 82596 Ethernet</td>
<td>network interface driver if_eideve</td>
<td>69</td>
</tr>
<tr>
<td>hkv3500. Intel 82596 Ethernet</td>
<td>network interface driver if_eihkve</td>
<td>73</td>
</tr>
</tbody>
</table>
Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler.</td>
<td>network interface interrupt</td>
<td>mbcIntr()</td>
</tr>
<tr>
<td>prints current value of</td>
<td>NIC registers.</td>
<td>ns83902RegShow()</td>
</tr>
<tr>
<td>interface/ Fujitsu MB86960</td>
<td>NICE Ethernet network</td>
<td>if_fn</td>
</tr>
<tr>
<td>interface driver.</td>
<td>Novell/Eagle NE2000 network</td>
<td>if_enet</td>
</tr>
<tr>
<td>initialization of</td>
<td>NVR4101SIU S1U</td>
<td>nvr4101SIUDevInit()</td>
</tr>
<tr>
<td>read all PHY registers</td>
<td>out.</td>
<td>dec2114SPIReadBack()</td>
</tr>
<tr>
<td>form Ethernet address into</td>
<td>packet.</td>
<td>endEtherAddressForm()</td>
</tr>
<tr>
<td>locate addresses in</td>
<td>packet.</td>
<td>endEtherPacketAddrGet()</td>
</tr>
<tr>
<td>return beginning of</td>
<td>packet data.</td>
<td>endEtherPacketDataGet()</td>
</tr>
<tr>
<td>initialize END</td>
<td>device.</td>
<td>wdbEndPktDevInit()</td>
</tr>
<tr>
<td>initialize pipe</td>
<td>device.</td>
<td>wdbPipePktDevInit()</td>
</tr>
<tr>
<td>initialize NETROM</td>
<td>device for WDB agent</td>
<td>wdbNetromPktDevInit()</td>
</tr>
<tr>
<td>initialize SLIP</td>
<td>device for WDB agent</td>
<td>wdbSlipPktDevInit()</td>
</tr>
<tr>
<td>UDP/IP. END based</td>
<td>packet driver for lightweight</td>
<td>wdbEndPktDrv</td>
</tr>
<tr>
<td>UDP/IP. pipe</td>
<td>packet driver for lightweight</td>
<td>wdbPipePktDrv</td>
</tr>
<tr>
<td>NETROM</td>
<td>packet driver for WDB agent</td>
<td>wdbNetromPktDrv</td>
</tr>
<tr>
<td>copy</td>
<td>packet to interface.</td>
<td>elePut()</td>
</tr>
<tr>
<td>copy</td>
<td>packet to interface.</td>
<td>enePut()</td>
</tr>
<tr>
<td>copy</td>
<td>packet to interface.</td>
<td>esmcPut()</td>
</tr>
<tr>
<td>copy</td>
<td>packet to interface.</td>
<td>ultraPut()</td>
</tr>
<tr>
<td>device. output</td>
<td>packet to network interface</td>
<td>cpmStartOutput()</td>
</tr>
<tr>
<td>device. output</td>
<td>packet to network interface</td>
<td>mbcStartOutput()</td>
</tr>
<tr>
<td>conditions. align</td>
<td>PCI address and check boundary</td>
<td>pciAutoAddAddrAlign()</td>
</tr>
<tr>
<td>initialize</td>
<td>PCI autoconfig library.</td>
<td>pciAutoConfigLibInit()</td>
</tr>
<tr>
<td>assign PCI space to single</td>
<td>PCI base address register.</td>
<td>pciAutoRegConfig()</td>
</tr>
<tr>
<td>configure device on</td>
<td>PCI bus.</td>
<td>pciDevConfig()</td>
</tr>
<tr>
<td>show routines of</td>
<td>PCI bus (IO mapped) library.</td>
<td>pciConfigShow()</td>
</tr>
<tr>
<td>perform</td>
<td>PCI bus scan.</td>
<td>aic7880GetNumOfBuses()</td>
</tr>
<tr>
<td>allocation facility.</td>
<td>PCI bus scan and resource</td>
<td>pciAutoConfigLib</td>
</tr>
<tr>
<td>nth occurrence of device by</td>
<td>PCI class code. find</td>
<td>pciFindClass()</td>
</tr>
<tr>
<td>read from</td>
<td>PCI config space.</td>
<td>aic7880ReadConfig()</td>
</tr>
<tr>
<td>read to</td>
<td>PCI config space.</td>
<td>aic7880WriteConfig()</td>
</tr>
<tr>
<td>read one byte from</td>
<td>PCI configuration space.</td>
<td>pciConfigInByte()</td>
</tr>
<tr>
<td>read one longword from</td>
<td>PCI configuration space.</td>
<td>pciConfigInLong()</td>
</tr>
<tr>
<td>read one word from</td>
<td>PCI configuration space.</td>
<td>pciConfigInWord()</td>
</tr>
<tr>
<td>write one byte to</td>
<td>PCI configuration space.</td>
<td>pciConfigOutByte()</td>
</tr>
<tr>
<td>write one longword to</td>
<td>PCI configuration space.</td>
<td>pciConfigOutLong()</td>
</tr>
<tr>
<td>write one 16-bit word to</td>
<td>PCI configuration space.</td>
<td>pciConfigOutWord()</td>
