VxWorks Device Driver Guide for the bc635/637VME Time and Frequency Processor GPS Receiver

User's Guide
April, 1997

CHAPTER ONE
INTRODUCTION

1.0 INTRODUCTION

1.1 SCOPE

This document describes the use of the VxWorks device driver for the Datum Inc. bc635/637VME and bc350/357VXI Time & Frequency Processor / GPS Receiver (henceforth referred to as BTFP) running in a real-time VxWorks system.

1.2 RELATED DOCUMENTS

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

Table One
Related Publications

Document Name	Date	Vendor
VxWorks Programmer's Guide (Release 5.3)	1995	Wind River Systems
VxWorks Reference Manual (Release 5.3 Beta)	1995	Wind River Systems
bc635VME/bc350VXI Time And Frequency Processor User's Guide	1994	Datum Inc., Datum Inc., Division
bc637VME/bc357VXI GPS Satellite Receiver Addendum User's Guide	1993	Datum Inc., Datum Inc., Division

1.3 PRODUCT OVERVIEW

The bc63VME/bc350VXI Time and Frequency Processor combines IRIG time code generation and processing on a single card and features a real time clock. The bc637VME/bc3357VXI contains additional circuitry to support a GPS antenna/receiver in addition to providing all the capability of the bc635VME/bc350VXI.

The BTFP device driver runs under the VxWorks Real-Time Operating System. Software support includes an object loadable driver, diagnostics, a sample application, and user's guide (this document).

1.4 COMPATIBILITY

This release of the BTFP VxWorks device driver is compatible with the following releases of VxWorks:

- VxWorks 5.1, SPARC, MVME167, MVME162
- VxWorks 5.1.1, SPARC, MVME167, MVME162
- VxWorks 5.2, SPARC, MVME167, MVME162
- VxWorks 5.3, PowerPC 604

SPARC and 68K targets do require different object files.

CHAPTER TWO
THEORY OF OPERATION

2.0 THEORY OF OPERATION

2.1 OVERVIEW

This chapter describes how to set up and use the BTFP device driver and interface hardware through a VxWorks real-time task. Before the device is accessed, the driver needs to be installed into the Device Table using the *btfpDevCreate* command. After this is complete, the device may be accessed using the *open*, *close*, *read*, *write*, and *ioctl* commands. Before continuing on in this manual, the user should be familiar with these functions, which are described in the VxWorks manuals.

Each VxWorks task that will use the device should have the following statements near the beginning of the program:

#include "/<path>/bctfp.app.h"

where "path" is the host directory containing the BTFP driver header files. This statement will get the interface header file for this device. The header file defines the constants needed to access the board. His header file has been reproduced as Appendix A of this manual, and the remainder of this manual assumes the constants in the header files are being used.

2.2 INSTALLING THE DRIVER INTO THE DEVICE TABLE

The BTFP driver is installed into the VxWorks device table with the *btfpDevCreate* call. This call must be made before any other VxWorks call attempts to access this device. The parameters for the *btfpDevCreate* call are:

btfpDevCreate(devname, csr, level, vector)

where the parameters are as follows for the BTFP:

devname	-	20 character name, usually "btfp0" or "btfp/1". The last character should indicate the board number.
csr	-	VME A16 address for the board, 0xZZZZ. The "ZZZZ" contains the VME A16 address set on the BTFP by the user according to the corresponding BTFP device. Refer to the Datum Inc. Manual for further information.

level	-	VME interrupt level (1-7). Level 1 suggested.
vector	-	VME interrupt vector (0-255). Vectors 128-255 are user vectors. Those vectors are highly recommended. Other vectors may interfere with VxWorks operation.

Examples:

```
btfpDevCreate("btfp0", 0xff00, 1,240);
btfpDevCreate("btfp1",0x2000, 1,132);
```

NOTE: VxWorks has a small bug which causes non-autovectors on a SPARC (vectors other than the ones assigned by VxWorks) to print “Uninitialized interrupt”. This is not a driver bug, or an error with a user’s choice of vector.

The *btfpDevCreate* call will not complete but will return a failure status if the BTFP board is not configured at the csr address passed to this call.

