946 Solaris,
965 IRIX 6.5,
983 Windows NT/2000,
993 VxWorks
&
1003 Linux
Support Software

Manual

Disclaimer

Please read and abide by the following paragraphs. Questions and comments should be directed to:

Technical Publications Department
SBS Technologies, Inc.
1284 Corporate Center Drive
St. Paul, MN 55121-1245
651-905-4700

SBS makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SBS assumes no responsibility for any errors that may appear in this document. The information in this document is subject to change without notice.

SBS does not authorize the use of its components in life support applications where failure or malfunction of the component may result in injury or death. In accordance with SBS's terms and conditions of sale, the user of SBS components in any and all life support applications assumes all risks arising out of such use and further agrees to indemnify and hold SBS harmless against any and all claims of whatsoever kind or nature (including claims of culpable conduct [strict liability, negligence or breach of warranty] on the part of SBS) for all costs of defending any such claims.

SBS does not authorize the use of its components in control and process applications where failure or malfunction of the component may result in radioactive releases, explosions, environmental damage/contamination, personal injury or death. In accordance with SBS's terms and conditions of sale, the user of SBS components in any and all control and process applications assumes all risks arising out of such use and further agrees to indemnify and hold SBS harmless against any and all claims of whatsoever kind or nature (including claims of culpable conduct [strict liability, negligence or breach of warranty] on the part of SBS) for all costs of defending any such claims.

U.S. GOVERNMENT RESTRICTED RIGHTS

The Support Software and documentation are provided with restricted rights. Use, duplication or disclosure by the Government is subject to the restrictions as set forth in subdivision © (1) (ii) of the Rights in Technical Data and Computer Software Clause of DFAR 252.227-7013 (October 1988) and in similar clauses in the FAR and NASA FAR Supplement. Manufacturer is SBS Technologies, Inc., 1284 Corporate Center Drive, St. Paul, MN 55121-1245.

.

Revision 1.0 20020819

Pub. No. 85221990

Preface

This manual describes SBS Support Software for SBS dataBLIZZARD[tm] and SBS adapters for PCI computers, and provides instructions for software installation, set up and use. In this manual, "adapter" applies to both dataBLIZZARD and other SBS adapters. All mentions of "Support Software" or "the software" refer to Models 965, 946, 1003, 993 and 983. If a reference is to one model only, that model will be indicated.

For information about jumper settings, the physical installation adapter cards and descriptions of registers available on each card refer to the hardware manual included with your SBS adapter.

To simplify installation and eliminate operation problems, we recommend that you review this manual and the appropriate hardware manual before beginning to install the Support Software.

About this manual:

- Chapter 1 introduces you to Support Software, its components, and lists system and hardware requirements.
- Chapter 2 gets you started with information about the software packages, important notes, and a listing of additional references.
- Chapter 3 describes Support Software example applications.
- Chapter 4 discusses the SBS Mirror Application Program Interface (API).
- Chapter 5 is an API reference that details each function.
- Chapter 6 contains installation instructions and porting information that is specific to Model 965.
- Chapter 7 describes Model 946 installation and porting.
- Chapter 8 has Model 1003 installation instructions and porting information.
- Chapter 9 contains installation and porting information for Model 993.
- Chapter 10 describes Model 983 installation, porting, and 983 specific example applications.
- Chapter 11 contains information about general software issues including porting and optimization.
- Appendices—There are five appendices for quick reference, including: a glossary of the terms and symbols used throughout this manual, summaries of ioctl() commands and kernel functions, and information about DMA operation.

Standard C notation is used in this manual to denote hexadecimal and octal numbers. Hexadecimal numbers are preceded by 0x and octal numbers by 0.

Table Of Contents

Chapter 1: Software Support Functions

1.0 Overview

SBS Support Software provides a device driver and example programs to help applications programmers with adapter and system configuration. Support Software drivers are available for all dataBLIZZARD models and adapter models 614, 616, 617, 618, 620, 628, 630, 7X2, 7X3, and RPQ600890 for PCI, CompactPCI and PMC computers running IRIX 6.5 (Model 965), Solaris (Model 946 with Nexus Extensions), Linux (Model 1003), VxWorks (Model 993), and Windows NT/2000 (Model 983).

The software package provides a device driver that allows access to dual-port and/or remote memory space from an application. This allows memory sharing between a PCI computer and another system.

SBS's device drivers provide the support routines required to access all dataBLIZZARD and adapter resources. Remote memory and Dual Port RAM, if configured, can be shared between the two systems. Programmed interrupts can be exchanged. Devices on the remote system can be controlled from the PCI computer, and remote memory can be accessed.

Support Software's memory mapping feature allows direct mapping to dual-port and/or remote memory, without software overhead. After setup, all access details are handled by hardware.

Example programs included in the Support Software demonstrate features of the adapter hardware and software, and are useful tools for:

- Debugging.
- Uploading and downloading binary data.
- Receiving and counting programmed interrupts.
- Testing hardware.

Subroutines and example programs may be modified for your specific hardware configuration.

➔ In the remainder of this manual, "adapter" refers to both dataBLIZZARD and other SBS adapters. If text refers to only one type of hardware, that hardware will be indicated by "adapters only" or "dataBLIZZARD only".

➔ "Support Software" or "the software" refers to all five software models. If a reference is to one model only, that model will be indicated.

Chapter 2: Getting Started

2.0 Getting The Software

The software is shipped on CD-ROM (part number 85702000) with each dataBLIZZARD and adapter. The same software can also be downloaded from the SBS web site (www.sbs.com) under *Current Software Revisions* of the *Software* page.

2.1 Additional References

- *IEEE Standard for a Versatile Backplane Bus: VMEbus*, Institute of Electrical and Electronics Engineers (IEEE), Inc.
- *The PCI Local Bus Specification* is available from the PCI Special Interest Group, JF2-51, 5200 NE Elam Young Parkway, Hillsboro, OR 97124-6497.
- *PCI BIOS Specification* is available from the PCI Special Interest Group, JF2-51, 5200 NE Elam Young Parkway, Hillsboro, OR 97124-6497.
- *PCI System Architecture*; Tom Shanley and Don Anderson; MindShare, Inc.; Addison-Wesley Publishing Company.
- *Microsoft Development Library CD* is available from Microsoft Corp., One Microsoft Way, Redmond, WA 98053-6399.
- *Microsoft Windows NT Device Driver Kit* is available from Microsoft Corp., One Microsoft Way, Redmond, WA 98053-6399.
- *Microsoft Visual C++*™ is available from Microsoft Corp., One Microsoft Way, Redmond, WA 98053-6399.
- *Linux Kernel Internals,* second edition.
- PCI specifications: PCI SIG (http://www.pcisig.com/specs.html).
- *Writing Device Drivers*, Sun Microsystems, Inc., 2550 Gracia Avenue, Santa Clara, CA 94043.
- *Solaris 2.X On-Line Manual Pages*, Section 2 (Application Interface); Section 9 (Device Driver Interface).
- *Solaris 2.X AnswerBook CD*, SunSoft.

2.2 Help!

Please have the following items and information handy when calling SBS for technical support:

- This manual.
- The SBS hardware manual.
- Software model and version number. If the driver is already installed on your system, you can get this information with the following command:

  # what /var/sysgen/boot/btpdd.o
- Size of Dual Port RAM, if any.
- Remote system configuration (installed devices and their configuration).

Technical support is available from 8:00 a.m. to 5:00 p.m. (Central Time) Monday - Friday, excluding holidays.

Contact SBS at:

Mailing Address:	SBS Technologies, Inc. 1284 Corporate Center Drive St. Paul, MN 55121-1245
Tel:	651-905-4700
FAX:	651-905-4701
Email:	support.commercial@sbs.com
Web:	http://www.sbs.com

Chapter 3: Example Applications

3.0 Introduction

Example applications provided in the Support Software demonstrate use of device driver features and facilitate device driver use.

Example applications:

APPLICATION	DESCRIPTION	API FUNCTIONS USED
dumpmem	Uses bt_mmap() to read and print to standard output 256 bytes of remote bus data.	bt_mmap()
readmem	Uses bt_read() to read and print to standard output 256 bytes of remote bus data.	bt_read()
bt_cat *	Similar to the UNIX cat program. Allows reading from or writing to the remote bus from std.o.	bt_read() bt_write()
datachk	Reads and writes from device using a specific pattern and verifies that no data or status errors occurred.	bt_read() bt_write()
bt_icbr	Registers for and counts a given interrupt type.	bt_icbr_install() bt_icbr_remove()
bt_info	Gets or sets driver parameters.	bt_get_info() bt_set_info()
bt_sendi	Sends an interrupt to the remote bus.	bt_send_irq()
bt_bind *	Binds a buffer to the remote bus, waits for user input, and then prints the first 256 bytes of the bound buffer.	bt_bind() bt_unbind()
bt_revs	Prints the software driver version and harware firmware version	bt_open()
VMEbus ONLY		
bt_tas	Performs a remote bus test and set atomic transaction.	bt_tas()
bt_cas	Performs a remote bus compare and swap atomic transaction.	bt_cas()
bt_reset	Resets the remote bus.	bt_reset()

***** *Not included in all drivers.*

Each of the files mentioned above is described in detail in the following sections of this chapter.

Most of the example applications use a -t option to select the logical device type. This implementation recognizes the following logical device types:

NAME	LOGICAL DEVICE TYPE
BT_DEV_A16	Alias for BT_DEV_IO
BT_DEV_A24	Remote VMEbus A24 space
BT_DEV_A32	Alias for BT_DEV_RR
BT_DEV_LM	Local memory device
BT_DEV_IO	Remote VMEbus A16 space or PCI I/O space
BT_DEV_RR	Remote VMEbus A32 space or PCI memory space
BT_DEV_MEM	Alias for BT_DEV_RR
BT_DEV_DEFAULT	Alias for BT_DEV_DP
BT_DEV_LDP	Local dual port memory if it exists
BT_DEV_RDP	Remote dual port memory if it exists
BT_DEV_DP	Alias for BT_DEV_RDP

See section 4.1 for more information about logical devices.

3.1 dumpmem Example Application

The dumpmem example application uses the device driver to create a memory-mapped pointer to the Local Memory Device, Dual Port RAM or to remote memory.

The dumpmem program uses the bt_mmap() interface to open a memory window to the selected device or remote memory and displays the beginning of the first 256 bytes as hexadecimal and ASCII characters.

dumpmem takes a unit number (0, 1, 2, etc.), a logical device (BT_DEV_RR, etc.), and a device offset. By default, dumpmem accesses the first 256 bytes of unit 0's Remote Dual Port RAM.

Usage: dumpmem –[tua]

dumpmem command line options:

OPTION	FUNCTION
-t <logical device>	Logical device type.
-u <unit number>	Unit; default is unit 0.
-a <addr>	Device address to memory map and read.

3.2 readmem Example Application

The readmem example application uses the bt_read() function to read from the local memory device, Dual Port RAM or remote bus memory. This example application uses only the core Mirror API. readmem is similar to dumpmem but uses a different method to transfer data.

After opening the device driver, readmem initializes the area into which data from the selected device will be read. The bt_read() call is used to read in all data at one time.

Usage: readmem –[tua]

readmem command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.
-a <addr>	Device address to read.

3.3 bt_cat Example Application

The bt_cat example application uses bt_read() and bt_write() API functions and the POSIX stdin/stdout mechanism to transfer data between the given logical device and the PCI computer. The program provides a convenient way to quickly download small sections of data or program code between the PCI bus computer and the remote system.

Usage:

bt_cat [-t <type>] [-u <unit>] [-a <addr>] [-l <length>] [-b <buflen>]

ARGUMENT	FUNCTION
-t <type>	logical device type.
-u <unit>	Unit number; default 0.
-a <addr>	Device address to read or write.
-l <length>	Length to read.
-b <buflen>	Size of internal buffer; how many bytes to read or write at one time.

If the –l option is given, bt_cat reads the given number of bytes from the remote system and writes them to the standard output. If the –l option is not given, bt_cat reads from the standard input and writes the data to the remote system until an End of File is encountered.

3.4 datachk Example Application

The datachk example application reads and writes from the device using a specific pattern and verifies that no data or status errors occurred. It uses bt_write() to write a buffer to a memory region on the remote bus and uses bt_read() to read the data back from the remote bus. This application uses only the core Mirror API.

Usage: datachk –[tualcmpswk]

datachk command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.
-a <addr>	Device address to access.
-l <numb>	Length of transfer.
-c <numb>	Number of repetitions to perform.
-m <misalign>	Amount to misalign the buffer.
-p <pattern>	Pattern with which to fill the buffer.
-s <seed>	Seed for the pattern.
-w <width>	Pattern width.
-k <addr>	Kernel address *(not supported by all drivers)*

3.5 bt_icbr Example Application

The bt_icbr example uses bt_icbr_install() to register to receive the given type of interrupt, and print a message when that interrupt occurs. Messages are only printed after input is received.

To exit the application, press q.

Usage: bt_icbr -[ui]

bt_icbr command line options:

OPTION	FUNCTION
-u <numb>	Unit number; default 0.
-i<numb>	Interrupt type to register for (bt_irq_t)

Because interrupts are registered for on a unit-wide basis, it does not matter which logical device is used.

3.6 bt_info Example Application

The bt_info example application allows easy access to the device driver parameters from section 5.3 using bt_get_info() and bt_set_info().

Usage: bt_info –[tupv]

bt_info command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <number>	Unit number; default 0.
-p <param>	Parameter name. The name of one a parameter listed in section 5.3.1. A parameter name must be specified.
-v <value>	Value to set the parameter to. If the –v option is not specified, the current value of the selected parameter is printed. Only parameters listed in section 5.3.1 may be changed.

Example:

To set unit 1's DMA threshold to 1000 (decimal):

bt_info –u 1 –p BT_INFO_DMA_THRESHOLD –v 1000

To see the current data width setting for the remote bus memory, unit 0:

bt_info –t BT_DEV_MEM –p BT_INFO_DATA_WIDTH

3.7 bt_sendi Example Application

The bt_sendi example application sends a programmed interrupt to the remote bus.

Usage: bt_sendi –[tu]

bt_sendi command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <number>	Unit number; default 0.

3.8 bt_tas Example Application

The bt_tas example application uses the bt_tas() function to do an atomic test and set to the remote bus.

Usage: bt_tas –[tua]

bt_tas command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.
-a <addr>	Device address to access.

➔ This example application cannot be run in PCI-to-PCI configurations.

➔ This example application is not supported by Model 1003.

3.9 bt_cas Example Application

The bt_cas example application uses the bt_cas() function to do an atomic compare and swap on the remote bus.

Usage: bt_cas –[tuacsd]

bt_cas command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.
-a <addr>	Device address to access.
-c <cmpval>	Compare value. If the remote memory location is this value, it is set to swapval.
-s <swapval>	Swap value. If the remote memory location is cmpval, it is set to this value.
-d <datasize>	Size (in bytes) of the remote memory location. Can be 1, 2, or 4 bytes in length.

➔ This example application cannot be run in PCI-to-PCI configurations.

➔ This example application is not supported by Model 1003.

3.10 bt_reset Example Application

The bt_reset example application uses the bt_reset() function to reset the remote bus.

Usage: bt_reset –[tua]

bt_reset command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.

➔ This example application cannot be run in PCI-to-PCI configurations.

➔ The remote reset jumper (SYS-6) on the VMEbus adapter card must be installed for the VMEbus to be reset.

3.11 bt_bind Example Application

The bt_bind example application uses the bt_bind() function to bind a buffer to the remote bus. It waits for user input, then prints the first 256 bytes of the bound buffer.

Usage: bt_bind –[tuls]

bt_bind command line options:

OPTION	FUNCTION
-t <type>	Logical device type.
-u <numb>	Unit number; default 0.
-l <length>	Length of the buffer to bind.
-s <swap>	Swapping value to use during bind (bt_swap_t)

➔ This example application is not supported by Model 1003.

3.12 bt_revs Example Application

The bt_revs example application opens the device driver and prints on to the screen the software version and the hardware version.

Usage: bt_revs –[t]

bt_revs command line options:

OPTION	FUNCTION
-t <type>	Logical device type.

Chapter 4: Using The Mirror API

4.0 Introduction

The SBS Mirror API supports accessing and controlling SBS adapters. The Mirror API provides a level of abstraction that allows source code compatibility with other operating systems and other SBS devices, as well as simplifying writing applications that use a range of SBS products.

The Mirror API needs the device driver to transfer data and perform certain administrative functions required by the device. The example applications use the library, the header "btapi.h", and all the header files included by this file.

Core Mirror API functions are Mirror API functions that are common to all implementations. In addition to these core functions, the core Mirror API includes routines that are hardware architecture or software environment specific.

By necessity, the exact behavior of some functions vary from driver to driver. These implementation-defined behaviors should not be relied upon by portable programs, but will be defined by any given driver.

4.1 Understanding Logical Devices

There are several resource types you may want to access on the same physical unit; each resource type is treated as a logical device with a separate device name. This facilitates access and keeps each device independent.

Logical devices that can be accessed are named as follows:

LOGICAL DEVICE	FUNCTION
Remote Dual Port BT_DEV_RDP or BT_DEV_DP	References the remote Dual Port RAM address space for the Dual Port RAM located on the remote adapter card. BT_DEV_RDP is supported on all SBS NanoBus products.
Local Dual Port BT_DEV_LDP	References the local Dual Port RAM address space for the Dual Port RAM located on the local adapter card. BT_DEV_LDP only exists on dataBLIZZARD products. Not supported by Model 1003.
Remote Bus I/O BT_DEV_IO or BT_DEV_A16	References Remote Bus I/O space. For VMEbus, this is the A16 (Short) space using Supervisory Data access, address modifier 2D. For the PCI bus, this is PCI I/O space. BT_DEV_IO is part of the core Mirror API. BT_DEV_A16 is supported on all SBS NanoBus products.
Remote Bus Memory BT_DEV_RR or BT_DEV_A32 or BT_DEV_MEM	References Remote Bus Memory space. For VMEbus, this defaults to the A32 (Extended) space using Supervisory Data access, address modifier 0D. BT_DEV_A32 is supported on all SBS NanoBus products. The 'BT_DEV_MEM' device is part of the core Mirror API. BT_DEV_RR and BT_DEV_A32 are supported on all SBS NanoBus products.
Remote Bus BT_DEV_A24	References a secondary Remote Bus Memory space assumed to use 24-bit addressing or less. For VMEbus, this defaults to the A24 (Standard) space using Supervisory Data access, address modifier 3D. For the PCI bus, this space will not exist.
Local Memory BT_DEV_LM	References a special buffer set up on the local system. The driver initializes the adapter to allow the remote system access to this buffer. BT_DEV_LM is supported on all SBS NanoBus products.
Default device BT_DEV_DEFAULT	For Mirror API, this device is aliased to the Local Dual Port device. This device is the default device used by example applications that only use the core Mirror API.
Diagnostic Device BT_DEV_DIAG	Allows access to memory regions necessary to implement Mirror API as well as spaces for internal SBS use only. This device is meant for internal SBS use only. Unauthorized use of this device may corrupt the driver and/or adapter.

The example applications use the bt_gen_name() routine to convert from the mnemonic name and unit number to the device name.

4.2 Initializing The Device And API

Routines in the Mirror API use an opaque object (a descriptor) to determine which logical device an operation is to be performed against. The underlying descriptor type is implementation defined. Do not make assumptions about the relationship between the descriptor used by the Mirror API and file descriptors on the system.

The software supports multiple cards in a single system. Each physical card is a separate unit number. Each unit has its own set of logical devices within that unit. The bt_gen_name() routine takes the physical unit number and logical device mnemonic as parameters and creates a string that uniquely identifies the device.

bt_gen_name() prototype:

```
char * bt_gen_name(into unit, bt_dev_t logical_device,
                   char *devname_p, size_t max_devname);
```

The application is responsible for allocating the buffer to hold the string. It passes in the address and size of this buffer, which bt_gen_name() then initializes with the device identification string. The return value of this routine can be passed directly into the bt_open() routine. The return value may not be either the buffer passed in or NULL. See section 5.1.2 for a detailed description of bt_gen_name() and its parameters.

The bt_open() routine does any initialization needed by the API to access a device. For Windows NT and many other systems, this includes a system call to open the device driver for access.

bt_open() prototype:

```
bt_error_t bt_open(bt_desc_t *btd_p, const char *devname_p,
                   bt_accessflag_t flags);
```

The first parameter is a pointer to the location that will contain the descriptor that bt_open() returns. This value is then used by all other Mirror API calls to track the unit and logical device being accessed.

The second parameter is the device identification string. Use the bt_gen_name() routine to create this string. The string format is operating system dependent.

The last parameter is the device read and write permission flags.

See section 5.1.3 for a detailed description of bt_open() and its parameters.

When bt_open() returns, the device descriptor is set. If an error occurred on the bt_open() call, this descriptor is only valid for use when calling the error reporting routines bt_perror() and bt_strerror().

After successfully opening the logical device, it is ready to use. The descriptor remains valid until released with the bt_close() routine.

4.3 Reading And Writing Data

The bt_read() and bt_write() functions provide a simple interface for transferring data. They provide serialization of all requests and use the current driver settings to determine swapping mode, data transfer size, and if the transfer should be performed via Programmed Input/Output (PIO) or Direct Memory Access (DMA).

bt_read() prototype:

```
bt_error_t bt_read(bt_desc_t btd, void *buffer_p,
                   bt_devaddr_t logical_device_address, size_t transfer_length,
                   size_t *actual_length_transferred_p);
```

bt_write() prototype:

```
bt_error_t bt_write(bt_desc_t btd, void *buffer_p,
                    bt_devaddr_t logical_device_address, size_t transfer_length,
                    size_t *actual_length_transferred_p);
```

The bt_read() and bt_write() routines provided by the library and the normal read() and write() functions that are part of the operating system are not the same. Differences include:

- Determination of errors and the amount of data transferred. The library always returns an error code to indicate success or failure. It has a separate parameter that indicates the amount of data transferred. Even if there is an error, the length parameter is updated to help determine if any part of the data was transferred.

- Library functions take the destination address as a parameter on the call. This parameter allows access to the full range of logical device addresses. Consequently, there is no need for a lseek() function to reposition within the remote device's address space. The application advances the current address so that subsequent calls to routines access the next address.

When performing a data transfer, DMA is automatically performed during bt_read() or bt_write() transfers that are longer than the DMA threshold value. See section 5.3.1 for information on setting DMA threshold values.

The adapter supports DMA transfers of 32-bit data aligned to any 4-byte boundary, and DMA transfers of 16-bit data aligned to any 2-byte boundary . DMA transfer length must always be multiples of 4 bytes. The device driver or library must perform PIO if the starting address cannot be aligned properly. If device driver performance is not as high as expected, check the alignment of the data buffers and destination addresses.

Several configuration parameters affect the way bt_read() and bt_write() transfer data: maximum data transfer size, byte swapping mode, and DMA threshold value. See section 5.3.1 for information about these settings.

4.4 Memory Mapping Support

Memory mapping allows an application to create a region of address space to directly access a resource. After the mapping is created, all references to this region of memory are completely handled by hardware. Via memory mapping, an application can directly access Remote Bus Memory, Remote Bus I/O, or Dual Port RAM.

Memory mapping is most efficient when small amounts of data are being accessed, or the data are at discontiguous addresses. The library provides the bt_mmap() and bt_unmmap() routines to create and release memory mapped regions to a logical device.

bt_mmap() prototype:

```
bt_error_t bt_mmap(bt_desc_t btd, void **map_p,
                   bt_devaddr_t logical_device_address, size_t map_length,
                   bt_accessflag_t flags, bt_swap_t swapping);
```

bt_unmmap() prototype:

```
bt_error_t bt_unmmap(bt_desc_t btd, void *map_p, size_t
                     map_length);
```

The bt_mmap() routine performs all required resource allocation and adapter hardware mapping register programming. If the logical device is Remote Bus Memory or I/O, the region must be unmapped and remapped after the remote adapter card is disconnected and reconnected. This condition can be detected by registering for error interrupts and watching for a power cycling.

The memory mapped region always becomes invalid after bt_unmmap() is called to release the region or the descriptor used for the bt_mmap() is closed with the bt_close() routine. After either action, the results of accessing this region are undefined.

It is important to call bt_unmmap() before closing a logical device. This allows the library and driver to release any resources used by the mapped region and reset the mapping registers. The length parameter for the bt_unmmap() call must match the length given in the original bt_mmapp() call.

4.5 Interrupt Call Back Routines

The library uses a call back routine to notify tasks that an interrupt occurred. An application can register an Interrupt Call Back Routine (ICBR) to be called anytime a remote bus device interrupt occurs, an Event is set, or an error interrupt occurs.

When the ICBR is registered, the application indicates:

- Which type of interrupt it is registering the routine for;
- The address of the routine to be called;
- A parameter that is passed as one of the arguments to the routine;
- A vector to match.

When an interrupt occurs, the Interrupt Service Routine (ISR) determines the type and vector for the interrupt sources and queues these for later interrupt dispatch.

Vectors are the non-zero return values from user ISRs or the IACK STATUS/ID value for VMEbus interrupts that are used to limit the times an ICBR is called to only those interrupts handled by a given user ISR.

The interrupt dispatcher receives the queued interrupt type and vector and searches the list of registered ICBRs for a match. If the interrupt type matches the registered type matches and the registered vector is BT_VECTOR_ALL or matches the queued interrupt vector, the dispatch causes the ICBR to execute.

On some systems, the context that the ICBR executes in may have limited functions. If the same ICBR is to be used on multiple operating systems, functions within the ICBR will be limited because some systems implement the ICBR within a signal handler or a lightweight thread.

Even in the most restricted contexts, access is available to:

- The full address space, including any memory mapped regions, of the task that originally registered the ICBR.
- Error checking and handling routines, bt_chkerr() and bt_strerror().

Not all implementations support I/O from within an ICBR. Consequently, bt_perror, which writes its output to stderr, may not be available on all systems from within an ICBR.

The ICBR runs in a POSIX thread within the application. The library creates a new thread for each library instance, and destroys the thread with the last call to bt_close(). There are no restrictions on what operations an ICBR can perform.

On Linux, the ICBR runs as a POSIX thread within the application. The library creates a new thread for the first call to bt_icbr_install(), and destroys the thread with the last call to bt_icbr_uninstall().

Because the interrupt data are queued, this queue could overflow. If an overflow occurs, any ICBRs registered for that type of interrupt are called with a BT_IRQ_OVERFLOW interrupt type. If the software cannot determine exactly which types of interrupts occurred during the overflow condition, all registered ICBRs are called with a BT_IRQ_OVERFLOW interrupt type.

4.6 Binding A Buffer To The Remote Bus

➔ Binding is not supported by Model 1003.

The Support Software offers two ways to share memory with the remote bus: local memory and binding a buffer to the remote bus.

Local memory is a logical device associated with a given unit. It is a buffer that the driver allocates out of kernel memory, and binds to the remote bus at boot. It can be opened using bt_open() and bt_gen_name() with a device type of BT_DEV_LM instead of BT_DEV_DEFAULT. Data can be transferred to and from the drivers buffer using bt_read() and bt_write(), and it can be mapped into the applications address space by using bt_mmap(). Other devices on the remote bus may access the local memory device by accessing the remote adapter card's remote memory window or remote memory device.

Another, more versatile but more complex way of sharing memory with the remote bus is by binding a buffer to the remote bus. This allows the applications buffer to show up on the remote bus, and be accessible to all devices on the remote bus. Thus, bt_bind() is the opposite of bt_mmap().

Unlike bt_mmap(), buffers bound to the remote bus must be aligned to a multiple of a given value, and their size in bytes must also be a multiple of a given value. This is necessary as most implementations can only bind whole pages, and either the whole page is bound or none of the page is bound. Consequently, in these implementations, the buffer to be bound must start at the beginning of a page, and must cover a whole number of pages.

The SBS API provides the INFO parameter BT_INFO_BIND_ALIGN that returns the alignment restrictions for the implementation. The size of the buffer must be an even multiple of this parameter. To go from an arbitrary length for the buffer to an aligned length, the code looks like:

```
bt_desc_t btd;              /* open descriptor */
size_t buffer_size;         /* size of the buffer to bind */
bt_data32_t align_size;     /* alignment restriction */
bt_error_t status;          /* return value */

/* get the alignment restriction */
status = bt_get_info(btd, BT_INFO_BIND_ALIGN, &align_size);
if (status != BT_SUCCESS) {
/* error */
}

/* align the size requirement to an even multiple of align_size */
if ((buffer_size % align_size) != 0) {
/* we need to add some to the buffer size to make it an even
multiple */
buffer_size += align_size - (buffer_size % align_size);
}
```

ISO 9899 Standard C has no portable way to align the start of the buffer. Most implementations have a flat address space where pointers are interchangeable with integers. The code above is used to adjust the length of the buffer. This would also work to adjust the beginning address of the buffer. However, with segmented architectures (MS-DOS is a popular example) and non-linear address systems (where the address is actually a hash value) the code that would work in a flat address space would not work. To work around this, the SBS API provides a macro, BT_ALIGN_PTR, returns the amount to add to the pointer to align it. This will always be less than BT_INFO_BIND_ALIGN. It is a good idea to allocate an "extra" amount of memory, BT_INFO_BIND_ALIGN bytes in size, in the buffer to make sure the buffer will fit.

```
void * orig_ptr;              /* buffer we malloc'ed */
void * buf_ptr;               /* buffer we will bind */
bt_data32_t align_size;       /* alignment size from above */
size_t buffer_size;           /* buffer size from above */

/* call malloc to allocate the buffer.  Note that we add a full
      align_size to the amount to allocate to make sure we can align
      the beginning of the buffer to an even multiple of align_size */

orig_ptr = malloc ( buffer_size + align_size );
if (orig_ptr == NULL) {
/* error */

}

/* we need to keep orig_ptr around to pass to free(), so we put
      the aligned buffer pointer into buf.  We cast orig_ptr to a pointer
      to bt_data8_t, to make sure we're adding bytes. */


buf_ptr = (void *) (((bt_data8_t *) orig_ptr) + BT_ALIGN_PTR(orig_ptr, align_size));
```

After the buffer is aligned, bt_bind() can be called to bind the memory to the remote bus. bt_bind() returns the offset into the remote window the buffer was bound at, and a bind descriptor. A bind descriptor is a mechanism for bt_bind() to pass information to bt_unbind(); it has no other use.

```
   bt_desc_t btd;                                  /* open descriptor */
   bt_binddesc_t bind_desc;                        /* bind descriptor to initialize */
   bt_devaddr_t window_off; = BT_BIND_NO_CARE      /* written with the window offset */
   void * buf_ptr;                                 /* buffer we will bind */
   size_t buffer_size;                             /* buffer size from above */
   bt_accessflag_t flags = BT_RDWR;                /* accesses requested- both read and write. */
   bt_swap_t swap = BT_SWAP_DEFAULT;               /* swapping method to use. */
   bt_error_t status;                              /* return value */

status = bt_bind(btd, &bind_desc, & window_offset, buf_ptr,

buffer_size, flags, swap);

      if (status != BT_SUCCESS) {
      /* Error */
   }
```

Once bound, the buffer can still be accessed normally by pointer dereference or array subscripting. Other devices can access the buffer by accessing the proper location in the remote adapter card's remote bus window. Because some SBS adapters cannot support remote access concurrently with DMA accesses, it may be necessary to create and hold an application level lock to make sure it does not make any concurrent accesses. A successful call to bt_bind() will return a window offset. This is the offset from the remote adapter's remote memory window that should be used to access the bound buffer. The BT_BIND_NO_CARE value that is passed in above, indicates that the driver is free to place the buffer at any open offset for the remote system to access.

After all accesses are complete, and before the device descriptor is closed or the buffer is freed, the buffer should be unbound:

```
bt_desc_t btd;                  /* open descriptor from above */
bt_binddesc_t bind_desc;        /* bind descriptor from above */
void * orig_ptr;                /* pointer originally returned from   malloc() above */

   status = bt_unbind(btd, bind_desc);
   if (status != BT_SUCCESS) {
   /* Error */
}

   /* we can free the buffer now */
   free(orig_ptr);
```

4.7 Doing Atomic Transactions on the VMEbus

Support Software provides two functions for doing atomic read/write transactions on the VMEbus: bt_tas() and bt_cas(). The function bt_tas() does an atomic bit test and set. It tests and sets the high order bit of the given byte, and returns the value the byte had before the high bit was set. These functions only work when the remote bus is VME.

```
   bt_desc_t btd;
   bt_devaddr_t addr;
   bt_data8_t prev_val;
   bt_status_t status;

   status = bt_tas(btd, addr, &prev_val);
   if (status = BT_SUCCESS) {
   /* Error */
}
   if ((prev_val & 0x80u) == 0) {
   printf ("Bit was clear and is now set.\n");
   } else {
   printf ("Bit was already set.\n");
}
```

The function bt_cas() does an atomic compare and swap to a memory location on the remote bus. First the value is read. If the value read is equal to a given compare value, the location is written with a swap value before the bus is released, otherwise the location remains unmodified.

```
    bt_desc_t btd;
    bt_devaddr_t addr;
    bt_data32_t cmpval, swapval, prevval;
    bt_status_t status;

    status = bt_cas(btd, addr, cmpval, swapval, &prevval);
    if (status != BT_SUCCESS) {
    /* Error */
}
    if (prevval == cmpval) {
    printf ("Value swapped- Value is now swapval.\n");
    } else {
    printf ("Value not swapped- Value is still prevval.\n");
}
```

4.8 Hardware Access Routines

➔ Hardware Access Routines are not supported by Model 1003.

