VxWorks Device Driver Guide for the Datum bc635/637 PCI Time Frequency Processor Card

Document No. VS-DRV-DATUMBC635
(for software release 1.0+)

October 12, 2000
Release 1.1

Revision History

Manual Release Level	Release Date	Comments
1.0	5/28/00	Original Release
1.1	10/16/00	Added sample code to ch6

1.0 Preface

Scope

This document describes the use of the Datum VxWorks driver for the bc635 PCI/PMC card. For information on the underlying VxWorks I/O System (IOS), refer to the VxWorks Programmer's Reference Guide.

Related Documents

The following table lists the related documents referred to in this manual.

Title	Author	Date
VxWorks Programmer's Guide (Release 5.3.1, 5.4)	Wind River Systems	1998
bc635/bc637 PCI/CPCI/PMC Time & Frequency Processor User's Guide (rev E)	Datum	1999

2.0 Overview

Product
This driver controls the PCI bus version of the bc635 Time/Frequency Processor (TFP) as well as the bc637 (GPS option). This driver operates under the VxWorks Operating System and is compatible with VxWorks 5.3.1 as well as VxWorks 5.4.

The bc635/637 TFP supportst he following features.

Time-of-day
Time-of-day may be captured on-demand where the time reference source may be:

1. **Time Code** - IRIG B, IRIG A or IEEE-1344
2. **Free Running** - The internal 10Mhz oscillator can keep time if the chosen time reference is lost.
3. **1 PPS** - External 1 Pulse Per Second input in conjunction with the internal 10Mhz oscillator.
4. **RTC** - Battery backed on-board real-time clock.
5. **GPS** - GPS time reference source.
   Note: GPS as time reference source is supported by hardware version bc635/637 PCI – 12083 only.

Time may be read and set using decimal time (eg., unix time using *struct tm* (located in time.h)) or binary time (eg., unix time using *struct timeval* (located in sys/times.h)).

Flywheel Operation
If the time reference is lost by the TFP, then the internal 10Mhz oscillator will continue to keep time based on the last valid time reference.

External Event Timestamping
If an external event occurs, the TFP can be programmed to timestamp it and return the timestamp to the application.

Time Coincidence Strobe
The TFP may be programmed to generate a single strobe pulse of a 1 PPS strobe pulse at some later time. This feature is typically used to implement an alarm function.

Periodic or Heartbeat Pulse Train Generation
The TFP can be programmed to generate a user-specified pulse train (also known as *periodic* or heartbeat).

Hardware Configuration

There are no hardware configuration jumpers associated with this card

Register I/O and Interrupts

This driver accesses the bc635/637 using 32 bit register reads and writes using standard VxWorks PCI bus access routines. The bc635/637 does generate interrupts and this driver processes those interrupts.

3.0 Using the Device Driver Interface

This section describes how to initialize and use the bc635 PCI card under the VxWorks IOS. The IOS calls used to invoke card operations are *open, close and ioctl*. Before continuing on with this manual, the user should be familiar with the basic interface to these functions which are generically described in the VxWorks Programmer's Reference Guide.

Creating bc635 VxWorks IOS Devices

The following sequence of vxWorks calls will install a single instance of a bc635 PCI card into the VxWorks IOS. Once this is accomplished the user's application may invoke open, close, ioctl calls to operate the bc635 TFP (Time/Frequency Processor):

```
-> ld < sysBC635.o
-> ld < bc635.o
-> sysBC635Init
-> tfpDrv
-> tfpDevInstall (0, 0x19)
```

In the above sequence, the object files sysBC635.o and bc635.o are loaded into RAM by the vxWorks ‘ld' function. This is a one time only operation per boot of VxWorks.

The *sysBC635Init* function is a BSP specific routine that searches for a single instance of a PCI bc635 device and programs the PCI base address that the card is to respond to. This is one time only operation per boot of VxWorks. This routine uses BSP based PCI programming routines to program the PCI Base Address Register 0 (BAR0) address for *0xFD040000*. It also programs the PCI Base Address Register 1 (BAR1) address as *0xFD041000*. If these addresses are not applicable to your BSP, then an alternative routine(s) must be used to program the *BAR0*/*BAR1* addresses for your card. This routine has been tailored for the Motorola MV2600/2700 boards. If these cards are not in use then this routine may be used as a guide for creation of a routine that will function for your specific BSP.

Also, your BSP may automatically program *BAR0/BAR1* addresses for all PCI devices installed. If so, you need not call a routine like *sysBC635Init*. Under the BSP in use, the *BAR0/BAR1* addresses used may be addresses as seen from the PCI bus and not the CPU. The bc635 driver must view the card addresses as seen from the CPU. If the BSP in use uses a fixed offset to view PCI addresses from the CPU, then the global variable *bc635PciMemOffset* is to be used to account for this. If you need to create a *sysBC635Init* routine specific to your BSP, then you must set this variable. Note that under the Motorola MV2604/2700, this value is 0.

The *tfpDrv* routine initializes the TFP (Time/Frequency Processor) VxWorks driver into the VxWorks IOS. This is a one time only operation per boot of vxWorks.

The *tfpDevInstall* routine initializes/installs a specific instance of a bc635 PCI card into the VxWorks IOS. The first argument is the instance number of the card to install. The second argument is the interrupt number assigned to the card. Refer to your VxWorks Board Support Packages (BSP) documentation for information on what interrupt numbers

to use for PCI devices. If there are two physical bc635 cards installed, then this routine will be called twice. For example,

```
-> tfpDevInstall (0, 0x19)
-> tfpDevInstall (1, 0x23)
```

These two calls will initialize the two cards (instance ‘0' and instance ‘1') into the VxWorks IOS where the cards will issue interrupts on interrupt numbers 0x19 and 0x23 respectively. The VxWorks IOS device names created for these cards will be “/tfp0” and “/tfp1” respectively.

Table 1: bc635 PCI VxWorks Driver *tfpDevInstall* Parameters	
index	integer (0 - 5), Specific bc635 PCI device to initialize into the IOS.
interrupt number	integer, Interrupt number that the card will use to issue interrupts to VxWorks.
return	OK(0) if initialization is successful. ERROR if unsucessful.

The following example initializes a bc635 PCI (PMC) card that is installed on an MVME2604 PowerPC VME Bus based Single Board Computer. When installed in this cards lone PMC site, the interrupt number the bc635 card will use is hard-wired to 0x19. It is a matter of reading your VxWorks BSP documentation so as to discover what interrupt numbers are to be used.

```
Host CPU: mvme2604 running VxWorks 5.4.

PCI Bus Base Address of card:
Interrupt number:             0x19
VxWorks IOS device name:      /tfp0

-> tfpDevInstall (0, 0x19)
```

Deleting a VxWorks IOS device

To delete a bc635 PCI VxWorks device, there are a couple of vxWorks system calls to invoke.
An example invocation that results in "/tfp0" device being removed from the IOS is:

```
-> tail=0
-> phdr=iosDevFind("/tfp0",&tail)
-> iosDevDelete(phdr)
```

You may now re-invoke tfpDevInstall so as to create a "/tfp0" device without having to reboot vxWorks.

Opening/Closing a bc635 IOS

The function *open* is used to obtain an IOS handle for a bc635 device. This handle is then used when invoking ioctl, close functions. To close the device after access has been completed, *close* must be called. Examples of these calls are listed below. The second and third arguments to the open function are not used by this driver.

```
int fd;

 /* open bc635 device */
if ((fd = open (“/tfp0”, O_RDWR, 0)) == ERROR)
     {
     (void) printf (“open failed: “);
     printErrno (errnoGet ());
     return (ERROR);
     }

< perform ioctl operations here>

 /* close bc635 device */
(void) close (fd);
```

Read and Write Operations

The IO functions *read* and *write* are not supported by this driver.

I/O Controls

The IOS routine *ioctl* is used to perform all user accessible TFP operations. What follows are usage descriptions of each I/O control operation.

SELTIMINGMODE

COMMAND: SELTIMINGMODE

PURPOSE: Select time derivation source.

INPUTS:

Type	Name/Description
int	modeVal - Mode that time is based on.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This operation will select the timing mode that the card is to use when determining time.

The following values for modeVal may be used::

```
MODE_TIMECODE(0)
MODE_FREERUN(1)
MODE_EXT1PPS(2)
MODE_RTC(3)
MODE_GPS(6)
```

The *MODE_TIMECODE* is to be set when IRIG-A, IRIG-B, IEEE-1344 is to be used as a time source.

The *MODE_FREERUN* is to be set when the cards on-board 10Mhz oscillator is used as a reference.

The *MODE_EXT1PPS* is to be set when an external device is providing a 1 pulse-per-second pulse via the external 1PPS input line.

The *MODE_RTC* is to be set when the on-board battery backed real-time clock is to be used as the time synchronization source.

The *MODE_GPS* is to be set when time is to be derived via GPS (Global Positioning System). For this mode to operate you must be using a bc637 card and have a GPS antenna attached.

The card will synchronize its internal 10Mhz oscillator to the time reference (mode) selected. It does this by extracting a 1 PPS from the time reference. If synchronization is lost once achieved, then the TFP is in *flywheel* mode and the internal 10Mhz oscillator is solely used to derive time.

EXAMPLE: The following example code performs the *SELTIMINGMODE ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source:

:

```
#include “vxWorks.h”
#include “stdio.h”
```

```
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf (“ioctl(SELTIMINGMODE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf (“ioctl(SELTCFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
```

```
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf ("ioctl(SELTCMOD) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
    if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
        {
        (void) printf ("ioctl(SETPROPDELAY) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

        return (ERROR);
        }

/* ready to retrieve time using time retrieval ioctls */

}/*end of test() */
```

SELTCFORMAT

COMMAND: SELTCFORMAT

PURPOSE: Select the time code format.

INPUTS:

Type	Name/Description
int	fmtVal - Timecode format to use.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This operation will select the timecode format that the card is to use when the timing reference (mode) selected is *MODE_TIMECODE (IRIG-A, IRIG-B or IEEE-3144)*.

The following values for fmtVal may be used::

```
TC_IRIGA
TC_IRIGB
TC_IEEE
```

The *TC_IRIGA* is to be set when *IRIG-A* is the expected timecode input.

The *TC_IRIGB* is to be set when *IRIG-B* is the expectied timecode input.

The *TC_IEEE* is to be set when *IEEE-1344* is the expected timecode input.

This ioctl need only be called if *MODE_TIMECODE* had been set as the timing mode using the *ioctl:SELTIMINGMODE*.

EXAMPLE: The following example code performs the *SELTCFORMAT ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source:

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
```

```
        {
        (void) printf ("open of %s failed: ", "/bc635/0");
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf ("ioctl(SELTIMINGMODE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf ("ioctl(SELTCFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf ("ioctl(SELTCMOD) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
```

```
    if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
        {
        (void) printf ("ioctl(SETPROPDELAY) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

        return (ERROR);
        }

/* ready to retrieve time using time retrieval ioctls */

}/*end of test() */
```

SELTIMEFORMAT

COMMAND: SELTIMEFORMAT

PURPOSE: Select how time is formatted when delivered to the caller.

INPUTS:

Type	Name/Description
int	fmtVal - Time format used when reading time.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This operation will select the time format that the card will use when delivering the current time to the caller via subsequent ioctls. Once set, the caller must set and retrieve time using the format selected.

The following values for fmtVal may be used::

```
TIME_DECIMAL
TIME_BINARY
```

The *TIME_DECIMAL* format results in a time breakdown of the following:

```
seconds (0 - 59)
minutes (0 - 59)
hours (0 - 23)
days (0 - 366
year (1970 - 2050)
usecs (0 - 999,999)
hnsecs (hunreds of nsecs (0 - 9,999,999)
```

If *TIME_DECIMAL* is selected, then you must use the *ioctl:RDTIMETM* or *ioctl:RDTIMETMREQ* to retrieve time and the *ioctl:SETTIMETM* to set time.

The *TIME_BINARY* format results in time representation as the number of seconds from midnight, January 1, 1970 (UTC). If this time format is selected, then you must use the *ioctl:RDTIMETV* or *ioctl:RDTIMETVREQ* to retrieve time and the *ioctl:SETTIMETV* to set time.

EXAMPLE: The following example code performs the *SELTIMEFORMAT ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source and to set/read time in the *TIME_DECIMAL* format.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
```

```
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf (“ioctl(SELTIMINGMODE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf (“ioctl(SELTCFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
```

```
            {
            (void) printf ("ioctl(SELTCMOD) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        /* assume no(0) time offset to compensate for */
        if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
            {
            (void) printf ("ioctl(SETPROPDELAY) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

            return (ERROR);
            }

    /* ready to retrieve time using time retrieval ioctls */

    }/*end of test() */
```

SELTCMOD

COMMAND: SELTCMOD

PURPOSE: Select timecode modulation type on the timecode input.

INPUTS:

Type	Name/Description
int	modVal - Timecode modulation format to expect.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This operation will tell the card what timecode modulation type to expect when the timing mode selected is MODE_TIMECODE(IRIG-A, IRIG-B, IEEE1344).

The following values for modVal may be used::

```
MOD_AM
MOD_DCLS
```

The *MOD_AM* is to be selected when an *amplitude modulated sine wave* is used.