Device Names

Each VxWorks device has been given a unique name that is used with the *open* command to open the device. The device name for the BTFP takes on the following form:

btfp# or btfp/#

The ’#’ is a digit from 0 to 9; this is needed because the VxWorks BTFP driver can support up to ten boards. If VxWorks has one BTFP device, its device name would be "btfp0" or "btfp/0". If VxWorks has ten BTFP devices, the first BTFP device would have the device name of "btfp0" or "btfp/0", the second BTFP device would have the device name of "btfp1" or "btfp/1", and so on to the tenth BTFP device which would have the device name of "btfp9" or "btfp/9".

2.3 OPENING/CLOSING THE DEVICE

To obtain a file descriptor for accessing the device, an *open* call must be performed. To close the device so that other tasks may access it, *close* must be called with the corresponding file descriptor. Examples of the calls are listed below.

```
int btfpFd, stat;

btfpFd = open ("btfp0",0,0);
if (btfpFd < 0) /* process for an error condition */

...

stat = close (btfpFd);
if (stat != 0) /* process for an error condition */
```

IMPORTANT: It is recommended that only one task have a BTFP open at any one time. If more tasks open the device, it could potentially lead to problems if the second task changes registers on the BTFP without the first task knowing about it. The driver will not prevent more tasks from opening the device at this time.

2.4 THE DATUM INC. TIME FREQUENCY PROCESSOR BOARD

The BTFP processes timecode information in any of the following formats: IRIGA, IRIGB, 2137, NASA36, and XR3. It is also capable of keeping time with a drift of less than 2ms/hour without a signal source. In addition, the bc637/bc357 models are capable of receiving GPS signals and providing highly accurate position information in a timely manner. For additional information regarding hardware performance, hardware installation, and hardware use, see the Datum manuals.

2.5 BTFP I/O CONTROL COMMANDS

The VxWorks driver interface provides the *ioctl* routine which handles miscellaneous commands to/from the BTFP device driver. These commands allow setup and control of the BTFP interface. The *ioctl* commands can read and write the board registers, set/clear the BTFP function bits, and perform board diagnostics.

The *ioctl* command passes the *ioctl* command and the address of a buffer to the BTFP device driver. The following sections will define and explain the *ioctl* commands for the BTFP.

Unless otherwise stated, ioctls will work only with unsigned values. Using signed values where not explicitly required to do so will cause unpredictable results.

NOTE: Performing some ioctl commands when two tasks have opened one device can have unpredictable results. The driver doesn’t prevent more than one task opening the device, so if this is done, precautions should be taken to insure that all tasks set up the board to the desired mode before continuing with an operation as another task may have left the board in a different state from the expected state. Critical sections of code should be marked with VxWorks semaphores or taskLocks.

2.5.1 INTCONN

Command
INTCONN

Purpose
To attach a signal handler to the BTFP driver.

Inputs

Argument Type	Description
void (*func)(int)	Pointer to your function handler.

Outputs

Returns	Description
OK	signal handler connected to the driver.
ERROR	signal handler failed to connect to the driver (note: this shouldn't happen because the driver just assigns a value.)

Description
This driver function hooks a user supplied function into the driver interrupt. When the driver interrupt code is called during an unmasked BTFP interrupt (the BTFP mask register is discussed later), the interrupt code checks to see if the user has supplied a signal handler and wants to be notified of interrupts (see the SIGNAL ioctl). If both conditions are true, the user's signal handler is called with the BTFP's INTSTAT register (discussed later). On return from the user handler, the driver interrupt clears the appropriate INTSTAT bits to enable new interrupts to occur.

If the user doesn't want to be notified of interrupts, interrupts from the BTFP are NOT automatically shut off. The driver interrupt routine continues to process (ignore) interrupts. However, if a Data Packet Available interrupt occurs, the driver interrupt code reads the output fifo. This enables the user to shut off signal notification, yet keep the BTFP outfifo up to date. This helps prevent the user from reading stale data from the fifo. It doesn't eliminate the problem completely, however, so it is a good idea to clear the fifo periodically, especially in GPS mode.