The NanoBus family allows for the adapter card to DMA to or from any PCI bus location. When the location is a user's buffer, bt_read() or bt_write() must be used. However, when the location is another PCI card, the bt_hw_read() and bt_hw_write() routines must be used. See also sections 5.2.11 and 5.2.12.

The bt_hw_read() and bt_hw_write() implementations only use DMA mode. If the bus_addr, xfer_off or xfer_len are not aligned properly, the routines will return an error. Also, because only DMA is supported, it is illegal to transfer from local memory or local dual port. In addition, for PCI to PCI applications, remote dual port is not supported.

The bt_hw_read() and bt_hw_write() functions do not call bt_clrerr() or do the equivalent before starting the transfer. However, errors generated in the transfer will affect both the return value of these functions and the return values of later calls to bt_chkerr().

The bt_bind() and bt_unbind() functions of the core Mirror API allow a user's buffer to be bound to a unit. Besides binding application memory, you may wish to bind a hardware resource or device to a unit. this process would allow the remote system to access the hardware resource as if it were local. The hardware resource must be accessible from the bus that the SBS adapter card is installed in and the application must do the work of determining a bus address that can be used to access the resource they are interested in making accessible. The bt_hw_bind() and bt_hw_unbind() routines allow a local bus address to be made accessible from the remote system. See also sections 5.2.13 and 5.2.14.

When a local bus address is bound to a unit is accessible via PIO or DMA from the remote system until it is unbound. Thus, bound resources will reduce the maximum DMA transfer size for the bt_read(), bt_write(), bt_hw_read(), and bt_hw_write() routines.

The local bus address to be bound must be a multiple of BT_INFO_BIND_ALIGN.

The bt_binddesc_t descriptor is used to pass all needed information from a call to bt_hw_bind() to the corresponding call to bt_hw_unbind(), not including the device descriptor (device argument) passed to bt_hw_bind(). Which logical device the descriptor references is irrelevant – bt_hw_bind() binds the buffer to the associated unit's memory space or in the case of VME, wherever the REM RAM window is jumpered.

The device descriptor passed to bt_hw_unbind() must be the same one as was passed to the original bt_hw_bind() call. Not using the same descriptors results in undefined semantics.

Multiple calls to bt_unbind() with the same bind descriptor has undefined semantics.

Chapter 5: API Reference

5.0 Introduction

Chapter 6 documents the following functions provided by the Mirror API:

- bt_str2dev()
- bt_gen_name()
- bt_open()
- bt_close()
- bt_chkerr()
- bt_clrerr()
- bt_perror()
- bt_strerror()
- bt_init()
- bt_read()
- bt_write()
- bt_get_info()
- bt_set_info()
- bt_icbr_install()
- bt_icbr_remove()
- bt_lock()
- bt_unlock()
- bt_mmap()
- bt_unmmap()
- bt_dev2str()
- bt_ctrl()
- bt_bind()
- bt_unbind()

The following routines are NanoBus specific:

- bt_reg2str()
- bt_cas()
- bt_tas()
- bt_get_io()
- bt_put_io()
- bt_or_io()
- bt_reset()
- bt_send_irq()
- bt_status()

The following sections of this chapter describe the functions in detail.

5.1 Mirror API Routines

Mirror API routines detailed in sections 5.1.1 - 5.1.23 can be ported to other architectures.

5.1.1 Convert From String To Logical Device

bt_str2dev()

DESCRIPTION	Converts from a string containing the device name to a logical device type.
PROTOTYPE	bt_dev_t bt_str2dev(const char *name_p)
ARGUMENT	name_p String containing the device name.
COMMENTS	Example applications often use this routine when parsing the command line.
RETURN VALUES	Logical device type

5.1.2 Generate Device Name

bt_gen_name()

DESCRIPTION	Creates a string containing the device name for a particular unit and logical device type.
PROTOTYPE	char * bt_gen_name(int unit, bt_dev_t logical_device, char *devname_p, size_t max_devname)
ARGUMENTS	unit Unit number to reference. Valid range is 0 to 31. logical_device Logical device type to reference. devname_p Address of buffer in which to store the device name. max_devname Size of the buffer to hold device name.
COMMENTS	Return value should be passed unexamined to bt_open(). For Model 1003, the return value is the raw device name to open. If you are using the Mirror API, pass this directly to bt_open(). Programs that directly access the device driver, bypassing the Mirror API, can still use this routine to generate the device name.
RETURN VALUES	devname_p On success. NULL or other arbitrary pointer value On error.

5.1.3 Open A Logical Device For Access

bt_open()

DESCRIPTION	Opens the specified device for access by the Mirror API and returns the descriptor to use when accessing the device.
PROTOTYPE	bt_error_t bt_open(bt_desc_t *btd_p, const char *devname_p, bt_accessflag_t flags)
ARGUMENTS	btd_p Address to hold the descriptor created by bt_open. This must be a pointer to type bt_desc_t. devname_p Device name. Usually created by bt_gen_name() routine. flags The permission flags to indicate that reading and/or writing is to be allowed for this device. Valid flags include: BT_RD Read access allowed. BT_WR Write access allowed. BT_RDWR Both read and write access are allowed.
COMMENTS	May recognize NULL or other arbitrary pointer values to indicate an error (i.e. logical device not supported). If bt_open() returns an error value (anything other BT_SUCCESS), the descriptor can be used to call bt_perror() or bt_strerror() for the given error code. It is not valid for any other use, including other possible error codes. The descriptor returned in btd_p is only valid in the process that called bt_open() and any threads it spawns.
RETURN VALUES	BT_SUCCESS On success. BT_ENOSUP Logical device not supported (communicated from bt_gen_name() by special pointer value). BT_EINVAL Invalid parameter; possibly bt_gen_name() problem.

5.1.4 Close The Logical Device

bt_close()

DESCRIPTION	Closes the specified device, releasing the descriptor.
PROTOTYPE	bt_error_t bt_close(bt_desc_t btd)
ARGUMENT	btd Descriptor returned by the original bt_open() routine.
COMMENTS	There should be exactly one call to bt_close() for each successful call to bt_open(). If bt_close() returns an error value (anything other than BT_SUCCESS), the descriptor can be used to call bt_perror() or bt_strerror() for the given error code. It is not valid for any other use, including other possible error codes.
RETURN VALUES	BT_SUCCESS On success. Other error value On error.

5.1.5 Check For Errors On A Unit

bt_chkerr()

DESCRIPTION	Checks for errors on a unit. All logical devices on the same unit share the error status.
PROTOTYPE	bt_error_t bt_chkerr(bt_desc_t btd);
ARGUMENT	btd Descriptor returned by the original bt_open() routine.
COMMENTS	The error is maintained until an application clears them with bt_clrerr(), or reinitializes the adapter with bt_init().
RETURNS	BT_SUCCESS No error. Appropriate error value Otherwise.

5.1.6 Clear Errors On A Unit

bt_clrerr()

DESCRIPTION	Clears any error conditions on a unit.
PROTOTYPE	bt_error_t bt_clrerr(bt_desc_t btd)
ARGUMENT	btd Descriptor returned by the original bt_open() routine.
COMMENTS	None.
RETURN VALUES	BT_SUCCESS All errors were cleared. Other value Outstanding errors could not be cleared.

5.1.7 Print Error Message To stderr

bt_perror()

DESCRIPTION	Prints a description of a Mirror API error code to stderr.
PROTOTYPE	bt_error_t bt_perror(bt_desc_t btd, bt_error_t status, const char * message_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. status Value returned by one of the library functions indicating an error. message_p String with which to prefix any error messages.
COMMENTS	This function can accept invalid handles to print messages for error values returned from bt_open() and bt_close().
RETURN VALUES	BT_SUCCESS On success. Other error On failure.

5.1.8 String Error Message

bt_strerror()

DESCRIPTION	Creates a string containing a description of a Mirror API error code.
PROTOTYPE	char * bt_strerror(bt_desc_t btd, bt_error_t status, const char * message_p, char * buf_p, size_t buf_len)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. status Value returned by one of the library functions indicating an error. message_p String with which to prefix error messages. buf_p Address of the buffer to put the error message in. buf_len Size of the buffer.
COMMENTS	Will return NULL if the complete string does not fit in the buffer. This function can accept invalid handles to print messages for error values returned from bt_open() and bt_close().
RETURN VALUES	message_p On success. NULL On error.

5.1.9 Initialize A Unit

bt_init()

DESCRIPTION	Initializes a unit, resetting registers and bringing the device into a known state. This causes the remote device to be identified and clears a BT_EPWRCYC error code.
PROTOTYPE	bt_error_t bt_init(bt_desc_t btd)
ARGUMENT	btd Descriptor returned by the original bt_open() routine.
COMMENTS	On some systems (but not on Models 965, 1003 and 983) this may invalidate any mapped regions or bound buffers. As such, it should only be used as a last resort. Use bt_clrerr() instead.
RETURN VALUES	BT_SUCCESS Device was re-initialized. Other error value Device could not be re-initialized; a power cycle of both the system and the VMEbus is needed.

5.1.10 Read Data From Device

bt_read()

DESCRIPTION	Reads data from a logical device into an application's data buffer.
PROTOTYPE	bt_error_t bt_read(bt_desc_t btd, void *buffer_p, bt_devaddr_t transfer_addr, size_t transfer_length, size_t *actual_length_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. buffer_p Address of the data buffer to read data into. transfer_addr The logical device address from which to read data. transfer_length Transfer length (in bytes). actual_length_p The number of bytes actually read from the device. If there is an error, this will be less than the amount requested.
COMMENTS	The transfer is automatically performed via DMA if the length is greater than the DMA threshold and the buffers are properly aligned.
RETURN VALUES	BT_SUCCESS All data was successfully transferred. Other error value All data was not successfully transferred.

5.1.11 Write Data To Logical Device

bt_write()

DESCRIPTION	Writes data to a logical device from an application's data buffer.
PROTOTYPE	bt_error_t bt_write(bt_desc_t btd, void *buffer_p, bt_devaddr_t transfer_addr, size_t transfer_length, size_t *actual_length_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. buffer_p Address of the data buffer to write to the device. transfer_addr The logical device address to send data to. transfer_length The transfer length (in bytes). actual_length_p The number of bytes actually written to the device. If there is an error, this will be less than the amount requested.
COMMENTS	The transfer is automatically performed via DMA if the length is greater than the DMA threshold and the buffers are properly aligned.
RETURN VALUES	BT_SUCCESS All data was successfully transferred. Other error value All data was not successfully transferred.

5.1.12 Get Device Configuration Settings

bt_get_info()

DESCRIPTION	Gets the current value of a device configuration parameter. See the full list of parameters in section 5.3.
PROTOTYPE	bt_error_t bt_get_info(bt_desc_t btd, bt_info_t param, bt_devdata_t *value_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. param The parameter to read. value_p Address to store the parameter into.
COMMENTS	Some parameters are defined for all implementations but only supported on certain implementations. An error return of BT_ENOSUP indicates that the parameter is valid, but this specific implementation does not support it.
RETURN VALUES	BT_SUCCESS On success. Other error value On failure.

5.1.13 Set Device Configuration Settings

bt_set_info()

DESCRIPTION	Changes the current value of a device parameter. Not all parameters can be changed.
PROTOTYPE	bt_error_t bt_set_info(bt_desc_t btd, bt_info_t param, bt_devdata_t value)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. param The parameter to read. value The current value of that parameter.
COMMENTS	Some parameters are defined for all implementations but only supported on certain implementations. An error return of BT_ENOSUP indicates that the parameter is valid, but this specific implementation either doesn't support the parameter or doesn't support the value you attempted to set it to. To determine if the implementation has support for the parameter, find out if bt_get_info() also returns BT_ENOSUP.
RETURN VALUES	BT_SUCCESS On success. BT_ENOSUP Either the parameter or the value you attempted to set it to is not supported on this implementation. BT_EINVAL The value you attempted to set the parameter to is invalid. Other error value On failure.

5.1.14 Install An Interrupt Call Back Routine

bt_icbr_install()

DESCRIPTION	Installs an ICBR on that unit for a specific interrupt type.
PROTOTYPE	bt_error_t bt_icbr_install(bt_desc_t btd, bt_irq_t irq_type, bt_icbr_t *icbr_p, void *param_p, bt_data32_t vector)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. irq_type Type of interrupt this ICBR handles. icbr_p Address of the ICBR. param_p An opaque object that is passed through to the ICBR as a parameter. This is usually a pointer to a data structure. vector BT_VECTOR_ALL if the ICBR should be called for any occurrence of that interrupt type. Otherwise, the vector matches the vector for that interrupt type.
COMMENTS	The prototype for the application's ICBR is: void application_icbr(void * param_p, bt_irq_t irq_type, bt_data32_t vector); The first parameter (param_p) is whatever value was given when bt_icbr_install() was called. The API and driver pass this value through to the ICBR. The second parameter (irq_type) is the interrupt type that actually occurred. If the interrupt data queue has overflowed, this will be BT_IRQ_OVERFLOW. Otherwise, it will be the same type as given when the ICBR was installed. The last parameter (vector) is the value of the vector when this interrupt occurred. Only bt_chkerr(), bt_clrerr(), and bt_strerror() are guaranteed to be callable from within the ICBR on all platforms. However, Models 965 and 1003 run the ICBR within a lightweight (POSIX) thread allowing all functions to be called. On Windows and Linux, BT_IRQ_OVERFLOW should not be used when registering an ICBR. Any ICBR may be called with BT_IRQ_OVERFLOW as its type to indicate that the ICBR's queue has overflowed.
RETURN VALUES	BT_SUCCESS ICBR was installed. Other error value ICBR was not installed.

5.1.15 Remove An Interrupt Call Back Routine

bt_icbr_remove()

DESCRIPTION	Removes a previously installed ICBR. Returns BT_EINVAL if it cannot find a matching entry.
PROTOTYPE	bt_error_t bt_icbr_remove(bt_desc_t btd, bt_irq_t irq_type, bt_icbr_t *icbr_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. irq_type Type of interrupt that this ICBR handles. icbr_p Address of the ICBR.
COMMENTS	None.
RETURN VALUES	BT_SUCCESS ICBR was removed. Other error value On failure.

5.1.16 Lock A Unit

bt_lock()

DESCRIPTION	Provides any serialization required by the hardware architecture such that memory mapped access does not interfere with the device driver.
PROTOTYPE	bt_error_t bt_lock(bt_desc_t btd, bt_msec_t wait_len)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. wait_len Number of milliseconds to wait for the lock. There are two special values for this parameter: BT_FOREVER Never has a time out while waiting for the lock. BT_NO_WAIT Will not wait for the lock to become available.
COMMENTS	This function does nothing in this and future revisions of the 965, 946, 983 and 993 software. While pre-dataBLIZZARD adapters do require synchronization between the use of mmap pointers and bt_read, bt_write, bt_bind, bt_cas, bt_tas, and bt_reset, it was too much of a performance penalty to use bt_lock. Therefore, users should implement their own locking scheme, such as pthread_mutex_lrch, to provide protection from the things mentioned above. For Linux, we recommend that an application lock the unit before any memory mapped or bound buffer accesses are attempted. This is required because the NanoBus hardware cannot perform concurrent PIO and DMA transfers.
RETURN VALUES	BT_SUCCESS Unit was locked. BT_EBUSY Lock timed out. Other error value On failure.

5.1.17 Unlock A Previously Locked Unit

bt_unlock()

DESCRIPTION	Unlocks a unit previously locked by this task using this descriptor. Both the descriptor and the task ID must match those of the task that originally received the lock.
PROTOTYPE	bt_error_t bt_unlock(bt_desc_t btd)
ARGUMENT	btd Descriptor returned by the original bt_open() routine.
COMMENTS	This function does nothing in this and future revisions of the 964, 946, 983 and 993 software. While pre-dataBLIZZARD adapters do require synchronization between the use of mmap pointers and bt_read, bt_write, bt_bind, bt_cas, bt_tas, and bt_reset, it was too much of a performance penalty to use bt_unlock. Therefore, users should implement their own unlocking scheme, such as pthread_mutex_lrch, to provide protection from the things mentioned above. For Linux, the thread that locked the unit must be the one to unlock it.
RETURN VALUES	BT_SUCCESS Unit was unlocked. Other error value On failure.

5.1.18 Create A Memory Mapped Pointer Into A Logical Device

bt_mmap()

DESCRIPTION	Returns a memory mapped pointer to the adapter address space. The logical unit determines the type of memory space used.
PROTOTYPE	bt_error_t bt_mmap(bt_desc_t btd, void **map_p, bt_devaddr_t logical_addr, size_t map_length, bt_accessflag_t flags, bt_swap_t swapping)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. map_p Address of the pointer to the logical device. This is set by the bt_mmap() routine. logical_addr The logical device address to memory map to. This is the same value as would be used when doing a bt_read() or bt_write() to the device. map_length The number of bytes to memory map into the application's space. flags Permission flags to affect this memory mapped section. The following flags are currently supported: BT_RD Allow reads from the memory mapped location. BT_WR Allow writing to the memory mapped location. BT_RDWR Allow both reading and writing to the memory mapped region. swapping The swapping mode to use for this memory mapped section.
COMMENTS	For the Local Memory Device, swap bits are not used, and only BT_SWAP_NONE or BT_SWAP_DEFAULT is allowed for the swap value. Neither the length nor the address need to be aligned. The API may map extra space before or after the region to fulfill any alignment requirements. If the BT_RD and BT_WR flags set in the btAccessFlags parameter were not also set in the bt_open() call, this function will fail with BT_EACCESS. If the BT_WR flag is not set, subsequent use of the mapped memory for PIO writes generates a protection violation. However, if only the BT_WR flag is set, subsequent use of the mapped memory for PIO reads can not generate a violation. Multiple successful calls to bt_mmap() with identical units, logical devices, and remote addresses return different addresses and use different sets of mapping registers in the adapter hardware. However, each address returned will ultimately access the same address on the remote device.

(Table continued on next page.)

(Table continued from previous page.)

COMMENTS	For user PIO writes and reads, the mapping register swap bits are obtained from the parameter swapping rather than the swap bits set or obtained with bt_set_info() or bt_get_info(). The pointer returned is valid only in the context of the calling process. It is valid in the context of all threads within that process.
RETURN VALUES	BT_SUCCESS Region was mapped. Other error value Region was not mapped.

5.1.19 Unmap A Memory Mapped Location

bt_unmmap()

DESCRIPTION	Unmaps a previously created memory mapped pointer and releases any resources associated with that memory mapping.
PROTOTYPE	bt_error_t bt_unmmap(bt_desc_t btd, void *map_p, size_t map_len)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. map_p Value returned by the original bt_mmap() call. The address of the memory mapped region. map_len The number of bytes requested by the original bt_mmap() call.
COMMENTS	The map_len must be the same value as was passed in to bt_mmap() and map_p must be the value returned from bt_mmap(); otherwise the behavior is undefined.
RETURN VALUES	BT_SUCCESS Region was unmapped. Other error value On failure.

5.1.20 Convert From A Logical Device Type To A String

bt_dev2str()

DESCRIPTION	Returns a string with the suffix used in the device identification string to indicate a particular logical device.
PROTOTYPE	const char * bt_dev2str(bt_dev_t type)
ARGUMENT	type One of the defined logical device types.
COMMENTS	None.
RETURN VALUES	Pointer to constant string holding the logical device name On success. NULL Otherwise.

5.1.21 Call Directly Into The Driver I/O Control Function

bt_ctrl()

DESCRIPTION	Directly calls the device driver ioctl() entry point. This is useful for a program that directly called the device drive, but now needs to be converted to using the API.
PROTOTYPE	bt_error_t bt_ctrl(bt_desc_t btd, int ctrl, void * param_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. ctrl The command code (one of the BIOC_ values from btio.h) of the ioctl() to call. param_p The parameter for that particular ioctl() call. The use of this depends on which ioctl() is being called.
COMMENTS	bt_ctrl() is not portable across the various Mirror API implementations. It is intended as a temporary measure when trying to convert an application from directly accessing the device driver to using the Mirror API. On operating systems without an ioctl() entry point, this routine always returns BT_ENOSUP. To use this, the application would have to include both the “btapi.h” header file for the Mirror API and the “btio.h” header file for direct device driver access.
RETURN VALUES	BT_SUCCESS ioctl was successful. Other error value On failure.

5.1.22 Map An Application Supplied Buffer

bt_bind()

DESCRIPTION	Maps an application supplied buffer onto the remote bus.
PROTOTYPE	bt_error_t bt_bind(bt_desc_t btd, bt_binddesc_t* desc_p, bt_devaddr_t* rem_addr_p, void *buf_p, size_t buf_len, bt_accessflag_t flags, bt_swap_t swapping)
ARGUMENTS	btd Logical device handle returned from bt_open(). desc_p Pointer to the bind handle to initialize. rem_addr_p Pointer to the location to store the offset into the unit's remote memory window to which the buffer was bound. buf_p Pointer to the start of the buffer to bind. buf_len Length of the buffer to bind. flags Access rights requested on the bind. swapping Swapping method to use on remote accesses to the buffer.
COMMENTS	The bt_binddesc_t descriptor is used to identify the bind to undo in a call to bt_unbind(). It should be treated as an opaque data type. Which logical device the descriptor references is irrelevant; bt_bind() binds the buffer to the associated unit. Calls to bt_bind() will need to have their buffers aligned. The INFO parameter, BT_INFO_BIND_ALIGN, is provided for this purpose. If the buffer (buf_p) passed to bt_bind() is not aligned on a BT_INFO_BIND_ALIGN byte boundary, or is not a positive (non-zero) multiple of BT_INFO_BIND_ALIGN bytes long (buf_len), bt_bind() will return BT_EINVAL. The macro BT_ALIGN_PTR is provided to allow alignment of an arbitrary buffer. rem_addr_p should be initialized to a value of BT_BIND_NO_CARE before calling bt_bind() if you do not care what offset the driver binds the buffer to. Otherwise the requested offset should be passed into the driver in this field. A program should lock the adapter while any remote accesses are occurring, and not do DMA or PIO accesses at the same time. NanoBus adapters do no support concurrent bi-directional PIO or concurrent PIO and DMA. The bt_bind() functions exists on all SBS API products. If it is not supported, it will return BT_ENOSUP. Binding is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS On success. BT_ENOMEM Insufficient resources to bind the buffer. BT_EBUSY Buffer could not be bound due to conflicts with other bound buffers. BT_ENOSUP If bt_bind is not supported. This always returned on Model 1003 drivers. Appropriate error number Otherwise.

5.1.23 Unbind A Bound Buffer

bt_unbind()

DESCRIPTION	Unbinds a bound buffer and releases any resources consumed by a previous call to bt_bind().
PROTOTYPE	bt_error_t bt_unbind(bt_desc_t btd, bt_binddesc_t desc)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. desc Bind descriptor returned from bt_bind().
COMMENTS	The bt_unbind() functions exists on all SBS API products. If it is not supported, it will return BT_ENOSUP. Unbinding is supported on Models 965 version 2.0, and 983 version 2.0. The device descriptor passed to bt_unbind() must be the same one that was passed to the original bt_bind() call. The bind descriptor passed must be the one that was returned from the original bt_bind() call. Not using the same descriptors results in undefined behavior. Multiple calls to bt_unbind() with the same bind descriptor has undefined behavior.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2 NanoBus Specific Mirror API Functions

All routines discussed in this section are specific to the NanoBus hardware design, and may not port to other hardware architectures.

5.2.1 Convert Register To String

bt_reg2str()

DESCRIPTION	Given a bt_reg_t enumeration of a register, returns a null-terminated ASCII string containing the printable form of the register's name. The inverse of this function (bt_str2reg()) is not implemented.
PROTOTYPE	const char * bt_reg2str(bt_reg_t reg)
ARGUMENT	reg Register number that a name is needed for.
COMMENTS	The bt_reg2str() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function.
RETURN VALUES	Register name On success. NULL On error.

5.2.2 Compare And Swap Atomic Transactions

bt_cas()

DESCRIPTION	Does a compare and swap atomic transaction on the remote bus.
PROTOTYPE	bt_error_t bt_cas(bt_desc_t btd, bt_devaddr_t rem_off, bt_data32_t cmpval, bt_data32_t swapval, bt_data32_t *prevval_p, size_t data_size)
ARGUMENTS	btd Logical device handle returned from bt_open(). rem_off Address of the memory location to check. cmpval If the memory location is this value, swap_value is written to the location. swapval If the memory location is equal to compare_value, this value is written to the location. prevval_p Pointer to a bt_data32_t that is written with the value the memory location had before the swap value was written. data_size The size of the memory location to check. This must be one of BT_WIDTH_D8, BT_WIDTH_D16, or BT_WIDTH_D32.
COMMENTS	The bt_cas() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.3 Test And Set Atomic Transaction

bt_tas()

DESCRIPTION	Tests and sets a bit on the remote bus atomically. It acts on a single byte (bt_data8_t), and only checks the high-order bit of the byte. This function uses the address modifier set by BT_INFO_PIO_AMOD.
PROTOTYPE	bt_error_t bt_tas(bt_desc_t btd, bt_devaddr_t rem_off, bt_data8_t *prevval_p)
ARGUMENTS	btd Logical device handle returned from bt_open(). rem_off Address of the byte to test and set. prevval_p Pointer to a buffer to store the byte's value prior to modification.
COMMENTS	The bt_tas() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.4 Reads An Adapter CSR Register

bt_get_io()

DESCRIPTION	Reads an adapter CSR.
PROTOTYPE	bt_error_t bt_get_io(bt_desc_t device, bt_reg_t reg, bt_data32_t *result)
ARGUMENTS	device Logical device handle returned from bt_open(). reg Register to reads. result Pointer to buffer to hold the current value of the register.
COMMENTS	The bt_get_io() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function. Not all implementations will support all registers.
RETURN VALUES	BT_SUCCESS On success. BT_EINVAL Register not implemented on the current unit, or register is write-only. Appropriate error number Otherwise.

5.2.5 Writes An Adapter CSR Register

bt_put_io()

DESCRIPTION	Writes a new value into an adapter CSR.
PROTOTYPE	bt_error_t bt_put_io(bt_desc_t device, bt_reg_t reg, bt_data32_t value)
ARGUMENTS	device Logical device handle returned from bt_open(). reg Register to write to. value Value to write into the CSR.
COMMENTS	The bt_put_io() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function. Not all implementations will support all registers.
RETURN VALUES	BT_SUCCESS On success. BT_EINVAL Register not implemented on the current unit, or register is read-only. Appropriate error number Otherwise.

5.2.6 One Shot A Register

bt_or_io()

DESCRIPTION	One shots a value into a register. Bitwise ORs the value with the current value of the register and writes that value to the register.
PROTOTYPE	bt_error_t bt_or_io(bt_desc_t device, bt_reg_t reg, bt_data32_t value)
ARGUMENTS	device Logical device handle returned from bt_open(). reg Register to read. value Value to write to the register.
COMMENTS	The bt_or_io() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function. Not all implementations will support all registers.
RETURN VALUES	BT_SUCCESS On success. BT_EINVAL Register not implemented on the current unit, or register is read-only. Appropriate error number Otherwise.

5.2.7 Remote Reset

bt_reset()

DESCRIPTION	Performs a system reset on the remote bus.
PROTOTYPE	bt_error_t bt_reset(bt_desc_t device)
ARGUMENTS	device Logical device handle returned from bt_open().
COMMENTS	The bt_reset() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function. This implementation locks the unit and sleeps the length of time indicated by BT_INFO_RESET_DELAY before unlocking the unit. This allows the remote bus time to finish resetting.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.8 Send Interrupt To Remote Bus

bt_send_irq()

DESCRIPTION	Sends a programmed interrupt to the remote bus.
PROTOTYPE	bt_error_t bt_send_irq(bt_desc_t device)
ARGUMENTS	device Logical device handle returned from bt_open().
COMMENTS	The info parameter BT_INFO_USE_PT controls whether a PT or PR interrupt is sent to the remote system. The bt_send_irq() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.9 Send Vector to Remote Bus

bt_send_vector()

DESCRIPTION	Sends the given vector to the remote bus via programmed interrupt.
PROTOTYPE	bt_error_t bt_send_vector(bt_desc_t device, bt_data32_t vector)
ARGUMENTS	device Logical device handle returned from bt_open(). vector 32-bit vector to send to remote side. Must be less than BT_DRV_VECTOR_BASE in btngpci.h.
COMMENTS	The bt_send_vector() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.10 Status

bt_status()

DESCRIPTION	Returns the device status, including the Status Register.
PROTOTYPE	bt_error_t bt_status(bt_desc_t btd, bt_data32_t *status_p)
ARGUMENTS	btd Logical device handle returned from bt_open(). status_p Pointer to buffer to store status information into (length – bt_data32_t).
COMMENTS	The bt_status() function exists only on the SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using this function. For Model 1003, programs should check for BT_1003 before using this function.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.11 Read Data From Device To A Bus Address

bt_hw_read()

DESCRIPTION	Reads data from a logical device into a physical bus address. Only uses DMA Mode. Requests that cannot use DMA will return an error.
PROTOTYPE	bt_error_t bt_hw_read(bt_desc_t btd, bt_devaddr_t bus_addr, bt_devaddr_t xfer_off, size_t xfer_len, size_t* xfer_done_p)
ARGUMENTS	btd Descriptor returned by the original call to bt_open(). bus_addr A physical bus address of the data buffer to read data into. This is not the same as a virtual address received from a malloc() call or from a call to bt_mmap(). To read from application memory, use bt_read(). xfer_off The logical device address from which to read data. Cannot be used with BT_DEV_LDP or BT_DEV_LM. For PCI to PCI configurations, BT_DEV_DP and BT_DEV_RDP will also be illegal. xfer_len Transfer length (in bytes). xfer_done_p Pointer to the number of bytes actually read from the device. If there is an error, this may be less than the amount requested.
COMMENTS	This implementation only uses DMA mode. If the bus_addr, xfer_off or xfer_len are not aligned properly, the routines will return an error. Also, because only DMA is supported, it is illegal to transfer from local memory or local dual port. In addition, for PCI to PCI applications, remote dual port is not supported. This function does not call bt_clrerr() or do the equivalent before starting the transfer. However, errors generated in the transfer will affect both the return value of this function and the return values of later calls to bt_chkerr(). This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS All data were successfully transferred. Error value All data were not successfully transferred.

5.2.12 Write Data To Logical Device

bt_hw_write()

DESCRIPTION	Write data to a logical device from a physical bus address. Only uses DMA Mode. Requests that cannot use DMA will return an error.
PROTOTYPE	bt_error_t bt_hw_write(bt_desc_t btd, bt_devaddr_t bus_addr, bt_devaddr_t xfer_off, size_t xfer_len, size_t* xfer_done_p)
ARGUMENTS	btd Descriptor returned by the original call to bt_open(). bus_addr A physical bus address of the data buffer to write data from. This is not the same as a virtual address received from a malloc() call or from a call to bt_mmap(). To write from application memory, use bt_write(). xfer_off The logical device address from which to write data to. xfer_len Transfer length (in bytes). xfer_done_p Pointer to the number of bytes actually written to the device. If there is an error, this may be less than the amount requested.
COMMENTS	This implementation only uses DMA mode. If the bus_addr, xfer_off or xfer_len are not aligned properly, the routines will return an error. Also, because only DMA is supported, it is illegal to transfer from local memory or local dual port. In addition, for PCI to PCI applications, remote dual port is not supported. This function does not call bt_clrerr() or do the equivalent before starting the transfer. However, errors generated in the transfer will affect both the return value of this function and the return values of later calls to bt_chkerr(). This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS All data were successfully transferred. Error value All data were not successfully transferred.

5.2.13 Bind A Given Bus Address

bt_hw_bind()

DESCRIPTION	Binds a local bus resource to the remote bus.
PROTOTYPE	bt_error_t bt_hw_bind(bt_desc_t btd, bt_binddesc_t* desc_p, bt_devaddr_t* rem_addr_p, bt_devaddr_t* loc_addr, size_t buf_len, bt_accessflag_t flags, bt_swap_t swapping)
ARGUMENTS	btd Logical device handle returned from bt_open(). desc_p Pointer to the bind handle to initialize. rem_addr_p Pointer to the location to store the offset into the unit's remote memory window to which the buffer was bound. If BT_BIND_NO_CARE is passed in, then the buffer is bound to the first open spot of the remote memory window. Otherwise, the driver will bind the buffer at the *rem_addr_p offset of the remote memory window or return an error if it is already used. loc_addr Local physical bus address to bind. buf_len Length of the bus region to bind. flags Access rights requested on the bind. swapping Swapping method to use on remote accesses to the region.
COMMENTS	The bt_binddesc_t descriptor is used to identify the bind to undo in a call to bt_unbind(). It should be treated as an opaque data type. Which logical device the descriptor references is irrelevant; bt_bind() binds the buffer to the associated unit. Calls to bt_bind() will need to have their bus addresses aligned to BT_INFO_BIND_ALIGN. The rem_addr_p either indicates the offset into the remote memory window that the buffer should be bound at or that the first open offset should be used (BT_BIND_NO_CARE). This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS On success. BT_ENOMEM Insufficient resources to bind the buffer. BT_EBUSY Buffer could not be bound due to conflicts with other bound buffers. BT_ENOSUP If bt_bind is not supported. Appropriate error number Otherwise.