The *MOD_DCLS* is to be selected when a *DC level shift* type of modulation is expected

EXAMPLE: The following example code performs the *SELTCMOD ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source and to set *amplitude modulated sine wave* as the modulation type.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
```

```
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf ("ioctl(SELTIMINGMODE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf ("ioctl(SELTCFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf ("ioctl(SELTCMOD) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
    if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
        {
        (void) printf ("ioctl(SETPROPDELAY) failed: ");
```

```
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

            return (ERROR);
            }

    /* ready to retrieve time using time retrieval ioctls */

    }/*end of test() */
```

SETPROPDELAY

COMMAND: SETPROPDELAY

PURPOSE: Set propagation delay compensation.

INPUTS:

Type	Name/Description
int	propVal - Value in usecs to offset derived time. Value may be negative.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This operation may be used to set a time offset relative to the reference input. If there is a known time delay due to cable length between the time reference generator and the TFP, this ioctl is used to take that delay into account when the TFP is to derive time. The value is set in terms of microseconds and has a range of -9,999,999 to 9,999,999 usecs.

EXAMPLE: The following example code performs the *SETPROPDELAY ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source. The propagation delay in this example is -15 usecs.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
```

```
        (void) printf (“ioctl(SELTIMINGMODE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf (“ioctl(SELTCFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf (“ioctl(SELTCMOD) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
    if (ioctl (fd, SETPROPDELAY, -15) == ERROR)
        {
        (void) printf (“ioctl(SETPROPDELAY) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }
```

```
/* ready to retrieve time using time retrieval ioctls */

}/*end of test*/
```

RDTIMETMREQ, RDTIMETM

COMMAND: RDTIMETMREQ, RDTIMETM

PURPOSE: Read decimal time from the TFP.

INPUTS:

Type	Name/Description
TFP_TM *	*pTfpTm* - Pointer to *TFP_TM* structure

RETURNS: The *ioctl* returns *OK* or *ERROR*. Also on return the *pTfpTm* structure is filled with current time data.

DESCRIPTION: This ioctl is to be called when the *TIME_DECIMAL* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

The *ioctl:RDTIMETMREQ* will return *latched* time. Before reading the *TFP* time registers, time is latched by the *TFP* into these registers.

The *ioctl:RDTIMETM* will return *non-latched* time. With this form, whatever time the time registers currently hold are returned. The *ioctl:TIMEREQUEST* can be called before this ioctl to specifically latch the time.

This ioctl is used to read the current time from the *TFP*. The caller passes the address of a *TFP_TM* structure defined in bc635.h. The driver will then fill this structure with the current time and return. Embedded in this structure is the UNIX time structure called *tm*. The *tm* structure is defined in the *vxWorks* file time.h and is a standard structure for holding time of day in unix. The values contained in the *tm* structure is called major time by the *TFP*. The minor time values of microseconds and hundreds of nanoseconds are also returned in the *TFP_TM* structure.

The following is the definition of the *TFP_TM* structure:

```
typedef struct
   {
   struct tm tm;
   UINT32 usec;
   UINT32 hnsec;
   } TFP_TM;
```

The vxWorks defined definition of the *tm* structure is:

```
struct tm
    {
    int tm_sec;     /* seconds after the minute     -
[0, 59] */
    int tm_min;     /* minutes after the hour       -
[0, 59] */
```

```
                        int tm_hour;    /* hours after midnight         -
[0, 23] */
                        int tm_mday;    /* day of the month             -
[1, 31] */
                        int tm_mon;     /* months since January         -
[0, 11] */
                        int tm_year;    /* years since 1900     */
                        int tm_wday;    /* days since Sunday            -
[0, 6] */
                        int tm_yday;    /* days since January 1         -
[0, 365] */
                        int tm_isdst;   /* Daylight Saving Time flag */
                        };
```

On return from this call, the *tm* structure gives the time-of-day. In addition, the *usec* and *hnsec* members give current number of microseconds and hundreds of nanoseconds.

The following members of the tm structure are used:

```
tm_sec - range: 0 - 59
tm_min - range: 0 - 59
tm_hour - range: 0 - 23
tm_yday - range: 0 - 365
tm_year - range: 1970 - 2050
```

Note that the *tm_year* member has a range different than the standard unix range listed above. The other members of the unix tm strcture are not filled in by this ioctl call.

The status member contains 3 bits that indicate the following status conditions:

```
Bit 0: flywheeling
Bit 1: Time offset
Bit 2: Frequency offset
```

Bit *#0* indicates the *flywheeling* state. If bit 0 of *status* is set, then the *TFP* is not tracking the time reference source. If Bit 0 is not set, then the *TFP* is tracking the time reference.

Bit *#1* is indicates the synchonization accuracy of the *TFP* relative to the timing source. This bit is updated approximately once per second. When the *TFP's* 10Mhz oscillator is synchronized to less than 5 microseconds in time code mode *(e.g., mode = MODE_TIMECODE)* and 2 microseconds in other modes, this bit is cleared.

Bit *#2* is an indication of the *TFP* on-board oscillator frequency offset relative to the timing source. This bit is updated approximately once per second and reflects the short-term stability of the *TFP's* oscillator.

EXAMPLE:

The following example code performs the *RDTIMETMREQ ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source and to set/read time in the *TIME_DECIMAL* format. The ioctl described here is then called and the current time displayed.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
```

```
#include “time.h”
#include “bc635.h”

void test ()
    {
    int fd;
     TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf (“ioctl(SELTIMINGMODE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf (“ioctl(SELTCFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }
```

```
    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf ("ioctl(SELTCMOD) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
    if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
        {
        (void) printf ("ioctl(SETPROPDELAY) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

        return (ERROR);
        }

    /* ready to retrieve time in 'decimal' form */
    if (ioctl (fd, RDTIMETMREQ, &tfpTM) == ERROR)
        {
        (void) printf ("ioctl(RDTIMETMREQ) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* use unix derived time display function to show time
*/
    (void) printf ("%s\n", asctime (&tfpTm.tm));

}/*end of test() */
```

RDTIMETVREQ, RDTIMETV

COMMAND: RDTIMETVREQ, RDTIMETV

PURPOSE: Read binary time from the TFP.

INPUTS:

Type	Name/Description
TFP_TV *	*pTfpT*v - Pointer to *TFP_T*V structure

RETURNS: The *ioctl* returns *OK* or *ERROR*. Also on return the *pTfpT*v structure is filled with current time data.

DESCRIPTION: This ioctl is to be called when the *TIME_BINARY* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

The *ioctl:RDTIMETVREQ* will return *latched* time. Before reading the *TFP* time registers, time is latched by the *TFP* into these registers.

The *ioctl:RDTIMET*V will return *non-latched* time. With this form, whatever time the time registers currently hold are returned. The *ioctl:TIMEREQUEST* can be called before this ioctl to specifically latch the time.

This ioctl is used to read the current time from the *TFP*. The caller passes the address of a *TFP_T*V structure defined in bc635.h. The driver will then fill this structure with the current time and return. Embedded in this structure is the UNIX time structure called *t*v. The *tv* structure is defined in the *vxWorks* file time.h and is a standard structure for holding time of day in unix. The values contained in the *t*v structure is called major binary time by the *TFP*. The minor time values of microseconds and hundreds of nanoseconds are also returned in the *TFP_T*V structure.

The following is the definition of the *TFP_T*V structure:

```
typedef struct
   {
   struct timeval tv;
   UINT32 hnsec;
   UINT8 status;
   } TFP_TV;
```

The vxWorks defined definition of the *t*v structure is:

```
struct timeval tv
    {
    time_t tv_sec;
    long tv_usec;
    };
```

The *tv_sec* member is the number of seconds from Jan 1, 1970. The tv_usec member is the number of microseconds. The hnsec member gives the number of nanoseconds in terms of hundreds of nanoseconds. The status member contains 3 bits that indicate the following status conditions:

```
Bit 0: flywheeling
Bit 1: Time offset
Bit 2: Frequency offset
```

Bit *#0* indicates the *flywheeling* state. If bit 0 of *status* is set, then the *TFP* is not tracking the time reference source. If Bit 0 is not set, then the *TFP* is tracking the time reference.

Bit *#1* is indicates the synchonization accuracy of the *TFP* relative to the timing source. This bit is updated approximately once per second. When the *TFP's* 10Mhz oscillator is synchronized to less than 5 microseconds in time code mode *(e.g., mode = MODE_TIMECODE)* and 2 microseconds in other modes, this bit is cleared.

Bit *#2* is an indication of the *TFP* on-board oscillator frequency offset relative to the timing source. This bit is updated approximately once per second and reflects the short-term stability of the *TFP's* oscillator.

EXAMPLE: The following example code performs the *RDTIMETVREQ ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source and to set/read time in the *TIME_BINARY* format. The ioctl described here is then called and the current time displayed.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     TFP_TV tfpTv;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* want to set/retrieve time in binary format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_BINARY) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
```

```
        (void) close (fd);
        return (ERROR);
        }


    /* ready to retrieve time in ‘binary' form */
    if (ioctl (fd, RDTIMETVREQ, &tfpTv) == ERROR)
        {
        (void) printf (“ioctl(RDTIMETVREQ) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* display number of seconds since Jan. 1, 1970. */
    (void) printf (“Number of seconds since Jan. 1, 1970:
%d\n”,
                   tfpTv->tv.tv_sec);

}/*end of test() */
```

TIMEREQUEST

COMMAND: TIMEREQUEST

PURPOSE: Latch TFP time registers.

INPUTS:

Type	Name/Description
int	Always 0/

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl latches the current time in the *TFP* to the time registers. The time is then held in these registers until the next call of this ioctl.

This ioctl can be used to read time in conjunction with *ioctl:RDTIMETM* or *ioctl:RDTIMETV*. See example below.

The ioctls *ioctl:RDTIMETMREQ* and *ioctl:RDTIMETVREQ* perform this function as well as return the current time.

EXAMPLE: The following example code performs the TIMEREQUEST *ioctl* in the context of completely configuring the driver and card to derive time using *IRIG-B* as the time reference source and to set/read time in the *TIME_DECIMAL* format.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “time.h”
#include “bc635.h”

void test ()
    {
    int fd;
     TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }
```

```
    /* select time reference (will use IRIG-B input) */
    if (ioctl (fd, SELTIMINGMODE, MODE_TIMECODE) == ERROR)
        {
        (void) printf ("ioctl(SELTIMINGMODE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card which type of timecode input to look for */
    if (ioctl (fd, SELTCFORMAT, TC_IRIGB) == ERROR)
        {
        (void) printf ("ioctl(SELTCFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* want to retrieve time in decimal format */
     if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* tell card what type of timecode modulation to expect
*/
    if (ioctl (fd, SELTCMOD, MOD_AM) == ERROR)
        {
        (void) printf ("ioctl(SELTCMOD) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* assume no(0) time offset to compensate for */
    if (ioctl (fd, SETPROPDELAY, 0) == ERROR)
        {
        (void) printf ("ioctl(SETPROPDELAY) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
```

```
            }

            return (ERROR);
            }

        /* latch time */
        if (ioctl (fd, TIMEREQUEST, 0) == ERROR)
            {
            (void) printf ("ioctl(TIMEREQUEST) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }


        /* ready to retrieve time in 'decimal' form */
        if (ioctl (fd, RDTIMETM, &tfpTM) == ERROR)
            {
            (void) printf ("ioctl(RDTIMETMREQ) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        /* use unix derived time display function to show time
*/
        (void) printf ("%s\n", asctime (&tfpTm.tm));

    }/*end of test() */
```

SETTIMETM

COMMAND: SETTIMETM

PURPOSE: Set the the time-of-day on the TFP using *decimal* format.

INPUTS:

Type	Name/Description
TFP_TM *	*pTfpTm* - Pointer to *TFP_TM* structure

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called when the *TIME_DECIMAL* time format had been selected using the *ioctl:SELTIMEFORMAT*.

This ioctl is used to set the current time for the *TFP*. All subsequent time retrievals using the *ioctl:RDTIMETMREQ* will be current relative to this the new time-of-day. The caller passes the address of a *TFP_TM* structure defined in bc635.h. This structure is to be filled by the caller. For information on the *TFP_TM* structure, refer to the manual section regarding the *ioctl:RDTIMETMREQ*. Note that the *status* member is not to be set by the caller.

The *tm_year* member set by the caller will not take effect until a power-cycle of system reset is performed on the *TFP*. This is a feature of the *TFP*. The *ioctl:SETYEAR* need not be called if this ioctl is used.

EXAMPLE: The following example code performs the *SETTIMETM ioctl*. Note that the ioctls used to configure the card are not called here. See the *ioctl:RDTIMETMREQ* for an example that configures the card completely.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
```

```
            return (ERROR);
            }

        /* want to set/retrieve time in decimal format */
         if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
            {
            (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        /* ready to set time in 'decimal' form */

        /* set this date */
        tfpTm.tm.tm_year = 2000;
        tfpTm.tm.tm_yday = 255;
        tfpTm.tm.tm_hour = 13;
        tfpTm.tm.tm_min = 33;
        tfpTm.tm.tm_sec = 48;

        if (ioctl (fd, SETTIMETM, &tfpTm) == ERROR)
            {
            (void) printf ("ioctl(SETTIMETM) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        (void) close (fd);

    }/*end of test() */
```

SETTIMETV

COMMAND: SETTIMETV

PURPOSE: Set the the time-of-day on the TFP using *binary* format.

INPUTS:

Type	Name/Description
TFP_TV *	*pTfpTv* - Pointer to *TFP_TV* structure

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called when the *TIME_BINARY* time format had been selected using the *ioctl:SELTIMEFORMAT*.

This ioctl is used to set the current time for the *TFP*. All subsequent time retrievals using the *ioctl:RDTIMETVREQ* will be current relative to this the new time-of-day. The caller passes the address of a *TFP_TV* structure defined in bc635.h. This structure is to be filled by the caller. For information on the *TFP_TV* structure, refer to the manual section regarding the *ioctl:RDTIMETVREQ*. Note that the *status* member is not to be set by the caller.

The *tm_year* member set by the caller will not take effect until a power-cycle of system reset is performed on the *TFP*. This is a feature of the *TFP*. The *ioctl:SETYEAR* need not be called if this ioctl is used.

