SNAPPER VxWorks PowerPC Library

Programmer's Manual

DataCell Limited

v4.0.1

Disclaimer

While every precaution has been taken in the preparation of this manual, DataCell Ltd assumes no responsibility for errors or omissions. DataCell Ltd reserves the right to change the specification of the product described within this manual and the manual itself at any time without notice and without obligation of DataCell Ltd to notify any person of such revisions or changes.

Copyright Notice

Trademarks

Part Information

Part Number: SNP-MAN-VXW-LIB

Version v4.0.1 September 1999

Printed in the United Kingdom.

Contact Details

Europe & ROW	Web	www.datacell.co.uk
	Sales	info@datacell.co.uk
	Support	techsupport@datacell.co.uk
USA	Web	www.datacell.com
	Sales	info@datacell.com
	Support	techsupport@datacell.com

Head Office:
DataCell Limited.
Falcon Business Park, 40 Ivanhoe Road,
Finchampstead, Berkshire, RG40 4QQ, UK

Tel +44 (0) 1189 324324
Fax +44 (0) 1189 324325

Table of Contents

Introduction

This manual describes the low-level functions of the VxWorks driver library for the Snapper-PMC series of acquisition cards. This library provides a hardware platform independent method of accessing Snapper cards over the PMC bus, but allows the user to add enhanced functionality specific to a particular platform via a set of installable callout functions.

Concepts

The VxWorks specific driver module is shipped as a series of four VxWorks libraries, in GNU - PPC archive format, a set of header files, and sample source modules for user customisation.

LIBTMG.A

This library contains all the functions of the TMG image manipulation and processing library, with the exception of display routines, which are not supported under VxWorks. The TMG library is a pure software module, in that it does not perform any hardware operations. In addition to this, libtmg.a is self contained, not requiring to be linked with any more than the standard ANSI 'C' library. If NFS is configured into the system, then file I/O services are also available.

LIBFPGA.A

This library is a pure data library, and contains no executable code. It contains the programming files for the Snapper and Mapper FPGA devices.

LIBSNAP.A

This library contains all the BASE_xxx , SNP24_xxx, DIG16_xxx and MODULE_xxx functionality. Note that the only Snapper boards currently available in PMC format are the Snapper-24/8 and Snapper-DIG16. This library does not directly make any hardware accesses, which are all performed by libsnapdrv.a. When a program is linked, this library must appear ahead of libsnapdrv.a.

LIBSNAPDRV.A

This library contains all the low-level, VxWorks specific routines. Almost all of its functionality can be replaced at attach time by user installable routines.

FPGA.C

This module is supplied in source code, and contains a table of all available FPGA devices. If compiled and linked in with an application, the application will be able to run with any type and revision of Snapper board manufactured up to the date of the current software release. However, this will cause the entire contents (some 300Kbytes) of data in libfpga.a to be linked in with the application. It is possible to reduce this data overhead by only including in the table those FPGA files which will actually be required by the application. Due to the fact that an average embedded design would only normally use a single Snapper in a single configuration, this would normally represent program size saving in excess of 250 Kbytes. The file fpga.txt in the library installation directory should be consulted for more information on FPGA configuration on the current release.

MVME2604.C

This is the source code module for the initialisation of the Snapper library on the Motorola MVME2604 board.

PCORE603.C

This is the source code module for the initialisation of the Snapper library on the Force Computers PCORE6603 board.

Building an Application

The following is an example makefile.

MAKEFILE

```
# GNU Makefile for Snapper applications
#
# The CPUTYPE must be defined before including any of the system Make rules
# Currently this must be one of:-
#    604 for PowerPC - 604 series processors (604, 604e, 604ev etc.)
#    603 for PowerPC - 603 series processors (603, 603e, 603ev, EC603 etc.)
CPUTYPE         = 604
# Next define the Tornado installation directory
WIND_BASE       = c:/tornado
# and the cross development platform (see the Tornado Programmer's Guide)
WIND_HOST_TYPE  = x86-win32
# Next, the base installation directory of the Snapper libraries is required
SNAPBASE        = c:/snapper
# The following 3 definitions are generated automatically from the
# preceding information
SNAPSRC         = $(SNAPBASE)/src
SNAPLIB         = $(SNAPBASE)/lib/PPC$(CPUTYPE)
CPU             = PPC$(CPUTYPE)
# Finally, specify the development toolset used.
# Only GNU is supported at present.
TOOL            = gnu
#
# Load the default make rules,
#
include $(WIND_BASE)/target/h/make/defs.bsp
include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)
include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)
#
# Add the Snapper include path to the C-Preprocessor list
CC_INCLUDE += -I$(SNAPSRC)/include
#
# And the target include paths, which will be required to compile the
# xxxAttach routines
CC_INCLUDE += -I$(WIND_BASE)/target/config
#
# Add the required Snapper #defines.
CC_DEFINES += -D_VXWORKS -D_FPGA_INTERNAL -D_GNU_TOOL
# And the optional board specific #defines.
# These should be:-
# -D_SNP24 if the Snapper-24 or Snapper-8 is to be used
# -D_DIG16 if the Snapper-DIG16 is to be used.
CC_DEFINES += -D_SNP24 -D_DIG16
#
# Finally define the CPU taget for the compiler
CC_ARCH_SPEC += -mcpu=$(CPUTYPE)
#
SNAPLIBS = -ltmg -lsnap -lfpga -lsnapdrv
#
# Every makefile should contain a target "all"
all: MyApp
#
# Note that the fpga table must be linked in with the application object
#
MyApp: MyApp.o fpga.o
       ldppc -r -o MyApp MyApp.o fpga.o -L$(SNAPLIB) $(SNAPLIBS)
```