5.2.14 Unbind A Bound Local Bus Address

bt_hw_unbind()

DESCRIPTION	Unbinds a bound bus address and releases any resources consumed by a previous call to bt_hw_bind().
PROTOTYPE	bt_error_t bt_hw_unbind(bt_desc_t btd, bt_binddesc_t desc_p)
ARGUMENTS	btd Descriptor returned by the original bt_open() routine. desc Bind descriptor returned from bt_hw_bind().
COMMENTS	The device descriptor passed to bt_hw_unbind() must be the same one that was passed to the original bt_hw_bind() call. Not using the same descriptors results in undefined semantics. Multiple calls to bt_hw_unbind() with the same bind descriptor has undefined semantics. This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS On success. Appropriate error number Otherwise.

5.2.15 Gain Control Over The Given Semaphore

bt_take_sema()

DESCRIPTION	Attempts to get control of a given semaphore.
PROTOTYPE	bt_error_t bt_take_sema(bt_desc_t device, bt_reg_t sema, bt_msesc_t timeout)
ARGUMENTS	device Logical device handle returned from bt_open(). sema Semaphore register to take. timeout Number of milliseconds to wait for semaphore to become available. Note: first implementation will only accept BT_NO_WAIT.
COMMENTS	This function exists only on SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using bt_take_sema(). Not all implementations will support semaphore registers. This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS On success. BT_EINVAL Semaphore does not exist on the current unit or timeout value is not supported. BT_EBUSY Semaphore is currently owned by another adapter card. Appropriate error number Otherwise.

5.2.16 Release A Currently Owned Semaphore

bt_give_sema()

DESCRIPTION	Release a currently owned semaphore register.
PROTOTYPE	bt_error_t bt_give_sema(bt_desc_t device, bt_reg_t sema)
ARGUMENTS	device Logical device handle returned from bt_open(). sema Semaphore register to give.
COMMENTS	This function exists only on SBS NanoBus adapters. Programs should test that the preprocessor define BT_NBUS_FAMILY is defined before using bt_take_sema(). Not all implementations will support semaphore registers. This function is not supported on Model 1003.
RETURN VALUES	BT_SUCCESS On success. BT_EINVAL Semaphore does not exist on the current unit. BT_EBUSY Semaphore is currently owned by another adapter card. Appropriate error number Otherwise.

5.3 Device Configuration Parameters

These parameters control device driver configuration and operation. They are accessed by the bt_get_info() and bt_set_info() routines.

5.3.1 Parameters That Can Be Modified

Modifiable parameters can be read by bt_get_info() and changed with bt_set_info().

PARAMETER	BT_INFO_BLOCK
TYPE	boolean
DESCRIPTION	Force block transfer when reading or writing data. Default setting is TRUE.
PARAMETER	BT_INFO_PAUSE
TYPE	boolean
DESCRIPTION	When doing block transfers, rearbitrate for the bus more than required. This allows other bus masters faster arbitration for the bus, but reduces the transfer rate. Default setting is FALSE.
PARAMETER	BT_INFO_DATAWIDTH
TYPE	bt_width_t
DESCRIPTION	Determines the maximum size transfer used for PIO or DMA. Valid values are: BT_WIDTH_D8, BT_WIDTH_D16, BT_WIDTH_D32, and BT_WIDTH_ANY. The BT_WIDTH_ANY setting allows the device driver to choose the transfer size. The driver always selects the highest performance data transfer method. for Model 1003, the default setting is BT_EWIDTH_D32 since PCs rarely support D64. For all other software drivers, the default setting is BT_WIDTH_ANY.
PARAMETER	BT_INFO_DMA_AMOD
TYPE	int
DESCRIPTION	Address modifier to use for DMA transfers.
PARAMETER	BT_INFO_PIO_AMOD
TYPE	int
DESCRIPTION	Address modifier to use for PIO transfers.

(Parameters continued on next page.)

(Parameters continued from previous page.)

PARAMETER	BT_INFO_INC_INHIBIT
TYPE	boolean
DESCRIPTION	Model 1003 only. Can only be used with RQP600XXX adapter cards. Prevents the DMA controller from incrementing the remote bus address.
PARAMETER	BT_INFO_MMAP_AMOD
TYPE	int
DESCRIPTION	Address modifier to use when creating memory mapped sections. Used by the driver at the time the bt_mmap() call is made.
PARAMETER	BT_INFO_SWAP
TYPE	bt_swap_t
DESCRIPTION	Swapping mode to use. Data swapping is needed when data are shared between two systems with different byte ordering. The adapter hardware swaps data based on the assumed data size. This allows the hardware to correctly order the data regardless of transfer size used to move the data. The valid swapping modes are: BT_SWAP_DEFAULT Sets it to the default swapping mode for the adapter. BT_SWAP_NONE for Model 614 and 615 adapters, and BT_SWAP_BSBD for Models 616 and 617 adapters. BT_SWAP_NONE No swapping is performed. BT_SWAP_BSNBD Byte swap on non-byte data. BT_SWAP_WS Word swap. BT_SWAP_WS_BSNBD Word swap and byte swap on non-byte data. BT_SWAP_BSBD Byte swap on byte data. BT_SWAP_BSBD_BSNBD Byte swap on byte and non-byte data. BT_SWAP_BSBD_WS Byte swap on byte data and word swap. BT_SWAP_BSBD_WS_BSNBD Byte swap on byte and non-byte data, and word swap. For all logical devices except the Local Memory Device, BT_INFO_SWAP sets the mapping register swap bits used in subsequent bt_read() and bt_write() operations. The swap bits for user PIO read/writes are set through bt_mmap()

(Description continued on next page.)

(Description continued from previous page.)

BT_INFO_SWAP Description continued	For the Local Memory Device, BT_INFO_SWAP sets the swap bits for all subsequent accesses to the local memory via the remote adapter card. Setting the swap bits has no effect when the local system accesses the Local Memory Device. For more information on swapping, refer to your adapter hardware manual.
PARAMETER	BT_INFO_DMA_THRESHOLD
TYPE	unsigned int
DESCRIPTION	Minimum length of transfer at which DMA is attempted. Used by the device driver to determine when to use PIO instead of DMA for a read or write.
PARAMETER	BT_INFO_DMA_POLL_CEILING
TYPE	unsigned int
DESCRIPTION	Maximum length of DMA for which polled mode is used. Polled mode DMA causes the device driver to busy-wait for a DMA to complete, rather than allow other tasks to run. It is more efficient only if the transfer is small enough to complete in less time than is required to process an interrupt. Setting the BT_INFO_DMA_POLL_CEILING to a value less than the BT_INFO_DMA_THRESHOLD causes all DMAs to interrupt when done, disabling Polled Mode DMA.
PARAMETER	BT_INFO_TRACE
TYPE	bit-mask
DESCRIPTION	Software tracing level. This setting is global to all units and logical devices using the driver. Changing it on one logical device causes it to change on every unit and every logical device. It consists of various bit definitions that enable/disable various trace messages based on predefined functional sections. See the btngpci.h file in the include directory for a list of possible flags.
PARAMETER	BT_INFO_DMA_WATCHDOG
TYPE	bt_msec_t
DESCRIPTION	Maximum amount of time any DMA is allowed to take. This is used to detect “stuck” DMAs and complete them with an error.
PARAMETER	BT_INFO_USE_PT
TYPE	boolean
DESCRIPTION	TRUE Use PT interrupt for bt_send_irq() and bt_send_vector(). FALSE Use PR interrupt. Not supported on Model 1003.

(Parameters continued on next page.)

(Parameters continued from previous page.)

PARAMETER	BT_INFO_RESET_DELAY
TYPE	bt_msec_t
DESCRIPTION	Amount of time to block during bt_reset() / bt_setup() to allow the local bus to complete resetting. Not supported on Model 1003.
PARAMETER	BT_INFO_REM_RESET_DELAY
TYPE	bt_msec_t
DESCRIPTION	Maximum amount of time to block during bt_reset() to allow the remote bus to complete resetting. Model 1003 only.
PARAMETER	BT_INFO_TRANSMITTER
TYPE	boolean
DESCRIPTION	TRUE if the unit is configured as an adapter. Model 1003 only.

5.3.2 Parameters That Are Read Only

These parameters can only be read by bt_get_info(). Attempts to change them with bt_set_info() will return an error BT_EINVAL.

PARAMETER	BT_INFO_LOC_PN
TYPE	int
DESCRIPTION	SBS part number of the local adapter card.
PARAMETER	BT_INFO_REM_PN
TYPE	int
DESCRIPTION	SBS part number of the remote adapter card.
PARAMETER	BT_INFO_LM_SIZE
TYPE	unsigned int
DESCRIPTION	Size (in bytes) of the local memory device.
PARAMETER	BT_INFO_BIND_ALIGN
TYPE	size_t
DESCRIPTION	Bind alignment requirement (see section 4.6).
PARAMETER	BT_INFO_BIND_COUNT
TYPE	int
DESCRIPTION	Maximum number of binds. More than this number of bound buffers, of any size, will always fail.
PARAMETER	BT_INFO_BIND_SIZE
TYPE	size_t
DESCRIPTION	Largest possible bind (a bind request larger than this will always fail).
PARAMETER	BT_INFO_LOG_STAT
TYPE	bt_devdata_t
DESCRIPTION	Status of the logical device. Not supported on Model 1003.

(Parameters continued on next page.)

(Parameters continued from previous page.)

PARAMETER	BT_INFO_UNIT_NUM
TYPE	int
DESCRIPTION	Unit number of the descriptor.
PARAMETER	BT_INFO_TOTAL_COUNT
TYPE	bt_devdata_t
DESCRIPTION	Total number of interrupts received since boot.
PARAMETER	BT_INFO_EVENT_COUNT
TYPE	bt_devdata_t
DESCRIPTION	Total number of programmed interrupts received since boot.
PARAMETER	BT_INFO_ERROR_COUNT
TYPE	bt_devdata_t
DESCRIPTION	Total number of error interrupts received since boot.
PARAMETER	BT_INFO_IACK_COUNT
TYPE	bt_devdata_t
DESCRIPTION	Total number of remote bus interrupts received since boot.
PARAMETER	BT_INFO_ICBR_Q_SIZE
TYPE	bt_devdata_t
DESCRIPTION	Number of interrupt vectors that can be queued between the driver and the Mirror API without losing one. Not supported on Model 1003.
PARAMETER	BT_INFO_KMEM_SIZE
TYPE	bt_devdata_t
DESCRIPTION	Running total, in bytes, of kernel memory used by the driver. Not supported on Model 1003.

Chapter 6: Model 965

6.0 Introduction

Chapter 6 describes installation of Model 965 Support. It includes general information about the installation procedure, and gives a brief description of how to verify that the adapter is installed correctly and the device driver is loaded properly.

Model 965 Support Software provides a device driver and example programs to help applications programmers with adapter and system configuration. It currently supports all dataBLIZZARD models and adapter models 617, 618, 620, 628, 630, 7X2, 7X3, and RPQ600890 for PCI, CompactPCI and PMC computers running IRIX 6.5.

6.0.1 Components

Model 965 Support Software consists of the following components:

- An IRIX device driver with installation and removal script.
- A statically linked library implementing SBS's Mirror API. This API, found on most SBS Connectivity Products software, allows for easier porting between products.
- Example programs demonstrating how to map remote and/or dual-port memory into an application's memory space using the device driver.
- Example programs demonstrating the bt_read(), bt_write() functions for moving data blocks.
- Example programs demonstrating requirements for sending and receiving interrupts.

6.0.2 System And Hardware Requirements

- SBS recommends at least 128M bytes of RAM in SGI Origin, Octane, and O2 machines. Insufficient RAM can cause the Model 965 device driver to fail to load, or can hang the machine.
- Model 965 works with: all dataBLIZZARDs, all 7X2,7X3, 630, 628, 620 and 618 adapters, all RPQ 600890 cards, and Model 617 adapters with Part Number 85221511 PCI adapter cards (part numbers are located on a white label affixed to the adapter card).
- Model 965 ***does not*** work with: Model 617 adapters with Part Number 85221510 PCI adapter cards (part numbers are located on a white label affixed to the adapter card), all Model 616, 615 and 614 adapters.
- Operating systems: Model 965 works only with IRIX 6.5; it does not support IRIX 6.4 or IRIX 6.3.

6.1 Installation

6.1.1 Installation Notes

- Refer to the README file for revision history information.
- Files are stored in *tar* format.
- File or directory names in the form ./filespec relate to the directory in which the Support Software is installed. All files are located in a directory that is named for the software model and version number. For example, if version 2.0 of the software is installed in the /usr/local directory, the full path specification for the ./src directory is /usr/local/965/v2.0/src.
- Chapter 4 lists the contents of the ./src directory and describes the function of each file.
- Before example programs can run successfully, the device driver must be installed, the PCI and remote adapter cards must be installed, the adapter cable connected, and the remote system powered on. For dataBLIZZARD and Model 7X2/7X3 adapters, the remote system's device driver must be loaded and its local memory device enabled or a buffer bound to use any remote memory device.

6.1.2 Installing Support Software

Before extracting files:

1. Login as root.
2. Create a directory for Support Software tar files. Use the following commands (# denotes system prompt):

   ```
   # cd /usr/local
   # mkdir SBS
   ```

3. Change directories to the one you just created. Use the following command:

   ```
   # cd SBS
   ```

4. Retrieve the archive file from either the CD-ROM or SBS's web site (www.sbs.com), and extract it using the following command.

   ```
   # tar –xf 85222001.tar
   ```

6.1.3 Installing Device Driver

➔ You should be logged in as root and in the /usr/local/SBS directory.

6.1.3.1 Manual Installation

1. Move to the SBS ./sys directory:

   # cd 965/vx.x/sys
   (vx.x = version number)

2. Use the following command to install the device driver and related system files:

   # make install

   This command executes all other commands required to configure and install the device driver on your system.

3. Reboot the system.

➔ Step 3 must be completed or the system will not recognize the installed device driver.

➔ The PCI adapter must be installed for installation to continue.

4. Check that the adapter is installed correctly. The command

   ls /hw/bit3/965

   should list a directory for each PCI adapter installed (named "unit0", "unit1", etc.). If the ls command returns the error "No such file or directory" or the command fails to list any units, the driver did not load.

   If the driver fails to load, check that the PCI adapter cards are installed and firmly seated in the bus slots. Insufficient memory may cause the driver resource allocation to fail, causing the driver to fail to load. SBS recommends at least 128M bytes of RAM for Origin 200, Octane, and O2 systems.

5. Compile the dumpmem example program using the makefile provided in the ./src directory:

   #cd /usr/local/SBS/src
   #make dumpmem

6. If Dual Port RAM is installed, enter the command ./dumpmem (located in the ./src directory) to verify that the driver and adapter are working correctly. The program should display the data in the first section of dual-port memory in hexadecimal and ASCII format.

➔ If the dumpmem program fails, check the cable connection and jumper settings on the remote adapter card. For the VMEbus, pay special attention to the remote adapter card's System (SYS) and Bias jumper blocks and anything related to bus arbitration on the remote system.

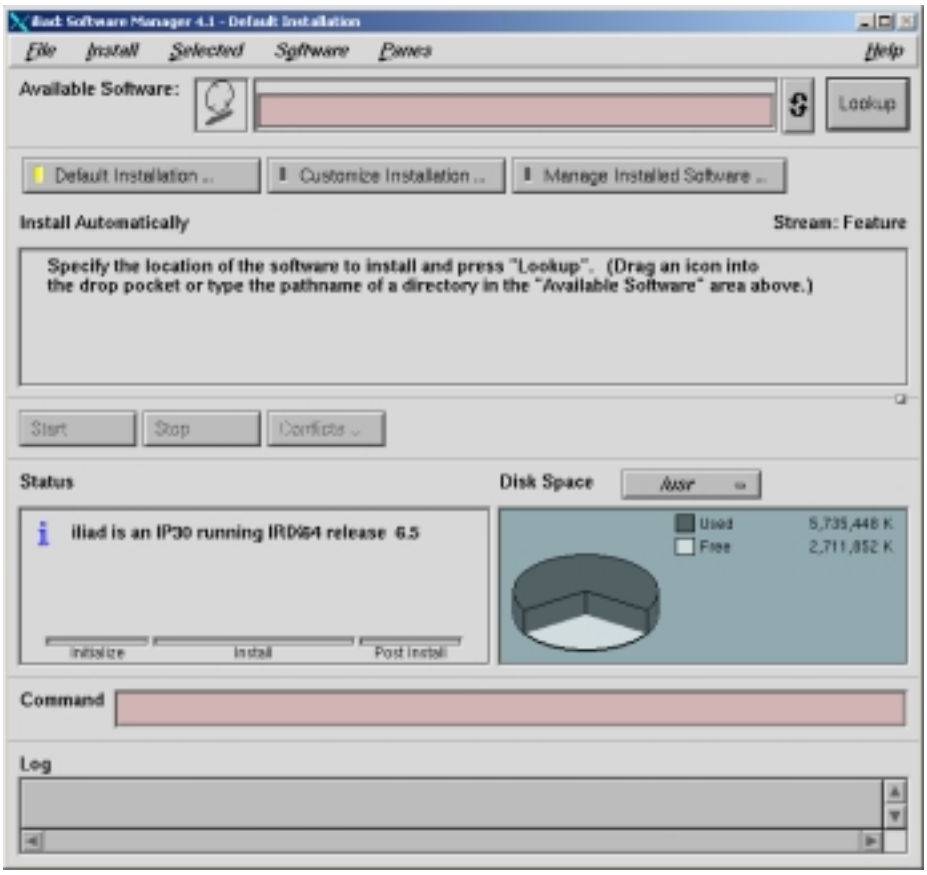
6.1.3.2 Software Manager Installation

1. Start the Software Manager as root:

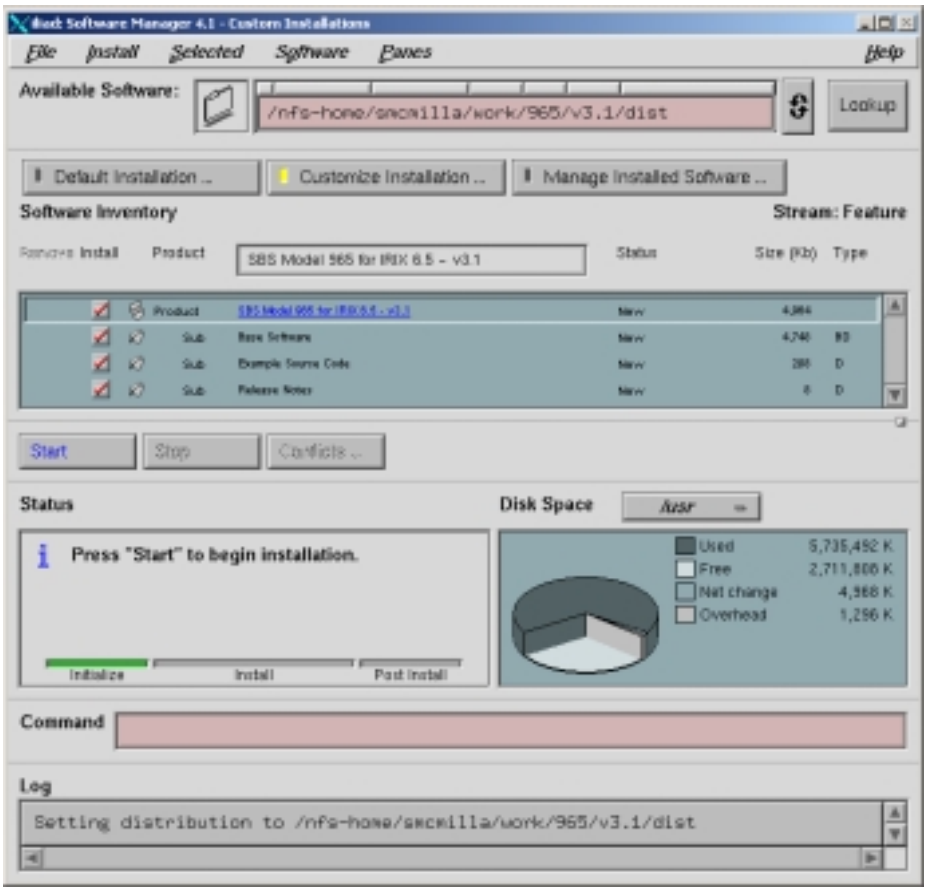
   # swmgr

2. Enter the directory where the tar file was extracted plus /dist in the Available Software button.

3. Click the Lookup button.

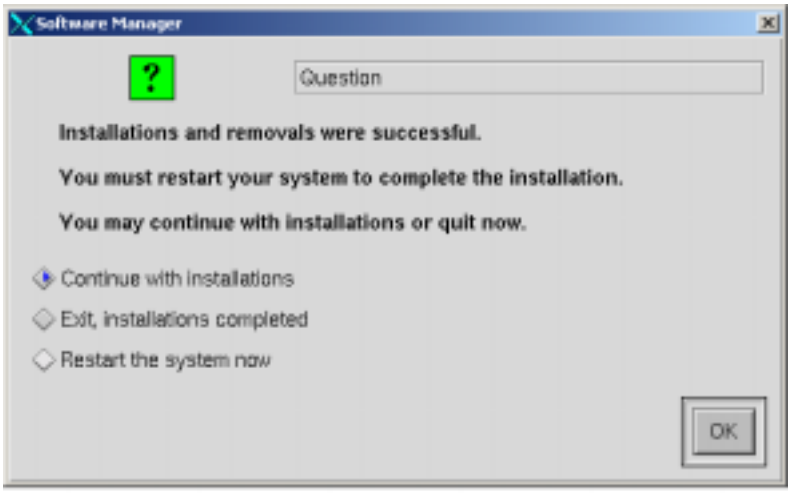


4. Click the Start box to begin installation. Or, click the Customize Installation button to customize the installation, then click the Start button.



5. Click the Restart the System Now line.

6. Click the OK button.



6.1.4 Configuring The Software

1. In most cases, you will not need to change the default settings. If no reconfiguration is required, go to step 2.

   Default configuration:

   Local memory enabled and sized at 64K bytes,

   Driver will only display error and warning messages (error messages resulting from programming errors will not be displayed).

   The default interrupt queue size will be used.

   If configuration changes are required, make sure you are in the ./sys directory. Then, using your choice of editor, edit the btp_flag.c file that contains the following configuration routine:

   ```
   void  btp_cnfg_flags(u_short unit, bt_cfg_param_t *config_p);
   ```

   This routine, called by the driver at initialization, is passed the physical unit number and pointer to the configuration settings for that unit.

   bt_cfg_param_t structure in btpio.h:

   ```
   /*
   **          Structure to pass the configuration information into the
   **          driver
   **
   **          See comments in sys/btp_flag.c for additional info
   */
   typedef struct bt_cfg_param_d {
      bt_data32_t config_flags;                 /* No config_flags currently defined */
      bt_data32_t trace_level;          /* Trace level */
      bt_data32_t lmem_size;            /* Local memory device size (bytes) */
      bt_data32_t q_size;               /* Interrupt queue size (# of interrupts stored */
   } bt_cfg_param_t;
   ```

Please note that support for the rram_start_addr configuration parameter has been removed. This parameter was incompatible with PCI to PCI support. VMEbus users will have to align their REM-RAM starting address to a multiple of 16M bytes. See section 5.3 for more information.

The trace_level value is one of the following:

VALUE	DESCRIPTION
BT_TRC_ERROR	Display only error messages.
BT_TRC_WARN	Display warning and error messages.
BT_TRC_INFO	Display informational, warning, and error messages.

Refer to section 5.3.1 for details on changing the trace level, although under normal operation there is no need to change the trace level from its default value.

➔ Operating the device driver with the trace level above the default, BT_TRC_WARN, severely degrades driver performance.

➔ If configuration changes are required, use caution when modifying btp_flag.c because the device may become inoperable if modifications are not made correctly.

➔ The lmem_size value gives the size in bytes of the local memory device; a value of 0 disables it.

➔ The q_size value give the number of interrupts that can be queued between the driver and awaiting ICBRs.

2. After adding, removing, or moving SBS PCI adapter cards to different PCI slots, the mkdev script in the sys directory should be rerun to reconfigure the driver.

3. Rebuild and re-install the device driver if you made any changes in steps 1 - 3. Use the following command:

   # make install

   The make install command executes all commands required to configure and install the device driver on your system.

➔ Make sure you are in the correct directory before executing make install. For example, if you loaded software version 2.0 in /usr/local, your working directory should be /usr/local/965/v2.0/sys.

4. Reboot the system to activate the new kernel.

6.2 Compiling Example Programs

➔ Only the source code to the example programs is distributed with the Support Software.

To compile the software:

Change directories to the ./src directory containing the example programs, then compile the example programs. Use the following commands:

```
# cd /usr/local/SBS/965/vx.x/src
(vx.x = version number)

# make all
```

To recompile a specific program you may have changed, use the command:

```
# make filename
```

6.3 Removing The SBS Support Software

In certain instances you may need to remove the Model 965 Support Software from a system.

To remove the SBS Support Software from a system, please follow the instructions below.

1. Remove all SBS adapters from the system and reboot.

2. Change directories to the ./sys directory. Use the following command.

   ```
   #cd /usr/local/SBS/965/vx.x/sys
   ```

3. Remove the device driver and configuration information from the system. Use the command below. You must be logged in as root.

   ```
   #make uninstall
   ```

4. Change directories to the top-level directory in which the software was originally installed. Use one of the two commands below.

   ```
   #cd ../../..
   ```

   or

   ```
   #cd /usr/local/SBS
   ```

5. Remove all versions of the SBS software. Use the following command.

   ```
   #rm -rf 965
   ```

Chapter 7: Model 946

7.0 Introduction

Chapter 7 describes installation of Model 946 Support. It includes general information about the installation procedure, and gives a brief description of how to verify that the adapter is installed correctly and the device driver is loaded properly.

SBS Model 946 Support Software with Nexus extensions for the PCI bus provides a loadable device driver for the SPARCstation® and example programs to help application programmers with adapter and system configuration. It currently supports the following SBS adapters:

- All dataBLIZZARDs.
- Model 616 that connects a PCI computer to an A32 VMEbus system.
- Model 617 with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 618 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 620 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection (no loopback diagnostics).
- Model 628 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection.
- Model 630 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection (no loopback diagnostics).

The software provides a device driver and installation tools necessary to quickly port Solaris VMEbus devices drivers to new Sun PCI workstations.

SBS's loadable device driver provides support that mimics the Solaris DDI routines for VMEbus drivers. Routines are supplied to map any VMEbus address to a virtual Solaris address, to probe the VMEbus by reads or writes, to install a device interrupt handler for an VMEbus interrupt level and vector, and to map Solaris memory so it can be accessed by VMEbus devices.

Example programs are included that aid in determining if the adapter hardware as well as the support software is functioning correctly.

Currently, Model 946 supports Solaris 2.5.1 or higher.

7.0.1 System & Hardware Requirements

PCI Bus: Root privileges to install the support software.

VMEbus: The remote reset jumper (SYS-5) on the VMEbus adapter card must be in place to use the remote VMEbus reset function.

7.1 Installation

7.1.1 Installation Notes

- Refer to the README file for revision history information.
- Files are stored in *tar* format. Files may also be compressed if the name ends with .2.
- File or directory names in the form ./filespec relate to the directory in which the Support Software is installed. All files are automatically placed in the /opt/SBSECm946/vX.Y directory. Where X.Y is the version of the software being installed.
- Chapter 4 lists the contents of the ./src directory and describes the function of each file.
- The PCI adapter card must be installed before the device driver can be loaded. The driver can successfully load itself even when the VMEbus system is not connected.
- Before example programs will run successfully, the device driver must be installed, the cable connected, and the VMEbus system powered on.

7.1.2 Installing Support Software

➔ # denotes a system root prompt.

Before extracting files:

1. Login as root.

2. Check that the adapter is installed correctly. Use the prtconf command located in the /usr/sbin directory; enter the command as follows:

   ```
   # prtconf | grep 108a
   ```

   If the adapter card is installed in the PCI system, the output generated by this command will include a section similar to this:

   ```
   pci108a,1, instance #0 (driver not attached)
   ```

   If the PCI adapter card is not installed or is incorrectly installed, device pci108a,? will not be reported.

   For additional information about the pci108a,1 device, as well as other devices in the system, enter:

   ```
   # prtconf -p -v | more
   ```

3. Retrieve the archive file from either the CD-ROM or SBS's web site (www.sbs.com), and extract it using the tar command located in the /usr/sbin directory. If the file you downloaded or received ends with a .Z, you will need to uncompress it using the uncompress command as demonstrated below.

   ```
   # uncompress 85221907.tar.Z
   # tar –xf 85221907.tar
   ```

4. Install the Model 946 Support Software using the /usr/sbin/pkgadd command and then follow the on-screen instructions.

```
# pkgadd SBSECm946
```

5. Make sure the following sub-directories were created in the /opt/SBSECm946/vX.Y directory:

```
# ls –l /opt/SBSECm946/vX.Y
```

SUB-DIRECTORY	CONTENTS
./sys	The device driver and installation script.
./src	Source files and makefile for all example programs.

7.1.3 Changing The Driver's Configuration

Certain driver parameters are inspected only when the driver loads. These parameters are called boot time configurable parameters and are controlled through the driver's configuration file, btp.conf. A copy of the driver's default configuration file is kept in the sys directory and can be modified with any text editor. The value for the boot time configuration parameters can be modified by changing the appropriate value in the btp.conf file, copying this file to /kernel/drv directory and reloading the SBS device driver. The process is described below.

1. Log into the root account.

2. Change to the sys directory:

```
# cd /opt/SBSECm946/vX.Y/sys
```

3. If this is the first time you have modified the btp.conf file, change the permissions to allow writing:

```
# chmod 644 btp.conf
```

4. Unload the driver (make utility can often be found in /usr/ccs/bin directory):

```
# make unload
```

5. Modify the boot time parameters of interest using a text editor. A description of the boot time parameters follows.

NAME	DEFAULT	LEGAL	DESCRIPTION
rram_addr	0xffffffff	0x0 – 0xffff0000	Setting of low REM RAM jumper block on VME
latency_timer	0x0	0x0, 0x20 – 0xe0	PCI Latency timer, 0 -> system default
dp_enable	0x0	0x0, 0x1	Dual port enable, 0x1 -> enabled
lm_enable	0x0	0x0, 0x1	Local memory enable, 0x1 -> enabled
trace_flags	0x3	0x0 – 0xffffffff	Trace flags bit map, see btnbus.h file for definition

6. Reload the driver (make utility can often be found in /usr/ccs/bin directory):

```
# make load
```

7.1.4 Checking The Installation

1. Issue the package information command, pkginfo, located in the /usr/sbin directory. The results should indicate that this package has been installed.

2. Check that the adapter is installed correctly and the device driver loaded properly. Use the prtconf command located in the /usr/sbin directory; enter the command as follows:

   ```
   # prtconf | grep 108a
   ```

   If the driver successfully installed, the output generated by this command will include a line similar to one of the following:

   ```
   pci108a,1 instance #0    (For Model 617 installed as Unit 0)
   pci108a,3 instance #1    (For Model 616 installed as Unit 1)
   pci108a,10 instance #2   (For Model 618 installed as Unit 2)
   ```

   (The unit number may be different for your system.)

   If this section looks like the following, the adapter card is installed in the PCI chassis; however, the device driver was not loaded correctly:

   ```
   pci108a,1, (driver not attached)
   ```

   If the PCI adapter card is not installed or is incorrectly installed, device pci108a will not be reported and the device driver will not load correctly.

3. If Dual Port RAM is installed and enabled, enter the command

   ```
   # ./dumpmem
   ```

   (the dumpmem program is located in the ./src directory) to verify that the driver and adapter are working correctly. The program should display the data in the first section of dual-port memory in a hexadecimal and ASCII format.

4. To access memory on the VMEbus system, try using the dumpmem program with the given flags.

   Your dumpmem command line entry should be similar to the following example (should be replaced by the address of at least 256 bytes of memory on your VMEbus).

   ```
   # ./dumpmem –t re -a 0x80000000
   ```

5. If the dumpport or dumpmem program fails, check the cable connection and jumper settings on the VMEbus adapter card. Pay special attention to the VMEbus adapter card's system jumper block and anything related to bus arbitration on the VME system.