EXAMPLE: The following example code performs the *SETTIMETV ioctl*. Note that the ioctls used to configure the card are not called here. See the *ioctl:RDTIMETMREQ* for an example that configures the card completely.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     TFP_TV tfpTv;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
```

```
            return (ERROR);
            }

        /* want to set/retrieve time in binary format */
         if (ioctl (fd, SELTIMEFORMAT, TIME_BINARY) == ERROR)
            {
            (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        /* ready to set time in 'binary' form */

        /* set the time */
        tfpTv.tv.tv_sec = 1,235,444;

        if (ioctl (fd, SETTIMETV, &tfpTv) == ERROR)
            {
            (void) printf ("ioctl(SETTIMETV) failed: ");
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        (void) close (fd);

    }/*end of test() */
```

SETYEAR

COMMAND: SETYEAR

PURPOSE: Set the the year for the TFP.

INPUTS:

Type	Name/Description
int	*yearVal* - Range is from 1970 - 2050.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called to set the year. The value is a decimal number in the range of 1970 to 2050.

Note that this value does not take effect on the TFP until a power-cycle or system reset is performed. This is a feature of the TFP..

EXAMPLE: The following example code performs the *SETYEAR ioctl*. Note that the ioctls used to configure the card are not called here. See the *ioctl:RDTIMETMREQ* for an example that configures the card completely.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     int yearVal;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }


    /* set year  to 2010 */
    yearVal = 2010;
```

```
    if (ioctl (fd, SETYEAR, yearVal) == ERROR)
        {
        (void) printf ("ioctl(SETYEAR) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    (void) close (fd);

}/*end of test() */
```

SETPERIODIC

COMMAND: SETPERIODIC

PURPOSE: Program the periodic output of the TFP.

INPUTS:

Type	Name/Description
struct periodic	*perStruct* - Values describing periodic output characteristics.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called so as to program the periodic output of the *TFP*. This is especially useful when a pulse train is desired of a certain pulse width and frequency.

The *periodic output* can be optionally synchronized to the *TFP 1pps* signal. The *1pps* synchronization works besst when the periodic output frequency is an integer value, otherwise, the *1pps* signal will cut short one of the periodic output cycles.

The following structure is to be filled by the caller:

```
struct periodic
   {
   UINT8 sync1pps;
   UINT16 n1;
   UINT16 n2;
   }
```

If the sync1pps value is set to 1, then the periodic output is synchronized to the TFP 1pps signal.

The n1, n2 values are determined according to the following description:

```
Terms:
  n1: Counter divider number 1
n2: Counter divider number 2
Duty Cycle: Percentage of high pulse width to
signal period

This signal is generated by dividing down a 1 Mhz
clock. The 1 Mhz clock is derived directly from the
10 Mhz oscillator, thus the periodic output is
synchronous with the timing source. The periodic
output frequency can range from 250 kHz (n1 = n2 = 2)
to less than 1 Hz, and is determined by the
relationship:
    Frequency = 1,000,000 / (n1 * n2) Hz
                (where 2 <= n1, n2 <= 65535)
```

```
Duty Cycle = (1 - (1 / n2)) * 100 %
```

Note: If n1 or n2 is set to 2, 1pps synchronization will not work correctly, though the *periodic output* frequency will be correct.

EXAMPLE: The following example code performs the *ioctl:SETPERIODIC* where a 100 Hz pulse train is formed with a 75% duty cycle.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     struct periodic pulseDescrip;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    pulseDescrip.sync1pps = PERIODIC_SYNC;   /* sync to 1
pps */
    pulseDescrip.n1 = 2500;
    pulseDescrip.n1 = 4;

    if (ioctl (fd, SETPERIODIC, &pulseDescrip) == ERROR)
        {
        (void) printf (“ioctl(SETPERIODIC) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    (void) close (fd);

}/*end of test() */
```

SELFREQUENCYOUT

COMMAND: SELFREQUENCYOUT

PURPOSE: Select the output frequency for the TFP's TTL output signal.

INPUTS:

Type	Name/Description
int	*freqOut* - Choose one of the following defines listed below.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called so as to select an output frequency for the TFP TTL output signal.

One of the following *freqOut* values is to be used:

```
FREQ_10MHZ - 10 Mhz output frequency selected.
FREQ_5MHZ - 5 Mhz output frequency selected.
FREQ_1MHZ - 1 Mhz output frequency selected.
```

EXAMPLE: The following example code performs the *ioctl:SELFREQ*UENCYOUT where a 5 Mhz output TTL signal frequency is selected.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     struct periodic pulseDescrip;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }
```

```
    if (ioctl (fd, SELFREQUENCYOUT, FREQ_5MHZ) == ERROR)
        {
        (void) printf ("ioctl(SETPERIODIC) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    (void) close (fd);

}/*end of test() */
```

CONTROLEVENT

COMMAND: CONTROLEVENT

PURPOSE: Set event time recording parameters.

INPUTS:

Type	Name/Description
int	*ctrlVal* - Choose one of the following defines listed below.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called so as to set a specific control parameter with regards to recording a timestamp of an event.

One of the following *defines* is to be set. This ioctl is to be called multiple times when setting multiple parameters:

```
EVENT_RISING
EVENT_FALLING
CAPLOCK_ENABLE
CAPLOCK_DISABLE
EVCAP_ENABLE
EVCAP_DISABLE
EVCAP_PERIODIC
EVCAP_EVENT
```

Set *EVCAP_ENABLE* to enable timestamp of events. Set *EVCAP_PERIODIC* or *EVCAP_EVENT* as well.

Set *EVCAP_DISABLE* to disable timestamp of events.

Set *EVCAP_EVENT* when detecting external events. You must set *EVENT_RISING* or *EVENT_FALLING* as well.

Set *EVCAP_PERIODIC* when the events to be timestamped are the periodic output of the *TFP*.

Set *EVENT_RISING* so as to detect an event on a rising edge.

Set *EVENT_FALLING* so as to detect an event on a falling edge.

Set *CAPLOCK_ENABLE* to enable event capture lockout so that the timestamp captured on the next event is not overwritten when new events occur. After the timestamp is read using one of the ioctls *RDEVENTTM* or *RDEVENTTV*, then the control parameter *CAPLOCK_DISABLE* can be set so as to re-enable event timestamping.

Set *CAPLOCK_DISABLE* to disable event capture lockout so that the timestamp captured on the next event will be overwritten if new events occur.

EXAMPLE: The following example code performs the *ioctl:CONTROLEVENT* so as to capture timestamps for external events (non TFP-periodic) on the falling edge. The event capture lockout parameter is set so that the subsequent timestamps read are for the most recent external event.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* read event time in ‘decimal' format */
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture timestamps for external events (non-tfp-
periodic) */
    if (ioctl (fd, CONTROLEVENT, EVCAP_EVENT) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture event on the falling edge */
    if (ioctl (fd, CONTROLEVENT, EVENT_FALLING) == ERROR)
```

```
    {
    (void) printf (“ioctl(CONTROLEVENT) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

 /* disable timestamp lockout mode
    (timestamps are for most recent events) */
if (ioctl (fd, CONTROLEVENT, CAPLOCK_DISABLE) == ERROR)
    {
    (void) printf (“ioctl(CONTROLEVENT) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* enable event capture */
if (ioctl (fd, CONTROLEVENT, EVCAP_ENABLE) == ERROR)
    {
    (void) printf (“ioctl(CONTROLEVENT) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

 /* wait for next event to occur */
 if (ioctl (fd, BC635_INT_WAITONEVENT, 0) == ERROR)
    {
    (void) printf (“ioctl(BC635_WAITONEVENT) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

 /* event occurred, now get timestamp */
 if (ioctl (fd, RDEVENTTMREQ, &tfpTm) == ERROR)
    {
    (void) printf (“ioctl(RDEVENTTMREQ) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }
```

```
    /* use unix derived time display function to show time
*/
    (void) printf ("%s\n", asctime (&tfpTm.tm));

    (void) close (fd);

}/*end of test() */
```

RDEVENTTMREQ, RDEVENTM

COMMAND: RDEVENTTMREQ, RDEVENTTM

PURPOSE: Read decimal time of an event from the TFP.

INPUTS:

Type	Name/Description
TFP_TM *	*pTfpTm* - Pointer to *TFP_TM* structure

RETURNS: The *ioctl* returns *OK* or *ERROR*. Also on return the *pTfpTm* structure is filled with current time data.

DESCRIPTION: This ioctl is to be called when the *TIME_DECIMAL* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

The *ioctl:RDEVENTTMREQ* will return *latched* event time. Before reading the *TFP* event time registers, event time is latched by the *TFP* into these registers.

The *ioctl:RDEVENTTM* will return *non-latched* event time. With this form, whatever event time the event time registers currently hold are returned. The *ioctl:EVENTREQUEST* can be called before this ioctl to specifically latch the time.

Before calling this ioctl, the ioctl:CONTROLEVENT must have been called to configure the TFP for event time processing.

This ioctl is used to read the current time for an event from the *TFP*. The caller passes the address of a *TFP_TM* structure defined in bc635.h. The driver will then fill this structure with the current event time and return. Embedded in this structure is the UNIX time structure called *tm*. The *tm* structure is defined in the *vxWorks* file time.h and is a standard structure for holding time of day in unix. The values contained in the *tm* structure is called major time by the *TFP*. The minor time values of microseconds and hundreds of nanoseconds are also returned in the *TFP_TM* structure.

The following is the definition of the *TFP_TM* structure:

```
typedef struct
   {
   struct tm tm;
   UINT32 usec;
   UINT32 hnsec;
   } TFP_TM;
```

The vxWorks defined definition of the *tm* structure is:

```
struct tm
    {
    int tm_sec;     /* seconds after the minute     -
[0, 59] */
```

```
                        int tm_min;     /* minutes after the hour       -
[0, 59] */
                        int tm_hour;    /* hours after midnight         -
[0, 23] */
                        int tm_mday;    /* day of the month             -
[1, 31] */
                        int tm_mon;     /* months since January         -
[0, 11] */
                        int tm_year;    /* years since 1900     */
                        int tm_wday;    /* days since Sunday            -
[0, 6] */
                        int tm_yday;    /* days since January 1         -
[0, 365] */
                        int tm_isdst;   /* Daylight Saving Time flag */
                        };
```

On return from this call, the *tm* structure gives the time-of-day. In addition, the *usec* and *hnsec* members give current number of microseconds and hundreds of nanoseconds.

The following members of the tm structure are used:

```
tm_sec - range: 0 - 59
tm_min - range: 0 - 59
tm_hour - range: 0 - 23
tm_yday - range: 0 - 365
tm_year - range: 1970 - 2050
```

Note that the *tm_year* member has a range different than the standard unix range listed above. The other members of the unix tm strcture are not filled in by this ioctl call.

The status member contains 3 bits that indicate the following status conditions:

```
Bit 0: flywheeling
Bit 1: Time offset
Bit 2: Frequency offset
```

Bit *#0* indicates the *flywheeling* state. If bit 0 of *status* is set, then the *TFP* is not tracking the time reference source. If Bit 0 is not set, then the *TFP* is tracking the time reference.

Bit *#1* is indicates the synchonization accuracy of the *TFP* relative to the timing source. This bit is updated approximately once per second. When the *TFP's* 10Mhz oscillator is synchronized to less than 5 microseconds in time code mode *(e.g., mode = MODE_TIMECODE)* and 2 microseconds in other modes, this bit is cleared.

Bit *#2* is an indication of the *TFP* on-board oscillator frequency offset relative to the timing source. This bit is updated approximately once per second and reflects the short-term stability of the *TFP's* oscillator.

EXAMPLE:

The following example code performs the *ioctl:RDEVENTTMREQ* so as to capture a timestamp for the next external event once event capture is enabled. The external events (non TFP-periodic) are on the falling edge. The event capture lockout parameter is set so that the subsequent timestamps read are for the most recent external event.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* read event time in ‘decimal' format */
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture timestamps for external events (non-tfp-
periodic) */
    if (ioctl (fd, CONTROLEVENT, EVCAP_EVENT) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture event on the falling edge */
    if (ioctl (fd, CONTROLEVENT, EVENT_FALLING) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
```

```
        }

     /* disable timestamp lockout mode
        (timestamps are for most recent events) */
    if (ioctl (fd, CONTROLEVENT, CAPLOCK_DISABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable event capture */
    if (ioctl (fd, CONTROLEVENT, EVCAP_ENABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

     /* wait for next event to occur */
     if (ioctl (fd, BC635_INT_WAITONEVENT, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_WAITONEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

     /* event occurred, now get timestamp */
     if (ioctl (fd, RDEVENTTMREQ, &tfpTm) == ERROR)
        {
        (void) printf ("ioctl(RDEVENTTMREQ) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

     /* use unix derived time display function to show event
time */
     (void) printf ("%s\n", asctime (&tfpTm.tm));

     (void) close (fd);
```

```
}/*end of test() */
```

RDEVENTTVREQ, RDEVENTTV

COMMAND: RDEVENTTVREQ, RDEVENTTV

PURPOSE: Read binary time for an event from the TFP.

INPUTS:

Type	Name/Description
TFP_TV *	*pTfpT*v - Pointer to *TFP_T*V structure

RETURNS: The *ioctl* returns *OK* or *ERROR*. Also on return the *pTfpT*v structure is filled with current time data.

DESCRIPTION: This ioctl is to be called when the *TIME_BINARY* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

The *ioctl:RDEVENTTVREQ* will return *latched* event time. Before reading the *TFP* event time registers, event time is latched by the *TFP* into these registers.

The *ioctl:RDEVENTT*V will return *non-latched* event time. With this form, whatever event time the event time registers currently hold are returned. The *ioctl:EVENTREQUEST* can be called before this ioctl to specifically latch the event time.

This ioctl is used to read the current event time from the *TFP*. The caller passes the address of a *TFP_T*V structure defined in bc635.h. The driver will then fill this structure with the current event time and return. Embedded in this structure is the UNIX time structure called *t*v. The *tv* structure is defined in the *vxWorks* file time.h and is a standard structure for holding time of day in unix. The values contained in the *t*v structure is called major binary time by the *TFP*. The minor time values of microseconds and hundreds of nanoseconds are also returned in the *TFP_T*V structure.