</tr>
<tr>
<td>support for PCI drivers.</td>
<td>PCI Configuration space access</td>
<td>pciConfigLib</td>
</tr>
<tr>
<td>print header of specified</td>
<td>PCI device.</td>
<td>pciHeaderShow()</td>
</tr>
<tr>
<td>writeable status/ quiesce</td>
<td>PCI device and reset all</td>
<td>pciAutoDevReset()</td>
</tr>
<tr>
<td>print information about</td>
<td>PCI devices.</td>
<td>pciDeviceShow()</td>
</tr>
<tr>
<td>space access support for</td>
<td>PCI drivers.</td>
<td>pciConfigLib</td>
</tr>
<tr>
<td>driver. END style DEC 21x4x</td>
<td>PCI Ethernet network interface</td>
<td>dec21x4xEth</td>
</tr>
<tr>
<td>driver. END-style DEC 21x40</td>
<td>PCI Ethernet network interface</td>
<td>dec21x40Eth</td>
</tr>
<tr>
<td>disable specific</td>
<td>PCI function.</td>
<td>pciAutoFuncDisable()</td>
</tr>
<tr>
<td>configure all nonexcluded</td>
<td>PCI headers. automatically</td>
<td>pciAutoCfg()</td>
</tr>
<tr>
<td>/configure all nonexcluded</td>
<td>PCI headers; obsolete.</td>
<td>pciAutoCfg()</td>
</tr>
<tr>
<td>interrupt handler for shared</td>
<td>PCI interrupt.</td>
<td>pciInt()</td>
</tr>
<tr>
<td>connect interrupt handler to</td>
<td>PCI interrupt.</td>
<td>pciIntConnect()</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>INTERRUPT HANDLER FROM</td>
<td>PCI INTERRUPT DISCONNECT</td>
<td>pciIntDisconnect()</td>
</tr>
<tr>
<td>level initialization code for</td>
<td>PCI ISA/IDE XCelerator LOW</td>
<td>pciIntx()</td>
</tr>
<tr>
<td>library (SCSI-2), NCR 53C8xx</td>
<td>PCI SCSI I/O Processor (SIOP)</td>
<td>ncr810Lib()</td>
</tr>
<tr>
<td>address register assignment</td>
<td>PCI shared interrupt support</td>
<td>pciIntLib()</td>
</tr>
<tr>
<td>show</td>
<td>PCI space to single PCI base</td>
<td>pciAutoRegConfig()</td>
</tr>
<tr>
<td>show all configurations of</td>
<td>PCI topology</td>
<td>pciConfigTopoShow()</td>
</tr>
<tr>
<td></td>
<td>PMICIA chip</td>
<td>pmciaShow()</td>
</tr>
<tr>
<td></td>
<td>PMICIA CIS library</td>
<td>cisLib()</td>
</tr>
<tr>
<td></td>
<td>PMICIA CIS show library</td>
<td>cisShow()</td>
</tr>
<tr>
<td>get</td>
<td>PMICIA configuration register</td>
<td>cisConfigRegGet()</td>
</tr>
<tr>
<td>SET</td>
<td>PMICIA configuration register</td>
<td>cisConfigRegSet()</td>
</tr>
<tr>
<td>/and ATAPI CDROM (LOCAL and PCMCIA) disk device driver</td>
<td>ATA/IDE (LOCAL and PCMCIA) disk device driver</td>
<td>ataDrv()</td>
</tr>
<tr>
<td>show / ATA/IDE (LOCAL and PCMCIA) all show routines for</td>
<td>PMICIA disk device driver</td>
<td>ataShow()</td>
</tr>
<tr>
<td>enable</td>
<td>PMICIA drivers, initialize</td>
<td>pcmciaShowInit()</td>
</tr>
<tr>
<td>facilities, generic</td>
<td>PMICIA Etherlink III card</td>
<td>pcmciaLib()</td>
</tr>
<tr>
<td>initialize</td>
<td>PMICIA event-handling package</td>
<td>pcmciaInit()</td>
</tr>
<tr>
<td>handle task-level</td>
<td>PMICIA event-handling</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>Databook TCIC/2</td>
<td>Databook TCIC/2 Host Bus Adaptor Chip</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>library, Intel 82365SL</td>
<td>Databook TCIC/2 Host Bus Adaptor Chip</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>show library, Intel 82365SL</td>
<td>PMICIA host bus adaptor chip</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>create</td>
<td>PMICIA host bus adaptor chip</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>ISA address space map</td>
<td>PMICIA memory disk device</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>END style Intel Olicom</td>
<td>PMICIA memory onto specified</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>PCMCIA network interface</td>
<td>Olicom END</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>show library</td>
<td>Olicom END</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>install</td>