7.2 "Nexus-Link" Kernel Interface Routines

Model 946 Support Software has been specifically designed to ease the porting of Solaris VMEbus device drivers. It contains kernel level routines that mirror the DDI routines supplied by the Solaris operating system. When the Model 946 software is coupled with a Model 616, 617 or 618 PCI to VMEbus adapter, the combination provides an easy method for customers to upgrade from either VMEbus based or SBus-based Sun workstations to the new PCI-based Sun workstations.

The kernel level routines provided by Model 946 can be broken down in to four categories: Mapping the VMEbus, Accessing the VMEbus, Handling a VMEbus Interrupt and Preparing for VMEbus Device DMA.

Mapping VMEbus Memory

- btp_ddi_map_regs
- btp_ddi_unmap_regs

Accessing the VMEbus

- bt_ddi_peek8
- bt_ddi_peek16
- bt_ddi_peek32
- bt_ddi_peek
- bt_ddi_poke8
- bt_ddi_poke16
- bt_ddi_poke32
- bt_ddi_poke

Handling a VMEbus Interrupt

- btp_ddi_get_iblock_cookie
- btp_ddi_add_intr
- btp_ddi_remove_intr

Preparing for VMEbus Device DMA

- btp_ddi_dma_buf_setup
- btp_ddi_dma_free
- btp_ddi_dma_htoc

7.2.1 Mapping The VMEbus

Two routines are provided for mapping VMEbus addresses into kernel addresses so that a Solaris driver can access them. The two routines are described below. For a detailed description of the routine please see the Solaris man page for the corresponding DDI routine or the *Writing Device Drivers* manual by Sun Microsystems.

7.2.1.1 Map VMEbus Memory - btp_ddi_map_regs()

FUNCTION	Creates a mapping to the register set given in the xxx.conf file. Returns a kernel pointer to the region described.
PROTOTYPE	int btp_ddi_map_regs (u_int unit, dev_info_t *dip, u_int mumber, caddr_t *kaddrp, off_t offset, off_t len);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. mumber = Register set number from the requesting driver's xxx.conf file reg property kaddrp = Address of a pointer to remote memory (set upon success). offset = Offset into register space. len = Number of bytes to map
DESCRIPTION	Available in kernel mode only. bt_ddi_map_regs() is a macro that always uses unit 0. An amod value of 0x1 in the reg property will setup a mapping to Dual Port RAM. An amod value of 0x2 in the reg property will setup a mapping the local memory device.
DDI Routine	Mimics the ddi_map_regs() function.

7.2.1.2 Unmap VMEbus Memory - btp_ddi_unmap_regs()

FUNCTION	Releases a mapping previously created with btp_ddi_map_regs
PROTOTYPE	void btp_ddi_unmap_regs (u_int unit, dev_info_t *dip, u_int rnumber, caddr_t *kaddrp, off_t offset, off_t len);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. rnumber = Register set given to the prior btp_ddi_map_regs call. kaddrp = Address of the pointer created by the prior btp_ddi_map_regs call. offset = Offset into register space specified in the prior btp_ddi_map_regs call. len = Number of bytes to map specified in the prior btp_ddi_map_regs call.
DESCRIPTION	Available in kernel mode only. bt_ddi_unmap_regs() is a macro that always uses unit 0.
DDI Routine	Mimics the ddi_unmap_regs() function.

7.2.2 Accessing the VMEbus

There are two sets of routines that allow kernel mode drivers access to the VMEbus and obtain status information about the result. Normally, the kernel mode driver can dereference pointers obtained by the btp_ddi_map_regs() call to access VMEbus resources. However, if there is a chance that the access may result in a VMEbus error or the driver is interested in verifying that the access succeeded, it should use the routines described in sections 7.2.2.1 – 7.2.2.8. For example, during the driver's probe() routine, the VMEbus read may result in a bus error if the VMEbus card is not installed.

7.2.2.1 Reading An 8-Bit Value From The VMEbus

bt_ddi_peek8

FUNCTION	Cautiously tries to read an 8-bit value from the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_peek8 (dev_info_t *dip, int8_t *addr, int8_t *valuep);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Char pointer to the VMEbus location to read. valuep = Pointer to 8-bit storage. If read is successful, this is updated with the value read.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_peekc() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call. Value is only updated on success.
DDI Routine	Mimics the ddi_peek8() function.

7.2.2.2 Reading A 16-Bit Value From The VMEbus

bt_ddi_peek16

FUNCTION	Cautiously tries to read a 16-bit value from the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_peek16 (dev_info_t *dip, int16_t *addr, int16_t *valuep);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Short pointer to the VMEbus location to read. valuep = Pointer to 16-bit storage. If read is successful, this is updated with the value read.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_peeks() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call. Value is only updated on success.
DDI Routine	Mimics the ddi_peek16() function.

7.2.2.3 Reading A 32-Bit Value From The VMEbus

bt_ddi_peek32

FUNCTION	Cautiously tries to read a 32-bit value from the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_peek32 (dev_info_t *dip, int32_t *addr, int32_t *valuep);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Long pointer to the VMEbus location to read. valuep = Pointer to 32-bit storage. If read is successful, this is updated with the value read.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_peekl() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call. Value is only updated on success.
DDI Routine	Mimics the ddi_peek32() function.

7.2.2.4 Reading A VMEbus Value From A Given Unit

btp_ddi_peek

FUNCTION	Cautiously tries to read a value from the given pointer and checks for errors.
PROTOTYPE	int btp_ddi_peek (u_int unit, caddr_t kaddr_p, size_t width, bt_data32_t *val_p);
ARGUMENT	unit = Hardware unit/instance number to use. kaddr_p = Pointer to the VMEbus location to read. width = Number of bytes to read in a single transaction. val_p = Pointer to 32-bit storage. If read is successful, this is updated with the value read.
DESCRIPTION	Available in kernel mode only. kaddr_p must be obtained from the btp_ddi_map_regs() call.
DDI Routine	Provides the same functionality provided in ddi_peek??() functions.

7.2.2.5 Writing An 8-Bit Value To The VMEbus

bt_ddi_poke8

FUNCTION	Cautiously tries to write an 8-bit value to the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_poke8 (dev_info_t *dip, int8_t *addr, int8_t value);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Char pointer to the VMEbus location to write. value = 8-bit value to write.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_pokec() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call.
DDI Routine	Mimics the ddi_poke8() function.

7.2.2.6 Writing A 16-Bit Value To The VMEbus

bt_ddi_poke16

FUNCTION	Cautiously tries to write a 16-bit value to the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_poke16 (dev_info_t *dip, int16_t *addr, int16_t value);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Short pointer to the VMEbus location to write. value = 16-bit value to write.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_pokes() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call.
DDI Routine	Mimics the ddi_poke16() function.

7.2.2.7 Writing A 32-Bit Value To The VMEbus

bt_ddi_poke32

FUNCTION	Cautiously tries to write a 32-bit value to the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_poke32 (dev_info_t *dip, int32_t *addr, int32_t value);
ARGUMENT	dip = Device information pointer of the requesting driver. addr = Pointer to the 32-bit VMEbus location to write. value = 32-bit value to write.
DESCRIPTION	Available in kernel mode only. Always uses unit 0. bt_ddi_pokel() is a macro for Solaris 2.5 systems that always uses unit 0. addr must be obtained from the btp_ddi_map_regs() call.
DDI Routine	Mimics the ddi_pokel() function.

7.2.2.8 Writing A value To The VMEbus

btp_ddi_poke

FUNCTION	Cautiously tries to write a value to the given pointer and checks for errors.
PROTOTYPE	int bt_ddi_poke (u_int unit, caddr_t kaddr_p, size_t width, bt_data32_t Val);
ARGUMENT	unit = Hardware unit/instance number to use. kaddr_p = Pointer to the VMEbus location to write. width = Number of bytes to read in a single transaction. Val = Value to write.
DESCRIPTION	Available in kernel mode only. kaddr_p must be obtained from the btp_ddi_map_regs() call.
DDI Routine	Provides the same functionality provided in ddi_pokek??() functions.

7.2.3 Handling A VMEbus Interrupt

Many VMEbus devices generate an interrupt to indicate that it requires attention or has completed some operation. Therefore, the device driver must register an Interrupt Service Routine (ISR) with the Operating System, to respond to its device's interrupt. The following routines allow a driver to install or remove an ISR and provide protection from the ISR to the other parts of the driver.

7.2.3.1 Get Interrupt Block Cookie

btp_get_iblock_cookie()

FUNCTION	Retrieves an interrupt block cookie that can be used to initialize mutexes and locks for protecting driver code from the ISR.
PROTOTYPE	int btp_ddi_get_iblock_cookie (u_int unit, dev_info_t *dip, u_int inumber, ddi_iblock_cookie_t *iblock_cookiep);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. inumber = interrupt number from the interrupts property of the driver's xxx.conf file. iblock_cookiep = Pointer to a cookie storage item. The storage item will be initialized if the call succeeds.
DESCRIPTION	Available in kernel mode only. bt_ddi_get_iblock_cookie() is a macro that always uses unit 0.
DDI Routine	Mimics the ddi_get_iblock_cookie() function.

7.2.3.2 Register A VMEbus Interrupt Service Routine

btp_ddi_add_intr()

FUNCTION	Installs an interrupt service routine for the given VMEbus interrupt.
PROTOTYPE	int btp_ddi_add_intr (u_int unit, dev_info_t *dip, u_int inumber, ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *idevice_cookiep, u_int (* int_handler) (caddr_t), caddr_t int_handler_arg);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. inumber = interrupt number from the interrupts property of the driver's xxx.conf file. iblock_cookiep = Should always be NULL. idevice_cookiep = Should always be NULL. int_handler = Address of the interrupt service routine to install. int_handler_arg = Pointer to data structure that gets passed to int_handler() when it is called to handle a device interrupt.
DESCRIPTION	Available in kernel mode only. bt_ddi_add_intr() is a macro that always uses unit 0. Only one ISR may be installed for any given interrupt level/vector pair. The same ISR may be installed multiple times for different interrupt level/vector pairs.
DDI Routine	Mimics the ddi_add_intr() function.

7.2.3.3 Unregister A VMEbus Interrupt Service Routine

btp_ddi_remove_intr()

FUNCTION	Removes an installed interrupt service routine for the given VMEbus interrupt.
PROTOTYPE	int btp_ddi_remove_intr (u_int unit, dev_info_t *dip, u_int inumber, ddi_iblock_cookie_t *iblock_cookiep);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. inumber = interrupt number from the interrupts property of the driver's xxx.conf file. iblock_cookiep = Should always be NULL.
DESCRIPTION	Available in kernel mode only. bt_ddi_remove_intr() is a macro that always uses unit 0. btp_ddi_add_int() must have been previously called with the given inumber.
DDI Routine	Mimics the ddi_remove_intr() function.

7.2.4 Preparing For VMEbus Device DMA

The following routines allow a VMEbus device driver to DMA directly into a memory region described by a buf structure. Only three of the most popular DMA related routines are currently supported: btp_ddi_dma_buf_setup(), btp_ddi_dma_free(), and btp_ddi_dma_htoc(). If you use a different Solaris DDI DMA routine, please contact SBS Technologies for information on extending DMA support.

7.2.4.1 Buffer DMA Setup

btp_ddi_dma_buf_setup

FUNCTION	Prepares the system to DMA to the section of memory described in the buf structure.
PROTOTYPE	int btp_ddi_ dma_buf_setup (u_int unit, dev_info_t *dip, struct buf *bp, u_int flags, int (*waitfp) (caddr_t), caddr_t arg, ddi_dma_lim_t *lim, bt_ddi_dma_handle_t *handlep);
ARGUMENT	unit = Hardware unit/instance number to use. dip = Device information pointer of the requesting driver. bp = Pointer to buf structure describing memory region. flags = DMA specific flags. waitfp = Address of waiting function, must be NULL. arg = Address of argument to pass to the wait function, must be NULL. lim = Pointer to the structure describing the DMA limit requirements. handlep = Pointer to an empty DMA handle structure. This structure will be filled if the call succeeds.
DESCRIPTION	Available in kernel mode only. bt_ddi_dma_buf_setup() is a macro that always uses unit 0. The rram_addr parameter of the SBS btp.conf file must match the information in the limits structure. If the dlim_addr_hi and dlim_addr_lo are both below 16M bytes, an A24 VMEbus address is calculated. The limits structure is not fully parsed and the VMEbus device's DMA ability must be compatible with the adapter's REM-RAM window capability. The DMA handle type, bt_ddi_dma_handle_t, is an opaque type that should never be inspected or modified.
DDI Routine	Mimics the ddi_dma_buf_setup() function.

7.2.4.2 Free A DMA Mapping

btp_ddi_dma_free()

FUNCTION	Releases resources previously consumed in a DMA mapping with btp_ddi_dma_buf_setup() call.
PROTOTYPE	int btp_ddi_ dma_free (u_int unit, bt_ddi_dma_handle_t handle);
ARGUMENT	unit = Hardware unit/instance number to use. handle = DMA handle structure from the corresponding call to btp_ddi_dma_buf_setup().
DESCRIPTION	Available in kernel mode only. The DMA handle type, bt_ddi_dma_handle_t, is an opaque type that should never be inspected or modified.
DDI Routine	Mimics the ddi_dma_free() function.

7.2.4.3 DMA Convert Handle to Cookie

btp_ddi_dma_htoc()

FUNCTION	Converts a DMA handle to a DMA cookie containing the physical VMEbus addresses for the DMA.
PROTOTYPE	int btp_ddi_ dma_htoc (u_int unit, ddi_dma_handle_t handle, off_t, bt_ddi_dma_cookie_t *cookiep);
ARGUMENT	unit = Hardware unit/instance number to use. handle = DMA handle structure from the corresponding call to btp_ddi_dma_buf_setup(). cookiep = Pointer to DMA cookie structure to be filled out if call succeeds.
DESCRIPTION	Available in kernel mode only. The DMA cookie type, bt_ddi_dma_cookie_t, is the same as ddi_dma_cookie_t except for the bt_ prefixed to all of the members. The DMA handle type, bt_ddi_dma_handle_t, is an opaque type that should never be inspected or modified.
DDI Routine	Mimics the ddi_dma_htoc() function.

7.3 Notes & Suggestions For Using The 946 Device Driver

7.3.1 Writing Device Drivers

Model 946 Support Software is designed to provide a kernel level interface similar to the Solaris DDI. This manual describes the kernel level routines implemented in Model 946. However, it does not go into great detail on these routines or kernel level programming in general. For detailed information on either of these two topics, please refer to the appropriate manual pages or the *Writing Devices Drivers* manual by Sun Microsystems.

7.3.2 Porting VMEbus Device Drivers

For most VMEbus drivers, actual C source code conversion is the easiest porting task. Model 946 supports the DDI functions listed in the table below. Code conversion consists of simply prefixing bt_ to each function, no parameters need to be changed. Not all of the ddi_dma_ functions are supported. If you are using a DMA function not listed, please contact SBS for specific porting instructions.

Solaris DDI Name	SBS Model 946 Name
ddi_pokec	bt_ddi_pokec
ddi_pokes	bt_ddi_pokes
ddi_pokel	bt_ddi_pokel
ddi_poked	bt_ddi_poked
ddi_poke8	bt_ddi_poke8
ddi_poke16	bt_ddi_poke16
ddi_poke32	bt_ddi_poke32
ddi_poke64	bt_ddi_poke64
ddi_peekc	bt_ddi_peekc
ddi_peeks	bt_ddi_peeks
ddi_peekl	bt_ddi_peekl
ddi_peekd	bt_ddi_peekd
ddi_peek8	bt_ddi_peek8
ddi_peek16	bt_ddi_peek16
ddi_peek32	bt_ddi_peek32
ddi_peek64	bt_ddi_peek64
ddi_map_regs	bt_ddi_map_regs
ddi_unmap_regs	bt_ddi_unmap_regs
ddi_add_intr	bt_ddi_add_intr
ddi_remove_intr	bt_ddi_remove_intr
ddi_dma_buf_setup	bt_ddi_dma_buf_setup
ddi_dma_free	bt_ddi_dma_free
ddi_dma_htoc	bt_ddi_dma_htoc

The following line must be added to the driver's main source code module. This line should be added before the "struct cb_ops" standard Solaris driver declaration.

```
static char _depends_on[] = "drv/btp";
```

The include file below must be added to all files that reference Model 946's routines or types.

```
#include <sys/btpvme.h>
```

After the source code has been converted, the driver configuration file must be modified. The ported driver will actually be a pseudo device to the PCI workstation. This is accomplished through the driver configuration file. The class property (for VMEbus driver) or the parent property (for drivers using a SBus Nexus) will need to be changed to "parent=pseudo". Also an instance property must be added to each device defined. Please see the "vme" and "pseudo" manual pages for detailed information. Below is a sample conversion for a given VMEbus device. No changes to the "reg" or "interrupts" properties need to be made.

```
Original
name="btv" class="vme";

Ported
name="btv" parent="pseudo" instance=0;
```

After the driver configuration file has been modified, device link creation must be examined. Since the driver is no longer a VMEbus driver, but now is a pseudo driver, the location of the device files created in the driver's attach routine will have changed. They will now be located under the /devices/pseudo directory but will have the same name as before. Most installation scripts create symbolic links to the device files in the /dev directory. The commands that create these links will have to be changed to take into account the new file locations under the /devices/pseudo directory. If you used the /etc/devlink.tab file to automatically create the links, no changes are necessary.

7.3.3 Limitations

Model 946 Support Software has the following limitations.

- bt_ddi_map_regs() – Slightly less then 32M bytes of VMEbus address space can be mapped at any given time.
- bt_ddi_dma_buf_setup() – Up to 16M bytes of host PCI memory can be allocated for DMAs at a time.
- bt_ddi_dma_buf_setup() – Does limited looking at the limits structure. User must manually setup the REM-RAM jumpers to match the limits structure and set the rem_ram_addr parameter of the btp.conf file to match jumpers. If dlim_addr_hi and dlim_addr_lo are below 16M bytes, A24 addressing is assumed and the upper 8 address bits are cleared during the bt_ddi_dma_htoc.
- If applications are going to use Model 946's read() or write() functions while drivers use the pointers acquired with bt_ddi_map_regs(), the ioctl() parameter THRESHOLD must be set to 17M bytes. This disables the DMA engine.

Chapter 8: Model 1003

8.0 Introduction

Chapter 8 describes installation of Model 1003 Support. It includes general information about the installation procedure, and gives a brief description of how to verify that the adapter is installed correctly and the device driver is loaded properly.

SBS Model 1003 Support Software for Intel™ x86-compatible PCI bus computers provides a device driver for Red Hat 6.0 (Kernel 2.2.5–15) Linux, Red Hat 7.0 (Kernel 2.2.16-22) Linux, or Red Hat 7.2 (kernel 2.4.7-10) and example applications to help application programmers with adapter and system configuration. Other 2.2.X kernels and distributions may work, but are not officially supported. Model 1003 currently supports the following SBS products:

- dataBLIZZARD communication interfaces.
- Model 618/620 PCI to VMEbus fiber-optic adapters.
- Model 617 PCI to VMEbus adapters.
- Model 616PCI to VMEbus (no DMA) adapters.

The software package provides a device driver, plus all tools, including memory mapping, to access dual-port and/or remote memory space from an application. This allows memory sharing between a PCI bus computer and another system.

Model 1003 also includes an Application Program Interface (API) that provides routines required to access all adapter resources. Remote memory and Dual Port RAM, if configured, can be shared between the two systems. Programmed interrupts can be exchanged. Devices on the remote system can be controlled from Linux and remote bus memory can be accessed.

Model 1003's device driver allows direct mapping to Dual Port RAM and/or remote bus memory without software overhead. In addition, the Mirror API provides routines to map VMEbus addresses to an application's memory. After setup, all access is handled by hardware; the memory responds to all VMEbus accesses.

The example applications included in the Support Software demonstrate features of the adapter hardware and software, and are useful tools for:

- Debugging.
- Uploading and downloading binary data.
- Receiving and counting error interrupts.
- Testing hardware.

Subroutines and example applications may be modified for your specific hardware configuration or application's requirements.

8.0.1 Components

Model 1003 consists of the following components:

- A device driver with installation script for Linux 2.2.X kernel.
- Mirror API Library to access the device.
- Example applications that demonstrate using the Mirror API.
- An example user Interrupt Service Routine (ISR).

8.0.2 System And Hardware Requirements

Linux: Intel x86-compatible computer with a PCI bus with Linux 2.2 kernel, such as Red Hat 6.0 or 7.0 or other distribution.

- Kernel source code for the currently running kernel.
- Kernel module support built into the kernel.

VMEbus: The remote reset jumper (SYS-5) on the VMEbus adapter card must be in place to use the remote VMEbus reset function.

The Address Modifier Register jumper (SYS-1) on the VMEbus adapter card must be removed.

Although Model 1003 is designed to work with a variety of Linux distributions, it has been tested only against a limited set. Currently, the software has been tested to work with Red Hat 6.0 Linux for Intel systems and Red Hat 6.1 Linux for Intel systems, and Red Hat 7.0 Linux for Intel systems.

8.1 Installation

8.1.1 Installation Notes

- Refer to the README file for revision history information.
- Files are stored in *tar* format.
- File or directory names in the form ./filespec relate to the directory in which the Support Software is installed. All files are located in a directory that is named for the software model and version number. For example, if version 2.0 of the software is installed in the /usr/local directory, the full path specification for the ./src directory is /usr/local/1003/v1.0/src.
- Chapter 3 lists the contents of the ./src directory and describes the function of each file.
- Before example programs can run successfully, the device driver must be installed, the PCI and remote adapter cards must be installed, the adapter cable connected, and the remote system powered on.

8.1.2 Installing Support Software

Before extracting files:

1. Login as root.
2. Create a directory for Support Software tar files. Use the following commands (# denotes system prompt):

   ```
   # cd /usr/local
   # mkdir SBS
   ```

3. Change directories to the one you just created. Use the following command:

   ```
   # cd SBS
   ```

4. Retrieve the archive file from either the CD-ROM or SBS's web site (www.sbs.com), and extract it using the following command.

   ```
   # tar –xf 85222001.tar
   ```

8.1.3 Installing Device Driver

➔ You should be logged in as root and in the usr/local/SBS directory.

1. Move to the SBS ./sys directory:

   # cd 1003/vx.x/sys
   (vx.x = version number)

   Check that the adapter is installed correctly; the following command should list all SBS (vendor_id = 108a) adapters (device_id = 1, 2, 3, or 10:40):

   cat /proc/pci | grep 108a

   bridge: PCI device 108a:0040 (Bit3 Computer Corp.) (rev 66),

2. Use the following command to install the device driver and related system files:

   # make install

   This command executes all other commands required to configure and install the device driver on your system.

➔ The PCI adapter must be installed for installation to continue.

3. Check that the adapter is installed correctly. The command

   cat /proc/pci

   should list a device named "btp". If the command fails to list any units, the driver did not load.

   If the driver fails to load, check that the PCI adapter cards are installed and firmly seated in the bus slots. Insufficient memory may cause the driver resource allocation to fail, causing the driver to fail to load.

   For Mandrake Linux, use the following commands to verify the correct loading:

   % cat /proc/modules | grep i btp

   (device btp should appear in the output)

   % cat /proc/pci | grep i 108a

   (the vendor ID should appear in the output)

4. Compile the dumpmem example program using the makefile provided in the ./src directory:

   #cd /usr/local/SBS/vX.X/src
   #make dumpmem

5. If Dual Port RAM is installed, enter the command ./dumpmem (located in the ./src directory) to verify that the driver and adapter are working correctly. The program should display the data in the first section of dual-port memory in hexadecimal and ASCII format.

   ➔ The xmit flag must be enabled (default is enabled).

   To adjust the configuration flag, see section 8.1.3.

6. To access remote bus memory, try using the dumpmem program (located in the ./src directory). Enter the following command:

   ./dumpmem -t BT_AXSRR -a <addr>

   Where <addr> is the location of memory on the VMEbus A32 address space.

   See section 4.1 for a list of logical devices' mnemonic names.

   ➔ The xmit flag must be enabled (default is enabled).

   To adjust the xmit flag, see section 8.1.3.

➔ If the dumpmem program fails, check the cable connection and jumper settings on the remote adapter card. Pay special attention to the remote adapter card's System (SYS) and Bias jumper blocks and anything related to bus arbitration on the remote system.

8.2 Configuring The Software

1. In most cases, you will not need to change the default settings. If no reconfiguration is required, go to step 2.

   Default configuration (set to enabled):

   - Transmitter status enabled, allowing the driver to access the remote bus.
   - All remote bus interrupters assumed to be ROAK.
   - Local Memory device is enabled.

   By default, the software is configured for use as a transmitter, to display all warning messages, and for a VMEbus REM RAM starting address jumpered to 0.

   By default, local memory (lm_enable) is enabled with a default size of 64K bytes.

   The configuration parameters are stored in the btp.conf file that the SBS script reads when it installs the Model 1003 driver with the insmod command.

The following parameters can be customized when loading the device driver:

PARAMETER	DESCRIPTION
bt_major	Major device number to request. By default, it is 0 (zero) allowing the kernel to choose the number.
trace	Device driver tracing level. Used to control which trace messages the driver displays. See section 5.3.1, BT_INFO_TRACE.
icbr_q_size	The number of ICBR entries that should be allocated for the queue. Once set, this value cannot be changed without unloading and reloading the driver.
xmit	Determines if this system is a transmitter. Defaults to enabled (non-zero).
roak	Determines if all interrupters are assumed to be ROAK (Release-On-Acknowledge) devices. Defaults to true (zero).
lm_enable	Determines if local memory device is enabled for any of the units. Default is disabled (zero).
lm_size[]	Array of local memory sizes. If the local memory device is enabled, this determines the size of local memory to allocate. If it is set to 0 (zero), local memory will be disabled for that unit only.
lm_raddr[]	Array of local memory remote addresses. These are the starting address used by the remote system to access the local memory device on the Linux system. Default is 0 (zero). This is only needed when the remote memory window is not aligned on a 16M byte address boundary.

The lm_raddr[] should be set to the value of the REM RAM LO jumper value on the remote card.

Refer to section 5.3.1 for details on changing the trace level, although under normal operation there is no need to change the trace level from its default value.

➔ When enabled, local memory access uses PCI system resources and may affect local system performance.

➔ Operating the device driver with the trace level above the default, BT_TRC_WARN, severely degrades driver performance.

2. After adding or removing SBS PCI adapter cards, the mkbtp script in the sys directory should be rerun to reconfigure the driver (see section 8.2, step 3).

3. Rebuild and re-install the device driver if you made any changes in steps 1 - 3. Use the following command:

```
# make install
```

The make install command executes all commands required to configure and install the device driver on your system.

➔ Make sure you are in the correct directory before executing make install. For example, if you loaded software version 2.0 in /usr/local/SBS, your working directory should be /usr/local/SBS/1003/v2.0/sys.

8.3 Loading The Driver

The mkbtp script can be used to load the device driver and create the /dev/btp* device nodes. The installation procedure automatically calls this script after copying the device driver and configuration file to /lib/modules.

Add the mkbtp script to your boot sequence to have the driver loaded each time the system is booted.

8.4 Compiling Example Programs

➔ Only the source code to the example programs is distributed with the Support Software.

To compile the software:

Change directories to the ./src directory containing the example programs, then compile the example programs. Use the following commands:

```
# cd /usr/local/SBS/1003/vx.x/src
```

(vx.x = version number)

```
# make all
```

To recompile a specific program you may have changed, use the command:

```
# make filename
```

8.5 Removing The SBS Support Software

In certain instances you may need to remove the Model 1003 Support Software from a system; for example, to install the software on a different system in compliance with the software license.

To remove the SBS Support Software from a system, please follow the instructions below.

1. Remove all SBS adapters from the system and reboot.

2. Change directories to the ./sys directory. Use the following command.

   ```
   #cd /usr/local/SBS/1003/vx.x/sys
   ```

3. Remove the device driver and configuration information from the system. Use the command below. You must be logged in as root.

   ```
   #make uninstall
   ```

4. Change directories to the top level directory in which the software was originally installed. Use one of the two commands below.

   ```
   #cd ../../..
   ```

   or

   ```
   #cd /usr/local/SBS
   ```

5. Remove all versions of the SBS software. Use the following command.

   ```
   #rm -rf 1003
   ```

8.6 Detailed Interrupt Handling

The first time the function bt_icbr_install() is called, the library spawns a thread within the task. The thread then blocks waiting for an interrupt from the driver. When a hardware interrupt occurs, Linux calls the device driver's interrupt handler. The device driver (possibly with the help of one or more User ISRs) acknowledges the interrupt, and then wakes up the relevant threads waiting for interrupts. The thread, after determining that the error is relevant, calls the actual ICBR.

A single queue is used for all ICBR registrations. If interrupts are occurring faster than an ICBR is handling them, that ICBR will receive a queue overflow.

ICBRs run in a separate thread than the main program; consequently, the ICBR interact with the main program without errors and on multiprocessor systems, the ICBR and main program can run simultaneously. We recommend that the programmer be familiar with the POSIX thread routines pthread_mutex_enter() and pthread_mutex_exit(), ant that these routines (or ones with similar purposes) be used to synchronize access to communal resources. Also, because the ICBR runs in a separate thread, it can call any function.

Any ICBR may receive queue overflow interrupts. ICBRs should be written to handle these calls.

8.7 usrisr Example User ISR

The usrisr is the Example User ISR.

8.8 Programming Considerations

This section contains several related topics on writing and porting applications for the Model 1003 driver.

8.8.1 Building Applications With The Mirror API

➔ The API can be used to build Win32 applications for Linux. It cannot be used to build 16-bit applications.

➔ The API is implemented as a library. To build your application, you must link it with the btp library. With the GNU compilers and binary utilities, this is done by specifying –lbtp on the command line.

➔ In addition, the preprocessor symbol BT1003 needs to be defined before including btapi.h either by having

```
# define BT1003
```

within the source code, or having

```
-DBT1003
```

on the command line.

8.8.2 Porting Applications

8.8.2.1 Porting Applications From UNIX Direct Device Interface

Somewhat more work is required to port applications from the old UNIX interface to the Mirror API on Linux. The Mirror API provides the function bt_ctrl() that on UNIX is an interface to the ioctl() call. Note: ioctl() cannot be called with a bt_desc_t.

In addition, many of the comments in section 8.8.2.2.1 – 8.8.2.2.3 are applicable.

To convert a program from the direct driver interface to the Mirror API:

1. Change the program to use bt_gen_name(), and bt_str2dev() routines to generate the device names. Include the btapi.h header file in addition to the btio.h header file.
2. Replace all the calls to BIOC_LOCK and BIOC_UNLOCK with calls to bt_lock() and bt_unlock().
3. Change the open() and close() routines to use bt_open() and bt_close(). Change the program to use a bt_desc_t to identify the device instead of an integer.
4. Change the mmap() and munmap() routines to use bt_mmap() and bt_unmmap().
5. Rewrite any code that used signal handlers for interrupt notification to use ICBRs. This should simplify the code and make the driver more efficient when notifying an application.
6. Convert all other ioctl() calls to use bt_ioctl(). This is only a temporary measure to allow you to get the program running.
7. Debug.
8. Change the bt_ioctl() calls to the equivalent Mirror API routines, after which, you will no longer need to include the btio.h header file.

8.8.2.2 Writing Portable Applications Using The Mirror API

This section deals with the issues arising from using the SBS API in a portable way.

8.8.2.2.1 Using NanoBus Or Model 1003 Specific Extensions

When writing code that will be ported and that uses the Mirror API, be aware of the generality of the functions used. All functions in the Mirror API fall into one of three categories: supported on all SBS products, supported on all SBS products of the same family, and supported only on one or a small set of SBS products.

For example, bt_open() is a function that is supported on all SBS Mirror API products. A program may assume that this function exists and works as described on any SBS API product.

An example of a function that is only supported on a given family of products is bt_tas(). All NanoBus-based products, including Model 1003, support this function. However, products based on other hardware designs, such as the NanoPort family of hardware, may not support this program. To help programs determine at compile time which family-based functions are available, every Mirror API product defines a preprocessor symbol that indicates the family. For example, all NanoBus-based products define the preprocessor symbol BT_NBUS_FAMILY. Programs can test for the existence of these functions:

```
# ifdef BT_NBUS_FAMILY
        bt_tas(btd, addr, prev_val_p);
# else /* BT_NBUS_FAMILY */
# error This program only supports NanoBus-based programs!
# endif /* BT_NBUS_FAMILY */
```

The function bt_gettrace() function is not supported on Model 1003; however, to achieve similar functionality for code specific to Model 1003, test for the preprocessor symbol BT1003.

8.8.2.2.2 BT_ENOSUP Error Return Value

A supported function may return BT_ENOSUP, a special error return value that indicates a requested service is not available. Common reasons this may occur are:

- Using the function bt_ctrl() to access an unsupported ioctl() call. No ioctl() calls are supported by the Model 1003; therefore, all calls to bt_ctrl() will return BT_ENOSUP.
- Attempting to open an unsupported device. For example, the Node I/O device that is a legal device in the NanoBus family but is not supported on the Model 1003. Attempting to open this device will cause bt_open() to return BT_ENOSUP.
- Attempting to use bt_bind() on a product that does not currently support it (Model 1003 does not support bt_bind()).