The following is the definition of the *TFP_T*V structure:

```
typedef struct
   {
   struct timeval tv;
   UINT32 hnsec;
   UINT8 status;
   } TFP_TV;
```

The vxWorks defined definition of the *t*v structure is:

```
struct timeval tv
    {
    time_t tv_sec;
    long tv_usec;
    };
```

The *tv_sec* member is the number of seconds from Jan 1, 1970. The tv_usec member is the number of microseconds. The hnsec member gives the number of nanoseconds in terms of hundreds of nanoseconds. The status member contains 3 bits that indicate the following status conditions:

```
Bit 0: flywheeling
Bit 1: Time offset
Bit 2: Frequency offset
```

Bit *#0* indicates the *flywheeling* state. If bit 0 of *status* is set, then the *TFP* is not tracking the time reference source. If Bit 0 is not set, then the *TFP* is tracking the time reference.

Bit *#1* is indicates the synchonization accuracy of the *TFP* relative to the timing source. This bit is updated approximately once per second. When the *TFP's* 10Mhz oscillator is synchronized to less than 5 microseconds in time code mode *(e.g., mode = MODE_TIMECODE)* and 2 microseconds in other modes, this bit is cleared.

Bit *#2* is an indication of the *TFP* on-board oscillator frequency offset relative to the timing source. This bit is updated approximately once per second and reflects the short-term stability of the *TFP's* oscillator.

EXAMPLE:

The following example code performs the *ioctl:RDEVENTTVREQ* in the process of reading a timestamp for the next external event once event capture is enabled. The external events (non TFP-periodic) are on the falling edge. The event capture lockout parameter is set so that the subsequent timestamps read are for the most recent external event.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TV tfpTv;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* read event time in ‘decimal' format */
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
```

```
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture timestamps for external events (non-tfp-
periodic) */
    if (ioctl (fd, CONTROLEVENT, EVCAP_EVENT) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture event on the falling edge */
    if (ioctl (fd, CONTROLEVENT, EVENT_FALLING) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* disable timestamp lockout mode
       (timestamps are for most recent events) */
   if (ioctl (fd, CONTROLEVENT, CAPLOCK_DISABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

   /* enable event capture */
   if (ioctl (fd, CONTROLEVENT, EVCAP_ENABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }
```

```
    /* wait for next event to occur */
    if (ioctl (fd, BC635_INT_WAITONEVENT, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_WAITONEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* event occurred, now get timestamp */
    if (ioctl (fd, RDEVENTTVREQ, &tfpTv) == ERROR)
        {
        (void) printf ("ioctl(RDEVENTTVREQ) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* display number of seconds since Jan. 1, 1970 for this
event */
    (void) printf ("Number of seconds since Jan. 1, 1970:
%d\n",
                   tfpTv->tv.tv_sec);

    (void) close (fd);

}/*end of test() */
```

EVENTREQUEST

COMMAND: EVENTREQUEST

PURPOSE: Latch TFP event time registers.

INPUTS:

Type	Name/Description
int	Always 0.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl latches the current event time in the *TFP* to the time registers. The event time is then held in these registers until the next call of this ioctl.

This ioctl can be used to read time in conjunction with *ioctl:RDEVENTTM* or *ioctl:RDTEVENTTV*. See example below.

The ioctls *ioctl:RDVENTTMREQ* and *ioctl:RDEVENTTVREQ* perform this function as well as return the current time.

EXAMPLE: The following example code performs the *ioctl:EVENTREQUEST* in the process of reading a timestamp for the next external event once event capture is enabled. The external events (non TFP-periodic) are on the falling edge. The event capture lockout parameter is set so that the subsequent timestamps read are for the most recent external event.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }
```

```
    /* read event time in ‘decimal' format */
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture timestamps for external events (non-tfp-
periodic) */
    if (ioctl (fd, CONTROLEVENT, EVCAP_EVENT) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture event on the falling edge */
    if (ioctl (fd, CONTROLEVENT, EVENT_FALLING) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* disable timestamp lockout mode
       (timestamps are for most recent events) */
   if (ioctl (fd, CONTROLEVENT, CAPLOCK_DISABLE) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

   /* enable event capture */
   if (ioctl (fd, CONTROLEVENT, EVCAP_ENABLE) == ERROR)
        {
        (void) printf (“ioctl(CONTROLEVENT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
```

```
        return (ERROR);
        }

    /* wait for next event to occur */
    if (ioctl (fd, BC635_INT_WAITONEVENT, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_WAITONEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* latch TFP event time registers */
    if (ioctl (fd, EVENTREQUEST, 0) == ERROR)
        {
        (void) printf ("ioctl(EVENTREQUEST) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* read TFP event time registers */
    if (ioctl (fd, RDEVENTTM, &tfpTm) == ERROR)
        {
        (void) printf ("ioctl(RDEVENTTM) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* use unix derived time display function to show event
time */
    (void) printf ("%s\n", asctime (&tfpTm.tm));

    (void) close (fd);

}/*end of test() */
```

BC635_INT_WAITONEVENT

COMMAND: BC635_INT_WAITONEVENT

PURPOSE: Wait for event interrupt to occur.

INPUTS:

Type	Name/Description
int	Always 0

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called after the *ioctl:CONTROLEVENT* has been called to configure the *TFP* for event processing and after event capture has been enabled using *ioctl:CONTROLEVENT*.

This ioctl will block waiting for an event to occur as configured by the *ioctl:CONTROLEVENT*.

EXAMPLE: The following example code performs the *ioctl:BC635_INT_WAITONEVENT* in the process of reading a timestamp for the next external event once event capture is enabled. The external events (non TFP-periodic) are on the falling edge. The event capture lockout parameter is set so that the subsequent timestamps read are for the most recent external event.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* read event time in ‘decimal' format */
```

```
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture timestamps for external events (non-tfp-
periodic) */
    if (ioctl (fd, CONTROLEVENT, EVCAP_EVENT) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* capture event on the falling edge */
    if (ioctl (fd, CONTROLEVENT, EVENT_FALLING) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* disable timestamp lockout mode
       (timestamps are for most recent events) */
   if (ioctl (fd, CONTROLEVENT, CAPLOCK_DISABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

   /* enable event capture */
   if (ioctl (fd, CONTROLEVENT, EVCAP_ENABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
```

```
        }

    /* wait for next event to occur */
    if (ioctl (fd, BC635_INT_WAITONEVENT, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_WAITONEVENT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* event occurred, now get timestamp */
    if (ioctl (fd, RDEVENTTMREQ, &tfpTm) == ERROR)
        {
        (void) printf ("ioctl(RDEVENTTMREQ) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* use unix derived time display function to show event
time */
    (void) printf ("%s\n", asctime (&tfpTm.tm));

    (void) close (fd);

}/*end of test() */
```

BC635_INT_WAITONPERIODIC

COMMAND: BC635_INT_WAITONPERIODIC

PURPOSE: Wait for next TFP periodic pulse interrupt to occur.

INPUTS:

Type	Name/Description
int	Always 0

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl will block waiting for the *TFP* to generate a *PCI* interrupt when the next periodic pulse generated by the *TFP* occurs. The *ioctl:SETINTMASK* need not be explicitly called to enable this type of interrupt, this ioctl will enable the generation of the periodic interrupt..

EXAMPLE: The following example code performs the *ioctl:BC635_INT_WAITONPERIODIC.*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     struct periodic pulseDescrip;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    pulseDescrip.sync1pps = PERIODIC_SYNC;   /* sync to 1
pps */
    pulseDescrip.n1 = 2500;
    pulseDescrip.n1 = 4;
```

```
    if (ioctl (fd, SETPERIODIC, &pulseDescrip) == ERROR)
        {
        (void) printf ("ioctl(SETPERIODIC) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

  /* wait for next 'periodic' interrupt to generated by the
TFP */
  if (ioctl (fd, BC635_INT_WAITONPERIODIC, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_INT_WAITONPERIODIC)
failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    (void) close (fd);

}/*end of test() */
```

BC635_INT_WAITON1PPS

COMMAND: BC635_INT_WAITON1PPS

PURPOSE: Wait for next 1 PPS interrupt to occur.

INPUTS:

Type	Name/Description
int	Always 0

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl will block waiting for the *TFP* to generate a *PCI* interrupt when the next 1 PPS pulse occurs. The *ioctl:SETINTMASK* need not be explicitly called to enable this type of interrupt, this ioctl will enable the generation of the 1 PPS interrupt..

EXAMPLE: The following example code performs the *ioctl:BC635_INT_WAITON1PPS.*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
     struct periodic pulseDescrip;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }


  /* wait for next ‘periodic' interrupt to generated by the
TFP */
  if (ioctl (fd, BC635_INT_WAITON1PPS, 0) == ERROR)
        {
```

```
        (void) printf (“ioctl(BC635_INT_WAITON1PPS) failed:
“);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    (void) close (fd);

}/*end of test() */
```

RDINTSTAT, SETINTSTAT

COMMAND: RDINTSTAT, SETINTSTAT, RDINTMASK, SETINTMASK

PURPOSE: Used to poll/clear TFP interrupts.

INPUTS:

Type (RDINTSTAT, RDINTMASK)	Name/Description
int	*intStatVal* - Interrupt status/mask register value read

Type (SETINTSTAT, SETINTMASK)	Name/Description
int *	*intStatVal* - Interrupt status/mask register value to set.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: The interrupt source status register contains information regarding the *TFP* interrupt sources. The *TFP* sets a register bit in the *INTSTAT* register corresponding to the type of interrupt source when interrupt source has generated an interrupt. The *ioctl:RDINTSTAT* call can be used to poll the TFP for the occurrence of an interrupt source. The corresponding bit can be cleared using the i*octl:SETINTSTAT*.

The *ioctl:SETINTMASK* is used to enable *TFP* interrupt sources. The interrupt enable mask value has the same form as is used for *ioctl:RDINTSTAT* and *ioctl:SETINTSTAT*.

The *ioctl:RDINTMASK* is used to read the current interrupt enable (mask) value.

Use the following defines to enable or poll/clear the corresponding interrupt source:

```
BCINT_EVENT - Event interrupt.
INT_PERIODIC - Periodic interrupt.
INT_STROBE - Strobe interrupt.
INT_1PPS - 1 PPS interrupt.
INT_PACKET - GPS packet interrupt.
INT_BITMASK - Used to clear all interrupt sources.
```

EXAMPLE: The following example code performs the *ioctl:SETINTMASK*, *ioctl:RDINTSTAT* and *ioctl:SETINTSTAT* so as to manage an event interrupt source:

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”
```

```
void test ()
    {
    int fd;
    UINT32 intStat;
    UINT32 intMask;
    BOOL done;
    int timeout;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* Enable event interrupts */
    intMask = BCINT_EVENT;

    if (ioctl (fd, SETINTMASK, intMask) == ERROR)
        {
        (void) printf (“ioctl(SETINTMASK) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

  /* Poll for event interrupt */
  done = FALSE;
  timeout = 0;
  for (; done == FALSE && timeout < 10;)
      {
      /* get interrupt status */
      (void) ioctl (fd, RDINTSTAT, &intStat) == ERROR)

      /* did event interrupt occur ? */
      if ((intStat & BCINT_EVENT) > 0)
           {
           (void) printf (“Event occurred\n”);

           /* clear */
           intStat = INT_BITMASK;
           (void) ioctl (fd, SETINTSTAT, intStat);
```

```
                done = TRUE;
                }

            taskDelay (sysClkRateGet () / 4);
            timeout ++;
            }

        (void) close (fd);

}/*end of test() */
```

SETGPSMDFLG

COMMAND: SETGPSMDFLG

PURPOSE: Read binary time for an event from the TFP.

INPUTS:

Type	Name/Description
int	*modeVal* - Enable or Disable value

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called to enable or disable GPS mode packet processing. You may use the following defines as the *modeVal*:

```
GPS_FLG_DIS(0)
GPS_FLG_ENA(1)
```

EXAMPLE: The following example code performs the *ioctl:SETGPSMDFLG* in the process of requesting a GPS packet:

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_GPS_PACKET gps_packet;
    UINT32 id;
    int i;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* enable GPS mode */
    if (ioctl (fd, SELTIMINGMODE, MODE_GPS) == ERROR)
```

```
        {
        (void) printf (“ioctl(SELTIMINGMODE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable GPS packet processing mode */
    if (ioctl (fd, SETGPSMDFLG, GPS_FLG_ENA) == ERROR)
        {
        (void) printf (“ioctl(SETGPSMDFLG) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    printf(“Enter Packet to receive (hex - i.e. 4a) : “);
    scanf (“%x”, &id);

    gps_packet.packet_id = id;
    gps_packet.packet_length = 0;

    ioctl (fd, REQGPSPACKET, (int) &gps_packet);

    if (gps_packet.packet_length == 0)
        printf(“Could not receive GPS System Message\n”);

    else
       {
       printf (“Packet id = %x, lenght=%d\n”,
               gps_packet.packet_id,
gps_packet.packet_length);
       for (i=0; i < gps_packet.packet_length-1; i++)
           printf (“[%x] = %x\n”, i,
gps_packet.packet_data[i]);
       }

    (void) close (fd);

}/*end of test() */
```

REQGPSPACKET

COMMAND: REQGPSPACKET

PURPOSE: Receive GPS packet from the TFP.

INPUTS:

Type	Name/Description
TFP_GPS_PACKET *	*pTfpGpsPacket* - GPS packet received.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl waits for the next GPS packet to be received. Control does not return to the caller until a packet is received. On return, the following packet definition is filled:

```
typedef struct
 {
 UINT8 packet_length;
 UINT8 packet_id;
 UINT8 packet_data[256]; /* one bigger than
necessary */
 } TFP_GPS_PACKET;
```

EXAMPLE: The following example code performs the *ioctl:REQGPSPACKET:*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_GPS_PACKET gps_packet;
    UINT32 id;
    int i;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
```

```
        return (ERROR);
        }

    /* enable GPS mode */
    if (ioctl (fd, SELTIMINGMODE, MODE_GPS) == ERROR)
        {
        (void) printf ("ioctl(SELTIMINGMODE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable GPS packet processing mode */
    if (ioctl (fd, SETGPSMDFLG, GPS_FLG_ENA) == ERROR)
        {
        (void) printf ("ioctl(SETGPSMDFLG) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    printf("Enter Packet to receive (hex - i.e. 4a) : ");
    scanf ("%x", &id);

    gps_packet.packet_id = id;
    gps_packet.packet_length = 0;

    ioctl (fd, REQGPSPACKET, (int) &gps_packet);

    if (gps_packet.packet_length == 0)
        printf("Could not receive GPS System Message\n");

    else
       {
       printf ("Packet id = %x, lenght=%d\n",
               gps_packet.packet_id,
gps_packet.packet_length);
       for (i=0; i < gps_packet.packet_length-1; i++)
           printf ("[%x] = %x\n", i,
gps_packet.packet_data[i]);
       }

    (void) close (fd);
```

```
}/*end of test() */
```

SENDGPSPACKET

COMMAND: SENDGPSPACKET

PURPOSE: Send GPS packet to TFP.