<td>PMICIA SRAM device driver</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>UNINITIALIZE</td>
<td>PMICIA SRAM memory driver</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>find first</td>
<td>PHY connected to DEC MII port</td>
<td>dec21x40PhyFind()</td>
</tr>
<tr>
<td>initialize and configure</td>
<td>PHY devices</td>
<td>dec21x40PhyIni()</td>
</tr>
<tr>
<td>read all</td>
<td>PHY registers out</td>
<td>dec2114SFPitimate()</td>
</tr>
<tr>
<td>initialize</td>
<td>PIPE packets out</td>
<td>dec2114SFPitimate()</td>
</tr>
<tr>
<td>lightweight UDP/IP: Processor (SIOP) library</td>
<td>Processor (SIOP) library</td>
<td>ncr710Lib()</td>
</tr>
<tr>
<td>(SCSI-1), NCR 53C710 SCSI I/O</td>
<td>Processor (SIOP) library</td>
<td>ncr710Lib()</td>
</tr>
<tr>
<td>(SCSI-2), NCR 53C710 SCSI I/O</td>
<td>Processor (SIOP) library</td>
<td>ncr810Lib()</td>
</tr>
<tr>
<td>NCR 53C8xx PCI S3 PLUS PCI host bus adaptor chip</td>
<td>Host Bus Adaptor Chip</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>raw I/O access</td>
<td>pmciaLib()</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>PROVIDE</td>
<td>raw I/O access</td>
<td>pmciaLib()</td>
</tr>
<tr>
<td>read all PHY registers out</td>
<td>dec21145SFPitimate()</td>
<td>dec21145SFPitimate()</td>
</tr>
<tr>
<td>read all PHY devices</td>
<td>dec2114SFPitimate()</td>
<td>dec2114SFPitimate()</td>
</tr>
<tr>
<td>read entire serial rom</td>
<td>decReadAllRom()</td>
<td>decReadAllRom()</td>
</tr>
<tr>
<td>read from PCI config space</td>
<td>aic8788ReadConfig()</td>
<td>aic8788ReadConfig()</td>
</tr>
<tr>
<td>read one byte from PCI</td>
<td>pciConfigInByte()</td>
<td>pciConfigInByte()</td>
</tr>
<tr>
<td>read one longword from PCI</td>
<td>pciConfigInWord()</td>
<td>pciConfigInWord()</td>
</tr>
<tr>
<td>read one word from PCI</td>
<td>pciConfigInWord()</td>
<td>pciConfigInWord()</td>
</tr>
<tr>
<td>read to PCI config space</td>
<td>aic8788WriteConfig()</td>
<td>aic8788WriteConfig()</td>
</tr>
<tr>
<td>read two bytes from serial</td>
<td>dec21140SromWordRead()</td>
<td>dec21140SromWordRead()</td>
</tr>
<tr>
<td>get PCMCIA configuration register</td>
<td>cisConfigRegGet()</td>
<td>cisConfigRegGet()</td>
</tr>
<tr>
<td>set PCMCIA configuration register</td>
<td>cisConfigRegSet()</td>
<td>cisConfigRegSet()</td>
</tr>
<tr>
<td>Bits in UART’s AUX control</td>
<td>register set and clear</td>
<td>coldfireAcrSetClr()</td>
</tr>
</tbody>
</table>
Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits in UART’s interrupt mask register. set and clear...</td>
<td>coldfireImrSetClr()</td>
<td>248</td>
</tr>
<tr>
<td>return current state of output register.</td>
<td>coldfireOmr()</td>
<td>249</td>
</tr>
<tr>
<td>and clear bits in output port of DUART auxiliary control</td>
<td>coldfireOprSetClr()</td>
<td>249</td>
</tr>
<tr>
<td>in DUART auxiliary control</td>
<td>m68681Acr()</td>
<td>304</td>
</tr>
<tr>
<td>of DUART interrupt-mask register.</td>
<td>m6881Ac()</td>
<td>304</td>
</tr>
<tr>
<td>bits in DUART interrupt-mask register.</td>
<td>m68681Imr()</td>
<td>306</td>
</tr>
<tr>
<td>output port configuration</td>
<td>m68681Ocr()</td>
<td>307</td>
</tr>
<tr>
<td>output port configuration</td>
<td>m68681OpcrSetClr()</td>
<td>308</td>
</tr>
<tr>
<td>state of DUART output port</td>
<td>m68681Opr()</td>
<td>308</td>
</tr>
<tr>
<td>bits in DUART output port</td>
<td>m68681OprSetClr()</td>
<td>309</td>
</tr>
<tr>
<td>to single PCI base address</td>
<td>pciAutoRegConfig()</td>
<td>379</td>
</tr>
<tr>
<td>for Configuration Address</td>
<td>pciConfigBdPack()</td>
<td>380</td>
</tr>
<tr>
<td>return aux control registers.</td>
<td>coldfireAcr()</td>
<td>245</td>
</tr>
<tr>
<td>return current interrupt mask register contents.</td>