REQUIRED VXWORKS MODULES

The Snapper libraries reference the following VxWorks components explicitly, in addition to the libraries which make up the standard ANSI 'C' environment:

cacheLib Cache control library.

clockLib POSIX clock handling library.

errnoLib Multi-tasking error reporting library.

intLib Interrupt library.

semLib General purpose semaphore library.

semCLib Counting semaphore library.

sigLib Signal delivery library.

taskLib Task control library.

timerLib POSIX timer library – only required if the built in timer routines are used (default).

vmLib Virtual memory library. Only if VxVMI component is operational.

wdLib VxWorks watchdog timer library.

Sample Applications

The following are minimal applications to install a Snapper board and acquire a single image.

SNAPPER-24 ON A MOTOROLA COMPUTER MVME2604 BOARD

The following program captures an image from a monochrome camera attached to the channel "red 1" of a Snapper-24, and saves it to an NFS mounted disk in TIFF format:

```
/* Standard Snapper includes */
#include <asl_inc.h>
#include <asl_vx.h>
/* Additional system include to support architecture and board includes */
#include <sysLib.h>
#include <dllLib.h>
/* Architecture specific include files */
#include <arch/ppc/ivppc.h>
#include "arch/ppc/mmu603Lib.h"
#include "arch/ppc/vxPpcLib.h"
#include "private/vmLibP.h"
/* MVME2604 specific includes */
#include "drv/pci/pciIomapLib.h"
#include <mv2604/config.h>
#include <mv2604/mv2600.h>
#include <mv2604/PciLocalBus1.h>

/* The system memory is offset to address PCI2DRAM_BASE_ADRS
 * as seen by a PCI bus master device.
 * This routine is used to override the compiled in default, which
 * assumes that the PCI bus master and processor system memory address
 * maps are the same
 */
int MVME2604AlenListPhysToBus(  struct AlenList *pPhysList,
                                struct AlenList *pBusList,
                                struct tDeviceInfo* pDevice)
{
   int nElements = 0;

   while(pPhysList != NULL){
      pBusList->Address.PhysicalAddress = pPhysList->Address.PhysicalAddress
                                        + PCI2DRAM_BASE_ADRS;
      pBusList->Length = pPhysList->Length; /* memory is contiguous */
      nElements++;      /* increase count of valid elements in pBusList */
      pPhysList = pPhysList->pNext;  /* traverse list */
      if (pPhysList != NULL){        /* Need another element in pBusList */
         pBusList = DRV_AlenList_Next_Element(pBusList);
         if ( pBusList == NULL ){
            return 0;
         }
      }
   }
   return nElements;
}
```

```
/* This routine is called by the application at initialisation time.
 * In a run-time environment, it would normally be called late
 * in the boot sequence.
 * Note that the two parameters are to enable DMA and to enable interrupts
 * for this slot.
 */
int InstallSnapper(ui32 bDmaEnable, ui32 bInterruptEnable)
{
   struct tDeviceInfo *pDevice1;
   struct tLocalDeviceInfo *pLDI1 = NULL;
   int    pciBusNo;
   int    pciDevNo;
   int    pciFuncNo;

   /* First call the BSP defined function to locate the PMC card,
    * using the Snapper ID and vendor
    */
   if (pciFindDevice (PCI_ASL_VENDOR_ID,
                      PCI_BIB_ID40, 0,
                      &pciBusNo, &pciDevNo, &pciFuncNo) != ERROR){
      /* If the board is present, set up its configuration space with
       * the value obtained from the BSP header files.
       */
      pciDevConfig (pciBusNo, pciDevNo, pciFuncNo,
                    PCI_IO_PMC_ADRS,
                    NULL,
                    (PCI_CMD_MASTER_ENABLE | PCI_CMD_IO_ENABLE));

      /* Allocate a device information structure, and initialise it to
       * the default state, which is all zeroed
       */
      pDevice1 = (struct tDeviceInfo*)MALLOC(sizeof(struct tDeviceInfo));
      bzero((char*)pDevice1,sizeof(struct tDeviceInfo));