8.8.2.3 ICBR Context Restrictions

ICBRs give implementations flexibility. Some products, including most UNIX implementations, use signals for interrupts. Some like the Model 1003 use events, monitored by separate threads. Some even call the ICBRs during interrupt context. Because of the wide range of contexts the ICBR may be called in, strict limitations are placed on what can be done within an ICBR. Only the functions bt_chkerr(), bt_clrerr(), and bt_strerror() are guaranteed to be callable from ICBR context. No other functions are guaranteed.

8.8.3 Extending Or Modifying The Example Applications

8.8.3.1 Modifying bt_icbr Code Structure

There are three ways to extend bt_icbr: allow it to receive other types of interrupts, have it do something other than simply print a message when an interrupt occurs, and improve the mechanism by which it sleeps waiting for interrupts.

To receive interrupt types other than error interrupts, change the arguments to the call bt_icbr_install(). Only error interrupts are supported on all Mirror API products. Other interrupt types such as IACK interrupts and programmed interrupts are NanoBus-specific. See section 4.5 for more information. The switch statement in main() that determines how to respond to the interrupt to properly handle the new type of interrupt will also need to be modified.

The program structure is slightly odd. It is limited in what it is guaranteed to do in an ICBR. Consequently, the bt_icbr only puts the information into a FIFO queue that the main program reads data from and then acts upon the data. The functions queue_insert() and queue_remove() are used to maintain the queue.

There is no way in ISO Standard C to poll standard in; even the function sleep() is not part of the ISO standard. To maintain the portability of the program, the main function uses getchar() to sleep. Every time input is read, it polls the FIFO queue for new interrupts. Programs with less stringent portability requirements may use sleep(), select(), or similar functions. Programs that only need to run on Linux may assume the ICBR is run in a separate thread and do all processing in the ICBR.

Chapter 9: Model 993

9.0 Introduction

Chapter 9 describes installation of Model 993 Support. It includes general information about the installation procedure, and gives a brief description of how to verify that the adapter is installed correctly and the device driver is loaded properly.

Model 993 Support Software for VxWorks provides a loadable device driver, a library implementing the SBS Mirror API™, and example applications to help applications programmers with hardware and system configuration. Model 993 currently supports:

- All dataBLIZZARDs.
- All 7X2 CompactPCI/PCI adapters.
- All 7X3 CompactPCI/PCI adapters (no loopback diagnostics).
- Model 616 that connects a PCI computer to an A32 VMEbus system.
- Model 617 with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 618 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 620 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection (no loopback diagnostics).
- Model 628 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection.
- Model 630 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection (no loopback diagnostics).

9.0.1 System And Hardware Requirements

- Tornado™ 2.0 / VxWorks® 5.4 with the pc486 Board Support Package (BSP) for a Intel/Cyrix/AMD (80486 or greater) PCI system.
- A VMEbus or PCI/CompactPCI remote chassis.
- SBS adapter, plus appropriate cable.

9.1 Installation

9.1.1 Installation Notes

- Refer to the README file for revision history information.
- Files are stored in *tar* format.
- File or directory names in the form ./filespec relate to the directory in which the Support Software is installed. All files are located in a directory that is named for the software model and version number. For example, if version 1.0 of the software is installed in the /usr/local directory, the full path specification for the ./src directory is /usr/local/993/v1.0/src.

9.1.2 Installing Support Software

Before extracting files:

1. Make sure the Tornado environment and your BSP are already installed on the host system.

2. Login on the host system to an account that allows access and modification to the directories where the Tornado environment is installed.

3. If you are using Microsoft Windows as a host, open a Command Prompt window and execute the TorVars command to initialize the Tornado environment. This must be done to use the tar command provided with the Tornado environment.

4. Create a directory for Support Software files. Use the following commands (# denotes system prompt):

```
# cd /usr/local
# mkdir SBS
```

5. Change directories to make the directory you just created the default directory. Use the following command:

```
# cd SBS
```

6. Retrieve the archive file from either the CD-ROM or SBS's web site (www.sbs.com), and extract it using the following command.

```
# tar –xf 85221950.tar
```

7. Make sure the sub-directories and files listed on the following page were created.

SUB-DIRECTORY	File	CONTENTS
993/vx.x/sys/ *(vx.x = version number)*		
	btp.stub.c	Example of installing a remote bus interrupt handler
	readme	Text file that contains release notes for the 993 driver
993/vx.x/src/		
	bt_icbr.c	Example program to test the receiving of error interrupts
	bt_info.c	Example program to get and set an INFO parameter
	bt_main.c	Routine to make a single main program in VxWorks to set up command line arguments
	bt_sendi.c	Example program to send a programmed interrupt to the remote bus
	bt_xyint.c	Example program for interrupt notification
	bt_xypol.h	Defines data structures and constants used by bt_xyint.c
	datachk.c	Example program to perform a data pattern transfer and verify the data
	dumpmem.c	Example program to memory map remote memory
	readmem.c	Example program to read remote memory
	bt_bind.c	Binds a buffer to the remote bus, waits for user input, and then prints the first 256 bytes of the bound buffer.
	bt_cat.c	Example program that allows reading from and writing to the remote bus from standard in/out.
	bt_revs	Example program that prints the software driver version and the hardware firmware version.
993/vx.x/include/		
	btapi.h	Header file for the SBS API
	btdef.h	Header file for shared definitions
	btio.h	Header file used by all drivers
	btngpci.h	Header file for dataBLIZZARD products
	btpapi.h	Header file for PCI specific products
	btpdef.h	Header file for specific adapters
	btpio.h	Header file for specific drivers
	bt_bsp_unique.h	Header file for BSP unique definitions

(Table continued on next page.)

(Table continued from previous page.)

SUB-DIRECTORY	File	CONTENTS
993/vx.x/object	btppentiumdd.obj	SBS' 993 device driver for the PC Pentium CPU
	btpmcp750dd.obj	SBS' 993 device driver for the mcp750 CPU
	btpk2dd.obj	SBS' 993 device driver for the k2 CPU
	btprl4dd.obj	SBS' 993 device driver for the RL4 CPU
	pentiumsrc.out	Example programs for the pcPentium CPU
	mcp750src.out	Example programs for the mcp750 CPU
	k2src.out	Example programs for the k2 CPU
	rl4src.out	Example programs for the RL4 CPU
	btpk2dd.a	The SBS 993 device driver for the K2 CPU
	btpmcp750dd.a	The SBS 993 device driver for the mcp750
	btppentiumdd.a	The SBS 993 device driver archive for the pcPentium CPU
	btprl4dd.a	The SBS 993 device driver archive for the RL4 CPU
	lib993k2.a	The SBS 993 device driver archive library for the K2 CPU
	lib993mcp750.a	The SBS 993 device driver archive library for the mcp750 CPU
	lib993pentium.a	The SBS 993 device driver archive library for the pcPentium CPU
	lib993rl4.a	The SBS 993 device driver archive library for the RL4 CPU
993/vx.x/porting	btpppc604dd.obj	CPU only portion of SBS' 993 device driver for powerPC 604 CPUs
	bt_bsp_unique.c	BSP only portion of SBS' 993 device driver
	btpx86dd.obj	CPU only portion of SBS' 993 device driver for pcPentium CPUs
	lib993ppc403.a	Archival library that provides the Mirror API interface for the PPC403 driver
	lib993ppc604.a	Archival library that provides the Mirror API interface for the PPC604 driver
	lib993x86.a	Archival library that provides the Mirror API interface for the Pentium 993 driver
	btppc403dd.ojb	CPU only portion of the SBS 993 device driver for PowerPC 403 CPUs

9.1.3 Initializing The Adapter Card In VxWorks

To use the device driver and software, the VxWorks system configuration must be adjusted. These files are stored in the directory for your specific BSP (pc486) under the $(WIND_BASE)/target/config directory.

9.1.4 Configuring VxWorks Memory Space

➔ The sysLib.c file must be modified if you are using a BSP that does not support auto PCI configuration, for example, the pcPentium BSP. For mcp750, k2 and other BSPs that support auto PCI configuration, this step is not required. Including the INCLUDE_SHOW_ROUTINE and running the pciDeviceShow() and PCIHeaderShow() commands, however, is very useful.

The NanoBus adapter card's memory space must be located in a non-cacheable memory area. This area is created in the data structure sysPhysMemDesc[] defined in the sysLib.c BSP file. The sysPhysMemDesc[] is an array of structures that define the physical memory in the system, including the physical address, virtual address, initial state of the memory, and a mask defining which state bits in the state value are to be set.

Most BSPs include an example of how to configure VxWorks memory space. The entry from the pc486 BSP looks like this:

```
PHYS_MEM_DESC sysPhysMemDesc [] =
   {
   /* adrs and length parameters must be page-aligned (multiples of 0x1000) */

   /* lower memory */
   {
   (void *) LOCAL_MEM_LOCAL_ADRS,
   (void *) LOCAL_MEM_LOCAL_ADRS,
   0xa0000,
   VM_STATE_MASK_FOR_ALL,
   VM_STATE.FOR_MEM_OS
   },

   /* video ram, etc */
   {
   (void *) 0xa0000,
   (void *) 0xa0000,
   0x60000,
   VM_STATE_MASK_FOR_ALL,
   VM_STATE_FOR_.IO
   },

   /* upper memory */
   {
   (void *) 0x100000,
   (void *) 0x100000,
   LOCAL_MEM_SIZE - 0x180000,          /* it is changed in sysMemTop() */
   VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
VM_STATE_MASK_FOR_ALL,
   VM_STATE_VALID    | VM_STATE_WRITABLE    | VM_STATE_CACHEABLE, VM_STATE_FOR_MEM_APPLICATION
   },
```

```
PHY_MEM_DESC sysPhysMemDesc [] =
   {
   /* adrs and length parameters must be page-aligned (multiples of 4KB/4MB) */

#if (VM_PAGE_SIZE == PAGE_SIZE_4KB)
   /* lower memory */
   {
   (void *) LOCAL_MEM_LOCAL_ADRS,
   (void *) LOCAL_MEM_LOCAL_ADRS,
   0xa0000,
   VM_STATE_MASK_FOR_ALL,
   VM_STATE_FOR_MEM_OS
   },

   /* video ram, etc */
   {
   (void *) 0xa0000,
   (void *) 0xa0000,
   0x60000,
   VM_STATE_MASK_FOR_ALL,
   VM_STATE_FOR_IO
   },

   /* upper memory for OS */
   {
   (void *) 0x100000,
   (void *) 0x100000,
   0x080000,
   VM_STATE_MASK_FOR_ALL,
   VM_STATE_FOR_MEM_OS
   },

   /* upper memory for Application */
   {
   (void *) 0x180000,
   (void *) 0x180000,
   LOCAL_MEM_SIZE – 0x180000,  /* it is changed in sysMemTop() */
   VM_STATE_MASK_FOR_ALL,
   VM_STATE_FOR_MEM_APPLICATION
   },
```

Assume that a VxWorks kernel with INCLUDE_PCI and INCLUDE_SHOW_ROUTINES defined has been made. Install the PCI adapter card into the system, power up and launch a windsh window to access the PCI system. Execute the following command in the window that displays information about each PCI device on PCI bus number 0 (use a different bus number if appropriate):

```
-> pciDeviceShow(0)
Scanning function 0 of each PCI device on bus 0
Using configuration mechanism 1

  bus         device      function    vendorID    deviceID    class
  0000 0000   0000 0007   0000 0000   0000 8086   0000 7110   0001 0600
  0000 0000   0000 000a   0000 0000   0000 9005   0000 001f   0000 0100
  0000 0000   0000 000e   0000 0000   0000 108a   0000 0040   0080 0600
  0000 0000   0000 000f   0000 0000   0000 1042   0000 3030   0000 0600
value = 0 = 0x0
->
```

The SBS connectivity products vendorID is 0x108a. Model 616 has a deviceID of 3; the Model 617 has a deviceID of 1; Model 618 has a deviceID of 0x10. Please make note of the device number (**0x0e** for this example), function number, and bus number of the dataBLIZZARD adapter. Now execute a pciHeaderShow() command using the bus number, device number, and function number that was previously determined:

```
-> pciDeviceShow(1)
Scanning function 0 of each PCI device on bus 1
Using configuration mechanism 1
bus         device      function    vendorID    deviceID    class
00000001    00000013    00000000    0000108a    00000040    00800600
value = 0 = 0x0
-> pciHeaderShow (1, 0x13, 0)
vendor ID =                     0x108a
device ID =                     0x0040
command register =              0x0017
status register =               0x0400
revision ID =                   0x41
class code =                    0x06
sub class code =                0x80
programming interface =         0x00
cache line =                    0x08
latency time =                  0xe0
header type =                   0x00
BIST =                          0x00
base address 0 =                0x00001801
base address 1 =                0xfa700000
base address 2 =                0xfa780000
base address 3 =                0xfc000000
base address 4 =                0x00000000
base address 5 =                0x00000000
cardBus CIS pointer =           0x00000000
sub system vendor ID =          0x0000
sub system ID =                 0x0000
expansion ROM base address =    0x00000000
interrupt line =                0x09
interrupt pin =                 0x01
min Grant =                     0x00
max Latency =                   0x00
value =             0 =         0x0
->
```

Edit the sysPhysMemDesc[] array in your BSP's $(WIND_BASE)/target/config/<BSP>/syslib.c file. Insert the following text before "#ifdef INCLUDE_PCI", substituting the base addresses displayed by the previous execution of pciHeaderShow() for the variables base_addr_1 (CSR), base_addr_2 (mapping registers) and base_addr_3 (memory windows):

```
#define base_addr_1 0xfa700000
#define base_addr_2 0xfa780000
#define base_addr_3 0xfc000000

{
(void *) base_addr_1,
(void *) base_addr_1,
0x1000,
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID      | VM_STATE_WRITABLE      | VM_STATE_CACHEABLE_NOT
},

{
(void *) base_addr_2,
(void *) base_addr_2,
0x80000, /* 0x10000 for non-dataBLIZZARD H/W */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID      | VM_STATE_WRITABLE      | VM_STATE_CACHEABLE_NOT
},

{
(void *) base_addr_3,
(void *) base_addr_3,
0x2000000,
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID      | VM_STATE_WRITABLE      | VM_STATE_CACHEABLE_NOT
},
```

9.1.5 Allocating PCI Memory

dataBLIZZARD may require more PCI memory than currently configured in VxWorks. The mcp750 uses a define to determine the size of the PCI memory space that is mapped in the sysPhysMemDesc[] array for PCI auto configuration.

- For mcp750 BSO v1.2/2 and less: Edit the file mv2600.h and change the define CPC_PCI_MEM_SIZE at 0x04000000.
- For mcp750 BSP v1.2/3 and higher: Edit the file config.h and change the define PCI_MSTR_MEMIO_SIZE to 0x04000000.

9.1.5.1 mcp750 J Fix

mcp750 version J will not function because of a running change made to the super I/O chip from PC87307 to PC97307. To use the cards, you will need to follow Wind River system SPR #67558 to resolve the problems in ns8730xSuperIo.h and ns8730xSuperIo.c.

9.1.5.2 Rebuilding VxWorks

After completing changes to the sysLib.c configuration file, the system must be rebuilt. We recommend rebuilding VxWorks and booting the system with this new configuration before attempting to load the device driver. Follow directions in the Tornado User's Guide for "Building a VxWorks System Image".

If this has already been done once, there should be an entry under the Tornado's "Projects" menu to make your BSP. Look under the Projects menu for the name of your BSP. A VxWorks Targets menu should be under the Projects menu. The Targets menu has separate menu items for each of the various binary formats you can create.

9.1.6 Installing The Library And Device Driver

After VxWorks is configured, install the library, header files, and device driver.

1. Change directories to the ./sys directory:

```
#cd /usr/local/SBS/993/vx.x/sys
```

2. Load the device driver for your CPU type:

```
-> cd "<installdir>/993/vx.y/objects""
-> ld <btppentiumdd.obj
     or
  ld <btpppc604dd.obj
```

9.1.6.1 Configuring The Device Driver

Two routines are used to configure the device driver: btpDrv() and btpDevCreate(). Both routines can be used at any time after exception handling is initialized.

The btpDrv() routine adds the device driver entry points to the system table.

Prototype:

```
void btpDrv(void);
```

The btpDevCreate() configures each physical unit and adds the device to the I/O system.

Prototype:

```
STATUS btpDevCreate(unsigned int unit, int isr_prio, int isr_stack, size_t lm_size);
```

Arguments are listed and described below.

ARGUMENT	DESCRIPTION
unit	The unit number between 0 and 15 inclusive.
isr_prio	The VxWorks task priority at which the Interrupt Service Routine (ISR) should run. The device driver uses a separate task to do all interrupt processing, only waking the ISR task is done at the hardware interrupt level. The ISR task name is in the format "t%s_isr", where %s is replaced by the device name.
isr_stack	Total stack size for the ISR task. Zero causes the software to use the default size. All normal methods provided by VxWorks can be used to track stack usage by the task.
lm_size	Size of the local memory device.

Example of the call to btpDevCreate():

```
status = btpDevCreate(0, 55, 0, 0x40000)
```

This would add a device named '/btp0' to the system, create a task with priority 55 and with a local memory device of 0x40000 bytes that would normally be pending.

The iosDevShow function can be used to display the device.

An example session that shows configuring the driver and one unit would have output similar to the following (—> denotes the WindShell prompt):

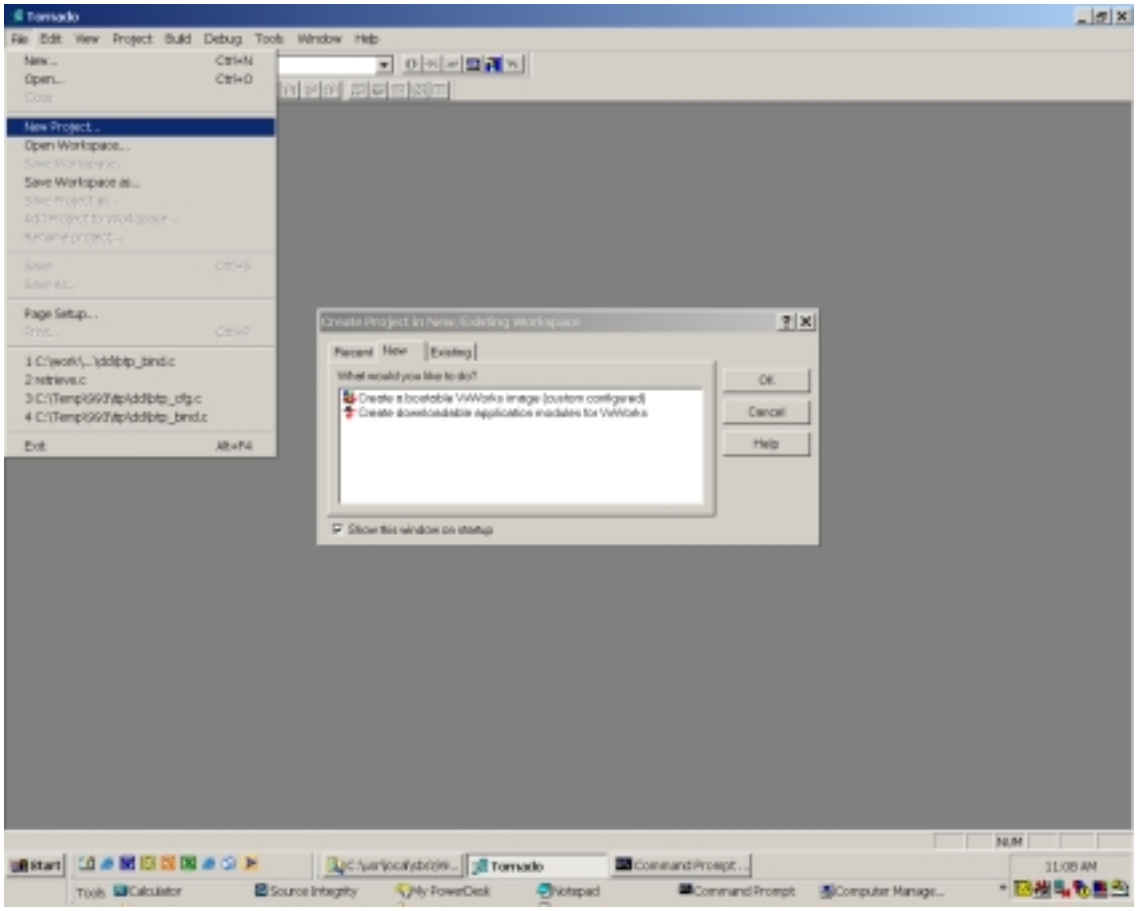
```
—>btpDrv()
value = 9 = 0x9
—>btpDevCreate (0, 55, 0, 0x4000)
value = 0 = 0x0

-> iosDevShow
drv name
  0 /null
  1 /tyCo/0
  1 /tyCo/1
  2 /pcConsole/0
  2 /pcConsole/1
  7 risky:
  8 /vio
  9 /btp0
value = 0 = 0x0
->
```

9.1.7 Compiling Example Applications

To compile the example applications:

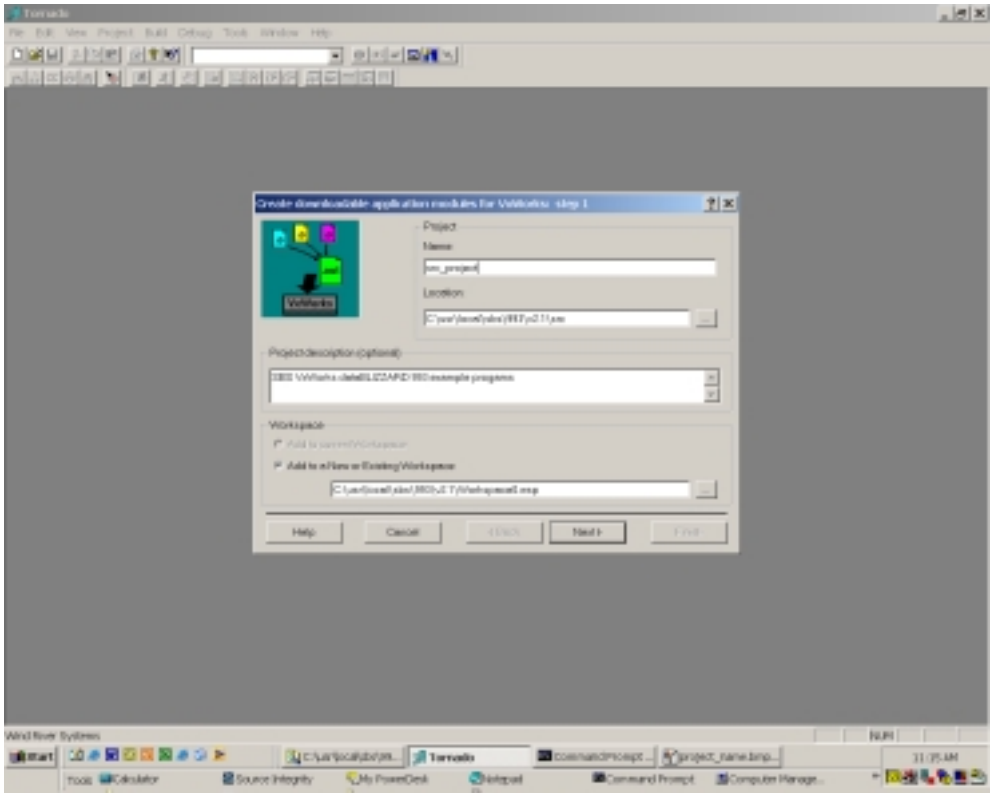
1. Create a project by selecting New Project from the Tornado pull down menu.



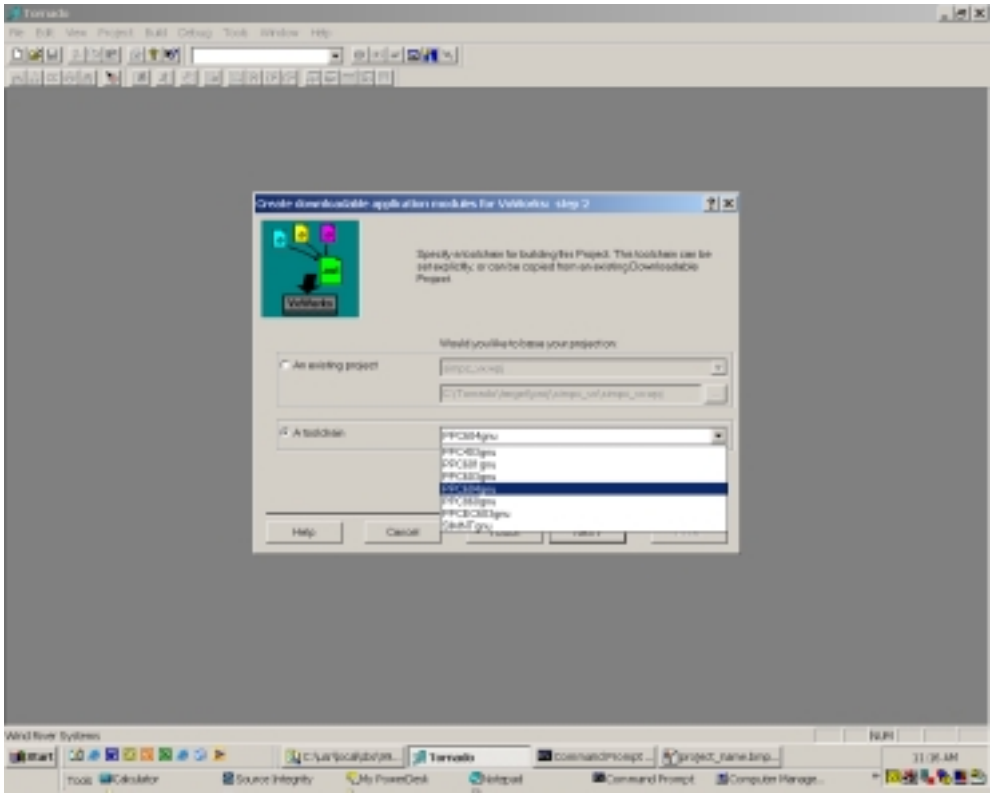
2. Select Create downloadable modules for VxWorks, click OK.

3. Name the project and define its location to be the directory to which the driver was extracted.

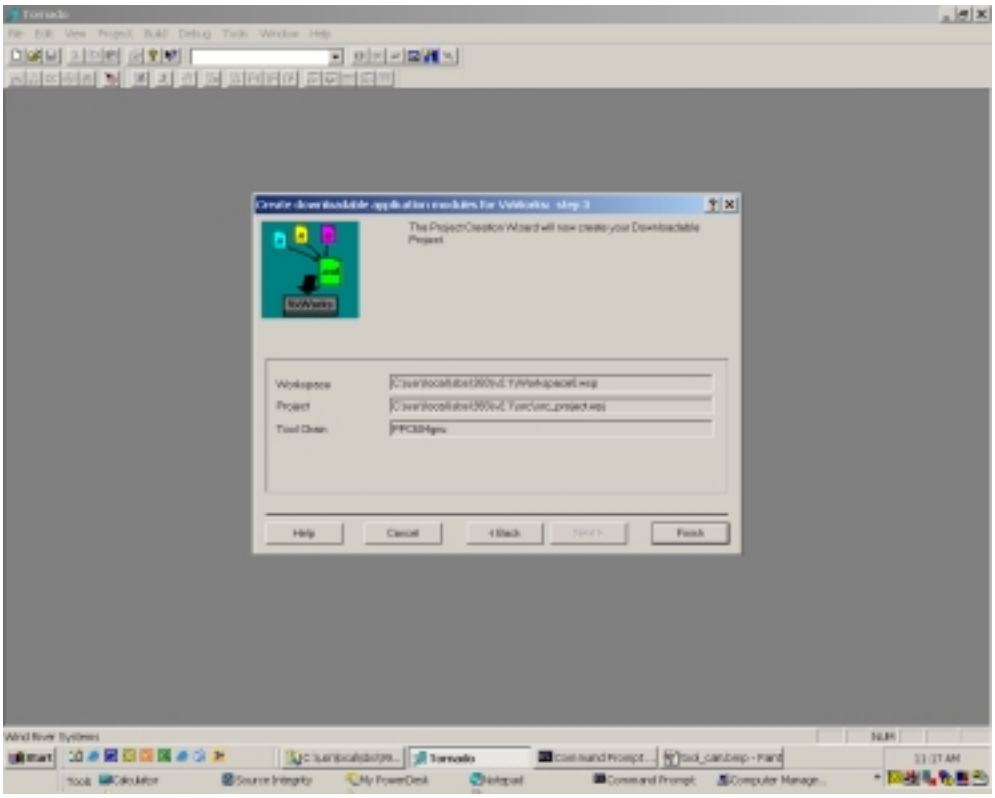
   Also, name the workspace file and its location.



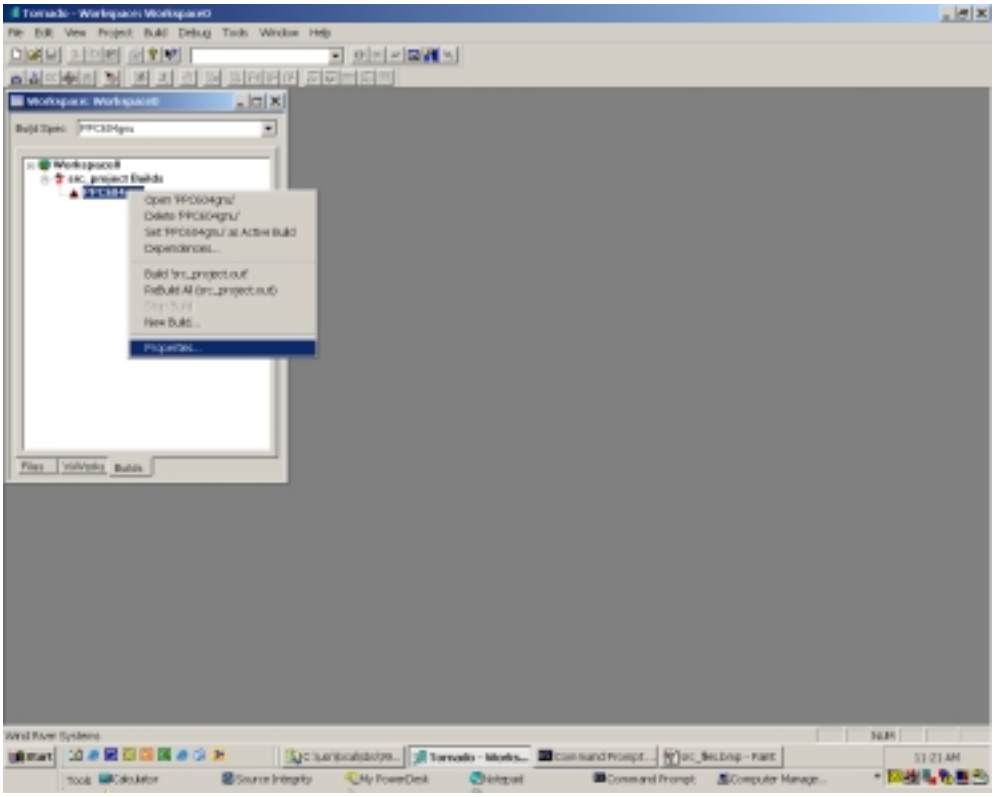
4. Select a toolchain based on the processor family you are porting to. Use PENTIUMgnu for CT7 or Pentium BSPs; and use PPC604gnu for k2 and mcp750 BSPs.