<td>coldfireIm()</td>
<td>247</td>
</tr>
<tr>
<td>perform masked longword register update.</td>
<td>pciConfigModifyByte()</td>
<td>385</td>
</tr>
<tr>
<td>perform masked longword register update.</td>
<td>pciConfigModifyLong()</td>
<td>386</td>
</tr>
<tr>
<td>perform masked longword register update.</td>
<td>pciConfigModifyWord()</td>
<td>387</td>
</tr>
<tr>
<td>display dec 21140/21140 status registers. display values of...</td>
<td>m87030Show()</td>
<td>314</td>
</tr>
<tr>
<td>all readable MB87030 SPIC get contents of MII registers. display values of...</td>
<td>miiRegsGet()</td>
<td>326</td>
</tr>
<tr>
<td>all readable NCR 53C710 SIOP registers. display values of...</td>
<td>ncr710Show()</td>
<td>339</td>
</tr>
<tr>
<td>all readable NCR 53C710 SIOP registers. display values of...</td>
<td>ncr710ShowScsi2()</td>
<td>340</td>
</tr>
<tr>
<td>all readable NCR5390 chip registers. display values of...</td>
<td>ncr810Show()</td>
<td>345</td>
</tr>
<tr>
<td>all readable NCR5390 chip registers. display values of...</td>
<td>ncr810ShowScsi2()</td>
<td>345</td>
</tr>
<tr>
<td>all readable SIM 53C8xx SIOP chip points current value of NIC registers. display values of...</td>
<td>ncr5390Show()</td>
<td>350</td>
</tr>
<tr>
<td>all readable SIM 53C8xx SIOP chip points current value of NIC registers. display values of...</td>
<td>ns83902RegShow()</td>
<td>356</td>
</tr>
<tr>
<td>all readable WD33C93 chip registers. display values of...</td>
<td>wd33c93Show()</td>
<td>348</td>
</tr>
<tr>
<td>all readable WD33C93 chip registers. display values of...</td>
<td>wd33c93Show()</td>
<td>348</td>
</tr>
<tr>
<td>all readable WD33C93 chip registers. display values of...</td>
<td>wd33c93Show()</td>
<td>348</td>
</tr>
<tr>
<td>set hardware-dependent registers for NCR 53C710.</td>
<td>ncr710SetHwRegisterScsi2()</td>
<td>338</td>
</tr>
<tr>
<td>set hardware-dependent registers for NCR 53C8xx SIOP.</td>
<td>ncr810SetHwRegister()</td>
<td>337</td>
</tr>
<tr>
<td>set hardware-dependent registers for NCR 53C8xx SIOP.</td>
<td>ncr810SetHwRegister()</td>
<td>344</td>
</tr>
<tr>
<td>set pointer to MII optional registers.</td>
<td>miiPhyOptFuncSet()</td>
<td>325</td>
</tr>
<tr>
<td>set pointers to MII optional registers.</td>
<td>miiPhyOptFuncMultiSet()</td>
<td>325</td>
</tr>
<tr>
<td>read all PHF ROM.</td>
<td>dec21140SromWordRead()</td>
<td>258</td>
</tr>
<tr>
<td>read two bytes from serial ROM.</td>
<td>dec21140SromWordRead()</td>
<td>258</td>
</tr>
<tr>
<td>display lines of serial ROM for dec21140.</td>
<td>dcViewRom()</td>
<td>256</td>
</tr>
<tr>
<td>route PIRQ[A:D].</td>
<td>iPIX4IntrRoute()</td>
<td>291</td>
</tr>
<tr>
<td>Siemens SAB 82532 UART tty driver.</td>
<td>sab82532()</td>
<td>192</td>
</tr>
<tr>
<td>partially initialize WD33C93 SBIC structure. create and...</td>
<td>wd33c93CtrlCreate()</td>
<td>434</td>
</tr>
<tr>
<td>and partially initialize SBIC structure. create...</td>
<td>wd33c93CtrlCreateScsi2()</td>
<td>435</td>
</tr>
<tr>
<td>user-specified fields in SCC.</td>
<td>m68332DevInit()</td>
<td>300</td>
</tr>
<tr>
<td>initialize SCC.</td>
<td>m68332DevInit()</td>
<td>301</td>
</tr>
<tr>
<td>read entire serial</td>
<td>dcReadAllRom()</td>
<td>255</td>
</tr>
<tr>
<td>display lines of serial ROM for dec21140.</td>
<td>dcViewRom()</td>
<td>256</td>
</tr>
<tr>
<td>Controller driver.</td>
<td>iop1()</td>
<td>291</td>
</tr>
<tr>
<td>Motorola MC68360 SCC UART serial driver.</td>
<td>m68360()</td>
<td>120</td>
</tr>
<tr>
<td>driver for Symbios SYM895 SCSI Controller.</td>
<td>sym895Lib()</td>
<td>204</td>
</tr>
<tr>
<td>interrupt service routine for SCSI Controller.</td>
<td>sym895Intr()</td>
<td>425</td>
</tr>
<tr>
<td>Keyword</td>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>(SCSI-1). NCR 53C90 Advanced</td>