      /* Fill in addressing information, as specified in the BSP routines */
      pDevice1->dwBusNum = (ui32)pciBusNo;
      pDevice1->dwDeviceNumber = (ui32)pciDevNo;
      pDevice1->dwFunctionNumber = (ui32)pciFuncNo;
      pDevice1->PhysicalAddress = PCI_IO_PMC_ADRS;
      pDevice1->VirtualAddress = (void*)(CPU_PCI_IO_ADRS + PMC_DEV_SPACE);
      pDevice1->dwBusType = DRV_BUS_TYPE_PCI;

      /* Fill in interrupt information.
       * This is only used if bInterruptEnable is true
       */
      pDevice1->nIRQLevel = PMC_INT_LVL1;  /* from the BSP headers */
      pDevice1->nIRQVector = (ui32)INUM_TO_IVEC(pDevice1->nIRQLevel);
      /* Install override function for address translation */
       * and leave all the other function pointers NULL (i.e. use defaults)
       */
      pDevice1->pFnAlenListPhysToBus = MVME2604AlenListPhysToBus;

      /* Fill in options for this card */
      pDevice1->bDmaEnable = bDmaEnable;
      pDevice1->bInterruptEnable = bDmaEnable;

      /* And install the device */
      pLDI1 = SnapperAttach(pDevice1);
   }
   if (pLDI) return 0; /* Success */
   else return -1; /* failure */
}
```

```
int SnapMonoRed(int nBoard)
{
   Thandle hSnapper;
   Thandle hBase;
   Thandle hImage;
   ui32 Height;
   Terr Err;

   ASL_err_set_reporting(ASL_ERR_SET_HANDLER, ASL_err_display);

   hBase = BASE_create((ui32)nBoard);
   hSnapper = BASE_get_parameter(hBase, BASE_MODULE_HANDLE);
   hImage = TMG_image_create();

   SNP24_initialize(hSnapper,SNP24_CCIR_DEFAULT);
   SNP24_set_format(hSnapper,SNP24_FORMAT_Y8_ON_RED,TMG_Y8);
   SNP24_set_sync(hSnapper, SNP24_SYNC_OFF_RED1);
   SNP24_set_capture(hSnapper, SNP24_SUB_X1);
   SNP24_set_image(hSnapper, hImage);
   Height = TMG_image_get_parameter(hImage,TMG_HEIGHT);
   TMG_image_set_parameter(hImage, TMG_LINES_THIS_STRIP, Height);
   taskDelay( CLOCKS_PER_SEC );
   if ( SNP24_is_locked(hSnapper) != TRUE ){
      /* Perform error recovery */
   }
   SNP24_capture(hSnapper, SNP24_START_AND_WAIT);
   SNP24_read_video_data(hSnapper, hImage, TMG_RUN);
   TMG_image_set_outfilename(hImage,"/sun/pattern.tif");
   TMG_image_write(hImage,TMG_NULL, TMG_TIFF,TMG_RUN);
   BASE_destroy(hBase);
   TMG_image_destroy(hImage);
   return 0;
}
```

Function and Structure List

DEVICE STRUCTURES

ALENLIST MANIPULATION

DEFAULT ADDRESS MAPPING FUNCTIONS

INTERRUPT HANDLER

INSTALLATION FUNCTION

The functions and structures are described in alphabetical order in the following pages.

AlenList Structure

CONCEPT

Alenlists are address/length lists. Each element describes a range of contiguous addresses. Each list is a set of contiguous ranges that concatenated form a single memory allocation.

DEFINITION

```
typedef struct AlenList{
    struct AlenList *pNext;
    union {
        void  *VirtualAddress;
        ui32   PhysicalAddress;
    } Address;
    size_t Length;
    ui32   Flags;
} AlenList;
```

MEMBERS

pNext	Pointer to the next element in the list.
VirtualAddress	The mapped address of the beginning of the range represented by this element, as seen from the calling processor.
PhysicalAddress	The physical address of the beginning of the range represented by this element. Note that the physical address for a given range may be different, depending on the viewpoint (e.g. Processor or PCI Bus-master).
Length	The length in bytes of the range represented by this element.
Flags	Flags describing this address range. Currently unused, and must be set to 0 for future compatibility.

SEE ALSO

DRV_AlenList_Next_Element, *DRV_AlenList_Count*, *DRV_AlenList_Destroy*.

DeviceInfo Structure

CONCEPT

This structure must be filled out correctly and passed to the *SnapperAttach* routine. It is fairly complex, and includes information about the location and mapping of the Snapper board, as well as a set of overrides for the default built-in driver functions.