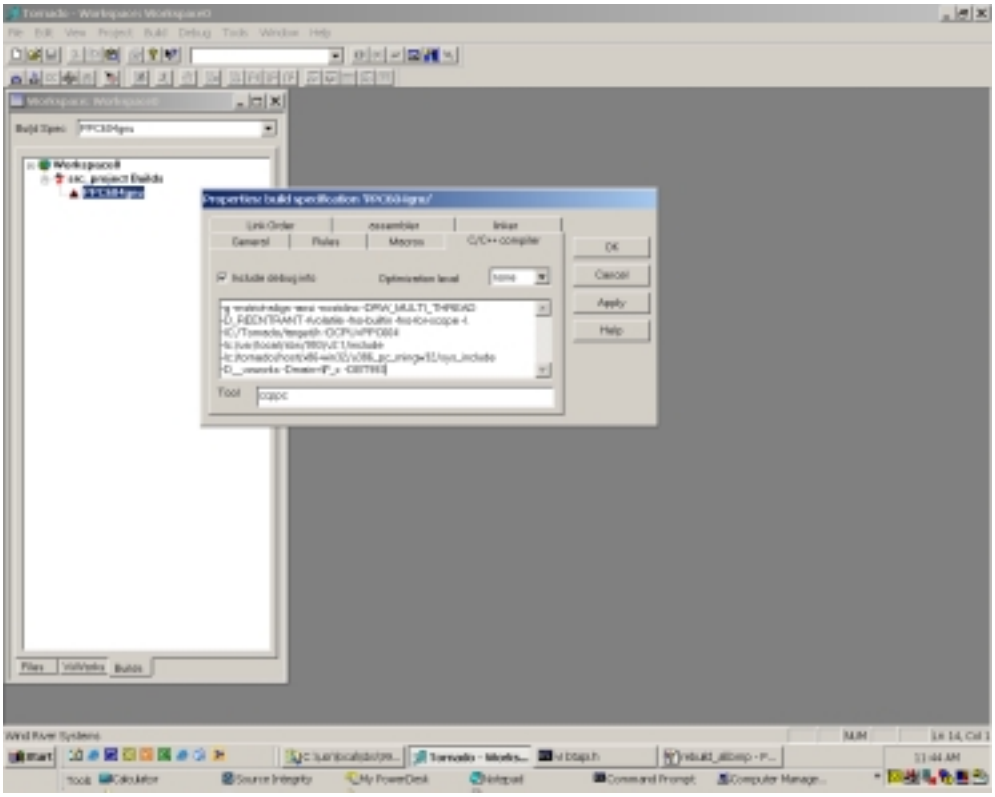
5. Select Finish to complete the project and workspace definitions.



6. Define the build environment and properties for the project by selecting the Build tab in the Workspace window and right click on the Toolchain. From the Toolchains pull down window double click on Properties.

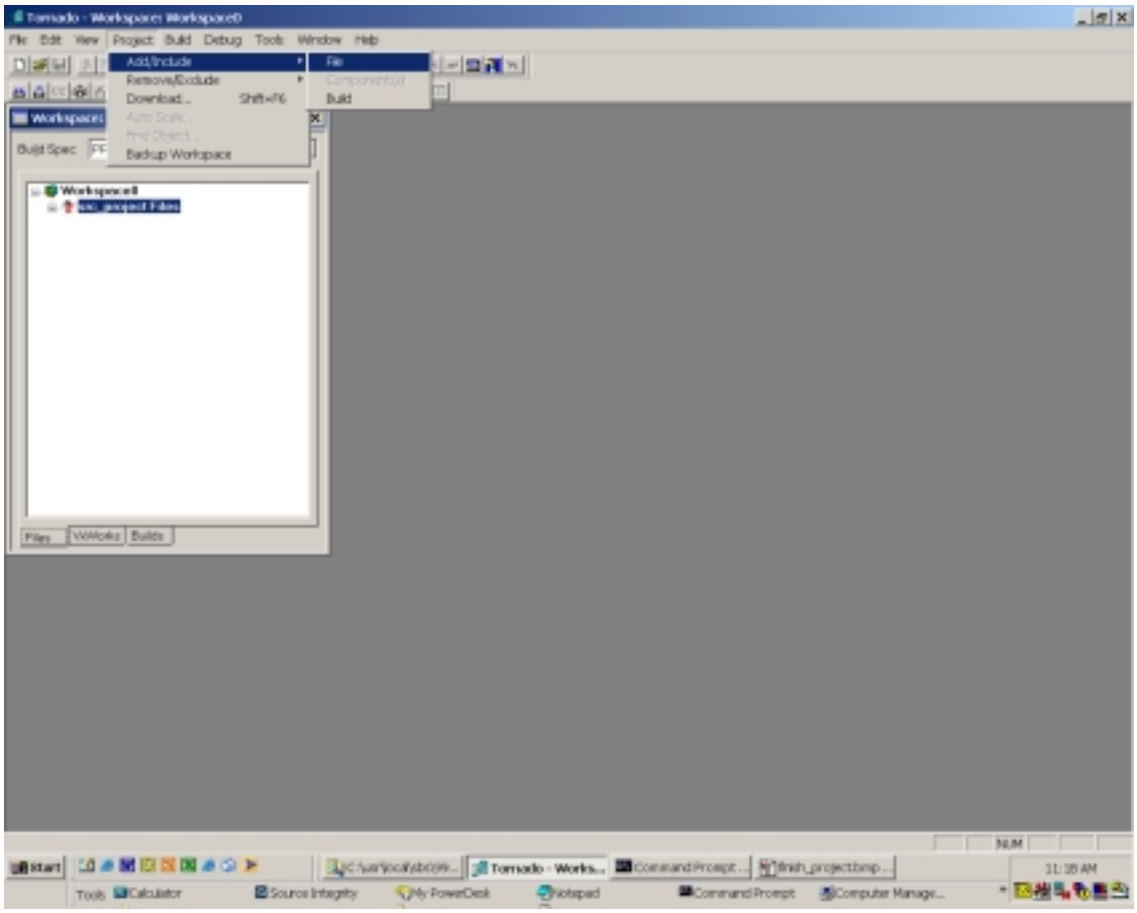


7. Select the C/C++ Compiler tab from the Properties window and add the following options:

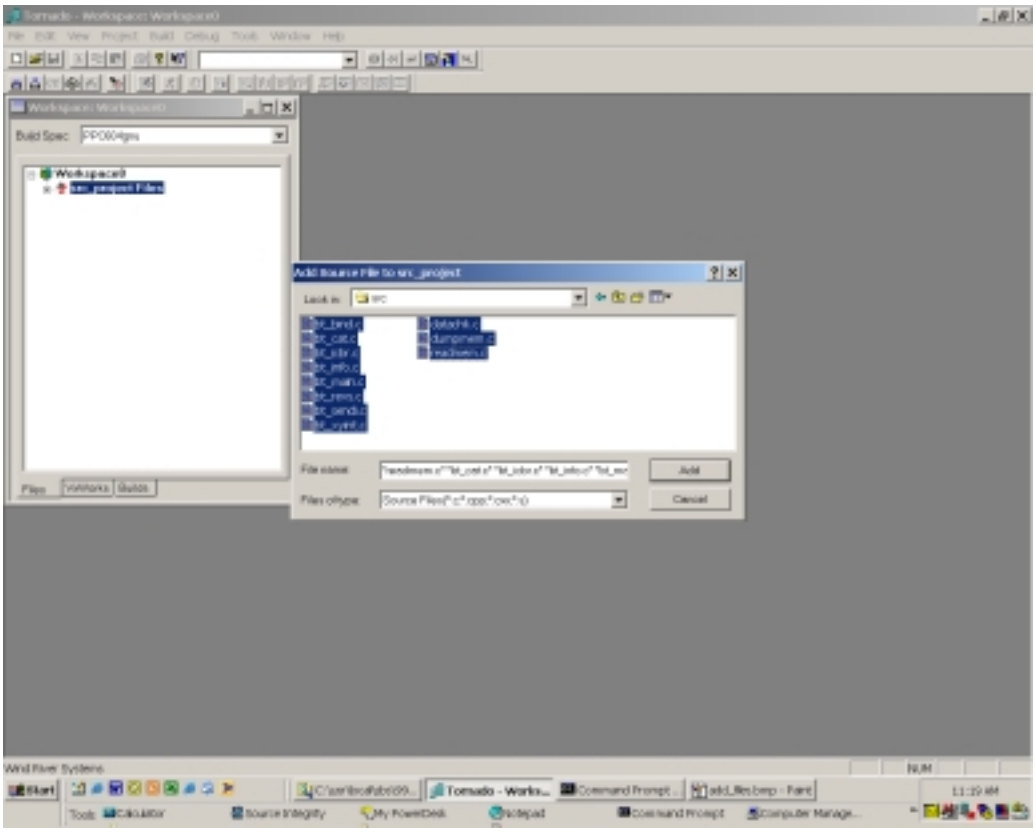


Driver	Option
All	-I c:/usr/local/SBS/993/vx.x/include
	-I c:/tornado/host/x86 –win32/u386_pc_mingw32/sys_include
	-D _ _vxworks
	-DBT993
	-Dmain = $*_x
mcp750	-DMCP750_BSP
	-I c:/tornado/target/config/mcp750
k2	-Dk2_BSP
	-I c:/tornado/target/config/powerk2
pcPentium	-DPCPENTIUM_BSP
	-I c:/tornado/target/config/pcPentium

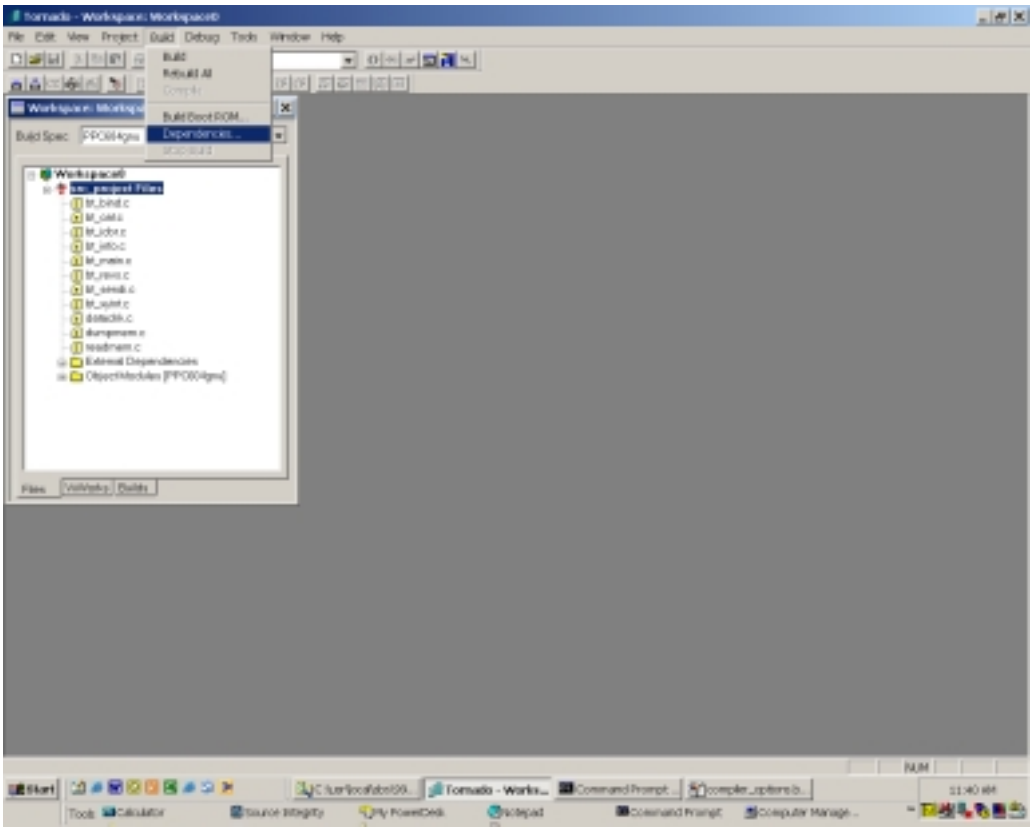
8. Add the src example applications to the project by selecting Add/include files from the Project menu.



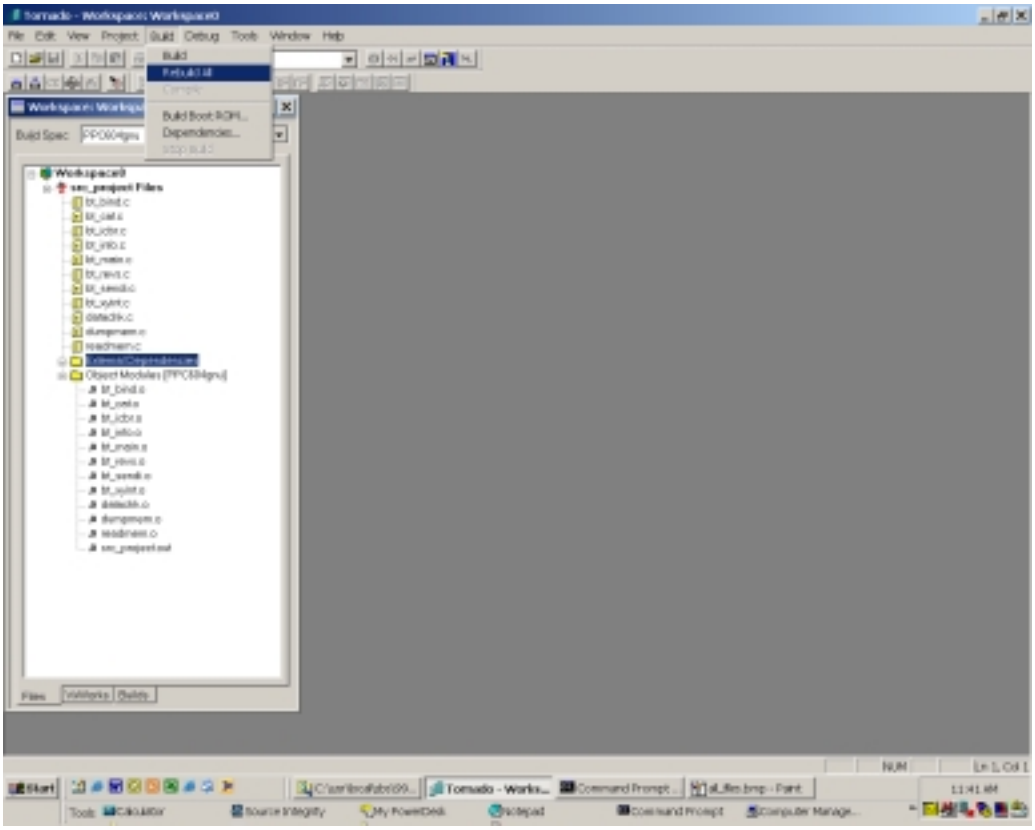
Select the example application .c files contained in the src directory.



9. Build the dependencies for the example application by selecting Dependencies from the Build menu.



10. Compile the example application by selecting Rebuild all from the Build menu.



9.1.8 Checking The Installation

After the device driver is loaded, example programs compiled, and the pcpentium_src.out file downloaded, you can use example applications to check the installation and that the software can correctly access the local hardware:

```
-> cd “c:/usr/local/SBS/993/vx.x/objects:
    pentium.src.out
or
   mcp750_src.out
```

To check that the driver is installed and is communicating with the remote adapter card, run the bt_info program:

```
-> bt_main (“bt_info –p REM_PN”)
```

The call should return the part number of the remote adapter card. If you get an error, check your cables and try this command:

```
-> bt_main (“bt_info –p TRACE”)
```

The call should return the current driver trace flags. If you get an error, the driver was unable to open the local adapter card. Reboot your system and reload the driver.

9.1.9 Running The Example Applications

The example applications can be run after initializing the adapter card. All example programs assume a traditional command line interface. Because VxWorks does not have this type of command line interface, a routine is needed to set up passing argc and argv arguments to the example applications.

The program bt_main passes these arguments to the example applications. It accepts a single string as an argument. The program parses the string and creates the argc and argv arguments before calling an example application. For example:

```
bt_main("datachk -t DP -l 0x8000")
```

will call datachk with argc = 5 and argv = {"datachk", "-t", "DP", "-l", "0x8000"}. These are the same values the program would expect on a system with command line processing.

9.2 Direct Access To The Device Driver

Section 9.2 describes how to directly access the SBS device driver instead of using the Mirror API library and documents use of lseek(), read(), write(), and a few select available ioctl() functions. If more than these functions are needed, use the Mirror API library. There is no significant performance advantage to directly accessing the device driver.

To directly access the device driver, you will need to use routines contained in the VxWorks ioLib library. You should already be familiar with these routines. In addition, you will need to include the btio.h header file that comes with the Model 993 Support Software.

9.2.1 Accessing The Correct Logical Device

The device driver uses the same concept of logical devices as the Mirror API. To access a specific logical device, call open() with the device name corresponding to that device. Section 4.1 explains logical devices and includes the device name used for each logical device.

For VxWorks, use the bt_gen_name() routine from the library to create the correct device name. Provided that the return value is not NULL, use this in the open() call.

After opening the device driver, use the lseek(), read(), and write() calls to transfer data between the system and that logical device. In addition, there are a limited number of ioctl() calls that can be called directly from an application.

9.2.2 read() And write() Functions

The read() and write() functions are the older method for transferring data from a device to an application. Developers of current software should use the bt_read and bt_write functions; read() and write() functions are only included for backwards compatibility with existing applications. These functions provide serialization of all requests and automatically update the current position within the device as data are transferred.

Using a combination of lseek(), read(), and write() functions, data can be positioned anywhere within the logical device address space. The read() and write() interface updates the current position so that subsequent calls to read() or write() are offset by the length of the last read() or write(). Pipes to the device driver can be maintained.

The read() and write() functions are affected in the same way by all device configuration controls that affect bt_read() and bt_write(). These include the controls for the DMA threshold, address modifier used, and data transfer size used. Use the bt_info example application to change these parameters.

The read() and write() functions return the number of bytes transferred. If an error prevents any data from being transferred, ERROR is returned. If the amount transferred is less than the amount requested, a partial transfer completed before the error occurred.

Three ioctl() functions provide additional information about the type of error encountered; see section 9.2.4.

9.2.3 lseek() Function

The lseek() function is used to position read() or write() operations to a specific Remote Bus Memory address. Here, lseek() is used differently than in a standard UNIX file; all lseek() references are based from a physical memory address rather than a file offset.

The address referenced when the device performs a lseek() of SEEK_SET to zero depends on the logical unit used.

LOGICAL DEVICE	ADDRESS
Remote Dual Port	Start of the Remote Dual Port Memory
Remote Bus I/O	Bus address 0, A16 address space
Remote Bus Memory	Bus address 0, A32 address space
Remote A24 space	Bus address 0, A24 address space

One drawback to using lseek() is that the offset parameter to the function is a signed integer. Fortunately, VxWorks does not look at the sign bit when doing a lseek(), allowing the device driver to treat it as an unsigned quantity.

Using lseek() to SEEK_SET uses the offset given as the address to reference. The offset parameter is treated as an unsigned quantity. This allows the full 4G bytes of A32 space on the remote bus to be accessed.

Using lseek() to SEEK_CUR adds the offset given to the current position (address) to determine the new address to reference. A positive value causes the device driver to reference a higher address. A negative address positions the device at a lower address. The device driver treats the result as an unsigned quantity.

Example:

```
{
        current_location = lseek(file, 10, SEEK_SET);
        /* Address 10 */
        current_location = lseek(file, 10, SEEK_CUR);
        /* Address 20 (10 + 10) */
        current_location = lseek(file, -5, SEEK_CUR);
        /* Address 15 (20-5) */
        current_location = lseek(file, -5, SEEK_SET);
        /* Address 0xfffffffb (-5 treated as an unsigned value) */
        current_location = lseek(file, -5, SEEKCUR);
        /* Address 0xfffffff6 (0xfffffffb-5) */
        current_location = lseek(file, 16, SEEK_CUR);
        /* Address 0x6 (overflowed the offset) */
}
```

Be aware that the device driver and operating system both ignore underflow and overflow when using lseek() with SEEK_CUR. This can result in the value ERROR being indistinguishable from the offset 0xffffffff, the last address in A32 space.

Using lseek() to SEEK_END is undefined for the Model 993 device driver.

9.2.4 Checking For And Handling Errors

Although a number of ioctl() functions are provided by the device driver, most are intended to only be used by the Mirror API library. The following ioctl()s are documented for direct device access:

ioctl()	FUNCTION
BIOC_INIT	Initializes the device driver. Equivalent to the bt_init() routine in the library.
BIOC_CHKERR	Checks if errors occurred on the NanoBus adapter. This includes detecting if the cable is disconnected or the remote bus is switched off. Equivalent to the bt_chkerr() routine in the library.
BIOC_CLRERR	Clears any errors on the interface. Equivalent to the bt_clrerr() routine in the library.

If more error checking and handling functions than these are needed, we strongly recommend using the Mirror API library. The library provides a portable interface between the application and the device driver.

9.2.4.1 Initializing The Adapter

BIOC_INIT

FUNCTION	Restores the local and remote adapter cards to a known (default) state. Causes the device driver to determine the part number of the remote adapter card.
ARGUMENT	bt_error_t
EQUIVALENT MIRROR API	bt_init()

Example:

```
bt_error_t retval;

if (ERROR == ioctl(file, BIOC_INIT, &retval)) {
      perror("BIOC_INIT failed");
      return FAILED;
}
if (BT_SUCCESS != retval) {
       /* Need to run makeStatTbl before this
          will work. */

      errnoSet(retval);
      perror("BIOC_INIT detected an error");
}
```

9.2.4.2 Check For Adapter Errors

BIOC_CHKERR

FUNCTION	Checks if any errors have occurred on the adapter since the last time they were cleared. An error during a read() or write() would be detected during the transfer and indicated at that time.
ARGUMENT	bt_error_t
EQUIVALENT MIRROR API	bt_chkerr()

Example:

```
bt_error_t retval;

if (ERROR == ioctl(file, BIOC_CHKERR, &retval)) {
      perror("BIOC_CHKERR failed");
      return FAILED;
}
if (BT_SUCCESS != retval) {
     if (retval == BT_ENOPWR) {
         printf("Please check that the cable is connected and"
                 " that the remote system is powered on.");

         return POWER_BAD;
     } else {
           /* Need to run makeStatTbl before
              this will work. */

           errnoSet(retval);
           perror("BIOC_INIT detected an error");
           return FAILED;
     }
}
```

9.2.4.3 Clear Error Status On The Adapter

BIOC_CLRERR

FUNCTION	Clears any accumulated errors on the adapter interface.
ARGUMENT	bt_error_t
EQUIVALENT MIRROR API	bt_clrerr()

Example:

```
bt_error_t retval;

if (ERROR == ioctl(file, BIOC_CLRERR, &retval)) {
    perror("BIOC_INIT failed");
    return FAILED;
}
if (BT_SUCCESS != retval) {
    /* Need to run makeStatTbl before this
       will work. */

    errnoSet(retval);
    perror("BIOC_INIT detected an error");
}
```

9.3 dataBLIZZARD Device Driver Porting

The dataBLIZZARD device driver can be ported to any PPC604, PPC403, or pcPentium VxWorks BSP. The drive driver is shipped with the CPU-only portion of the driver btpppc604dd.obj, btppc403dd.obj, and btppentiumdd.obj in the porting directory. The BSP portion of the device driver is delivered in source format in the file bt_bsp_unique.c, a file that must be customized for your specific BSP.

Functions that need to be customized for your BSP:

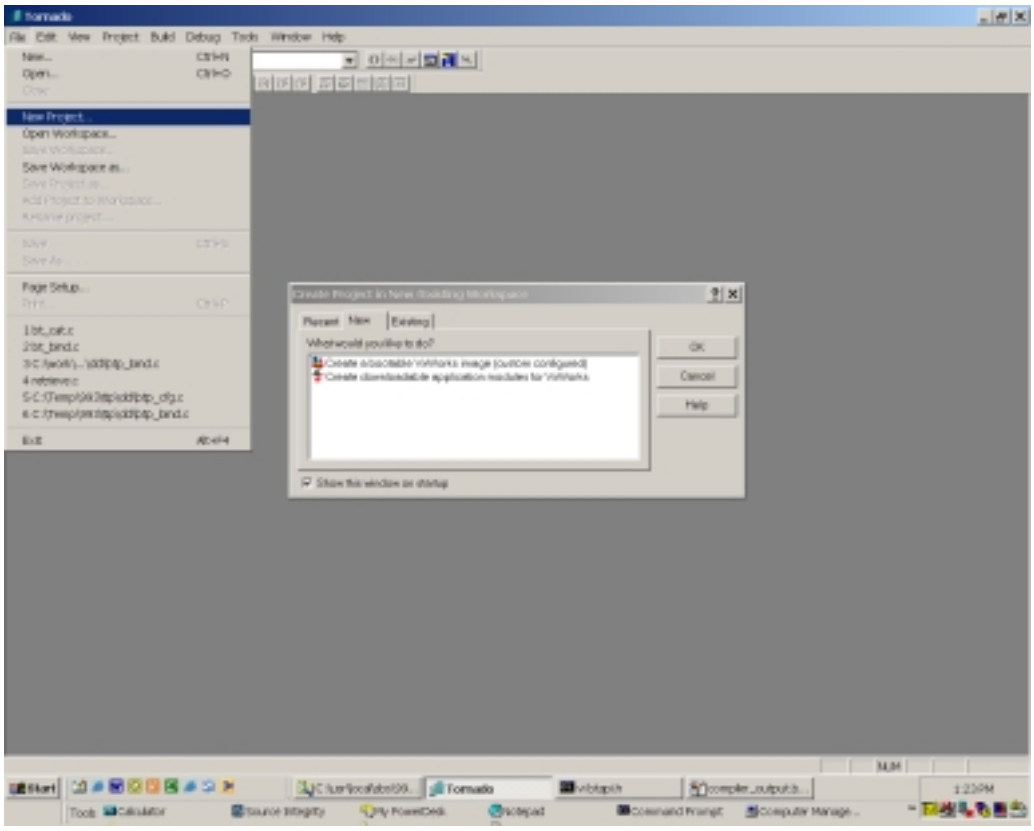
- bt_cpu2pci_advs() must be modified to convert an address seen from the CPU to the equivalent PCI address.
- bt_pci2cpu_advs() must be modified to convert an address seen from the PCI bus to a CPU address.
- bt_connect_irq() must be modified to connect a 'C' routine to a hardware interrupt.
- bt_enable_irq() must be modified to enable a hardware interrupt.
- bt_disable_irq() must be modified to disable a hardware interrupt.
- bt_get_model_name() and bt_get_bsp_rev() should be modified to return the model name of your BSP and the revision. Because these two routines are not required to support driver operation, not customizing them will not affect driver function.
- bt_get_sys_clk_rate() must be modified to return the system clock rate.

Details of how to modify the functions listed above and examples for the mcp750, pcPentium, CT7 and k2 are included in the source file bt_bsp_unique.c file.

9.4 Compiling vx_bsp_unique.c

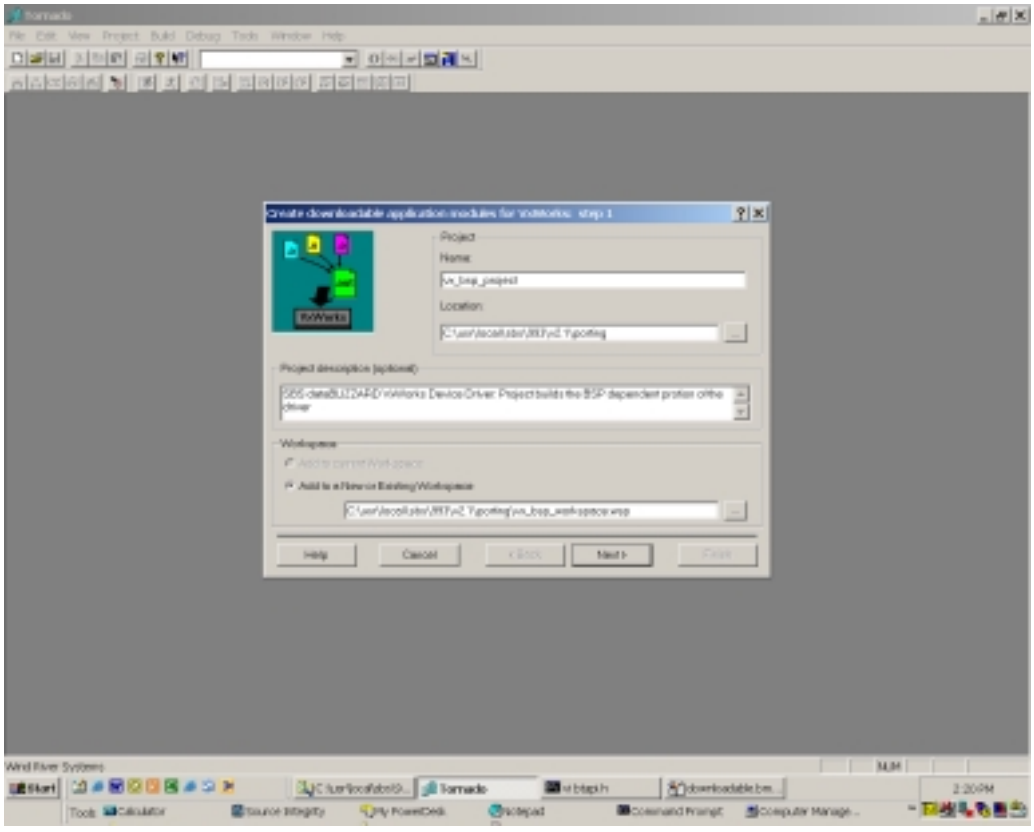
To compile the vx_bsp_unique.c:

1. Create a project by selecting New Project from the Tornado pull down menu.



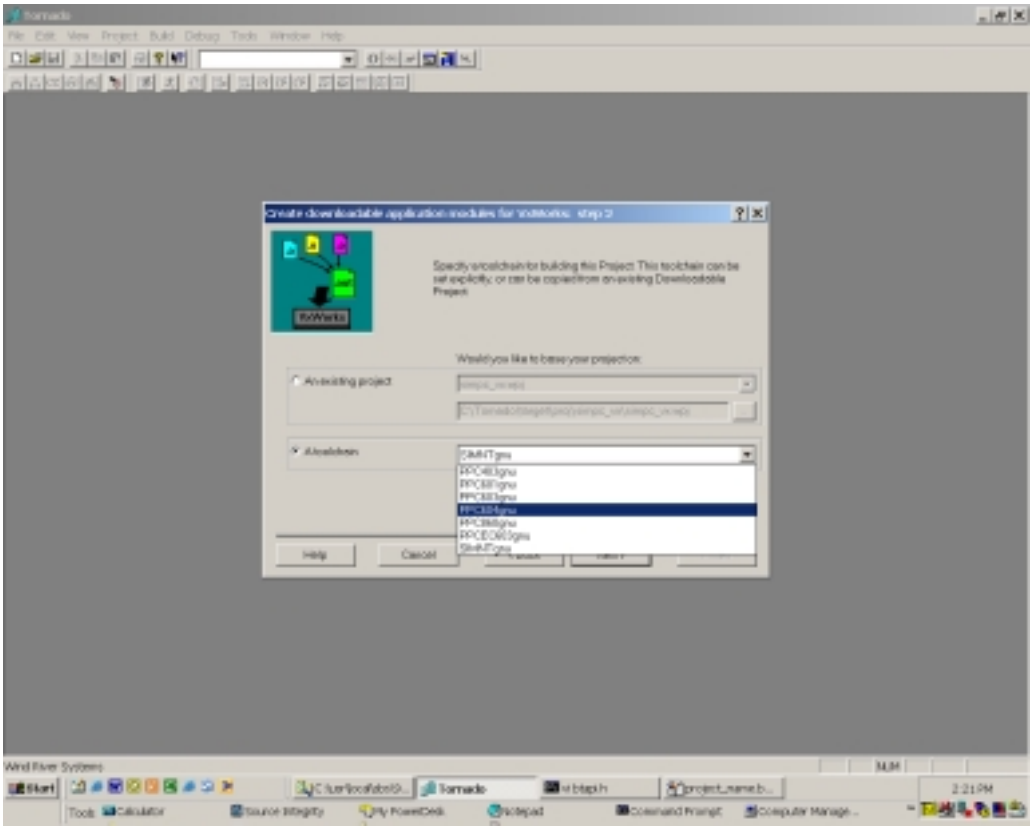
2. Select Create downloadable modules for VxWorks, click OK.