<td>SCSI Controller (ASC) library ........................................... ncr5390Lib1</td>
<td>154</td>
</tr>
<tr>
<td>(SCSI-2). NCR 53C90 Advanced</td>
<td>SCSI Controller (ASC) library ........................................... ncr5390Lib2</td>
<td>154</td>
</tr>
<tr>
<td>enable double speed</td>
<td>SCSI data transfers. ................................................ aic7880EnableFast20()</td>
<td>232</td>
</tr>
<tr>
<td>File. Adaptec 7880</td>
<td>SCSI Host Adapter Library .............................................. aic7880Lib</td>
<td>5</td>
</tr>
<tr>
<td>library (SCSI-1). NCR 53C710</td>
<td>SCSI I/O Processor (SIOP) ............................................. ncr710Lib</td>
<td>151</td>
</tr>
<tr>
<td>library (SCSI-2). NCR 53C710</td>
<td>SCSI I/O Processor (SIOP) ............................................. ncr710Lib2</td>
<td>151</td>
</tr>
<tr>
<td>library / NCR 53C8xx PCI</td>
<td>SCSI I/O Processor (SIOP) ............................................. ncr810Lib</td>
<td>152</td>
</tr>
<tr>
<td>library. Fujitsu MB87030</td>
<td>SCSI Protocol Controller (SPC) ....................................... mb87030Lib</td>
<td>126</td>
</tr>
<tr>
<td>I/O Processor (SIOP) library (SCSI-1). NCR 53C710</td>
<td>SCSI-1 I/O Processor (SIOP) ............................................. ncr710Lib</td>
<td>151</td>
</tr>
<tr>
<td>Interface Controller library (SCSI-1). NCR 53C90 Advanced</td>
<td>ncr5390Lib1</td>
<td>154</td>
</tr>
<tr>
<td>Interface Controller library (SCSI-2). NCR 53C710</td>
<td>ncr710Lib2</td>
<td>151</td>
</tr>
<tr>
<td>Interface Controller library (SCSI-2). NCR 53C90 Advanced</td>
<td>ncr5390Lib2</td>
<td>154</td>
</tr>
<tr>
<td>driver. Nat.</td>
<td>Semitronics DP8392B SONIC Ethernet ..................................... sn8392End</td>
<td>199</td>
</tr>
<tr>
<td>interface driver. Crystal</td>
<td>Semiconductor CS8900 network .......................................... if_cs</td>
<td>58</td>
</tr>
<tr>
<td>National</td>
<td>Semiconductor DP83902A ST-NIC .......................................... ns83902End</td>
<td>160</td>
</tr>
<tr>
<td>Ethernet network / National</td>
<td>Semiconductor DP8392B SONIC .......................................... if_sn</td>
<td>99</td>
</tr>
<tr>
<td>driver. Digital</td>
<td>Semiconductor SA-1100 UART tty ......................................... sa1100pio</td>
<td>190</td>
</tr>
<tr>
<td>network interface / National</td>
<td>Semiconductor ST-NIC Chip ............................................ nicEvbEnd</td>
<td>157</td>
</tr>
<tr>
<td>network interface / National</td>
<td>Semiconductor ST-NIC Chip ............................................ if_nicEvb</td>
<td>95</td>
</tr>
<tr>
<td>PCI</td>
<td>Shared Interrupt support. ................................................... pciIntLib</td>
<td>186</td>
</tr>
<tr>
<td>network interface driver.</td>
<td>shared memory backplane .................................................. if_sm</td>
<td>98</td>
</tr>
<tr>
<td>show information about VxWorks interface</td>
<td>shared memory network ....................................................... smNetShow()</td>
<td>415</td>
</tr>
<tr>
<td>show routines.</td>
<td>shared memory network driver ................................................ smNetShow</td>
<td>198</td>
</tr>
<tr>
<td>interrupt handler for shared PCI interrupt.</td>
<td>.................................................... pciInt()</td>
<td>395</td>
</tr>
<tr>
<td>PCIC chip.</td>
<td>show all configurations of ............................................... piciShow</td>
<td>391</td>
</tr>
<tr>
<td>PCMCIA chip.</td>
<td>show all configurations of ............................................... pcmiaShow</td>
<td>399</td>
</tr>
<tr>
<td>TCIC chip.</td>