Note that if required, the driver functions can be replaced on a per board basis, thus allowing for different access functions on boards on different PCI/PMC buses.

DEFINITION

```
struct tDeviceInfo {
   ui32 dwBusNum;
   ui32 dwDeviceNumber;
   ui32 dwFunctionNumber;
   enum{
      DRV_BUS_TYPE_UNDEFINED,
      DRV_BUS_TYPE_ISA,
      DRV_BUS_TYPE_PCI
   } dwBusType;
   ui32 bDmaEnable;
   ui32 bInterruptEnable;
   ui32 BusAddress;
   ui32 PhysicalAddress;
   void *VirtualAddress;
   ui32 nIRQLevel;
   ui32 nIRQVector;
   void (*pFnWritePortUlong)( struct tDeviceInfo*, void*, ui32);
   void (*pFnWritePortUshort)( struct tDeviceInfo*, void*, ui16);
   void (*pFnWritePortUbyte)( struct tDeviceInfo*, void*, ui8);
   ui32 (*pFnReadPortUlong)( struct tDeviceInfo*, void*);
   ui16 (*pFnReadPortUshort)( struct tDeviceInfo*, void*);
   ui8  (*pFnReadPortUbyte)( struct tDeviceInfo*, void*);
   void (*pFnWriteDataUlong)( struct tDeviceInfo*, void*, ui32);
   void (*pFnWriteDataUshort)( struct tDeviceInfo*, void*, ui16);
   ui32 (*pFnReadDataUlong)( struct tDeviceInfo*, void*);
   ui16 (*pFnReadDataUshort)( struct tDeviceInfo*, void*);
   int  (*pFnAlenListVirtToPhys)(AlenList *, AlenList *);
   int  (*pFnAlenListPhysToBus)(AlenList *, AlenList *, struct tDeviceInfo *);
   int  (*pFnAlenListCacheFlush)(AlenList *);
   int  (*pFnAlenListCacheInvalidate)(AlenList *);
   DRV_BOOLEAN (*pFnCreateTimer)(struct tVxDriverTimer *);
   DRV_BOOLEAN (*pFnDeleteTimer)(struct tVxDriverTimer *);
   DRV_BOOLEAN (*pFnCancelTimer)(struct tVxDriverTimer *);
   DRV_BOOLEAN (*pFnStartTimer)(struct tVxDriverTimer *);
   void (*pFnPreInterruptHook)(struct tDeviceInfo *);
   void (*pFnPostInterruptHook)(struct tDeviceInfo *);
   void (*ToggleDebugIndicator)(ui8);
};
```

MEMBERS

MAPPING INFORMATION

dwBusNum — Physical bus number of PCI/PMC bus to which Snapper is attached.
Returned by bus scan routines.
Used by address mapping, cache, interrupt and configuration access routines.

dwDeviceNumber — PCI device number of Snapper.
Returned by bus scan routines.
Used by configuration access routines.

dwFunctionNumber — PCI Function number of the Snapper card.
Returned by bus scan routines.
Used by configuration access routines.

dwBusType — Bus interface type of the Snapper.
Must be *DRV_BUS_TYPE_PCI.*

BusAddress — The address of the Snapper card on the PCI-local bus. (Note the Snapper card always appears in I/O space).
This is generally allocated as boot time by the system firmware, and is read back by the driver to determine the processor physical address of the Snapper hardware.

PhysicalAddress — The physical (i.e. hardware unmapped) address of the Snapper card as seen from the processor calling *SnapperAttach*.
On the PowerPC processor, all I/O space is mapped into main memory space. The physical address of the Snapper hardware is the combination of the BusAddress and the base address of the PCI I/O space. This base address can only be determined by reference to the hardware manual for the particular processor card.

VirtualAddress — The virtual (i.e. MMU mapped) address of the Snapper card as seen from the processor calling *SnapperAttach*. The processor physical address is mapped by the kernel to a virtual address which can used by the driver to access the Snapper hardware.

INTERRUPT INFORMATION

nIRQLevel — The interrupt level at which the Snapper will interrupt the calling processor.

Note that the Snapper only asserts PCI INTA. The actual interrupt line to which this interrupt line is attached is specific to the processor board design, and also a function of the bus number to which the Snapper is attached.

nIRQVector — The interrupt vector corresponding to the interrupt level of the Snapper.

This is usually derived from an operating system/processor specific macro which is a function of *nIRQLevel*.

REGISTER ACCESS FUNCTIONS

pFnWritePortUlong — Writes a 32 bit value to the specified register on a given Snapper.

pFnWritePortUshort — Writes a 16 bit value to the specified register on a given Snapper.

pFnWritePortUbyte — Writes a 8 bit value to the specified register on a given Snapper.

pFnReadPortUlong — Reads a 32 bit value from the specified register on a given Snapper.

pFnReadPortUshort — Reads a 16 bit value from the specified register on a given Snapper.

pFnReadPortUbyte — Reads a 8 bit value from the specified register on a given Snapper.