3. Name the Project and define the project location, and name the workspace.

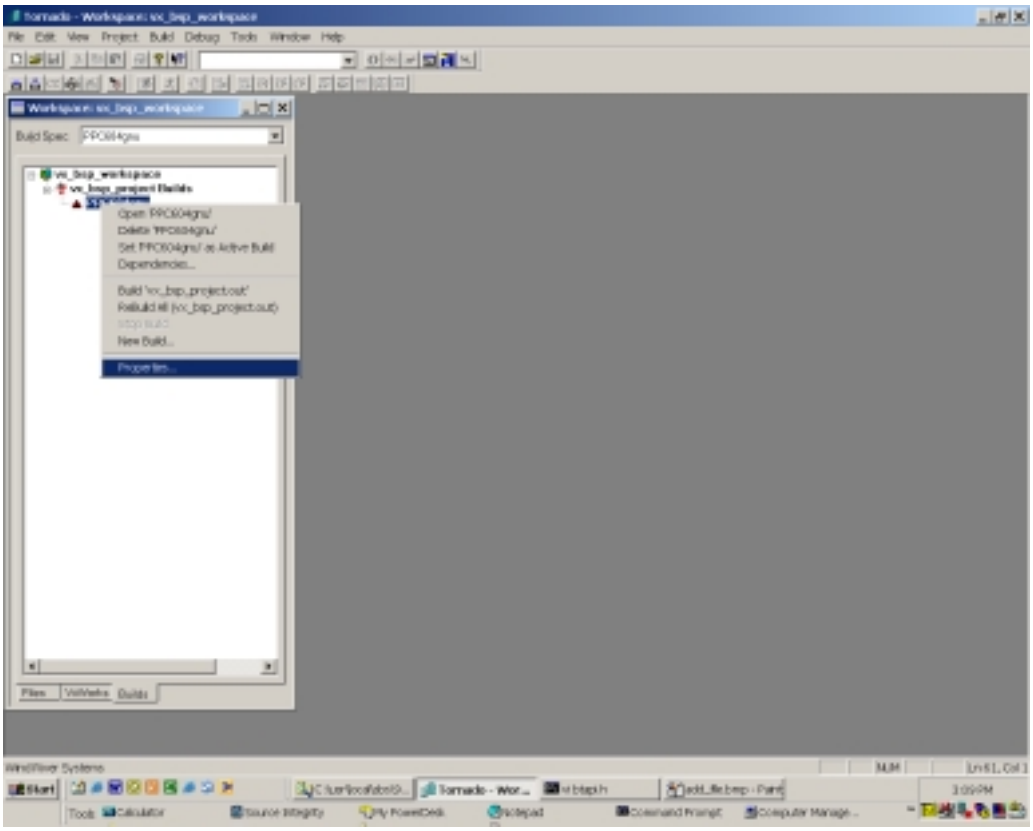


4. Select the toolchain based on the processor family you are porting to.

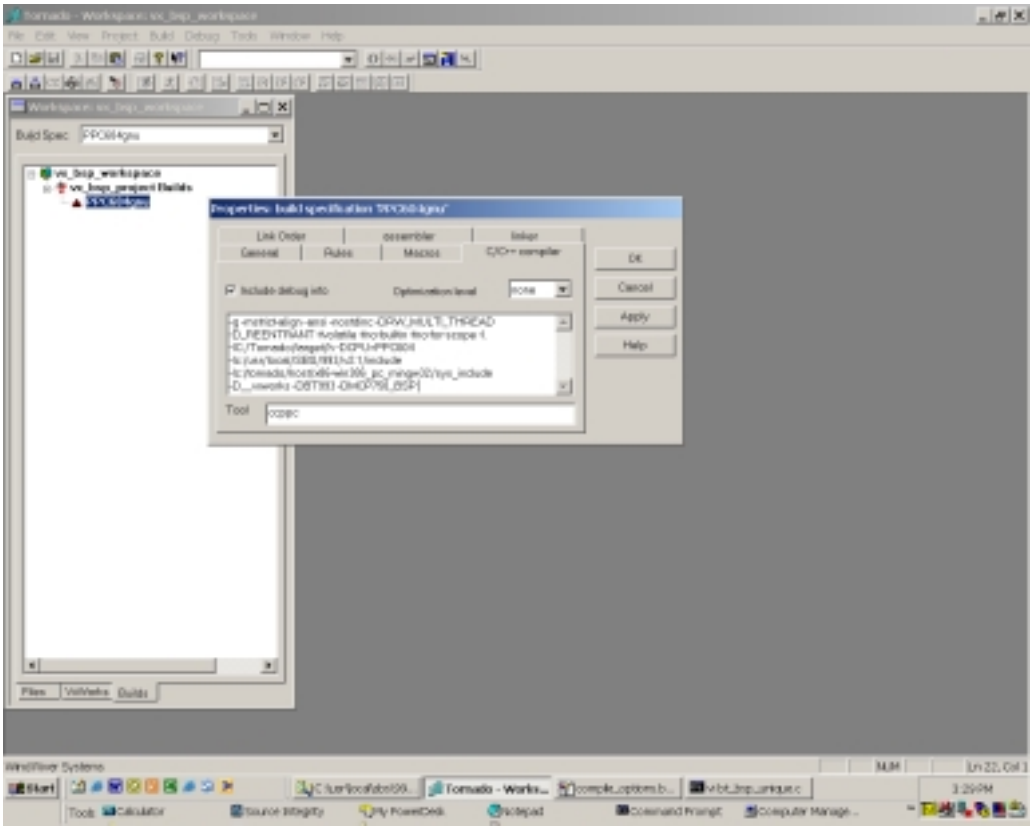
5. Select Finish to complete the project and workspace definitions.



6. Define the build environment and properties for the project by selecting the Build tab in the Workspace window and right click on the toolchain. From the Toolchains pulldown window, double click on Properties.



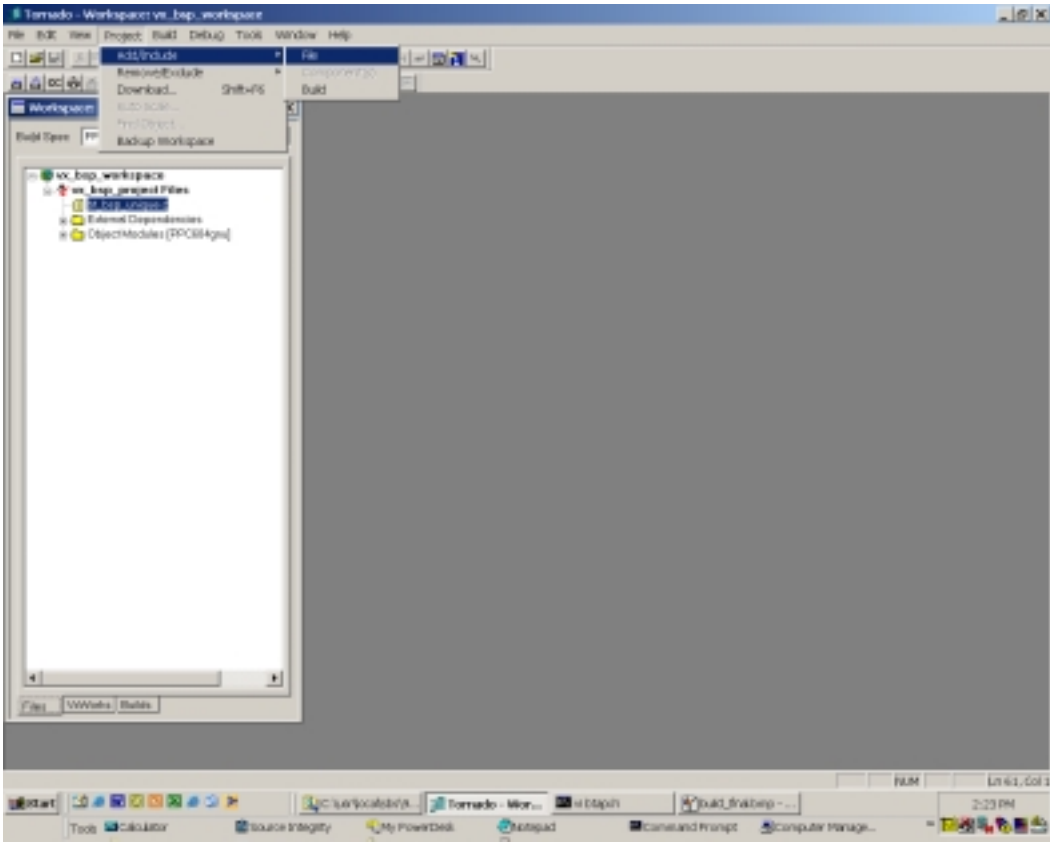
7. Select the C/C++ compiler tab from the Properties window.



Add the following options:

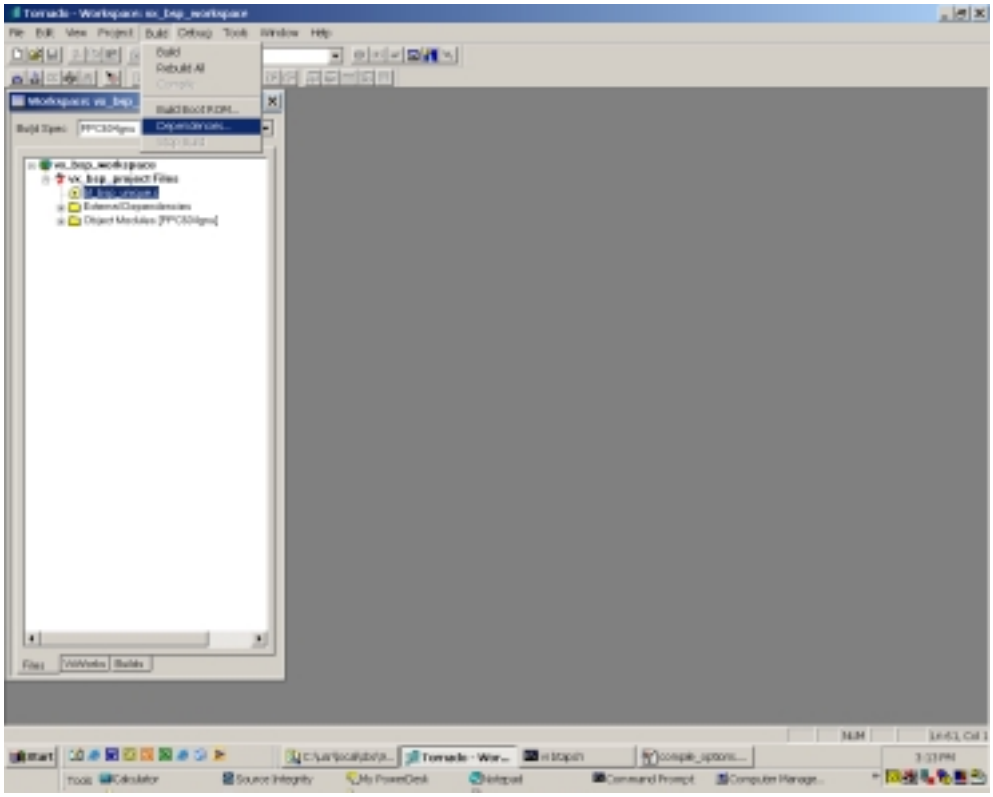
Driver	Option
All	-I c:/usr/local/SBS/993/vx.x/include
	-I c:/tornado/host/x86 –win32/u386_pc_mingw32/sys_include
	-D _ _vxworks
	-Dmain = $*_x
mcp750	-DMCP750_BSP
	-I c:/tornado/target/config/mcp750
k2	-Dk2_BSP
	-I c:/tornado/target/config/powerk2
pcPentium	-DpcPentium.BSP
	-I c:/tornado/target/config/pcPentium

8. Add the bt_bsp_unique.c file to the project by selecting Add/include files from the Project menu.

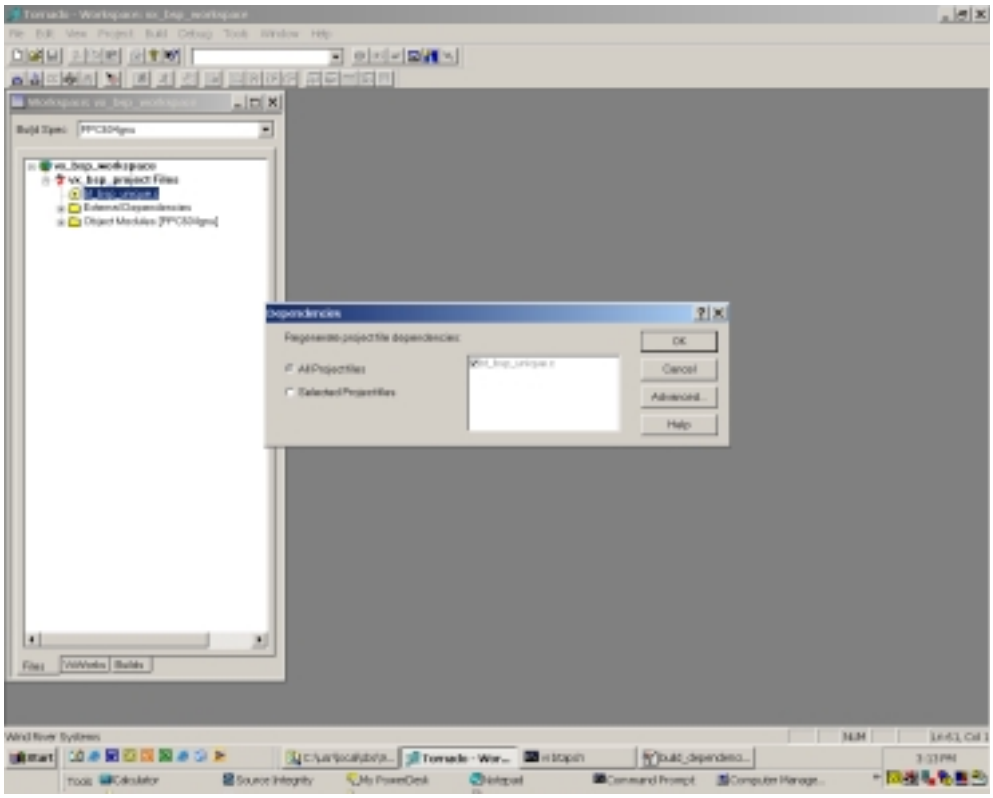


Select the file bt_bsp_unique.c contained in the Porting directory.

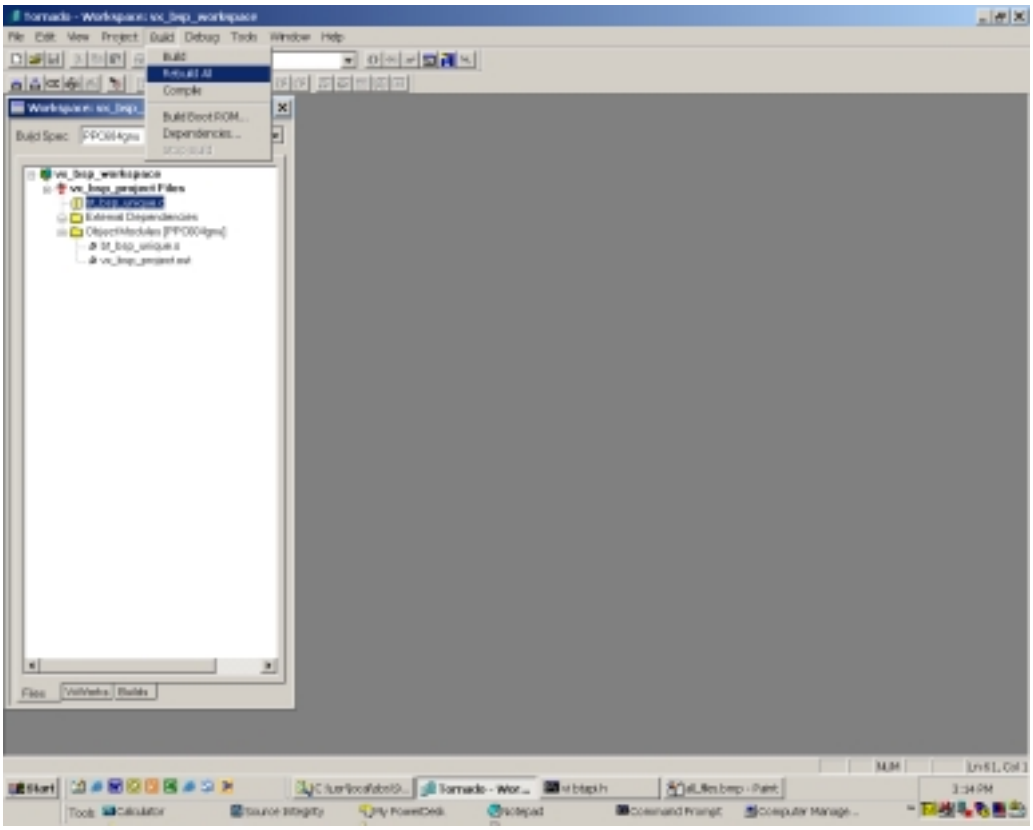
9. Build the dependencies for bt_bsp_unique.c by selecting Dependencies from the Build menu.



Select All Project files.



10. Compile the bt_bsp_unique.c file by selecting Rebuild all from the Build menu.



11. The file vx_bsp_unique.c, the BSP-independent portion of the driver, allows the user to port the SBS BSP independent device driver to any VxWorks BSP.

    a. Load the CPU portion of the driver into VxWorks. The driver object file will have unresolved symbols without the BSP independent driver object.

    b. Load the vx_bsp_unique.obj file that you modified for your BSP and compile.

    c. Reload the CPU portion of the driver to resolve any unresolved symbols.

    d. Load the device driver as outlined in section 9.1.6.

Chapter 10: Model 983

10.0 Introduction

Chapter 10 describes installation of Model 983 Support. It includes general information about the installation procedure, and gives a brief description of how to verify that the adapter is installed correctly and the device driver is loaded properly.

SBS Model 983 Support Software for Intel™ x86-compatible PCI bus computers provides a device driver for Microsoft Windows NT/2000 and example applications to help application programmers with adapter and system configuration. Model 983 currently supports the following SBS adapters:

- All dataBLIZZARDs.
- All 7X2 CompactPCI/PCI adapters.
- All 7X3 CompactPCI/PCI adapters (no loopback diagnostics).
- Model 615 PCI to Q22-bus adapter.
- Model 614 PCI to MULTIBUS I adapter.
- Model 616 that connects a PCI computer to an A32 VMEbus system.
- Model 617 with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 618 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection.
- Model 620 fiber-optic adapter with Slave Mode and Controller Mode DMA for PCI bus to VMEbus interconnection (no loopback diagnostics).
- Model 628 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection.
- Model 630 fiber-optic adapter with Slave Mode and Controller Mode DMA for CompactPCI bus to VMEbus interconnection (no loopback diagnostics).

10.0.1 Components

SBS Support Software consists of the following components:

- A device driver with installation script for Windows NT/2000.
- A BT_QCheck program that is useful for adapter functionality tests.
- Example applications dumpmem, btxyint, bt_bind, readmem, bt_info, bt_reset, bt_sendi, bt_cat, datachk, and bt_icbr that demonstrate using the Mirror API.
- An example application, dumptrc that demonstrates printing trace messages from the driver.
- An example user Interrupt Service Routine (ISR).
- The B3SetDef application to facilitate driver administration and configuration.

10.0.2 System And Hardware Requirements

Windows: Intel x86-compatible computer with a PCI bus with Windows NT/2000.

- For developing Windows console applications –

  Required: Windows compatible 32-bit compiler.

  Recommended: MSVC++ 4.0 or greater as the 32-bit compiler. *Microsoft Development Network (MSDN) Professional* membership.

- For developing Graphical User Interface (GUI) Windows applications -

  Required: Windows compatible 32-bit compiler. To use SBS GUI example source code, MSVC++ 4.0 or greater.

  Recommended: MSVC++ 4.0 or greater as the 32-bit compiler. *Microsoft Development Network (MSDN) Professional* membership.

- For developing user-written Interrupt Service Routines -

  Required: Windows compatible 32-bit compiler and Microsoft Windows NT Device Driver Kit (DDK).

  Recommended: MSVC++ 4.0 or greater as the 32-bit compiler. *Microsoft Development Network (MSDN) Professional* membership. Microsoft or third party kernel level debugger.

VMEbus: The remote reset jumper (SYS-5) on the VMEbus adapter card must be in place to use the remote VMEbus reset function.

The Address Modifier Register jumper (SYS-1) on the VMEbus adapter card must be removed.

MULTIBUS I: No special requirements.

10.1 Installation

10.1.1 Installation Notes

- Refer to the README file for revision history information.
- Microsoft Visual C++ 4.0 is required to compile SBS GUI example applications.
- For user interrupt handlers: a 32-bit compiler, and WinNT DDK™ are required.

  For applications: a 32-bit compiler capable of producing Windows NT/2000 application programs is required.
- Before example applications will run successfully, the device driver must be installed, the cable connected, and the remote system powered on.

10.1.2 Installation

➔ Any time during installation, clicking Cancel aborts the installation. After installation is complete, the B3SetDef program can be run to reconfigure the driver. Be sure to run the SBS uninstall procedure before running Setup again. See section 10.1.4 for Uninstall instructions.

After installing your adapter cards and booting Windows NT/2000:

1. Select Do not install a driver in the New Hardware Found dialog. Click OK.
2. Double click the Windows desktop My Computer icon to view the drives available on your computer.
3. Select a drive on which to create a temporary directory.
4. Create a File/New/Folder in which to download the software.
5. Close all windows.
6. Retrieve the Model 983 software (85221831.exe) from the web (www.sbs.com) or from the CD-ROM.
7. Double click the My Computer icon.
8. Open the download directory by double clicking.
9. Run the archive file by double clicking; this will extract the files.
10. Click Next in the Welcome dialog to continue setup. Clicking Cancel quits the setup program and closes any programs you have running.
11. Enter an installation directory using the browse button or accept the default of C:\Program Files\SBS\983\v*X* by clicking OK (*X* is the actual revision number of the Support Software).
12. Enter a Program Folder name or select one from the list of existing folder names. The default name is SBS 983 v*X* (where *X* is the current Model 983 revision number; for example, SBS 983 v2.0). Click Next to continue. Click Back to enter or select a different Program Folder name.
13. Select Yes, I want to restart my computer now in the Setup Complete dialog. Click Finish to complete installation and reboot your system.

    ➔ If you select No, I will restart my computer later, you must reboot your system before the device driver will be loaded and the example applications can be used or before B3SetDef can be run.

10.1.3 B3SetDef Program

B3SetDef is a program used to set the default values for various configurable driver parameters. The configurable parameters are divided into two types: trace parameters and adapter parameters.

Trace parameters affect all adapters installed in the system.

PARAMETER	DESCRIPTION
User Trace Length	The number of trace messages to be kept by the driver.
User Trace Flags	The set of flags that determine which trace messages will be printed by the driver.

Adapter parameters control accesses to individual adapters. Each adapter is identified by a unit number. If multiple adapters are installed, some experimentation may be required to discover which unit numbers correspond to which slot in the system.

PARAMETER	DESCRIPTION
DMA Pause	VMEbus specific. Causes the DMA controller to rearbitrate for the bus more often than required when doing a block transfer. This reduces the arbitration latency for other bus masters at some cost to the maximum DMA performance.
Remote Reset Delay	The length of time to wait after a remote reset before allowing remote accesses again.
DMA Threshold	The initial value for the BT_INFO_DMA_THRESHOLD parameter (see bt_get_info()).
DMA Poll Ceiling	The initial value for the BT_INFO_DMA_POLL_CEILING parameter (see bt_get_info()).
DMA Timeout	The maximum time for a DMA transfer to complete before the transfer is aborted.
Interrupt Node Count	The number of pending (unacquired) interrupts the driver can accommodate.
Local Memory Device Size	Size in bytes of the Local Memory Device (BT_DEV_LM). Must be a multiple of 4096. A value of zero disables the Local Memory Device.

10.1.4 Uninstall Procedure

➔ Run the uninstall program before re-running the setup program.

1. Open Add/Remove Programs control panel.
2. Double click the Uninstall 983 program icon.
3. Click Yes in the Confirm File Deletion dialog.
4. Click OK in the Remove Programs From Your Computer dialog.

10.1.5 Verifying The Installation

Assuming setup completed successfully, there are several tests that can be performed to make sure the driver is installed and functioning correctly.

10.1.5.1 Presence Of The Driver

In a DOS shell to show the driver, use the following command:

```
DIR %WINDIR%\SYSTEM32\DRIVERS\BT983.SYS
```

In a DOS shell to show the DLL, use the following command:

```
DIR %WINDIR% \SYSTEM32\BIT3_API.DLL
```

Make sure the program files directory as entered into the Setup programs Setup Type dialog (defaults to C:\Program files\SBS\983\vx.x) was created by the installation.

For Windows NT version 4.0, make sure a SBS group was added to the Programs menu under the Start Button. Minimally, this group should contain the readme.txt and uninstall programs.

For Window NT version 3.51, check that a program group was created. This group should have the same name as set in Setup and that defaults to SBS 983 vx.x. Minimally, this group should contain the readme.txt and uninstall programs.

10.1.5.2 Driver Functioning

If installed, the B3SetDef program can be use to detect the number of cards installed in the system and to configure those cards. If executable images of the example applications were installed, several can be used to test how the device driver is functioning:

- The btqcheck program provides a graphical user interface to several driver tests. See section 10.2.1.
- The dumpmem program can be used to test remote bus accesses. See section 3.1.
- The readmem program can be used to test remote bus accesses. See section 3.2.
- The bt_cat program can be used to test remote bus accesses. See section 3.3.
- The datachk program can be used to test remote bus accesses. See section 3.4.
- The dumptrc program can be used to view log messages. After booting, these messages should include the driver probing the PCI buses and finding any cards installed. See section 10.2.2.

10.2 Model 983 Specific Example Applications

10.2.1 btqcheck Example Application

btqcheck tests the adapter's and device driver's functionality. It runs a sequence of tests and displays the output in a window. Multiple tests can be run simultaneously because each test runs in a separate thread. Tests can easily be added or subtracted.

Start Test Dialog:

The Start Test Dialog is displayed by clicking "Start Test" in the Test Menu. It sets the test parameters to be run. After specifying the parameters, click on "OK" to start the tests, or "Cancel" to abort (does not start the tests).

The following parameters can be set:

PARAMETER	DESCRIPTION
Test to Run	List of tests that can be performed. Multiple selections are allowed. If more than one test is selected, they are performed iteratively in the order of appearance in the scroll box.
Unit Number	Unit number of the adapter card to test.
Logical Device	Logical device of the given unit to test.
Iteration Count	Number of times each test repeats.
Exit on Error	Exit the test on the first error encountered.
Verbose	Print test operation messages to the test window.
View Trace Messages	Print device driver trace messages to the test window.
View Read Data	Print data read from the logical device to the test window.
Base Address	Base (lowest) remote bus address the test can address.
Transfer Size	Size in bytes of the remote bus address space to use. The test can access remote bus addresses between Base Address and Base Address plus Size minus one, inclusive.
Pattern Type	Selects the type of data pattern that will be written to the logical device.
Pattern Data Width	Selects the width of the data pattern values used to fill the buffer that will be written to the logical device.
Increment Value	The amount by which each Pattern Data Width value written to the adapter will be increased or decreased. Used only if Pattern Type is set to Incrementing or Decrementing.
Initial Data	The starting value used to fill the buffer that will be written to the logical device.

Test Menu:

The Test Menu is btqcheck's replacement for the File Menu. Options under the Test Menu are:

OPTION	FUNCTION
START TEST	Starts a new test. Displays the Start Test Dialog, and opens a new Test Output window.
PARAMETERS	Displays the Run Time Parameters dialog that allows the following run time parameters to be set: ■ DMA Threshold ■ DMA Poll Ceiling ■ DMA Timeout ■ DMA Address Modifier ■ DMA Pause ■ Remote Reset Delay ■ Data Width ■ PIO Address Modifier
TRACE FLAGS	Displays the Trace Flags dialog that allows user selection of the trace message types that will be printed to the test window when View Trace Messages is selected.
CLOSE	Closes the currently selected Test Output window, killing the test if necessary.
STOP	Stops the current test without closing the Test Output window.
STOP ALL	Stops all currently executing tests without closing any Test Output windows.
PRINT	Prints the test output to a printer.
PRINT PREVIEW	Shows how the test output would look if printed.
PRINT SETUP	Sets printer options.
EXIT	Exits btqcheck and kills all currently executing tests.

Window Menu:

OPTION	FUNCTION
NEW WINDOW	Opens a new window duplicating the view of the currently selected Test Output window.
CASCADE	Cascades all currently opened windows.
TILE	Tiles all currently opened windows.
ARRANGE ICONS	Arranges the window icons.

Help Menu:

OPTION	FUNCTION
HELP TOPICS	Opens the help file index.
ABOUT btqcheck	Displays the "About" dialog.

Test Output Windows:
Each open document window in btqcheck displays output from a test run. Each test has a distinctive output. The scroll bars can be used to move through the output, and the windows can be maximized or minimized. Input to the tests, and edit commands (cut, paste, etc.) are not supported.

10.2.2 dumptrc Example Application

The dumptrc program is a simplistic console application that continuously reads any trace messages produced by the driver and prints them to standard output. dumptrc sleeps between each call to get the trace messages. The sleep length is set via the -s option. To exit the program, press q.

10.3 Porting Applications

10.3.1 Porting Applications From Previous Windows Drivers

There are many changes that only require simple textual substitutions; for example, BT_Read becomes bt_read. Some changes require more complex code changes. These are:

- Programs should include btapi.h instead of directly including btwapi.h.
- The semantics of opening a device have changed. If bt_gen_name() is not called in the call to bt_open() (see section 5.1.3), the return value must be kept and used in the call to bt_open(). It is possible for the return value of bt_gen_name() to be not equal to NULL and not be the array passed in.
- The access flags and the pointer to the map pointer arguments to bt_mmap have been exchanged. In addition, any map length to any map address is now supported. It is no longer necessary to map extra and then adjust the pointers.
- The functions BT_Setup(), BT_ClrStatus(), and BT_Reset(), all of which optionally return the device status, are no longer supported. The functions bt_init(), bt_clrerr(), and bt_reset() should be used. If the status is needed after one of these calls, the function bt_status() can be called.
- Interrupt handling is simpler than before. The functions BT_CreateInterrupt(), BT_DestroyInterrupt(), BT_RegisterInterrupt(), BT_UnregisterInterrupt(), BT_SignalInterrupt(), BT_WaitforInterrupt(), and BT_AcquireInterrupt() are no longer supported. Instead the functions bt_icbr_install() and bt_icbr_remove() should be used. The Model 983 driver has the ICBRs running in their own threads. For example, the code that responds to the interrupt can simply be placed into the ICBR.

10.3.2 Porting Applications From UNIX

Somewhat more work is required to port applications from the old UNIX interface to the Mirror API on Windows. The Mirror API provides the function bt_ctrl() that on UNIX is an interface to the ioctl() call. Note: ioctl() cannot be called with a bt_desc_t. As the Model 983 driver has no ioctl() interface, bt_ctrl() returns BT_ENOSUP. Consequently, every ioctl() call in the UNIX application will have to be rewritten to use the appropriate Mirror API call instead.

In addition, many of the comments in section 10.4.1 are applicable.

10.4 Extending Or Modifying The Example Applications

10.4.1 Modifying bt_icbr Code Structure

There are three ways to extend bt_icbr: allow it to receive other types of interrupts, have it do something other than simply print a message when an interrupt occurs, and improve the mechanism by which it sleeps waiting for interrupts.

To receive interrupt types other than error interrupts, change the arguments to the call bt_icbr_install(). Only error interrupts are supported on all Mirror API products. Other interrupt types such as IACK interrupts and programmed interrupts are NanoBus-specific. See section 4.5 for more information. The switch statement in main() that determines how to respond to the interrupt to properly handle the new type of interrupt will also need to be modified.

The program structure is slightly odd. It is limited in what it is guaranteed to do in an ICBR. Consequently, the bt_icbr only puts the information into a FIFO queue that the main program reads data from and then acts upon the data. The functions queue_insert() and queue_remove() are used to maintain the queue.

There is no way in ISO Standard C to poll standard in; even the function sleep() is not part of the ISO standard. To maintain the portability of the program, the main function uses getchar() to sleep. Every time input is read, it polls the FIFO queue for new interrupts. Programs with less stringent portability requirements may use sleep(), select(), or similar functions. Programs that only need to run on Windows may assume the ICBR is run in a separate thread and do all processing in the ICBR.

10.5 User Written Interrupt Handlers

You can extend the Model 983 device driver's internal Interrupt Service Routine (ISR) by writing your own interrupt handlers. This is done through a user written kernel mode device driver that must be installed in the system along with the Model 983 device driver.

Writing a kernel mode device driver for Windows NT is a complex task requiring knowledge of Windows NT operating system internals. This chapter assumes you are knowledgeable in writing NT drivers.

The source code for a driver that incorporates several sample user interrupt handlers is located in the .\examples\usrisr directory. The .\examples\usrisr directory contains the following files:

- bit3uisr.c: Main driver module that implements several user interrupt handlers and the code to register them with the Model 983 driver.
- bit3uisr.h: Header file for the driver.
- makefile: The standard Windows NT make file for building the kernel mode device drivers.
- sources: A file used with the Microsoft build utility containing macros that describe the driver directories and file names.

The bit3uisr driver was written using the Microsoft Windows NT 4.0.1 DDK and the Microsoft Visual C++ 4.0 compiler. These items must be installed on your system in order to build the bit3uisr device driver. Any discussions in this chapter assume use of these tools.

➔ The Model 983 driver's internal ISR is referred to as the 983 ISR in the remainder of this chapter.

10.5.1 Types Of User Interrupt Handlers

There are three types of user interrupt handlers: error, programmed, and cable (IACK). Each is called directly by the 983 ISR in the interrupt context. The user driver need not implement all three types, only those actually registered with the Model 983 driver will be called.

10.5.1.1 Error Interrupt Handlers

Error handlers are called if the 983 ISR detects that an error interrupt occurred. Any registered error handlers are called before the 983 ISR clears the error condition. If there is an error interrupt, programmed and cable interrupt handlers are not called.

10.5.1.2 Programmed Interrupt Handlers

Programmed interrupt handlers are called if the 983 ISR detects that a PT or PR interrupt occurred. First, a PT interrupt is checked for, and if it is active, any registered programmed handlers are called before clearing the PT interrupt. Next, a PR interrupt is checked for, and if active, any registered programmed handlers are called before clearing the PR interrupt.