<td>show all configurations of ............................................... ticcShow</td>
<td>430</td>
</tr>
<tr>
<td>about function. show configuration details ............................................... pciConfigFuncShow()</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>word.</td>
<td>show decoded value of command ........................................... piciConfigCmdWordShow()</td>
<td>381</td>
</tr>
<tr>
<td>word.</td>
<td>show decoded value of status ............................................ piciConfigStatusWordShow()</td>
<td>390</td>
</tr>
<tr>
<td>memory network.</td>
<td>information about shared .................................................. smNetShow()</td>
<td>415</td>
</tr>
<tr>
<td>PCMCIA host bus adaptor chip</td>
<td>show library. Intel 82365SL .............................................. piciShow</td>
<td>185</td>
</tr>
<tr>
<td>PCMCIA</td>
<td>show library. ............................................................... pcmiaShow</td>
<td>187</td>
</tr>
<tr>
<td>PCMCIA CIS</td>
<td>show library. ............................................................... cisShow</td>
<td>19</td>
</tr>
<tr>
<td>PCMCIA host bus adaptor chip</td>
<td>show library. Databook TCIC/2 ........................................... ticcShow</td>
<td>207</td>
</tr>
<tr>
<td>and PCMCIA) disk device driver</td>
<td>show LPT statistics. ....................................................... lptShow()</td>
<td>298</td>
</tr>
<tr>
<td>initialize ATA/IDE disk driver</td>
<td>show routine. ATA/IDE (LOCAL) .......................................... ataShow</td>
<td>14</td>
</tr>
<tr>
<td>shared memory network driver drivers. initialize all mapped library.</td>
<td>show routine for MII library. ............................................. miiShow()</td>
<td>327</td>
</tr>
<tr>
<td>show routines for PCMCIA</td>
<td>...................................................... piciShow()</td>
<td>399</td>
</tr>
<tr>
<td>show routines of PCI bus (IO)</td>
<td>...................................................... piciShow()</td>
<td>185</td>
</tr>
</tbody>
</table>
Keyword Index

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver.</td>
<td>Siemens SAB 82532 UART tty</td>
<td>sab82532</td>
</tr>
<tr>
<td>perform</td>
<td>single-step.</td>
<td>ncr710SingleStep()</td>
</tr>
<tr>
<td>enable/disable script</td>
<td>single-step.</td>
<td>ncr710StepEnable()</td>
</tr>
<tr>
<td>structure for NCR 53C710</td>
<td>SIOP: create control</td>
<td>ncr710CtrlCreate()</td>
</tr>
<tr>
<td>structure for NCR 53C710</td>
<td>SIOP: create control</td>
<td>ncr710CtrlCreateScsi2()</td>
</tr>
<tr>
<td>structure for NCR 53C710</td>
<td>SIOP: initialize control</td>
<td>ncr710CtrlInit()</td>
</tr>
<tr>
<td>structure for NCR 53C710</td>
<td>SIOP: initialize control</td>
<td>ncr710CtrlInitScsi2()</td>
</tr>
<tr>
<td>registers for NCR 53C710</td>
<td>SIOP: set hardware-dependent</td>
<td>ncr710SetHwRegister()</td>
</tr>
<tr>
<td>structure for NCR 53C8xx</td>
<td>SIOP: create control</td>
<td>ncr810CtrlCreate()</td>
</tr>
<tr>
<td>structure for NCR 53C8xx</td>
<td>SIOP: initialize control</td>
<td>ncr810CtrlInit()</td>
</tr>
<tr>
<td>registers for NCR 53C8xx</td>
<td>SIOP: set hardware-dependent</td>
<td>ncr810SetHwRegister()</td>
</tr>
<tr>
<td>NCR 53C710 SCSI I/O Processor</td>
<td>(SIOP) library (SCSI-1)</td>
<td>ncr710Lib</td>
</tr>
<tr>
<td>NCR 53C710 SCSI I/O Processor</td>
<td>(SIOP) library (SCSI-2)</td>
<td>ncr710Lib2</td>
</tr>
<tr>
<td>53C8xx PCI SCSI I/O Processor</td>
<td>(SIOP) library (SCSI-2). NCR</td>
<td>ncr810Lib</td>
</tr>
<tr>
<td>of all readable NCR 53C710</td>
<td>SIOP registers. /values</td>
<td>ncr710Show()</td>
</tr>
<tr>
<td>of all readable NCR 53C710</td>
<td>SIOP registers. /values</td>