Notes:

1. The PCI/PMC bus is inherently little endian. Thus for big endian processors, all 16/32 bit transfers require both byte and word swapping.
2. Most high performance processors require some form of cache and pipeline flush operation to be performed in order to guarantee that physical write operations have occurred. These operations must also be performed by these routines.
3. It is envisaged that these routines should only require replacing if there is a problem with intermediate hardware caches in a system, such as a bridge chip. In normal operation, the defaults should always be used.

DATA ACCESS FUNCTIONS

pFnWriteDataUlong	Writes a 32 bit value to the data port on a given Snapper.
pFnWriteDataUshort	Writes a 16 bit value to the data port on a given Snapper.
pFnReadDataUlong	Reads a 32 bit value from the data port on a given Snapper.
pFnReadDataUshort	Reads a 16 bit value from the data port on a given Snapper.

Notes:

1. The video data is a byte stream, and therefore has no inherent endian property. These routines are essentially the same as the Port routines above, but with no byte swapping.
2. Most high performance processors require some form of cache and pipeline flush operation to be performed in order to guarantee that physical write operations have occurred. These operations must also be performed by these routines.
3. It is envisaged that these routines should only required replacing if there is a problem with intermediate hardware caches in a system, such as a bridge chip. In normal operation, the defaults should always be used.
4. These routines are not used if *bDmaEnable* is non-zero.

DEVICE FLAGS

bDmaEnable	Non-zero to indicate that this Snapper should use DMA transfers for image data.
bInterruptEnable	Non zero to indicate that this Snapper should use Interrupts for end of DMA and serial communications.

Note: For Snapper-DIG16 operation, both *bDmaEnable* and *bInterruptEnable* must be set.

ADDRESS TRANSLATION FUNCTIONS

pFnAlenListVirtToPhys	Converts a set of processor virtual memory addresses to physical system memory addresses. The virtual addresses need only be valid on the calling processor. The output *AlenList* may contain more elements than the input length if the mapped physical memory is not contiguous. The default function derives the mapping from the *sysPhysMemDesc* structure in the BSP, and does not support VxVMI or VxMP.
pFnAlenListPhysToBus	Converts the system memory physical addresses to PCI bus master physical addresses, suitable for programming into a DMA engine. The actual mapping required is a function of the processor/PCI bridge chip setup and any intervening PCI/PCI bridge chips. The default function assumes a 1:1 mapping of system memory addresses to PCI bus master addresses. This is correct for a system using an MP106

bridge chip configured to memory map 'B' (CHRP).

This is the function which most commonly requires to be replaced.

CACHE CONTROL FUNCTIONS

pFnAlenListCacheInvalidate Invalidates the cache entries for the address range specified in the *AlenList*.

Called as part of the DMA read completion.

pFnAlenListCacheFlush Flushes the cache entries for the address range specified in the *AlenList*.

Called as part of DMA write setup.

Not used for the Snapper implementation, which only performs DMA read operations.

TIMER FUNCTIONS

pFnCreateTimer Called as part of *SnapperAttach* to create a timer.

The only valid field of the *struct tVxDriverTimer* (see *VxDriverTimer Structure*) is the *pInfo* field, which is a pointer to the *struct tLocalDeviceInfo* (see *DeviceInfo Structure*)structure for this device.

pFnDeleteTimer Deallocates the requested timer.

The timer may be still running on entry.

Not currently called, as driver unload is not supported.

pFnCancelTimer Cancels the requested timeout, preventing a call to the callback routine. Does nothing and returns *DRV_TRUE* if the timer is not running

pFnStartTimer Arms the requested timer.

This is the only function where *pFnTimeout* is guaranteed valid

INTERRUPT HOOK FUNCTIONS

pFnPreInterruptHook Called on entry to the driver interrupt service routine, before any I/O is performed.

This routine should be called if the hardware requires some specific operation to be performed before an interrupt can be serviced.

The default routine does nothing.

pFnPostInterruptHook Called on exit of the driver interrupt service routines, after all I/O operations have been completed.

This routine should be called if the hardware requires any specific operation to be performed before an interrupt service routine returns control to the kernel.

The default routine does nothing.

DRV_AlenList_New_Element

USAGE

struct AlenList DRV_AlenList_New_Element(struct AlenList *pElement)*

ARGUMENTS

pElement Pointer to an *AlenList* element somewhere in an *AlenList.*

DESCRIPTION

This function allocates a new element and appends it to the element referenced by *pElement*. If *pElement* does not reference the last element in the *AlenList*, then the list is traversed until the terminating element is reached.