10.5.1.3 Cable (IACK) Interrupt Handlers

Cable interrupt handlers (IACK handlers) are called if the 983 ISR detects a cable interrupt is active but no PT interrupt occurred. Cable handlers are called only if the cable interrupt level they were registered for matches an active cable interrupt number returned in the Interrupt Status Register.

Cable interrupts (except for PT) are generated on the remote bus. Consequently, the 983 ISR has no knowledge of how to clear such an interrupt. Therefore, cable interrupt handlers are responsible for clearing the interrupt on the remote bus. When a cable handler handles the interrupt it must return a non-zero return value.

10.5.2 Registering User Interrupt Handlers

Before a user interrupt handler can be called, it must be registered with the Model 983 driver by the user.

10.5.2.1 When To Register A User Interrupt Handler

The Model 983 driver is loaded during Windows NT initialization and is never unloaded. It supports registration of user interrupt handlers at any time after it is loaded.

The best way to ensure that the Model 983 driver is present when registering is to load the user driver after the Model 983 driver. This is done by controlling the driver load order with the ServiceGroupOrder key in the registry.

The Model 983 driver is assigned the group name of PCI Configuration. This group name is listed in the registry under the List value of the key \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ ServiceGroupOrder. Assign your user driver its own group name, and then place this group name after the PCI Configuration group name of the List value. Refer to section 10.5 for an example of how this is done.

For further information refer to section *Method 1* in the Microsoft Win32 Knowledge Base article *Two Methods to Control Device Driver Load Order* (ID Number Q115486). This article is included on the Microsoft Developer's Network Library CD.

10.5.2.2 How To Register A User Interrupt Handler

User driver registration is accomplished by sending an IOCTL_BTBRIDGE_REGISTER_UISR internal device control request to the Model 983 driver using IoCallDriver(). The function RegisterUserIsr(), provided in bit3uisr.c, shows how this is done. We recommend that RegisterUserIsr() be used rather than calling IoCallDriver directly.

The DriverEntry function in bit3uisr.c demonstrates use of RegisterUserIsr().

First, initialize a unicode string with the name of the device object in the Model 983 driver for the unit to register with; use RtlInitUnicodeString(). The device object name string passed to RtlInitUnicodeString() has the form:

```
L"\\Device\\BtwControl"
```

Next, IoGetDeviceObjectPointer() is called to obtain the device object of the unit to register on. The first argument takes the device object name unicode string described above. Always check the return value to ensure that the Model 983 driver is loaded and the desired unit exists.

Finally, RegisterUserIsr() is called once for each user interrupt handler being registered. More than one of each type of handler can be installed. When an interrupt occurs for a given unit number and handler type, the 983 ISR will call the handlers in the order registered until a handler indicates that it has handled the interrupt by returning a non-zero value.

10.5.2.2.1 RegisterUserIsr()

PROTOTYPE	BOOL RegisterUserIsr (PDEVICE_OBJECT pDeviceObject, PBT_UISR_REGR pRegr, PBT_UISR_INFO pInfo)
PURPOSE	Register a user interrupt handler with the Model 983 driver.
ARGUMENTS	PDEVICE_OBJECT pDeviceObject: Pointer to the device object of the driver and unit to register with. This can be obtained with IoGetDeviceObjectPointer(). Always register with the A32 logical device of the desired unit. PBT_UISR_REGR pRegr: Pointer to the user ISR registration structure. The caller must fill this structure as follows: ■ pRegr->btIntrFlag: Type of interrupt. Must be one of BT_INTR_ERR, BT_INTR_PRG, or BT_INTR_IACK. ■ pRegr->dwCIntLevel: Cable interrupt level being registered for on BT_INTR_IACK registrations. If bt_IntrFlag = BT_INTR_IACK this must be one of BT_CINT1 through BT_CINT7, otherwise it must be BT_CINT_NONE. ■ pRegr->pHandler: Pointer to user's interrupt handler. ■ pRegr->pParam: Parameter that will be passed to user's interrupt handler. This value is determined by the writer of the user interrupt handler. PBT_UISR_INFO pInfo: Pointer to user ISR info struct. This will be filled in by the Model 983 driver. It contains information needed by the user interrupt handler when accessing the adapter hardware.
RETURNS	BOOL: TRUE if successful; otherwise, FALSE.
COMMENTS	The user interrupt handler must be implemented in same driver that calls this function. This function sends an IOCTL_BTBRIDGE_REGISTER_UISR internal device control request to the Model 983 driver using IoCallDriver().
WARNING	All user ISRs must be unregistered before the user driver is unloaded.

10.5.2.2.2 BT_UISR_INFO Structure

PBT_UISR_INFO pInfo is the third argument of RegisterUserIsr(). Each user interrupt handler requires one of these structures. It will be filled by the Model 983 driver when registration is done.

The BT_UISR_INFO structure contains pointers to adapter resources. Section 10.5.4.2 explains how to use this information.

10.5.3 Unregistering A User Interrupt Handler

If your user driver cannot be unloaded, it is usually unnecessary to unregister user interrupt handlers. An exception is if the interrupt handling requirements change at run time.

➔ If your user driver can be unloaded, it is essential that all user interrupt handlers be unregistered before the driver is unloaded. If this is not done, the 983 ISR may attempt to call a handler that will use resources that are no longer present, resulting in a system crash.

10.5.3.1 How To Unregister A User Interrupt Handler

Unregistration of user interrupt handlers is accomplished by sending an IOCTL_BTBRIDGE_UNREGISTER_UISR internal device control request to the Model 983 driver using IoCallDriver(). The function UnregisterUserIsr(), provided in bit3uisr.c, shows how this is done. We recommend that UnregisterUserIsr() be used rather than calling IoCallDriver() directly.

10.5.3.2 UnregisterUserIsr()

PROTOTYPE	BOOL UnregisterUserIsr (PBT_UISR_REGR pRegr)
PURPOSE	Unregister a user interrupt handler previously registered with RegisterUserIsr().
ARGUMENTS	PDEVICE_OBJECT pDeviceObject: Pointer to the device object of the driver and unit to unregister with. This can be obtained with IoGetDeviceObjectPointer(). Always unregister with the A32 logical device of the desired unit. PBT_UISR_REGR pRegr: Pointer to the user ISR registration structure. The contents of this structure should be the same as the pReg structure passed in an earlier call to RegisterUserIsr().
RETURNS	BOOL: TRUE if successful, otherwise FALSE.

10.5.4 Writing A User Interrupt Handler

When writing a user interrupt handler, keep in mind that it will run in interrupt context. This means you must follow all rules for programming ISRs, including:

- Observe IRQL requirements when calling NT support routines.
- Never block.
- Never access pageable memory.
- The handler code must be non-pageable.
- Keep the handler as short and fast as possible.
- Limit the amount of stack space used.
- Do not touch the local adapter card's DMA registers. These are controlled by the Model 983 driver.

Also, be aware that the handler is invoked through a function call from the Model 983 driver's ISR, and it will return execution to that ISR when complete.

10.5.4.1 User Interrupt Handler Definition

PROTOTYPE	DWORD UserInterruptHandler (ULONG ulUnitNum, PVOID pParam, BT_INTRFLAG btIntrFlag)
PURPOSE	User interrupt handler.
ARGUMENTS	ULONG ulUnitNum: Unit number. PVOID pParam: User defined parameter. This value is obtained from the pRegr->pParam argument passed to RegisterUserIsr(). BT_INTRFLAG btIntrFlag: Type of interrupt. Will be one of BT_INTR_ERR, BT_INTR_PRG, or BT_INTR_IACK.
RETURNS	DWORD: 0: Interrupt not handled. Otherwise: Return value indicating interrupt serviced.
WARNING	This function is called in an interrupt context.

10.5.4.2 Accessing The Adapter Hardware

To access the adapter hardware, information about adapter resources is needed. This information is provided in the BT_UISR_INFO structure that is filled when the handler is registered (see section 10.5.2.2.2). Refer to the cable interrupt handler, SDmaHandler() in bit3user.c for an example of how this information can be used.

The adapter hardware resources available to user interrupt handlers are:

- A 4K byte window into the remote bus.
- A single mapping register to set up the base address and other characteristics of the remote 4K byte window.
- The local adapter card's node I/O registers.

10.5.4.2.1 Remote Bus Window

The pWin member of the BT_UISR_INFO structure is a pointer to the base of the 4K byte window into the remote bus. This allows the user interrupt handler to access any valid I/O or memory space on the remote bus.

Before using the remote bus window, the Mapping Register must be initialized (see section 10.5.4.2.2).

The user interrupt handler cannot use the remote bus window unless the local adapter card is configured as a transmitter.

10.5.4.2.2 Mapping Register

A single window Mapping Register in the adapter hardware is reserved for use by user interrupt handlers. The pMapReg member of BT_UISR_INFO is a pointer to this Mapping Register.

The Model 983 driver initializes the Mapping Register to invalid at DriverEntry time. This is the only time the Model 983 driver will touch this register.

The user written driver must load the Mapping Register with the base address of the 4K byte remote bus window before that window can be accessed. Use the LOAD_MAPREG macro (defined in btwuser.h) to load the Mapping Register.

The user interrupt handler should load the Mapping Register every time it executes. The only exception to this rule is if the Mapping Register contents will never change through the life the user driver. In that case, the Mapping Register can be loaded once before the remote bus window is accessed for the first time.

10.5.4.2.3 Node I/O Registers

The pNodeIo member of the BT_UISR_INFO structure is a pointer to the adapter hardware's node I/O registers in kernel virtual memory space. pNodeIo is a pointer to BT_REGMAP that gives a memory mapped representation of the node I/O registers. Individual registers can be accessed by de-referencing the appropriate member of BT_REGMAP.

For example, reading the Local Status Register into variable byLSR is coded as:

```
BT_UISR_INFO pInfo;     // assume this has been initialized by RegisterUserIsr()
BYTE byLSR;             // contents of local status register
byLSR = pInfo->pNodeIo->byLocStatus;     // read the register
```

➔ The BT_UISR_INFO structure must have byte packing. Therefore, be sure to set your compiler options or use packing pragmas appropriately before including bit3user.h. An example of how to do this is shown in bit3uisr.h.

10.5.4.3 Return Values

When a user interrupt handler is invoked, it must determine if it should handle the interrupt. Function XYCOMHandler() in bit3uisr.c shows how this is done in the case of a Xycom card interrupting. If the user interrupt handler does not handle the interrupt, it must return a zero value. If it handles the interrupt, a non-zero value should be returned.

For a given unit number and interrupt type, more than one handler can be registered. The 983 ISR will call each handler in the order it was registered until a non-zero value is returned, indicating the interrupt was handled. Once the interrupt has been handled, no more handlers of that type on that unit will be called during that instance of the 983 ISR.

➔ For error and programmed interrupt handlers, it is not essential that a user handler clear the interrupting condition because the 983 ISR will handle this task. However, for cable interrupts (except PT) the 983 ISR cannot clear the interrupt on the remote bus. Therefore, it is essential that a cable interrupt handler clear the interrupt and that it return a non-zero value to indicate that the interrupt has been handled. If the 983 ISR determines an active cable interrupt has not been handled, it will disable further interrupts to avoid hanging the system in an interrupt loop.

For programmed and cable interrupt handlers, the return value is saved along with other state information for later retrieval by application programs. These return values are passed to the ICBRs as the vector argument.

10.5.5 Installing A User Written Driver

The subject of device driver installation in Windows NT/2000 systems is beyond the scope of this manual. However, we suggest one simple installation method.

1. Install the Model 983 driver.

2. Copy bit3uisr.sys into the %SystemRoot%\system32\drivers directory.

3. Using the registry editor (regedt32.exe), add a key with the same name as your driver (for usrisr, Bit3uisr the .sys is left off) to HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.

4. Using the registry editor, add the following values to the key created in step 3:

   Start: REG_DWORD: 0x00000001

   Type: REG_DWORD: 0x00000001

   Group: REG_SZ: Bit3_User_ISR

5. Using the registry editor, locate the key:
   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
   Service GroupOrder.

   Edit the List value of this key, and add the string Bit3_User_ISR after the existing string PCI_Configuration.

6. Reboot the system.

For further information on driver installation, refer to the Windows NT DDK.

Chapter 11: General Software Issues

11.0 General Software Issues

11.1 Porting Applications From UNIX Direct Device Interface

Somewhat more work is required to port applications from the old UNIX interface to the Mirror . This only applies to Models 946 and 965 because the other software models always use Mirror API.

In addition, many of the comments in section 11.2 are applicable.

To convert a program from the direct driver interface to the Mirror API:

1. Change the program to use bt_gen_name(), and bt_str2dev() routines to generate the device names. Include the btapi.h header file in addition to the btio.h header file.

2. Remove all the calls to BIOC_LOCK and BIOC_UNLOCK and replace with operating system specific mechanisms if required.

3. Change the open() and close() routines to use bt_open() and bt_close(). Change the program to use a bt_desc_t to identify the device instead of an integer.

4. Change the mmap() and munmap() routines to use bt_mmap() and bt_unmmap().

5. Rewrite any code that used signal handlers for interrupt notification to use ICBRs. This should simplify the code and make the driver more efficient when notifying an application.

6. Convert all other ioctl() calls to use bt_ioctl(). This is only a temporary measure to allow you to get the program running.

7. Debug.

8. Change the bt_ioctl() calls to the equivalent Mirror API routines, after which, you will no longer need to include the btio.h header file.

11.2 Writing Portable Applications Using The Mirror API

This section deals with the issues arising from using the SBS API in a portable way.

11.2.1 Using NanoBus Or Model Specific Extensions

When writing code that will be ported and that uses the Mirror API, be aware of the generality of the functions used. All functions in the Mirror API fall into one of three categories: supported on all SBS products, supported on all SBS products of the same family, and supported only on one or a small set of SBS products.

For example, bt_open() is a function that is supported on all SBS Mirror API products. A program may assume that this function exists and works as described on any SBS API product.

An example of a function that is only supported on a given family of products is bt_tas(). All NanoBus-based products support this function. However, products based on other hardware designs, such as the NanoPort family of hardware, may not support this program. To help programs determine at compile time which family-based functions are available, every Mirror API product defines a preprocessor symbol that indicates the family. For example, all NanoBus-based products define the preprocessor symbol BT_NBUS_FAMILY. Programs can test for the existence of these functions:

```
        # ifdef BT_NBUS_FAMILY
bt_tas(btd, addr, prev_val_p);
        # else /* BT_NBUS_FAMILY */
        # error This program only supports NanoBus-based programs!
        # endif /* BT_NBUS_FAMILY */
```

Some functions only exist on a specific model. The token BT<MODEL#> can be used to test for all model specific functions. For example, the function bt_gettrace() function is only supported on Model 983; therefore, before using bt_get_trace(), you must test for the BT983 token. Use code similar to that above.

11.2.2 BT_ENOSUP Error Return Value

A supported function may return BT_ENOSUP, a special error return value that indicates a requested service is not available. Common reasons this may occur are:

- Using the function bt_ctrl() to access an unsupported ioctl() call. No ioctl() calls are supported by the Model 965; therefore, all calls to bt_ctrl() will return BT_ENOSUP.
- Attempting to open an unsupported device. For example, the Node I/O device that is a legal device in the NanoBus family but is not supported on all models. Attempting to open this device will cause bt_open() to return BT_ENOSUP.
- Attempting to use bt_bind() on a product that does not currently support it.

11.3 Be Careful Of Optimization

The C compiler provided with SGI UNIX supports the volatile type modifier defined by the ANSI C standard. However, other compilers may not provide this support. The only way to prevent some cc compilers from optimizing out successive accesses to the adapter registers or memory sections is to restrict the types of optimization. The highest level of optimization may cause the compiler to remove the necessary data references.

One example of where optimization could cause problems, is a mailbox using a special flag to indicate that data are changing. Applications using this type mailbox may have code similar to the following:

```
while (mailbox_p->is_ok != TRUE)   /* Empty while loop */ ;
```

Because the program has no code that changes the value at the pointer, the compiler may assume it only has to read the flag once. This assumption is logically valid, but does not apply when the value can be modified by something outside the application, such as another process on the PCI system or remote bus system. To prevent this optimization, the type qualifier volatile is used in the type declaration for the mailbox_p variable.

11.4 Using Structures

Care must be taken when passing structures between systems. Make sure all data types are the same bit length and that there are no holes in the structure, since different processors and compilers can generate different structure alignments.

The following structure demonstrates a few possible problems:

```
struct foo {
    int xyz;
    char abc[3];
    long lmn;
} *bar;
```

If this structure is used with an adapter to pass data between a PCI computer and a remote bus system, it may not produce identical results under all compilers. Some processors require that long data types be aligned on 4-byte boundaries. Others require alignment on a 2-byte boundary or have no alignment requirements.

Also, if the structure above uses the int data type, results may vary. Some compilers use a 16-bit integer, others use 32-bit integers, and some allow a compile-time switch to determine the size of integer types.

To find size and alignment problems, use the sizeof operator. If the two compilers generate different sizes for the same data structure, there is an alignment or data size problem.

One solution is to explicitly define any filler space required. Also, if the size of the integer type differs between the compilers, use either the short or long data types since most compilers use a 16-bit short and a 32-bit long data type.

If more control is needed, create your own data types for each data size using typedef statements. Then programs can use your custom data types instead of those defined in the C programming language.

C compilers usually begin structures on 4-byte boundaries. Also, standard memory allocation routines usually allocate on 4-byte boundaries. Determine if this is true of all systems you are using.

The example structure on the previous page works on many compilers when changed to the following:

```
struct foo {
        short   xyz;        /* assumed 16-bit data                 */
        short   filler1;    /* always align on 4-byte boundary  */
        char    abc[3];
        char    filler2;    /* always align on 4-byte boundary  */
        long    lmn;
} *bar;
```

There may be more compact ways to store the data, but if every element is aligned, this method makes it is easier to confirm correctness.

11.4.1 Memory Modifying Functions With Memory-Mapped Addresses

Take care when attempting to use a vendor's C library routines for performing memory-to-memory, file-to-memory, or similar types of memory movement or initialization operations with the SBS device. None of these routines are guaranteed to consistently use the same transfer size when accessing data. In particular, special care should be taken when using the types of routines listed below.

- memcpy(), memmove(), memset()
- strcpy(), strncpy()
- read(), write(), fread(), fwrite()

When source or destination addresses supplied to these routines are actually memory mapped locations corresponding to the SBS adapter itself, problems may occur. These problems result from the freedom each vendor has with respect to those functional implementation details within the library itself. How these routines are implemented may vary between versions of a manufacturer's operating system or across different platform models.

Understandably, a vendor may choose to take advantage of its own intimate knowledge of the underlying hardware and associated memory subsystem to code these routines for maximum efficiency. Unfortunately, the assumptions made do not always hold true for memory mapped devices that are inherently dependent on actual access width for proper program and/or device operation.

A vendor may use special instructions or hardware that does not allow access to the interface bus (where the SBS or other hardware devices are located). Vendors may also make optimizations that are only valid when the destination address is system memory.

For example, a version of memcpy() that checks the length of data to be copied may behave differently, dependent on how much data are to be moved. When small amounts of data are to be transferred, it may move the data as bytes. When larger amounts of data are transferred, it may attempt to use the floating point registers or special cache control instructions. Not all implementations allow the floating point unit to access the interface bus, nor guarantee that any other special hardware that works on the processor's internal bus will be supported out to the internal bus.

For these reasons, as well as to increase code portability, we recommend that you avoid using these types of routines when working with memory mapped pointers to the SBS device.

11.5 Extending or Modifying The Example Applications

11.5.1 Modifying The bt_icbr Code Structure

There are three ways to extend bt_icbr: allow it to receive other types of interrupts, have it do something other than simply print a message when an interrupt occurs, and improve the mechanism by which it sleeps waiting for interrupts.

To receive interrupt types other than error interrupts, change the arguments to the call bt_icbr_install(). Only error interrupts are supported on all Mirror API products. Other interrupt types such as IACK interrupts and programmed interrupts are NanoBus-specific. See section 4.5 for more information. The switch statement in main() that determines how to respond to the interrupt to properly handle the new type of interrupt will also need to be modified.

The program structure is slightly odd. It is limited in what it is guaranteed to do in an ICBR. Consequently, the bt_icbr only puts the information into a FIFO queue that the main program reads data from and then acts upon the data. The functions queue_insert() and queue_remove() are used to maintain the queue.

There is no way in the ISO Standard C to poll standard in; even the function sleep() is not part of the ISO standard. To maintain the portability of the program, the main function uses getchar() to sleep. Every time input is read, it polls the FIFO queue for new interrupts. Programs with less stringent portability requirements may use sleep(), select(), or similar functions. Programs that only need to run on Linux may assume the ICBR is run in a separate thread and do all processing in the ICBR.

Appendix A: Glossary

The following terms are used throughout this manual:

"0": Zero.

"1": One.

Adapter Node Input/Output: Any access to the I/O registers contained on either the PCI or remote bus adapter card. These are referred to as local node I/O and remote node I/O, respectively.

Address Modifier: A code designating the type of access (short, standard, or extended; non-privileged or supervisory) to occur on the VMEbus. VMEbus devices must receive their correct address modifier as well as the correct address or they will not respond to an access.

Bit: A single digit in a binary number (0 or 1).

Byte: 8 bits.

Cable Interrupt: An interrupt sent from a device on the remote bus system across the interface cable. The PT programmed interrupt also comes across the interface cable, but is considered as separate from the other cable interrupts.

Direct Memory Access Transfers (DMA): The adapter may be programmed to transfer large blocks of data across the cable to or from the remote bus chassis, rather than requiring a processor to move data.

DLL: Dynamically linked library.

Dual Port RAM: An optional dual-port memory card attached to remote bus adapter card.

Exchanging Interrupts: Sending interrupts to and receiving interrupts from the remote bus chassis. Also includes any processing an application should do to acknowledge the receipt of an interrupt.

G byte: Gigabyte. Two to the thirtieth power (exactly 1,073,741,824 bytes).

Hex: Hexadecimal notation. A numbering system that uses 16 digits (0123456789ABCDEF) to denote a number.

K byte: Kilobyte. Two to the tenth power (exactly 1024) bytes.

Local: Pertaining to the system accessing the adapter. Implies that it is not necessary to go across the interface cable to access the resource.

Longword: 32 bits.

M byte: Megabyte. Two to the twentieth power (exactly 1,048,576) bytes.

M Bytes/sec: Megabytes per second. Exactly 1,000,000 bytes per second.

MDI : Multiple document interface.

msec: Millisecond. 1/1,000 of a second.

nsec: Nanosecond. 1/1,000,000,000 of a second.

Physical Address: The actual or machine address of an item or device.

PIO: Programmed I/O.

PR Interrupts: See Programmed Interrupts.

Programmed Interrupts: Interrupts caused by setting a flip-flop in one of the adapter Node I/O registers. The two types of programmed interrupts are the PT (Programmed to Transmitter) interrupt and the PR (Programmed to Receiver) interrupt.

PT Interrupts: See Programmed Interrupts.

Receiver: An adapter card that is not allowed to transmit messages across the interface cable. Consequently, preventing it from accessing the Remote Node I/O, Remote Bus I/O, and Remote Bus memory, or a remotely-installed Dual Port RAM card.

Remote: Pertaining to the system accessing the adapter. Implies that the resource is located at the other end of the adapter interface cable.

Remote Bus Input/Output: Any access to the I/O registers of devices that are physically located in the remote bus chassis (*not* the remote adapter card). For VMEbus this is the A16 address space.

Remote Bus Interrupts: Interrupts generated by devices on the remote bus that are passed, via cable interrupt lines, to software residing in the PCI computer.

Remote Bus Memory: Any access to the memory space in the remote bus chassis: a shared memory section, a device buffer, or any device that responds to a memory access. Dual Port RAM located on the remote bus adapter card is not included.

Transmitter: An adapter card that is allowed to initiate message transfers across the interface cable. There must always be at least one transmitter in any pair of adapter cards.

usec: Microsecond. 1/1,000,000 of a second.

Virtual Address: An address that references a location in a virtual address space.

Virtual Address Space: A contiguous range of virtual memory locations.

Virtual Memory: A facility whereby the effective range of addressable memory locations provided to a process is independent of the size of main memory. The virtual address space of a process is independent of the size and location of physical memory.

Window: A range of addresses that the adapter responds to for a specific function; a reserved area of main memory.

Word: 16 bits.

Appendix B: Conventions Used In This Manual

- File or directory names in the form ./filespec relate to the directory in which Support Software is installed. All files are located in a directory named for the software model and version number. For example, if version 2.0 of the software is installed in the /usr/local directory, the full path specification for the ./src directory is /usr/local/965/v2.0/src.

- name() denotes a function. For example, mmap() denotes a function named mmap. These functions may require an argument.

- _t indicates typedef; names a data structure.

- # indicates a system prompt.

- | indicates exclusive or, choose exactly one option from the list.

- All numbers use C programming language conventions for denoting radix. A leading non-zero digit indicates decimal. A leading 0 indicates octal. A leading 0x indicates hexadecimal.

Appendix C: ioctl() Summary

Appendix C is a list of all ioctl() commands supported by the btpdd device driver. All ioctl() commands and structure definitions are declared in the <sys/btpio.h> file.

The device driver supports the following ioctl() commands:

General User Commands

BIOC_SETUP

FUNCTION	Restores the PCI adapter card, remote bus adapter card, and device driver to a known (default) state and returns the status of the adapter.
ARGUMENT	bt_status_t

BIOC_STATUS

FUNCTION	Returns the status of the device driver.
ARGUMENT	bt_status_t

BIOC_CLR_STATUS

FUNCTION	Returns and clears the status of the device driver.
ARGUMENT	bt_status_t

BIOC_IOREG

FUNCTION	Allows reads and writes to an adapter Node I/O register; however, user privilege is required to perform writes.
ARGUMENT	bt_ioaccess_t

BIOC_BIND

FUNCTION	Binds the user buffer to the I/O bus allowing remote bus devices to directly read from or write to the buffer.
ARGUMENT	bt_bind_t

BIOC_UNBIND

FUNCTION	Unbinds the user buffer from the I/O bus so that further remote bus device access to the buffer is invalid.
ARGUMENT	bt_bind_t

BIOC_RESET

FUNCTION	Performs a system reset on the remote bus, if configured, and performs a BIOC_SETUP returning the status of the adapter.
ARGUMENT	bt_status_t

Atomic Transactions

BIOC_TAS

FUNCTION	Provides an atomic Test And Set operation either on the remote bus or to remote Dual Port RAM.
ARGUMENT	bt_tas_t

BIOC_CAS

FUNCTION	Provides an atomic Compare And Swap either on the remote bus or to remote Dual Port RAM.
ARGUMENT	bt_cas_t

Interrupt Management

BIOC_THREAD_REG

FUNCTION	Registers an ICBR thread with the driver. This process creates a unique ID for the thread and places the thread on the thread list.
ARGUMENT	bt_thread_reg_t

BIOC_THREAD_UNREG

FUNCTION	Removes the given thread from the thread list and destroys any associated resources.
ARGUMENT	bt_thread_reg_t

BIOC_THREAD_ADD

FUNCTION	Increments the count for the given thread for the given interrupt type.
ARGUMENT	bt_thread_add_t

BIOC_THREAD_DELETE

FUNCTION	Decrements the count for the given thread for the given interrupt type.
ARGUMENT	bt_thread_add_t

BIOC_THREAD_WAIT

FUNCTION	Waits on the thread_event for the next interrupt.
ARGUMENT	bt_thread_wait_t

BIOC_THREAD_WAKE

FUNCTION	Wakes an ICBR thread so it can be canceled.
ARGUMENT	bt_thread_wait_t

BIOC_SND_INTR

FUNCTION	Sends a programmed interrupt to the remote bus.
ARGUMENT	void

Control and Configuration

BIOC_SET_PRIV

FUNCTION	Disables privilege checking in the device driver.
ARGUMENT	void

BIOC_CLR_PRIV

FUNCTION	Restores privilege checking in the device driver.
ARGUMENT	void

BIOC_PARAM

FUNCTION	Modifies several internal driver parameters.
ARGUMENT	bt_param_t

BIOC_DEV_ATTRIB

FUNCTION	Returns values for many device driver internal parameters.
ARGUMENT	bt_param_t

BIOC_LOG_ERROR

FUNCTION	Logs to the system error log any status error interrupts that occur.
ARGUMENT	void

BIOC_NOLOG_ERROR

FUNCTION	Discontinues logging status error interrupts to the system error log. The device driver continues to log other errors. This only affects the logging of status error interrupts.
ARGUMENT	void

BIOC_CFG

FUNCTION	Configuration routine used to access PCI configuration registers. *To be used **only** by SBS personnel for testing and debugging.*
ARGUMENT	bt_ioaccess_t

Device Access Control Commands

BIOC_LOCK

FUNCTION	Always returns success.
ARGUMENT	bt_lock_t

BIOC_UNLOCK

FUNCTION	Always returns success.
ARGUMENT	none

Hardware Access Routines

BIOC_HW_READ

FUNCTION	Read data from a given bus address to a given logical device.
ARGUMENT	bt_hw_xfer_t

BIOC_HW_WRITE

FUNCTION	Write data to a given bus address from a given logical device.
ARGUMENT	bt_hw_xfer_t

BIOC_HW_BIND

FUNCTION	Binds a given bus address to the adapter so the remote system can access it.
ARGUMENT	bt_bind_t

BIOC_HW_UNBIND

FUNCTION	Unbinds a given bus address previously bound.
ARGUMENT	bt_bind_t

Semaphore Routines

BIOC_SEMA_TAKE

FUNCTION	Get a semaphore for the application.
ARGUMENT	bt_sema_access_t

BIOC_SEMA_GIVE

FUNCTION	Release a semaphore taken by an application.
ARGUMENT	bt_sema_access_t

Appendix D: Kernel Functions

bt_rembus_install() (Kernel Mode only)

FUNCTION	Registers a kernel-level interrupt handler to the device driver. Available in Kernel Mode only.
PROTOTYPE	int bt_rembus_install (btp_dev_t device, bt_rembus_intr_t *handler_p);
ARGUMENTS	device = Opaque type describing device. handler_p = Pointer to the bt_rembus_intr_t structure that describes the handler to be installed.

bt_rembus_remove() (Kernel Mode only)

FUNCTION	Removes the kernel interrupt registration from the device driver lookup table. Available in Kernel Mode only.
PROTOTYPE	int bt_rembus_remove (btp_dev_t device, bt_rembus_intr_t *handler_p);
ARGUMENTS	device = Opaque type describing device. handler_p = Pointer to the bt_rembus_intr_t structure that describes the handler to be removed.

bt_kmap() (Kernel Mode only)

FUNCTION	Returns information necessary to access the adapter from the interrupt context. Available in Kernel Mode only.
PROTOTYPE	int bt_kmap (btp_dev_t device, bt_kmap_t *kmap_p);
ARGUMENTS	device = Opaque type describing device. kmap_p = Pointer to the bt_kmap_t structure.

bt_kunmap() (Kernel Mode only)

FUNCTION	Releases the resource allocated by the bt_kmap() call. Available in Kernel Mode only.
PROTOTYPE	int bt_kunmap (btp_dev_t device, bt_kmap_t *kmap_p);
ARGUMENTS	device = Opaque type describing device. kmap_p = Pointer to the bt_kmap_t structure.

Appendix E: DMA Operation

When the device driver receives a request for a read() or write() to the remote bus, the length of that request is checked. If the length is greater than or equal to a user defined number of bytes, the device driver transfers the data using the PCI adapter card's DMA engine.

DMA is performed automatically during the read() or write() function and in no other function.

Several parameters can be adjusted or switched to change the default transfer mode:

- DMA_ADDR_MOD: Address modifier can be changed so that VMEbus Block Mode devices can be serviced.
- THRESHOLD: DMA threshold can be changed from 0 on up.
- DMA_POLL_SIZE: DMA poll size can be changed upwards from 0.
- DMA_PAUSE: DMA pause can be set to pause after 16 transfers.
- DATA_SIZE: DMA will always be attempted using 32-bit quantities unless the DATA_SIZE parameter restricts it to 16- or 8-bit quantities. When DATA_SIZE is set to DATA16_SIZ, the PCI adapter uses 16-bit quantities for the DMA. When set to DATA8_SIZ, the PCI adapter does PIO only.

➔ Make sure the memory mapped pointers are not dereferenced or remote devices do not access PCI memory during a DMA. Use BIOC_LOCK and BIOC_UNLOCK to serialize these activities with the driver's DMA feature.

If 32-bit transfers are requested but the source and destination buffers cannot be aligned to a 4-byte boundary, the driver attempts 16-bit transfers. If 16-bit transfers cannot be aligned to a 2-byte boundary, the driver transfers in byte quantities.

Defaults:

TRANSFER METHOD	32-bit values
ADDRESS MODIFIER	Determined by the logical device in use
THRESHOLD VALUE	Defaults to start DMA process when a transfer is longer than or equal to 160 bytes

Index

C