<td>ncr710ShowScsi2()</td>
</tr>
<tr>
<td>of all readable NCR 53C8xx</td>
<td>SIOP registers. /values</td>
<td>ncr810Show()</td>
</tr>
<tr>
<td>of all readable SYM 53C8xx</td>
<td>SIOP registers. /values</td>
<td>sym895Show()</td>
</tr>
<tr>
<td>set baud rate for</td>
<td>SLIP interface.</td>
<td>slipBaudSet()</td>
</tr>
<tr>
<td>delete</td>
<td>SLIP interface.</td>
<td>slipDelete()</td>
</tr>
<tr>
<td>initialize</td>
<td>SLIP interface.</td>
<td>slipInit()</td>
</tr>
<tr>
<td>driver.</td>
<td>Serial Line IP (SLIP) network interface</td>
<td>if_sl</td>
</tr>
<tr>
<td>agent.</td>
<td>initialize</td>
<td>slipInitPktDevInit()</td>
</tr>
<tr>
<td>initialize</td>
<td>SLIP packet device for WDB</td>
<td>ifppc860DevInit()</td>
</tr>
<tr>
<td>display statistics for SMC 8013WC etc network</td>
<td></td>
<td>eleShow()</td>
</tr>
<tr>
<td>interface driver</td>
<td>SMC 8013WC Ethernet network</td>
<td>if_elec</td>
</tr>
<tr>
<td>network interface driver</td>
<td>SMC Elite Ultra Ethernet</td>
<td>if_ultra</td>
</tr>
<tr>
<td>handle</td>
<td>SMC interrupt.</td>
<td>ifppc860Int()</td>
</tr>
<tr>
<td>Motorola MPC800</td>
<td>SMC UART serial driver.</td>
<td>ppc860Sio</td>
</tr>
<tr>
<td>interface driver</td>
<td>SMC Ultra Elite END network</td>
<td>ultraEnd</td>
</tr>
<tr>
<td>initialize driver and</td>
<td>publischn network interface and</td>
<td>snattach()</td>
</tr>
<tr>
<td>Nat. Semi DP83932B</td>
<td>SONIC Ethernet driver.</td>
<td>sn83932End</td>
</tr>
<tr>
<td>/Semiconductor DP83932B</td>
<td>SONIC Ethernet network driver.</td>
<td>if_sn</td>
</tr>
<tr>
<td>control structure for MB87030</td>
<td>SPC: create</td>
<td>mb87030CtrlCreate()</td>
</tr>
<tr>
<td>control structure for MB87030</td>
<td>SPC: initialize</td>
<td>mb87030CtrlInit()</td>
</tr>
<tr>
<td>values of all readable MB87030</td>
<td>SPC registers. display</td>
<td>mb87030Show()</td>
</tr>
<tr>
<td>PCMCIA</td>
<td>SRAM device driver.</td>
<td>sramDrv</td>
</tr>
<tr>
<td>install PCMCIA</td>
<td>SRAM memory driver.</td>
<td>sramDrv()</td>
</tr>
<tr>
<td>Semiconductor DP83902A</td>
<td>ST-NIC. National</td>
<td>ns83902End</td>
</tr>
<tr>
<td>National Semiconductor</td>
<td>ST-NIC Chip network interface</td>
<td>nicEvBEnd</td>
</tr>
<tr>
<td>National Semiconductor</td>
<td>ST-NIC Chip network interface</td>
<td>if_nicEvB</td>
</tr>
<tr>
<td>parse initialization</td>
<td>string.</td>
<td>auxInitParse()</td>
</tr>
<tr>
<td>parse initialization</td>
<td>string.</td>
<td>e390xInitParse()</td>
</tr>
<tr>
<td>parse init</td>
<td>string.</td>
<td>e153c509Parse()</td>
</tr>
<tr>
<td>parse initialization</td>
<td>string.</td>
<td>ln97xInitParse()</td>
</tr>
<tr>
<td>parse initialization</td>
<td>string.</td>
<td>mb86960InitParse()</td>
</tr>
<tr>
<td>parse init</td>
<td>string.</td>
<td>mbcParse()</td>
</tr>
<tr>
<td>parse initialization</td>
<td>string.</td>
<td>nicEvBInitParse()</td>
</tr>
<tr>
<td>get token</td>
<td>string (modified version).</td>
<td>endTok_r()</td>
</tr>
<tr>
<td>display values of all readable</td>
<td>SYM 53C8xx SIOP registers.</td>
<td>sym895Show()</td>
</tr>
</tbody>
</table>
set Sym895 chip options. ...................... sym895SetHwOptions() 427
create structure for SYM95 device. ................ sym95CtrlCreate() 421
SCSI-2 driver for Symbios SYM95 SCSI Controller. sym95Lib 204
handle task-level PCMCIA events. ...................... pcmcia() 398
version). get token string (modified end Tok_r ) 275
agent. initialize TSFS device driver for WDB wdbTsfsDrv() 441
Motorola MC68302 bimodal tty driver. .................. m68302Sio 119
Motorola MC68332 tty driver. ...................... m68332Sio 120
MC68901 MFP tty driver. ...................... m68901Sio 124
MB 86940 UART tty driver. ...................... mb86940Sio 124