If *pElement* is *NULL*, then the function simply allocates and initialises an *AlenList* element.

The *pNext* value of the returned structure is set to *NULL*, to indicate that this is the terminating element.

The *Address* union and *flags* are not initialised.

RETURNS

The function returns a pointer to the newly allocated element, or *NULL* if the memory allocation failed.

BUGS / NOTES

If *pElement* is not *NULL*, then the *AlenList* must be valid, as no internal sanity checking is performed during the *AlenList* traversal.

SEE ALSO

AlenList Structure, *DRV_AlenList_Next_Element*, *DRV_AlenList_Count*, *DRV_AlenList_Destroy*.

DRV_AlenList_Next_Element

USAGE

struct AlenList DRV_AlenList_Next_Element(struct AlenList *pElement)*

ARGUMENTS

pElement Pointer to an *AlenList* element somewhere in an *AlenList.*

DESCRIPTION

This function traverses to the next element in the *Alenist* referenced by *pElement*, returning a pointer to the element. If *pElement* is the last element in the *AlenList*, a new element is allocated and appended.

If a new element has to be allocated, the *pNext* value of the returned structure is set to *NULL*, to indicate that this is the terminating element.

RETURNS

The function returns a pointer to the next element in the *AlenList*, or *NULL* on failure.

BUGS / NOTES

pElement must be a valid *struct AlenList**. On PowerPC platforms, be particularly careful as accesses through a *NULL* pointer are allowed if VxVMI is not installed.

SEE ALSO

AlenList Structure, *DRV_AlenList_New_Element*, *DRV_AlenList_Count*, *DRV_AlenList_Destroy*.

DRV_AlenList_Count

USAGE

*ui32 DRV_AlenList_Count(struct AlenList *pElement)*

ARGUMENTS

pElement Pointer to an *AlenList* element somewhere in an *AlenList.*

DESCRIPTION

This function counts the number of elements in an *AlenList* after and including the given element.

RETURNS

The number of elements in the *AlenList*.

BUGS / NOTES

pElement must be a valid *struct AlenList**. On PowerPC platforms, be particularly careful as accesses through a *NULL* pointer are allowed if VxVMI is not installed.

SEE ALSO

AlenList Structure, *DRV_AlenList_New_Element*, *DRV_AlenList_Next_Element*, *DRV_AlenList_Destroy*.

DRV_AlenList_Destroy

USAGE

*void DRV_AlenList_Destroy(struct AlenList *pElement)*

ARGUMENTS

pElement Pointer to an *AlenList* element somewhere in an *AlenList.*

DESCRIPTION

This function deallocates all the elements in an *AlenList* after and including the given element.

RETURNS

The function has no return value.

BUGS / NOTES

pElement must be a valid *struct AlenList**. On PowerPC platforms, be particularly careful as accesses through a *NULL* pointer are allowed if VxVMI is not installed.

SEE ALSO

AlenList Structure, *DRV_AlenList_New_Element*, *DRV_AlenList_Next_Element*, *DRV_AlenList_Count*.

DRV_Interrupt_Handler

USAGE

*void DRV_Interrupt_Handler(struct tLocalDeviceInfo *pLDI)*

ARGUMENTS

pLDI	Pointer to a local device info structure returned by *SnapperAttach*.

DESCRIPTION

This function handles a Snapper interrupt or interrupts. It should never be called directly, but must be installed if Snapper interrupts are enabled. By default, it will be installed automatically by *SnapperAttach* if *pDeviceInfo->bInterruptEnable* is non-zero.

RETURNS

The function has no return value.

EXAMPLE

Special BSP calls required before initialising an interrupt, therefore *intConnect* must be deferred until after *SnapperAttach* has completed.

```
/* pDevice already initialised */
pDevice->bUseDma = 1;        /* Arm DMA engine */
pDevice->bUseInterrupt  = 0; /* Don't attach interrupts (yet !)*/
pLDI  = SnapperAttach(pDevice); /* Initialise Snapper device entry */
if ( pLDI != NULL) /* SnapperAttach succeeded */
{
   /* perform special processing */
   sysRegisterPciInterrupt(pDevice->pIRQLevel);
   /* and then perform normal service routine installation */
   if ( intConnect(  (VOIDFUNCPTR)pDevice->pIRQVector,
                                  DRV_Interrupt_Handler,
                              (int)pLDI
      ) == OK ) /* intConnect succeeded */
   {
      intEnable(pLDI->nIRQLevel); /* Arm corresponding PowerPC interrupt */
      pDevice->bUseInterrupt = 1; /* and inform the driver */
   }
   else
   {
      /* Handle failure of interrupt registration */
  }
}
else
{
   /* Handle failure of SnapperAttach */
}
```

BUGS / NOTES

None.