INPUTS:

Type	Name/Description
TFP_GPS_PACKET *	*pTfpGpsPacket* - GPS packet received.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl sends a GPS packet to the TFP. The following packet definition is to be filled by the caller:

```
typedef struct
 {
 UINT8 packet_length;
 UINT8 packet_id;
 UINT8 packet_data[256]; /* one bigger than
necessary */
 } TFP_GPS_PACKET;
```

EXAMPLE: The following example code performs the *ioctl:SENDGPSPACKET:*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_GPS_PACKET gps_packet;
    UINT32 id;
    int i;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
```

```
        }

    /* enable GPS mode */
    if (ioctl (fd, SELTIMINGMODE, MODE_GPS) == ERROR)
        {
        (void) printf ("ioctl(SELTIMINGMODE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable GPS packet processing mode */
    if (ioctl (fd, SETGPSMDFLG, GPS_FLG_ENA) == ERROR)
        {
        (void) printf ("ioctl(SETGPSMDFLG) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* fill GPS packet to send */
    printf("Enter Packet to Send (hex - i.e. 4a) : ");
    scanf ("%x", &id);
    gps_packet.packet_id = id;

    printf("Enter Packet Length (decimal plus one) : ");
    scanf ("%d", &tmp);
    gps_packet.packet_length = (UINT8) tmp;

    for (i = 0; i < gps_packet.packet_length - 1; i++)
        {
        printf("Enter data (in hex) [%d] : ", i);
        scanf ("%x", &tmp);
        gps_packet.packet_data[i] = (UINT8) tmp;
        }

    ioctl (fd, SENDGPSPACKET, (int) &gps_packet);

    (void) close (fd);

}/*end of test() */
```

MANREQGPSPACKET

COMMAND: MANREQGPSPACKET

PURPOSE: Manually request a packet from GPS.

INPUTS:

Type	Name/Description
TFP_GPS_SENDRECEIVE_PACKET *	*pTfpGpsPacket* - GPS packet received.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl manually requests a GPS packet from the GPS. The *send* packet embedded in the following structure must be filled by the caller.

```
typedef struct
{
  TFP_GPS_PACKET send;
  TFP_GPS_PACKET receive;
} TFP_GPS_SENDRECEIVE_PACKET;

 typedef struct
  {
  UINT8 packet_length;
  UINT8 packet_id;
  UINT8 packet_data[256]; /* one bigger than
necessary */
  } TFP_GPS_PACKET;
```

EXAMPLE: The following example code performs the *ioctl:MANREQGPSPACKET:*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_GPS_SENDRECEIVE_PACKET gps_packet;
    UINT32 id;
    int i;

     /* open device - get handle */
```

```
if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
    {
    (void) printf (“open of %s failed: “, “/bc635/0”);
    printErrno (errnoGet ());
    return (ERROR);
    }

/* enable GPS mode */
if (ioctl (fd, SELTIMINGMODE, MODE_GPS) == ERROR)
    {
    (void) printf (“ioctl(SELTIMINGMODE) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* enable GPS packet processing mode */
if (ioctl (fd, SETGPSMDFLG, GPS_FLG_ENA) == ERROR)
    {
    (void) printf (“ioctl(SETGPSMDFLG) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* fill GPS packet to send */
printf(“Enter Packet to Send (hex - i.e. 4a) : “);
scanf (“%x”, &id);
gps_packet.packet_id = id;

printf(“Enter Packet Length (decimal plus one) : “);
scanf (“%d”, &tmp);
gps_packet.packet_length = (UINT8) tmp;

for (i = 0; i < gps_packet.packet_length - 1; i++)
    {
    printf(“Enter data (in hex) [%d] : “, i);
    scanf (“%x”, &tmp);
    gps_packet.packet_data[i] = (UINT8) tmp;
    }


/* manually request gps packet */
```

```
    ioctl (fd, MANREQGPSPACKET, (int) &gps_packet);

    /* check if GPS link was ok */
    if (gps_packet.receive.packet_length == 0)
        printf ("Could not receive GPS System Message\n");

    else
       {
       /* display packet received */
       printf ("Packet id = %x, length=%d\n",
               gps_packet.receive.packet_id,
               gps_packet.receive.packet_length);

       for (i=0; i < gps_packet.receive.packet_length - 1;
i++)
           printf ("[%x] = %x\n", i,
                   gps_packet.receive.packet_data[i]);
       }

    (void) close (fd);

}/*end of test() */
```

BC635_INT_WAITONGPS

COMMAND: BC635_INT_WAITONGPS

PURPOSE: Wait for next GPS event (packet) to arrive.

INPUTS:

Type	Name/Description
int	Always 0

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl will block waiting for the *TFP* to generate a *PCI* interrupt when the next GPS packet is received. The *ioctl:SETINTMASK* need not be explicitly called to enable this type of interrupt, this ioctl will enable the generation of the GPS packet received interrupt. Note that the *ioctl:REQGPSPACKET* and *ioctl:MANREQGPSPACKET* operate independent of this ioctl.

EXAMPLE: The following example code performs the *ioctl:BC635_INT_WAITONGPS*.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }


  /* wait for next ‘GPS' interrupt to generated by the TFP
*/
  if (ioctl (fd, BC635_INT_WAITONGPS, 0) == ERROR)
        {
```

```
            (void) printf (“ioctl(BC635_INT_GPS) failed: “);
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

        (void) close (fd);

}/*end of test() */
```

CONTROLSTROBE

COMMAND: CONTROLSTROBE

PURPOSE: Set strobe control parameters..

INPUTS:

Type	Name/Description
int	*ctrlVal* - Choose one of the following defines listed below.

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called so as to set a specific control parameter for the strobe function of the *TFP*.

One of the following *defines* is to be set. This ioctl is to be called multiple times when setting multiple parameters:

```
STROBE_SECUS
STROBE_USONLY
STROBE_ENABLE
STROBE_DISABLE
```

Set *STROBE_SECUS* to enable use of major and minor strobe time registers when setting the time at which the strobe output is to occur. Set this parameter when you wish to output the strobe some number of days ahead.

Set *STROBE_USONLY* to enable use of the minor (number of useconds) strobe time register only so as to strobe some number of microseconds ahead. With this parameter set, the strobe output continues once per second once started.

Set *STROBE_ENABLE* to enable strobe output.

Set *STROBE_DISABLE* to disable strobe output (the output is held low).

EXAMPLE: The following example code performs the *ioctl:CONTROLSTROBE* in the context of programming the strobe output to occur 24 hours ahead ot the time reference source. Just the strobe related ioctls are called here:

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
```

```
int fd;
TFP_TV tfpTv;

 /* open device - get handle */
if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
    {
    (void) printf (“open of %s failed: “, “/bc635/0”);
    printErrno (errnoGet ());
    return (ERROR);
    }

/* defensively disable strobe output for now */
if (ioctl (fd, CONTROLSTROBE, STROBE_DISABLE) == ERROR)
    {
    (void) printf (“ioctl(CONTROLSTROBE) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* enable use of major/minor strobe time */
if (ioctl (fd, CONTROLSTROBE, STROBE_SECUS) == ERROR)
    {
    (void) printf (“ioctl(CONTROLSTROBE) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* will use binary time to set strobe time */
if (ioctl (fd, SELTIMEFORMAT, TIME_BINARY) == ERROR)
    {
    (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* program strobe to occur 24hrs ahead */
tfpTv.tv.tv_sec = 60 * 60 * 24;
tfpTv.tv.tv_usec = 0;
if (ioctl (fd, SETSTROBETV, &tfpTv) == ERROR)
    {
```

```
            (void) printf (“ioctl(SETSTROBETV) failed: “);
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

         /* now enable strobe output */
         if (ioctl (fd, CONTROLSTROBE, STROBE_ENABLE) == ERROR)
            {
            (void) printf (“ioctl(CONTROLSTROBE) failed: “);
            printErrno (errnoGet ());
            (void) close (fd);
            return (ERROR);
            }

      /* a single strobe pulse will occur 24 hours ahead */
       (void) close (fd);

}/*end of test() */
```

SETSTROBETM

COMMAND: SETSTROBETM

PURPOSE: Set the strobe time registers usind decimal time.

INPUTS:

Type	Name/Description
TFP_TM *	*pTfpTm* - Pointer to *TFP_TM* structure

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called when the *TIME_DECIMAL* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

This ioctl is used to set the amount of time ahead of the current time reference that a strobe output will occur. This ioctl must be used in conjunction with *ioctl:CONTROLSTROBE*.

The following is the definition of the *TFP_TM* structure:

```
typedef struct
   {
   struct tm tm;
   UINT32 usec;
   UINT32 hnsec;
    UINT32 status;
   } TFP_TM;
```

The vxWorks defined definition of the *tm* structure is:

```
struct tm
    {
    int tm_sec;     /* seconds after the minute     -
[0, 59] */
    int tm_min;     /* minutes after the hour       -
[0, 59] */
    int tm_hour;    /* hours after midnight         -
[0, 23] */
    int tm_mday;    /* day of the month             -
[1, 31] */
    int tm_mon;     /* months since January         -
[0, 11] */
    int tm_year;    /* years since 1900     */
    int tm_wday;    /* days since Sunday            -
[0, 6] */
    int tm_yday;    /* days since January 1         -
[0, 365] */
    int tm_isdst;   /* Daylight Saving Time flag */
    };
```

For this ioctl only the *tm_hour*, *tm_min*, *tm_sec* and *usec* members are used. If number of days is to be included in the strobe time, then use the *ioctl:SETSTROBETV*.

EXAMPLE: The following example code performs the *SETSTROBETM* ioctl.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TM tfpTm;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* defensively disable strobe output for now */
    if (ioctl (fd, CONTROLSTROBE, STROBE_DISABLE) == ERROR)
        {
        (void) printf (“ioctl(CONTROLSTROBE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable use of major/minor strobe time */
    if (ioctl (fd, CONTROLSTROBE, STROBE_SECUS) == ERROR)
        {
        (void) printf (“ioctl(CONTROLSTROBE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* will use binary time to set strobe time */
```

```
    if (ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL) == ERROR)
        {
        (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* program strobe to occur 20hrs ahead */
    tfpTm.tm.tm_hour = 20;
    tfpTm.tm_min = 0;
    tfpTm.tm_sec = 0;
    tfpTm.usec = 0;

    if (ioctl (fd, SETSTROBETM, &tfpTm) == ERROR)
        {
        (void) printf ("ioctl(SETSTROBETM) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

     /* now enable strobe output */
     if (ioctl (fd, CONTROLSTROBE, STROBE_ENABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLSTROBE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

   /* a single strobe pulse will occur 20 hours ahead */
    (void) close (fd);

}/*end of test() */
```

SETSTROBETV

COMMAND: SETSTROBETV

PURPOSE: Set the strobe time registers using binary time.

INPUTS:

Type	Name/Description
TFP_TV *	*pTfpT*v - Pointer to *TFP_T*V structure

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl is to be called when the *TIME_BINARY* time format had been selected using the *ioctl:SELTIMEFORMAT*. Note that the *ioctl:SELTIMEFORMAT* need only be called once.

This ioctl is used to set the amount of time ahead of the current time reference that a *strobe* output will occur. This ioctl must be used in conjunction with *ioctl:CONTROLSTROBE*.

The following is the definition of the *TFP_T*V structure:

```
typedef struct
   {
   struct timeval tv;
   UINT32 hnsec;
   UINT8 status;
   } TFP_TV;
```

The vxWorks defined definition of the *t*v structure is:

```
struct timeval tv
    {
    time_t tv_sec;
    long tv_usec;
    };
```

For this ioctl only the *tv_sec* and *tv_usec* members are used. The *tv_sec* member can be programmed so that the strobe output occurs up to 48 days ahead.

EXAMPLE: The following example code performs the *SETSTROBET*V ioctl.

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”
```

```
void test ()
    {
    int fd;
    TFP_TV tfpTv;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* defensively disable strobe output for now */
    if (ioctl (fd, CONTROLSTROBE, STROBE_DISABLE) == ERROR)
        {
        (void) printf (“ioctl(CONTROLSTROBE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* enable use of major/minor strobe time */
    if (ioctl (fd, CONTROLSTROBE, STROBE_SECUS) == ERROR)
        {
        (void) printf (“ioctl(CONTROLSTROBE) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* will use binary time to set strobe time */
    if (ioctl (fd, SELTIMEFORMAT, TIME_BINARY) == ERROR)
        {
        (void) printf (“ioctl(SELTIMEFORMAT) failed: “);
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* program strobe to occur 40 days ahead */
    tfpTv.tv.tv_sec = 60 * 60 * 24 * 40;
    tfpTv.tv.tv_usec = 0;
```

```
    if (ioctl (fd, SETSTROBETV, &tfpTv) == ERROR)
        {
        (void) printf ("ioctl(SETSTROBETV) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

     /* now enable strobe output */
     if (ioctl (fd, CONTROLSTROBE, STROBE_ENABLE) == ERROR)
        {
        (void) printf ("ioctl(CONTROLSTROBE) failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* a single strobe pulse will occur 40 days ahead */
    (void) close (fd);

}/*end of test() */
```

BC635_INT_WAITONSTROBE

COMMAND: BC635_INT_WAITONSTROBE

PURPOSE: Wait for strobe output to occur.