NS 16550 UART tty driver. ...................... ns16550Sio 159
NEC VR4101 DSIU UART tty driver. ................. nvr4101DSIUo 161
NEC VR4101 SIU UART tty driver. ................. nvr4101SIUo 162
NEC VR4102 DSIU UART tty driver. ................. nvr4102DSIUo 162
Siemens SA-1100 UART tty driver. .................. sa1100Sio 190
Siemens SAB 82532 UART tty driver. ................ sab82532 192
ST 16C552 DUART tty driver. ...................... st16552Sio 202
ARM AMBA UART tty driver. ...................... ambaSio 8
initialize tty driver for WDB agent. ............. wdbVioDrv() 442
virtual tty I/O driver for WDB agent. ............ wdbVioDrv 220
Motorola MC68360 SCC UART serial driver. ........ m68360Sio 120
Motorola MPC800 SMC UART serial driver. ........ ppc860Sio 189
MB 86940 UART tty driver. ...................... mb86940Sio 124
NS 16550 UART tty driver. ...................... ns16550Sio 159
NEC VR4101 DSIU UART tty driver. ................. nvr4101DSIUo 161
NEC VR4101 SIU UART tty driver. ................. nvr4101SIUo 162
NEC VR4102 DSIU UART tty driver. ................. nvr4102DSIUo 162
Digital Semiconductor SA-1100 UART tty driver. .. sa1100Sio 190
Siemens SAB 82532 UART tty driver. ................ sab82532 192
ARM AMBA UART tty driver. ...................... ambaSio 8
communication functions for ULIP. initialize .......... wdbUlipPktDevInit() 441
communication interface for ULIP driver. WDB. ........ wdbUlipPktDrv 219
interface driver. SMC Ultra Elite END network. ........ ultraEnd 208
interface driver. SMC Elite Ultra Ethernet network. .... if_ult 102
display statistics for ulta interface and initialize .... ultaInit() 431
handle all interrupts in one vector. .................. coldfireInt() 248
all DUART interrupts in one vector. handle .......... m68681Int() 307
handle all interrupts in one vector. handle ........... z8530Int() 443
gt token string (modified version). ...................... end Tok_r ) 275
driver for WDB agent. virtual generic file I/O ....... wdbTsfsDrv 216
agent. virtual tty I/O driver for WDB ............... wdbVioDrv 220
NEC VR4101 DSIU UART tty driver. ................. nvr4101DSIUo 161
NEC VR4101 SIU UART tty driver. ................. nvr4101SIUo 162
NEC VR4102 DSIU UART tty driver. ................. nvr4102DSIUo 162
display values of all readable /and partially initialize WD3C39 chip registers. ............ wd3C39Show() 438
Controller library (SCSI-1). WD3C39 SCSI-Bus Interface....................... wd3C39Lib 211
Controller library (SCSI-2). WD3C39 SCSI-Bus Interface................. wd3C39Lib 212
Controller library (SCBIC)................. wd3C39SCBICLib 211
Keyword | Name | Page
---|---|---
NETROM packet driver for WDB agent. | wdbNetromPktDrv | 213
serial line pocket-size for WDB agent. | wdbSlipPktDrv | 215
generic file I/O driver for WDB agent. virtual | wdbTsfsDrv | 216
virtual tty I/O driver for WDB agent. | wdbVioDrv | 220
NETROM packet device for WDB agent. initialize | wdbNetromPktDevInit | 439
SLIP packet device for WDB agent. initialize | wdbSlipPktDevInit | 440
TSFS device driver for WDB agent. initialize | wdbTsfsDrv | 441
initialize tty driver for WDB agent. for ULIP driver. | wdbVioDrv | 442
show decoded value of command word. | pciConfigCmdWordShow | 381
show decoded value of status word. | pciConfigStatusWordShow | 390
translate character to output space. write one 16-bit word to PCI configuration space. write one longword to PCI | nvr410SIUCharToTxWord | 359
word. | pciConfigInWord | 384
word from PCI configuration | pciConfigOutWord | 389
write one 16-bit word to PCI | pciConfigOutWord | 389
write one byte to PCI | pciConfigOutByte | 388
write one longword to PCI | pciConfigOutLong | 388
Communications Controller/ Z8530 SCC Serial | z8530Sio | 221