SEE ALSO

DeviceInfo Structure, *SnapperAttach*.

DefAlenListVirtToPhys

USAGE

*int DefAlenlistVirtToPhys(struct AlenList *pVirtual, struct AlenList *pPhysical)*

ARGUMENTS

pVirtual	Pointer to an *AlenList* of processor virtual addresses.
pPhysical	Pointer to an *AlenList* of system memory addresses.

DESCRIPTION

This function converts a list of virtual processor addresses, as viewed from the calling processor, to a list of processor physical memory addresses. Both **pVirtual* and **pPhysical* must be zero terminated *AlenLists*. If required, the list referenced by **pPhysical* will be extended, but will never be truncated/deallocated.

RETURNS

The function returns the number of valid entries in **pPhysical*, or 0 if the addresses could not be translated.

BUGS / NOTES

The destination *AlenList* is never truncated. It is the responsibility of the caller application to eventually deallocate the *AlenList* using *DRV_AlenList_Destroy*.

The routine assumes that the virtual memory space is described by the global *sysPhysMemDesc* structure in the BSP. If the memory is mapped by some other means, such as directly via the BAT registers, or under VxVMI, then this function will fail and return 0.

SEE ALSO

DefAlenListPhysToBus.

DefAlenListPhysToBus

USAGE

*int DefAlenlistPhysToBus(struct AlenList *pPhysical, struct AlenList *pBus, struct tDeviceInfo *pDevice)*

ARGUMENTS

pPhysical	Pointer to an *AlenList* of system memory addresses.
pBus	Pointer to an *AlenList* of PCI bus master addresses.
pDevice	Pointer to a *DeviceInfo* structure describing where the DMA engine is located.

DESCRIPTION

This function converts the system physical addresses in **pPhysical* into DMA bus master addresses in **pBus*. These addresses are suitable for programming into a PCI DMA controller located on the bus referenced by *pDevice->dwBusNum*.

RETURNS

The function returns the number of valid entries in **pBus*, or 0 if the addresses could not be translated.

BUGS / NOTES

The destination *AlenList* is never truncated. It is the responsibility of the caller application to eventually deallocate the *AlenList* using *DRV_AlenList_Destroy*.

This routine assumes that the primary PCI bridge controller is an MPC106 running address map 'B'. For other hardware implementations, this default function will have to be replaced.

SEE ALSO

AlenList Structure, *DeviceInfo Structure*, *DefAlenListVirtToPhys*.

SnapperAttach

USAGE

*struct tLocalDeviceInfo SnapperAttach(struct tDeviceInfo *pDevice)*

ARGUMENTS

pDevice Pointer to a *DeviceInfo* structure describing where the DMA engine is located.

DESCRIPTION

This function creates a device instance for the Snapper card described in *pDevice* and allocates a *tLocalDevicInfo* structure.

If the *bUseInterrupts* flag is non-zero, the routine also connects an interrupt service routine and enables processor interrupts on the appropriate level.

The *tDeviceInfo* structure must be fully filled in prior to use, with any non overidden function pointers being set to *NULL*.

RETURNS

The function returns a valid *struct tLocalDeviceInfo* pointer (see *DeviceInfo Structure*) which may be analysed or modified by the caller.

If the attach fails, a *NULL* pointer is returned.

BUGS / NOTES

The function keeps the passed pointer in the *struct DeviceInfo*. It is therefore vital that the structure referenced by *pDevice* is never deallocated by the application. Do not make *struct tDeviceInfo* an automatic stack variable!

The returned value is also kept in a link list inside the driver, and is not a required parameter of any of the driver functions.

SEE ALSO

DeviceInfo Structure, DRV_Interrupt_Handler - for an example of how the returned *struct tLocalDeviceInfo* pointer may be used.

VxDriverTimer Structure

CONCEPT

The structure is used to control a generic timer function.

The built in timer functionality uses POSIX timers, which are linked into the SIGALRM signal handler. The default creation routine installs a *SIGALRM* handler for exclusive use of the Snapper driver. If this signal is required outside of the Snapper driver, the default timer functions must be overridden.

Note that the POSIX timer library is not linked in by default when generating a VxWorks boot image. The symbol *INCLUDE_POSIX_TIMERS* must be entered in the configured options in the BSP customisation manual. See the Tornado 1.0.1 Programmer's Guide for further information.

DEFINITION

```
struct tVxDriverTimer{
   timer_t Timer_id;
   ui32    Timeout;
   void  (*pFnTimeout)(void*);
   void   *pInfo;
};
```

MEMBERS

Timer_id	This is used by the built in routines to save the ID of the POSIX timer created. This field is not used if the built in routines are overridden.
Timeout	The requested timeout period in milliseconds
pFnTimeout	The function to be called when the timer expires. This value is only valid in a call to *pFnStartTimer*. The function must be called with the value specified in the *pInfo* field when *pFnCreateTimer* was called.
pInfo	At creation time, this value is set to the argument that must be passed to the *pFnTimeout* when the timer has expired. It is never accessed by the Snapper drivers after a successful creation call, and may be used at will by the application programmer.