INPUTS:

Type	Name/Description
int	Always 0

RETURNS: The *ioctl* returns *OK* or *ERROR*.

DESCRIPTION: This ioctl will block waiting for the *TFP* to generate a *PCI* interrupt when the time coincidence *strobe* output occurs. The ioctls *CONTROLSTROBE, SETSTROBETM, SETSTROBETV* must be called so as to configure *strobe* operation.

EXAMPLE: The following example code performs the *ioctl:BC635_INT_WAITONSTROBE.*

```
#include “vxWorks.h”
#include “stdio.h”
#include “ioLib.h”
#include “bc635.h”

void test ()
    {
    int fd;
    TFP_TV tfpTv;

     /* open device - get handle */
    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)
        {
        (void) printf (“open of %s failed: “, “/bc635/0”);
        printErrno (errnoGet ());
        return (ERROR);
        }

    /* defensively disable strobe output for now */
    if (ioctl (fd, CONTROLSTROBE, STROBE_DISABLE) == ERROR)
        {
        (void) printf (“ioctl(CONTROLSTROBE) failed: “);
        printErrno (errnoGet ());
```

```
    (void) close (fd);
    return (ERROR);
    }

/* enable use of major/minor strobe time */
if (ioctl (fd, CONTROLSTROBE, STROBE_SECUS) == ERROR)
    {
    (void) printf ("ioctl(CONTROLSTROBE) failed: ");
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* will use binary time to set strobe time */
if (ioctl (fd, SELTIMEFORMAT, TIME_BINARY) == ERROR)
    {
    (void) printf ("ioctl(SELTIMEFORMAT) failed: ");
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

/* program strobe to occur 40 days ahead */
tfpTv.tv.tv_sec = 60 * 60 * 24 * 40;
tfpTv.tv.tv_usec = 0;
if (ioctl (fd, SETSTROBETV, &tfpTv) == ERROR)
    {
    (void) printf ("ioctl(SETSTROBETV) failed: ");
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

 /* now enable strobe output */
 if (ioctl (fd, CONTROLSTROBE, STROBE_ENABLE) == ERROR)
    {
    (void) printf ("ioctl(CONTROLSTROBE) failed: ");
    printErrno (errnoGet ());
    (void) close (fd);
    return (ERROR);
    }

 /* wait for strobe to occur */
```

```
     if (ioctl (fd, BC635_INT_WAITONSTROBE, 0) == ERROR)
        {
        (void) printf ("ioctl(BC635_INT_WAITONSTROBE)
failed: ");
        printErrno (errnoGet ());
        (void) close (fd);
        return (ERROR);
        }

    /* a single strobe pulse will occur 40 days ahead */
    (void) close (fd);

}/*end of test() */
```

Undocumented IOCTLs

The following ioctl calls are supported by this driver but not documented in this manual. Examples of how to use the ioctls listed below are found in the sample code file *sample_vx.c*. Also, *sample_vx.c* is listed in section 6 of this manual.

SOFTWARERESET - Perform software reset of TFP.
SETGENTMOFFSET - Set generator Time Offset (default: 0)
SETLOCTMOFFSET - Set local time offset (default: 0)
GETDATA - Request TFP data
SETTCOUTFMT - Set TimeCode output format (default: IRIG-B)
CONTROLTIMEBASE - Set clock source, Control Jam-Sync, Force Jam-Sync
SETDAC - Load D/A converter
SETDISCGAIN - Set disciplining gain.
SYNCRTC - Sync external time to RTC.
DISCBATT - Disconnect battery to RTC chip.
SETCLKVAL - Set/Request Advance/Retard clock value (for 12083 boards only)
SETGPSTMFMT - Select GPS time format (default: UTC)
SETLOCTMFLG - Observe IEEE 1344 Local Time Flag (default: enabled)
SETYRINCFLG - Year auto-increment flag (default: yes)

4.0 Target CPU Considerations

The *sysBC635Init* routine is a BSP specific routine developed for the Motorola PowerPC Single-Board-Computer running *VxWorks*. It programs PCI base address 0 (*BAR0*) register for a single instance of the bc635/637 board as. The base address programmed is 0xFD04000. If this address is not applicable to your BSP, then an alternative routine(s) must be used to program the *BAR0* address for your card

5.0 Software Installation

The following table gives a list of the object and header files needed to operate this driver:

Filename	Description
bc635.o	Driver object file.
bc635.h	Driver header file containing defines and structure definitions required by the application.
sample_vx.c	Example code.

The file *bc635.h* must be installed in an area that is accessible to the application software at software build time.
The *sample_vx.c* file contains a menu-driven program that may be compiled and loaded into vxWorks via the vxWorks shell. The user then invokes sample at the shell prompt.

The following is an example VxWorks shell session for an MVME2604 PowerPC single-board-computer running *vxWorks 5.4*. There is a single bc635/637 installed on the MVME2604. The software is installed, then the driver is installed into the *VxWorks IOS*. The bc635/637 card installed is initialized for operation under *VxWorks*. The output of the *vxWorks* devs routine would list the device *"/tfp0"*. Lastly the *sample* menu-driven routine is invoked.

```
-> ld < sysBC635.o
-> ld < bc635.o
-> ld < sample_vx.o
-> sysBC635Init
-> tfpDrv
-> tfpDevInstall (0, 0x19)
-> sample
```