Porting Guide

GENERAL

Two example "C" files are provided on the distribution, which demonstrate how to initialise the Snapper VxWorks libraries. For each board upon which the Snapper PMC card is to be used, a custom version of this file is required. In essence, this initialisation file is an extension of the VxWorks BSP.

PCI BUS PROBING

The first stage of the install routine is to determine the location of the Snapper card(s) on the PCI bus. This is done by searching all the possible PCI slots, looking for a matching vendor and device ID. For the Snapper PMC card, the ID's are defined in the header files as *PCI_ASL_VENDOR_ID* and *PCI_BIB_ID40* respectively. There is no standard VxWorks function to do this, although most board vendors have a suitable helper function (e.g. *pciFindDevice* in the MVME2604 BSP).

Each board in a PCI system is referenced by its bus number, device number and function number. Note that the bus and device numbers are determined by the underlying hardware. The Snapper PMC card has a single function number.

MEMORY SPACES

There are conceptually three memory spaces used within the hardware, which are defined below. (The names of the memory spaces are not generic terms, but are used to aid understanding of their purpose).

Virtual Memory Space

When memory is allocated by an application program, ie to store an acquired image, the address used to reference the buffer is the virtual address. The virtual address is the address used by the processor core.

Processor Physical Memory Space

The processor physical address is the result of passing the virtual address through the memory management hardware. This is the address that is generated on the physical processor pins, (as the memory management hardware is generally internal to the processor chipset).

Bus Physical Memory Space

The bus physical memory address is that generated by the PCI bus master, ie Snapper hardware, in order to access the same physical address as the virtual or processor physical memory. The PCI bus and processor often have different hardware paths to the actual memory devices.

PCI BUS ADDRESS

The Snapper board has a single address space, located in PCI I/O space. In most VxWorks systems, the PCI bus is not configured by the system library. It is thus necessary to find a suitable space in the PCI I/O space in which to locate the Snapper board. This value must be written to the address0 offset in the Snapper card's PCI configuration space. Once an address has been written to the card, it must be enabled by setting the I/O enable and bus master bits in the command register in configuration space. Again there is no standard VxWorks function to do this, but most BSP's have a single function to perform both operations (e.g. *pciDevConfig* in the MVME2604 BSP).

PROCESSOR PHYSICAL ADDRESS

This is the processor physical address which is used to access the Snapper hardware, and is usually the PCI Bus Address plus a number of offsets, which can only be determined from the hardware.

PROCESSOR VIRTUAL ADDRESS

The processor physical address must then be mapped into the processor virtual address space, if not already covered either by the *sysPhysMemDesc* structure, or the BAT registers.

MAIN MEMORY PHYSICAL MAPPING

In order to perform DMA transfers, the Snapper driver must be able to convert processor physical memory addresses to PCI bus master addresses. In most systems, the two memory spaces are contiguous, but often offset by some value, which must be determined either from the header files, or from the manufacturer's hardware manual. If the correspondence is not 1:1, then the function *pFnAlenListPhysToBus* must be overridden. An example is provided for the MVME2604.

INTERRUPTS

The Snapper PMC card generates a single interrupt on the PCI INTA line. It is necessary to calculate the corresponding interrupt level and vector, which is a function of bus number and device number. This must be determined from the header files or the manufacturer's hardware manual.

DEVICE STRUCTURE

Before calling the *SnapperAttach* function, the *tLocalDeviceInfo* structure (see *DeviceInfo Structure*)must be allocated and filled in for each card. A reference to the structure is kept by the Snapper driver, and thus the storage used by the structure must persist until the driver is unloaded. All unused fields must be set to 0. As a very minimum, the following fields must be filled in:-

- *dwBusNum*
- *dwDeviceNumber*
- *dwFunctionNumber*
- *PhysicalAddress*
- *VirtualAddress*
- *nIRQLevel*
- *nIRQVector*
- *dwBusType* (must be *DRV_BUS_TYPE_PCI*)
- *bDmaEnabled* (must be non-zero for the Snapper DIG16 card)
- *bInterruptEnabled* (must be non-zero for the Snapper DIG16 card)

If all the other fields are left zero, then the following are assumed:

- Virtual to physical mapping can be determined from *sysPhysMemDesc*.
- Physical to bus mapping is 1 : 1 (no translation required).
- Image memory is not cached.
- *SIGALRM* is for the exclusive use of the Snapper driver.
- No extra pre/post interrupt processing is required.