6.0 Example

```
/*
 sample.c  - This is an interactive test program demonstrating the
             functionality of the bc635/637 VxWorks Device Driver.

             It is intended as example code, to assist a programmer in
             developing their own code using the driver.

             This test program lacks proper error checking, and also
             does not prevent erroneous actions.  (An example of an
             erroneous action would be to read a decimal time format when
             the board is in binary time mode).
*/
/*
  ****************************************************************************
** Copyright (C) 2000 Advanced Processing Laboratories Inc. (AP Labs)
** All rights reserved.
**
** This file and its contents are a product of AP Labs.
** All software distributed by AP Labs is done so under a software license.
** This file may not be copied or modified without execution of the
** appropriate software license agreement with AP Labs or explicit written
** permission of AP Labs.
**
** This file is provided as is, with no warranties of any kind including
** the warranties of design, merchantability and fitness for a particular
** purpose, or arising from a course of dealing, usage or trade practice.
**
** AP Labs shall have no liability with respect to the infringement of
** copyrights, trade secrets or any patents by this file or any part thereof.
**
** In no event will AP Labs be liable for any lost revenue or profits
** or other special, indirect and consequential damages, even if AP Labs
** has been advised of the possibility of such damages, as a result of the
** usage of this file and software for which this file is a part.
****************************************************************************
*/

/*****************************************************************************/

#include “vxWorks.h”
#include “stdio.h”
#include “logLib.h”
```

```
#include “taskLib.h”
#include “ioLib.h”
#include “errnoLib.h”
#include “sysLib.h”
#include “usrLib.h”
#include “selectLib.h”
#include “ctype.h”
#include “string.h”
#include “bc635.h”
#include “sysLib.h”

#define RC_OK           1
#define RC_ERROR   -1

/*****************************************************************************/
/* Global Variables */
/*****************************************************************************/
int     fd;
struct getdata_t get;

/* Forward declarations */

int     menu_main ();


/*****************************************************************************/
/* Description : main */
/*****************************************************************************/
int
sample ()
{

    if ((fd = open (“/tfp0”, 0, 0)) == ERROR)   /* open bc635pci device */
    {
        printf (“Error opening Device Driver ......... Exiting”);
        printErrno (errnoGet ());
        return (ERROR);
    }

    return menu_main ();

}


/*****************************************************************************/
```

```
/* Description: Set Oper Mode */
/*****************************************************************************/
void
pci_mode ()
{
    int     oper_mode = 0;

    printf ("\n\n\n\t    Select Operational Mode:\n\n");
    printf ("\t0. Time Code Mode\n");
    printf ("\t1. Free Running Mode\n");
    printf ("\t2. External 1PPS\n");
    printf ("\t3. Real Time Clock Mode\n");
    printf ("\t4. GPS Mode\n\n");
    printf ("\t   Select: ");
    scanf ("%i", &oper_mode);

    switch (oper_mode)
    {
        case 0:
            ioctl (fd, SELTIMINGMODE, (int) MODE_TIMECODE);     /* Timing Mode: time
                                                                 * code */
            break;
        case 1:
            ioctl (fd, SELTIMINGMODE, (int) MODE_FREERUN);      /* Timing Mode: Free
                                                                 * Running */
            break;
        case 2:
            ioctl (fd, SELTIMINGMODE, (int) MODE_EXT1PPS);      /* Timing Mode: 1PPS */
            break;
        case 3:
            ioctl (fd, SELTIMINGMODE, (int) MODE_RTC);  /* Timing Mode: RTC */
            break;
        case 4:
            ioctl (fd, SELTIMINGMODE, (int) MODE_GPS);  /* Timing Mode: GPS */
            break;
        default:
            printf ("Error setting opertional mode!!!!");
            break;
    }
}

/*****************************************************************************/
/* Description : Set Time Code Format */
/*****************************************************************************/
void
```

```
pci_time_code ()
{
    int     tc_fmt;
    int     tc_mod;

    printf (“\n\n\n\t    Select Time Code Format:\n\n”);
    printf (“\t1. IRIG A\n”);
    printf (“\t2. IRIG B\n\n”);
    printf (“\t   Select: “);
    scanf (“%d”, &tc_fmt);
    switch (tc_fmt)
    {
        case 1:
            ioctl (fd, SELTCFORMAT, TC_IRIGA);  /* IRIG A time code */
            break;
        case 2:
            ioctl (fd, SELTCFORMAT, TC_IRIGB);  /* IRIG B time code */
            break;
        default:
            printf (“Error Setting Time Code Format!”);
            break;
    }

    printf (“\n\n\n\t    Select Time Code Modulation:\n\n”);
    printf (“\t1. Modulated\n”);
    printf (“\t2. DC Level Shift\n\n”);
    printf (“\t   Select: “);
    scanf (“%d”, &tc_mod);
    switch (tc_mod)
    {
        case 1:
            ioctl (fd, SELTCMOD, MOD_AM);       /* Modulated */
            break;
        case 2:
            ioctl (fd, SELTCMOD, MOD_DCLS);     /* DCLS */
            break;
        default:
            printf (“Error Setting Time Code Modulation!”);
            break;
    }
}

/*****************************************************************************/
/* Description : Set Output Frequency */
/*****************************************************************************/
```

```
void
pci_out_freq ()
{
    int     out_freq = 3;

    printf (“\n\n\n\t    Select Output Frequency:\n\n”);
    printf (“\t1. 1  MHz\n”);
    printf (“\t2. 5  MHz\n”);
    printf (“\t3. 10 MHz\n\n”);
    printf (“\t   Select: “);
    scanf (“%d”, &out_freq);
    switch (out_freq)
    {
        case 1:
            ioctl (fd, SELFREQUENCYOUT, FREQ_1MHZ);     /* 1 MHz */
            break;
        case 2:
            ioctl (fd, SELFREQUENCYOUT, FREQ_5MHZ);     /* 5 MHz */
            break;
        case 3:
            ioctl (fd, SELFREQUENCYOUT, FREQ_10MHZ);    /* 10 MHz */
            break;
        default:
            printf (“Error Setting output frequency!”);
            break;
    }
}

/*****************************************************************************/
/* Description : Set Time Format */
/*****************************************************************************/
void
pci_time_format ()
{
    int     time_format = 0;

    printf (“Select Time Format\n  0. Decimal\n  1. Binary\n\nSelect: “);
    scanf (“%d”, &time_format);

    switch (time_format)
    {
        case 0:
            ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL);    /* decimal time format */
            break;
        case 1:
```

```
            ioctl (fd, SELTIMEFORMAT, TIME_BINARY);     /* Binary time format */
            break;
        default:
            printf ("\nError Setting Time Format!");
            break;
    }
}

/*****************************************************************************/
/* Description : Set Heartbeat Counters */
/*****************************************************************************/
void
pci_heartbeat ()
{
    struct periodic sper;
    int     hrt_mode;
    int     counter1 = 10;
    int     counter2 = 10;

    sper.sync1pps = PERIODIC_SYNC;

    printf ("\n\n\n\t    Heartbeat Counters and Mode:\n\n");
    printf ("\t1. Synchronous mode\n");
    printf ("\t2. Asynchronous mode\n\n");
    printf ("\t   Select: ");
    scanf ("%i", &hrt_mode);
    printf ("\n\tEnter 4 hex value for counetr1: \n");
    scanf ("%i", &counter1);
    printf ("\n\tEnter 4 hex value for counetr2: \n");
    scanf ("%i", &counter2);

    if (hrt_mode == 2)
        sper.sync1pps = PERIODIC_NOSYNC;
    if (counter1 >= 2 && counter1 <= 65535)
        sper.n1 = counter1;
    if (counter2 >= 2 && counter2 <= 65535)
        sper.n2 = counter2;

    ioctl (fd, SETPERIODIC, (int) &sper);
}


/*****************************************************************************/
/* Description : Set Time */
/*****************************************************************************/
```

```
void
pci_set_time ()
{
    int     choice;
    int     year = 1999;
    TFP_TM  tm;
    TFP_TV  tv;
    time_t tUsr_secs;
    time_t t1970_secs;
    struct tm tmpTm;

    printf (“\n\n\n\t Set Time Functions :\n\n”);
    printf (“\t1. Set Year\n”);
    printf (“\t2. Set Time TM\n”);
    printf (“\t3. Set Time TV\n”);
    printf (“\t   Select: “);

    scanf (“%d”, &choice);

    switch (choice)
    {
       case 1:
           printf (“\n\tEnter Year (1990 to 2050): \n”);
           scanf (“%d”, &year);
           if (year >= 1990 && year <= 2050)
              ioctl (fd, SETYEAR, year);      /* Set year */
           else
              printf (“\nError: Value Out of Range”);
           break;

       case 2:

           printf (“\n\tEnter Year [1970-2036]: “);
           scanf (“%d”, &(tm.tm.tm_year));
           printf (“\tEnter Year Day [0-366]: “);
           scanf (“%d”, &tm.tm.tm_yday);
           printf (“\t     Enter Hour [0-23]: “);
           scanf (“%d”, &tm.tm.tm_hour);
           printf (“\t      Enter Min [0-59]: “);
           scanf (“%d”, &tm.tm.tm_min);
           printf (“\t      Enter Sec [0-59]: “);
           scanf (“%d”, &tm.tm.tm_sec);

           ioctl (fd, SETTIMETM, (int)&tm);
```

```
        break;

    case 3:

        /*
         * We will prompt the user for year,month,day info
         * and then use the vxWorks mktime() routine
         * to generate a ‘binary' time of ‘seconds from 1970'.
         * But, vxWorks apparently calculates this based on
         * ‘seconds from 1900' (see time.h). So, we will
         * calculate ‘seconds from 1900 to 1970' and
         * ‘seconds from 1900 to user's date' and then subtract two.
         */

        /* prompt user for time/date info */
        printf (“\n\tEnter Year [1970-2036]: “);
        scanf (“%d”, &(tmpTm.tm_year));
        printf (“\t    Enter Month [0-11]: “);
        scanf (“%d”, &tmpTm.tm_mon);
        printf (“\tEnter Month Day [1-31]: “);
        scanf (“%d”, &tmpTm.tm_mday);
        printf (“\t     Enter Hour [0-23]: “);
        scanf (“%d”, &tmpTm.tm_hour);
        printf (“\t      Enter Min [0-59]: “);
        scanf (“%d”, &tmpTm.tm_min);
        printf (“\t      Enter Sec [0-59]: “);
        scanf (“%d”, &tmpTm.tm_sec);

        tmpTm.tm_yday = 0;
        tmpTm.tm_wday = 0;
        tmpTm.tm_isdst = 0;

        /* generate ‘1900 to user-date seconds' */
        tUsr_secs = mktime (&tmpTm);


        /* now get 1970 seconds */

        tmpTm.tm_sec = 0;
        tmpTm.tm_min = 0;
        tmpTm.tm_hour = 0;
        tmpTm.tm_mday = 1;
        tmpTm.tm_mon = 0;
        tmpTm.tm_wday = 0;
        tmpTm.tm_yday = 0;
```

```
            tmpTm.tm_isdst = 0;
            tmpTm.tm_year = 1970;

            /* generate ‘1900 to 1970 seconds' */
            t1970_secs = mktime (&tmpTm);

            /* now generate ‘1970 to user-date seconds' */
            tv.tv.tv_sec = tUsr_secs - t1970_secs;

            ioctl (fd, SETTIMETV, (int)&tv);

            break;

        default:
            break;
    }

}


/*****************************************************************************/
/* Description : Request Current Settings */
/*****************************************************************************/
void
pci_req_current_settings ()
{
    const char smode[7][10] = {“Time Code”, “Free Run”, “Ext 1PPS”, “RTC”, “Reserved”, “
Reserved”, “GPS”};
    const char stm_fmt[2][10] = {“Decimal”, “Binary”};
    int     tm_fmt = 0;
    int     year = 0;
    int     dac_val = 0;
    int     timing_off = 0;
    INT16   gen_off = 0;
    INT16   loc_off = 0;
    u_char  tc_fmt = ‘ ‘;
    u_char  tc_mod = ‘ ‘;
    u_char  clk_scr = ‘ ‘;
    UINT8   mode = 0;

    get.arg = GETDATA_MODE;     /* Timing Mode */
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        mode = (int) get.data.tmode;
    printf (“\nMode\t\t: %s”, smode[mode]);
```

```
    get.arg = GETDATA_TFORMAT;  /* Time Format */
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        tm_fmt = (int) get.data.tformat;
    printf (“\nTime Format\t: %s”, stm_fmt[tm_fmt]);

    get.arg = GETDATA_YEAR;     /* Year */
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        year = (int) get.data.year;
    printf (“\nYear\t\t: %d”, year);

    get.arg = GETDATA_PER;
    ioctl (fd, GETDATA, (int)&get);
    printf (“\nPeriodic Output\t: sync1pps=%x  n1=%x, n2=%x”,
       get.data.period.sync1pps, get.data.period.n1,
       get.data.period.n2);

    get.arg = GETDATA_TCFMT;    /* Time Code Format */
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        tc_fmt = (u_char) get.data.tcformat;
    printf (“\nTime Code\t: IRIG %c”, tc_fmt);

    get.arg = GETDATA_TCMOD;    /* Time Code Modulation */
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        tc_mod = (u_char) get.data.tcmod;
    printf (“\nCode Modulation\t: %c”, tc_mod);

    get.arg = GETDATA_TOFF;
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
        timing_off = (int) get.data.offset;
    printf (“\nTiming Offset\t: %d”, timing_off);

    get.arg = GETDATA_UTCINFO;
    ioctl (fd, GETDATA, (int) &get);
    printf (“\nUTC\t : enable=%d,leapsec=%d,leapevt=%d,evttime=%x”, get.data.utc.enable,
get.data.utc.leapsec,
       get.data.utc.leapevt, get.data.utc.evttime);

    get.arg = GETDATA_TCOUTFMT;
    ioctl (fd, GETDATA, (int) &get);
    printf (“\nTCOUTFMT\t: %d”, get.data.tcoutfmt);

    get.arg = GETDATA_LOCTMOFF;
    if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
         loc_off = get.data.locoff.offset;
    printf (“\nLoc Time Offset\t: %d”, loc_off);
```

```
get.arg = GETDATA_TCGENOFF;
if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
     gen_off = get.data.genoff.offset;
printf (“\nGen Time Offset\t: %d”, gen_off);

get.arg = GETDATA_LEAPSEC;
ioctl (fd, GETDATA, (int) &get);
printf (“\nLEAPSEC: ls_flag=%d, leap_tm=%d”,
   get.data.leapsec.ls_flag, get.data.leapsec.leap_tm);

get.arg = GETDATA_FWVER;
ioctl (fd, GETDATA, (int) &get);
printf (“\nFWVER\t: vmajor=%d, vminor=%d, month=%d, day=%d, year=%d”,
   get.data.version.vmajor, get.data.version.vminor,
   get.data.version.month, get.data.version.day,
   get.data.version.year);


get.arg = GETDATA_CLKSRC;
if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
    clk_scr = (u_char) get.data.clksrc;
if (clk_scr == ‘I')
    printf (“\nOscillator  \t: Internal”);
else
    printf (“\nOscillator  \t: External”);


get.arg = GETDATA_DAVAL;
if (!((ioctl (fd, GETDATA, (int) &get)) < 0))
    dac_val = (int) get.data.davalue;
printf (“\nDAC Value\t: %x”, dac_val);

get.arg = GETDATA_JAMSC;
ioctl (fd, GETDATA, (int) &get);
printf (“\nJAMSC\t\t: %d”, get.data.jsctl);

get.arg = GETDATA_OSCCTL;
ioctl (fd, GETDATA, (int) &get);
printf (“\nOCSCTL\t: discctl=%d, phasectl=%d, phasestep=%d, gain=%x, con=%x”,
   get.data.ocsctl.discctl, get.data.ocsctl.phasectl,
   get.data.ocsctl.phasestep, get.data.ocsctl.gain,
   get.data.ocsctl.con);

get.arg = GETDATA_BATTSTAT;
```

```
ioctl (fd, GETDATA, (int) &get);
printf (“\nBATSTATT\t: %x”, get.data.battstat);

get.arg= GETDATA_CLKVAL;
ioctl (fd, GETDATA, (int) &get);
printf (“\nCLKVAL\t\t: %x”, get.data.clkval);

get.arg = GETDATA_DTFW;
ioctl (fd, GETDATA, (int) &get);
printf (“\nDTFW\t\t: %c%c%c%c%c%c%c%c%c%c%c”,
  get.data.dtfw.dt01,
  get.data.dtfw.dt02,
  get.data.dtfw.dt03,
  get.data.dtfw.dt04,
  get.data.dtfw.dt05,
  get.data.dtfw.dt06,
  get.data.dtfw.dt07,
  get.data.dtfw.dt08,
  get.data.dtfw.dt09,
  get.data.dtfw.dt10,
  get.data.dtfw.dt11);

get.arg = GETDATA_ASSEMB;
ioctl (fd, GETDATA, (int) &get);
printf (“\nASSEMB\t\t: %x”, get.data.assemb);

get.arg = GETDATA_TFPMODEL;
ioctl (fd, GETDATA, (int) &get);
printf (“\nTFPMODEL\t: %c%c%c%c%c%c%c%c”,
  get.data.tfpm.tfp1,
  get.data.tfpm.tfp2,
  get.data.tfpm.tfp3,
  get.data.tfpm.tfp4,
  get.data.tfpm.tfp5,
  get.data.tfpm.tfp6,
  get.data.tfpm.tfp7,
  get.data.tfpm.tfp8);

get.arg = GETDATA_SERIAL;
ioctl (fd, GETDATA, (int) &get);
printf (“\nSERIAL Number\t: %x”, get.data.serialnum);
```

```
    return;

}

/*************************************************************************
* tmPrint - print ‘tm' struct
*
* RETURNS: OK
*/

LOCAL STATUS tmPrint
        (
                TFP_TM * pTm
)

{

    (void) printf (“\n   tm_sec: %d\n”, pTm->tm.tm_sec);
    (void) printf (“   tm_min: %d\n”, pTm->tm.tm_min);
    (void) printf (“  tm_hour: %d\n”, pTm->tm.tm_hour);
    (void) printf (“  tm_yday: %d\n”, pTm->tm.tm_yday);
    (void) printf (“  tm_year: %d\n”, pTm->tm.tm_year);
    (void) printf (“     usec: %d\n”, pTm->usec);
    (void) printf (“    hnsec: %d\n”, pTm->hnsec);
    (void) printf (“   status: x%01X\n”, pTm->status);

    return (OK);
}

/*************************************************************************
* tvPrint - print ‘tv' struct
*
* RETURNS: OK
*/
LOCAL STATUS tvPrint
        (
                TFP_TV * pTv
)

{
    struct tm tm;
    char tmbuf[80];
    size_t bufLen = sizeof(tmbuf);

    (void) printf (“   tv_sec: %lu (x%08X)\n”,
```

```
                pTv->tv.tv_sec, (int) pTv->tv.tv_sec);
    (void) printf (“     usec: x%ld\n”, pTv->tv.tv_usec);
    (void) printf (“    hnsec: x%x\n”, pTv->hnsec);
    (void) printf (“   status: x%01X\n”, pTv->status);

    gmtime_r (&pTv->tv.tv_sec, &tm);

    asctime_r (&tm, tmbuf, &bufLen);

    /* display date/time */
    (void) printf (“time/date: %s\n”, tmbuf);


    return (OK);
}

/*******************************************************************************/
/* Description : read_time_menu */
/*******************************************************************************/

void
read_time_menu ()
{
    int     choice;
    int     ioctl_choice;
    int     arg_choice;
    int     print_choice;
    TFP_TM  tfpTM;
    TFP_TV  tfpTV;


    printf (“\n\n\n  Read Time Formats:\n\n”);
    printf (“\t1. RDTIMETM (Unlatched, DECIMAL)\n”);
    printf (“\t2. RDTIMETMREQ (Latched, DECIMAL)\n”);
    printf (“\t3. RDTIMETV (Unlatched, BINARY)\n”);
    printf (“\t4. RDTIMETVREQ (Latched, BINARY)\n\n”);
    printf (“\t5. RDEVENTTM (Unlatched, DECIMAL)\n”);
    printf (“\t6. RDEVENTRTMEQ (Latched, DECIMAL)\n”);
    printf (“\t7. RDEVENTTV (Unlatched, BINARY)\n”);
    printf (“\t8. RDEVENTTVREQ (Latched, BINARY)\n\n”);
    printf (“\t9. TIMEREQUEST (Latch Time)\n”);
    printf (“\t10. EVENTREQUEST (Latch Event)\n”);
    printf (“\n Enter Choice : “);
    scanf (“%i”, &choice);
```

```
switch (choice)
{
    case 1:
        print_choice = TIME_DECIMAL;
        ioctl_choice = RDTIMETM;
        arg_choice = (int) &tfpTM;
        break;
    case 2:
        print_choice = TIME_DECIMAL;
        ioctl_choice = RDTIMETMREQ;
        arg_choice = (int) &tfpTM;
        break;
    case 3:
        print_choice = TIME_BINARY;
        ioctl_choice = RDTIMETV;
        arg_choice = (int) &tfpTV;
        break;
    case 4:
        print_choice = TIME_BINARY;
        ioctl_choice = RDTIMETVREQ;
        arg_choice = (int) &tfpTV;
        break;
    case 5:
        print_choice = TIME_DECIMAL;
        ioctl_choice = RDEVENTTM;
        arg_choice = (int) &tfpTM;
        break;
    case 6:
        print_choice = TIME_DECIMAL;
        ioctl_choice = RDEVENTTMREQ;
        arg_choice = (int) &tfpTM;
        break;
    case 7:
        print_choice = TIME_BINARY;
        ioctl_choice = RDEVENTTV;
        arg_choice = (int) &tfpTV;
        break;
    case 8:
        print_choice = TIME_BINARY;
        ioctl_choice = RDEVENTTVREQ;
        arg_choice = (int) &tfpTV;
        break;
    case 9:
        print_choice = TIME_BINARY + TIME_DECIMAL;  /* none */
        ioctl_choice = TIMEREQUEST;
```

```
            arg_choice = (int) 0;
            break;
        case 10:
            print_choice = TIME_BINARY + TIME_DECIMAL;  /* none */
            ioctl_choice = EVENTREQUEST;
            arg_choice = (int) 0;
            break;

        default:
            printf ("Not implemented\n");
            return;
    }

    if (ioctl (fd, ioctl_choice, arg_choice) == ERROR)
    {
        (void) printf ("ioctl (%d) failed: ", ioctl_choice);
        printErrno (errnoGet ());
        return;
    }

    if (print_choice == TIME_DECIMAL)
        (void) tmPrint (&tfpTM);
    else if (print_choice == TIME_BINARY)
        (void) tvPrint (&tfpTV);

}

/*******************************************************************************/
/* Description : pci_strobe_routines */
/*******************************************************************************/
void pci_strobe_routines ()
{
int strobe_secs;
TFP_TM tfpTM;

printf("\n\tEnter number of seconds till strobe : ");
scanf("%i", &strobe_secs);

if ((strobe_secs < 1) || (strobe_secs > 59))
   /* can't do this */
{
   printf("strobe must be between 1 and 59 seconds for this test \n");
   return;
}
```

```
ioctl (fd, SELTIMEFORMAT, TIME_DECIMAL); /* decimal time format */

/* Get current time */
ioctl (fd, RDTIMETMREQ, (int)&tfpTM);
printf("\nCurrent time is: \n");
tmPrint(&tfpTM);

if (tfpTM.tm.tm_sec + strobe_secs < 60)
   tfpTM.tm.tm_sec = tfpTM.tm.tm_sec + strobe_secs;
else
{
   tfpTM.tm.tm_sec = tfpTM.tm.tm_sec + strobe_secs - 60;
   if (tfpTM.tm.tm_min < 59)
      tfpTM.tm.tm_min = tfpTM.tm.tm_min + 1; /* doesn't handle the roll over */
   else
   {
      printf("Test program is not smart enough to strobe now\n");
      return;
   }
}

printf ("\nStrobe time is: \n");
tmPrint(&tfpTM);

ioctl (fd, SETSTROBETM, (int)&tfpTM);

ioctl (fd, BC635_INT_WAITONSTROBE, 0); /* Should be a task that starts up before the
                                strobe - but since our example is in seconds
                                we have time to it after */

printf("Got Strobe\n");
}

/*****************************************************************************/
/* Description : pci_interrupt_routines */
/*****************************************************************************/
void pci_interrupt_routines ()
{
    int choice;

    printf ("\n\n");
    printf ("1. Wait on Event\n");
    printf ("2. Wait on Periodic\n");
    printf ("3. Wait on Strobe\n");
    printf ("4. Wait on 1 PPS\n");
```

```
    printf (“5. Wait on GPS\n”);
    printf (“   Select : “);
    scanf (“%i”, &choice);

    switch (choice)
    {
    case 1:
        ioctl (fd, BC635_INT_WAITONEVENT, 0);
        printf (“Got EVENT\n”);
        break;
    case 2:
      {
        int i;
        for (i=0; i < 5; i++)
        {
           ioctl (fd, BC635_INT_WAITONPERIODIC,0);
           printf (“Got PERIODIC\n”);
        }
      }
        break;
    case 3:
        ioctl (fd, BC635_INT_WAITONSTROBE,0);
        printf (“Got STROBE\n”);
        break;
    case 4:
      {
        int i;
        for (i=0; i < 5; i++)
        {
           ioctl (fd, BC635_INT_WAITON1PPS, 0);
           printf (“Got 1PPS\n”);
        }
      }
        break;
    case 5:
        ioctl (fd, BC635_INT_WAITONGPS, 0);
        printf (“Got GPS\n”);
        break;
    default:
        break;
    }

}
/*****************************************************************************/
/* Description : pci_intstat_routines() */
```

```
/*****************************************************************************/
void pci_intstat_routines ()
{
     int choice;
     UINT32 intstat;

     printf (“\n\n”);
     printf (“1. Read Intstat \n”);
     printf (“2. Set Intstat \n”);
     printf (“3. Read Intmask \n”);
     printf (“4. Set Intmask \n”);
     printf (“   Select : “);
     scanf (“%i”, &choice);

     switch (choice)
     {
       case 1:
          intstat = 0;
          ioctl (fd, RDINTSTAT, (int)&intstat);
          printf (“INTSTAT = %x\n”, intstat & INT_BITMASK);
          break;
       case 2:
          printf (“/nEnter value : “);
          scanf (“%i”, &intstat);
          ioctl (fd, SETINTSTAT, intstat);
          break;
       case 3:
          intstat = 0;
          ioctl (fd, RDINTMASK, (int)&intstat);
          printf (“INTMASK = %x\n”, intstat & INT_BITMASK);
          break;
       case 4:
          printf (“/nEnter value : “);
          scanf (“%i”, &intstat);
          ioctl (fd, SETINTMASK, intstat);
          break;

       default:
          break;
      }
}
/*****************************************************************************/
/* Description : Main Menu */
/*****************************************************************************/
void pci_sw_reset ()
```

```
{
   ioctl (fd, SOFTWARERESET, 0);
}

void pci_set_offsets ()
{
   UINT32 offset;
   UINT32 halfhour;
   int    choice;
   printf (“\n\n 1. Set Generator time offset \n”);
   printf (“2. Set Local Time Offset \n”);
   printf (“   Select : “);
   scanf (“%i”, &choice);

   printf (“Enter Offset : “);
   scanf (“%i”, &offset);
   printf (“Enter Half Hour (0 or 1) : “);
   scanf (“%i”, &halfhour);

   switch (choice)
   {
      case 1:
       {
         struct tcgenoffset genoffset;

         genoffset.offset = (INT16) offset;
         genoffset.half_hour = (UINT8) halfhour;

         ioctl (fd, SETGENTMOFFSET, (int)&genoffset);

       }
         break;
      case 2:
      {
        struct loctmoffset locoffset;

         locoffset.offset = (INT16) offset;
         locoffset.half_hour = (UINT8) halfhour;

         ioctl (fd, SETLOCTMOFFSET, (int) &locoffset);

      }
         break;
      default:
         break;
```

```
   }
}

void pci_set_prop_delay()
{
  UINT32 delay;

  printf("Enter delay : ");
  scanf ("%i", &delay);

  ioctl (fd, SETPROPDELAY, delay);
}

void pci_set_leapsec_event ()
{
   INT32 tmp;
   UINT32 utmp;
   struct leapseconds leapsec;

   printf ("Enter Leap Sec Flag : ");
   scanf ("%i", &tmp);
   leapsec.ls_flag = (INT8) tmp;
   printf ("Enter Leap Time (Unix Format) : ");
   scanf ("%i", &utmp);
   leapsec.leap_tm = tmp;

   ioctl (fd, SETLEAPSECEVENT, (int) &leapsec);


}

void pci_gps_example ()
{
    TFP_GPS_PACKET gps_packet;
    UINT32 id;
    int i;

    printf("Enter Packet to receive (hex - i.e. 4a) : ");
    scanf ("%x", &id);
    gps_packet.packet_id = id;
    gps_packet.packet_length = 0;

    ioctl (fd, REQGPSPACKET, (int)&gps_packet);

    if (gps_packet.packet_length == 0)
```

```
        printf("GPS Example - Could not receive GPS System Message\n");

    else
    {
        printf("Packet id = %x, lenght=%d\n", gps_packet.packet_id,
gps_packet.packet_length);
        for (i=0; i < gps_packet.packet_length-1; i++)
            printf ("[%x] = %x\n", i, gps_packet.packet_data[i]);
    }
}

void pci_gps_example2 ()
{
    TFP_GPS_PACKET gps_packet;
    UINT32 id;
    int i,tmp;

    printf("Enter Packet to Send (hex - i.e. 4a) : ");
    scanf ("%x", &id);
    gps_packet.packet_id = id;
    printf("Enter Packet Length (decimal plus one) : ");
    scanf ("%d", &tmp);
    gps_packet.packet_length = (UINT8) tmp;
    for (i=0; i<gps_packet.packet_length-1;i++)
    {
        printf("Enter data (in hex) [%d] : ", i);
        scanf ("%x", &tmp);
        gps_packet.packet_data[i] = (UINT8) tmp;
    }

    ioctl (fd, SENDGPSPACKET, (int)&gps_packet);

}

void pci_gps_example3 ()
{
    TFP_GPS_SENDRECEIVE_PACKET gps_packet;
    UINT32 id;
    int i,tmp;

    printf("Enter Packet to Send (hex - i.e. 4a) : ");
    scanf ("%x", &id);
    gps_packet.send.packet_id = id;
    printf("Enter Packet Length (decimal plus one) : ");
    scanf ("%d", &tmp);
```

```
    gps_packet.send.packet_length = (UINT8) tmp;
    for (i=0; i<gps_packet.send.packet_length-1;i++)
    {
       printf("Enter data (in hex) [%d] : ", i);
       scanf ("%x", &tmp);
       gps_packet.send.packet_data[i] = (UINT8) tmp;
    }

    printf("Enter Packet to receive (hex - i.e. 4a) : ");
    scanf ("%x", &id);
    gps_packet.receive.packet_id = id;
    gps_packet.receive.packet_length = 0;



    ioctl (fd, MANREQGPSPACKET, (int)&gps_packet);

    if (gps_packet.receive.packet_length == 0)
       printf("GPS Example - Could not receive GPS System Message\n");

    else
    {
       printf("Packet id = %x, lenght=%d\n", gps_packet.receive.packet_id,
gps_packet.receive.packet_length);
       for (i=0; i < gps_packet.receive.packet_length-1; i++)
           printf ("[%x] = %x\n", i, gps_packet.receive.packet_data[i]);
    }



}

void pci_set_gps_mode ()
{

  int mode;

  printf ("\n\t 1) Disable");
  printf ("\n\t 2) Enable");
  printf ("\n\t    Select : ");
  scanf ("%i", &mode);

  ioctl (fd, SETGPSMDFLG, mode);
}
```

```
void pci_set_gps_time_format()
{

}

void pci_set_clock_source()
{
int clock;

printf ("\n\t Enter Clock Source (Internal = 1, External=2) : ");
scanf ("%i", &clock);

if (clock == 1)
   ioctl(fd, CONTROLTIMEBASE, CLOCK_INTERNAL);
else if (clock == 2)
   ioctl(fd, CONTROLTIMEBASE, CLOCK_EXTERNAL);
else
   printf ("Invalid value\n");

}

void pci_looping_time()
{
TFP_TV tfpTV;
int i;

ioctl (fd, SELTIMEFORMAT, TIME_BINARY); /* binary time format */

for (i = 0; i <= 10; i ++)
{
   /* Get current time */
   ioctl (fd, RDTIMETVREQ, (int)&tfpTV);
   printf("\nCurrent time is: \n");
   tvPrint(&tfpTV);
   taskDelay(60*2); /* every 2 seconds */
}
}

/*******************************************************************************/
/* Description : Main Menu */
/*******************************************************************************/
int
menu_main ()
{
```

```
int     menu;

do
{
    printf (“\n  Datum Inc. Bancomm-Timing Division\n\n”);
    printf (“    bc635PCI Time & Frequency Processor\n\n”);
    printf (“1.  Read time/Event Functions    14. Set Offsets \n”);
    printf (“2.  Select Operational Mode      15. Prop Delay \n”);
    printf (“3.  Select Time Code Format      16. Leap Sec \n”);
    printf (“4.  Select Output Freq           17. Capunlock \n”);
    printf (“5.  Select Time Format           18. GPS Example1 \n”);
    printf (“6.  Program Heartbeat            19. Set GPS Mode \n”);
    printf (“7.  Set Time Functions           20. Set GPS TM Fmt\n”);
    printf (“8.  Select Clock Source          21. GPS Example2\n”);
    printf (“9.  Current Settings             22. GPS Example3\n”);
    printf (“10. Strobe                       23. Looping Time\n”);
    printf (“11. Interrupt Routines\n”);
    printf (“12. Interrupt Stat Routines\n”);
    printf (“13. Software reset\n”);
    printf (“0.  Exit the Program \n\n”);
    printf (“    Select: “);
    scanf (“%i”, &menu);

    switch (menu)
    {
        case 1:
            read_time_menu ();
            break;
        case 2:
            pci_mode ();
            break;
        case 3:
            pci_time_code ();
            break;
        case 4:
            pci_out_freq ();
            break;
        case 5:
            pci_time_format ();
            break;
        case 6:
            pci_heartbeat ();
            break;
        case 7:
            pci_set_time ();
```

```
        break;
    case 8:
        pci_set_clock_source();
        break;
    case 9:
        pci_req_current_settings ();
        break;
    case 10:
        pci_strobe_routines ();
        break;
    case 11:
        pci_interrupt_routines ();
        break;
    case 12:
        pci_intstat_routines ();
        break;
    case 13:
        pci_sw_reset ();
        break;
    case 14:
        pci_set_offsets ();
        break;
    case 15:
        pci_set_prop_delay();
        break;
    case 16:
        pci_set_leapsec_event();
        break;
    case 17:
        ioctl (fd, CAPUNLOCK, 0);
        break;
    case 18:
        pci_gps_example();
        break;
    case 19:
        pci_set_gps_mode();
        break;
    case 20:
        pci_set_gps_time_format();
        break;
     case 21:
        pci_gps_example2();
        break;
    case 22:
        pci_gps_example3();
```

```
                break;
            case 23:
                pci_looping_time();
                break;
            case 0:
            default:
                return OK;
        }
    } while (1);
}
```