Example

The following code example will open a BTFP, attach a signal handler to the driver, enable signal notification, and unmask the 1PPS interrupt on the BTFP board. The interrupt will happen once per second and print a message.

```
#include "bctfp.app.h"

void myhandler(int arg)
{
        printf(“1PPS Signal\n”);
}
test()
{
        int dev,status,stat;
        args ioctlArgs;
        .
        .
        .
        /* Open the device */
        /* open example */
        dev = open ("btfp0");
        if (dev < 0)
        {



         printf("Error opening device btfp0, aborting.\n");
         /* close example */
         close(dev);
         return;
        }
        /* Issue IOCTL call */
        /* DEBUG example (turns off debugging) */
        stat = ioctl (dev, DEBUG, 0x0);

        /* PACKET example (turns off packet mode) */
        stat = ioctl (dev, PACKET, OFF);

        /* INTCONN example */
        stat = ioctl (dev, INTCONN, (int)myhandler);
        if (stat != 0)
        {
          printf("Error -- ioctl call INTCONN failed.\n");
          close(dev);
```

```
  return;
}

/* tell driver we want to be notified of ints */
ioctl (dev, SIGNAL, ON); /* signal command */

/* Clear the Mask register for 1PPS interrupts */
/* WRITEREG example */
ioctlArgs.reg_id = R_MASK;
ioctlArgs.reg_val = INT_1PPS;
ioctl(dev,WRITEREG,(int)&ioctlArgs);

/* Clear any previous interrups */
ioctlArgs.reg_id = R_INTMASK;
ioctl(dev,WRITEREG, (int)&ioctlArgs);

/* Message will be printed from here on out */
.
        .
        .
        close(dev);
}
```

2.5.2 SIGNAL

Command
SIGNAL

Purpose
To inform the driver whether user interrupt notification should be turned on or off.

Inputs

Argument	Description
ON	Turns on user interrupt notification (used with INTCONN).
OFF	Turns off user interrupt notification.

Outputs

Returns	Description
OK	Indicates driver state was changed.
ERROR	Indicates driver failed to change state. This error will not occur.

Description
Issuing this command with the ON argument will allow the driver to call the user interrupt handler installed with INTCONN. If the user has not installed an interrupt handler, no notification will occur. However, the driver will fail to clear the fifo on DPA interrupts. It is advised that only the SIGNAL OFF command be issued without installing a user interrupt handler if the out fifo needs to be updated.

The OFF argument has the opposite effect of the ON argument. Board interrupts that have been enabled will NOT be disabled as the driver interrupt is capable of handling all enabled interrupts that occur. If in the SIGNAL OFF mode a DPA interrupt occurs, the fifo is read and the data is discarded. This keeps the fifo current.

Example
See the example for INTCONN for an example of using the SIGNAL command.

2.5.3 PACKET

Command
PACKET

Purpose
To switch the driver to and from packet mode.

Inputs

Argument	Description
ON OFF	Turn on packet mode. Turn off packet mode.

Outputs

Returns	Description
OK	Indicates driver state was changed.
ERROR	Indicates driver failed to change state. This error will not occur.

Description
This routine will change the driver to and from packet mode. In packet mode, a read from the BTFP device will read packets from the outfifo until no more packets are available. The packet will be "stripped" according to the guidelines laid out in the Datum Inc. Manuals. A write to the outfifo will write the data in a buffer in a packet format. The packet will be "stuffed" if necessary according to the guidelines laid out in the Datum Inc. Manuals. Combined with SIGNAL mode, this can be a powerful way to achieve asynchronous operation of the board using the driver.

If packet mode is off, reads from the device will latch and read the time registers. The buffer output format is of the form:

```
xxx...xx...xx...xx...xxx...xxx...x\0
```

Where the X's represent digits in a time string. The first three comprise the number of days. The next two the number of hours. The next two are the minutes. The next two are the seconds. The next three are milliseconds. The next three are microseconds. The last one is 10E-7 seconds. The string is null terminated.

A write in packet off mode will wait for the 1PPS interrupt before writing a packet. Generally this is used to synchronize a packet 'B' write (Set major/minor time) to the board. The driver doesn't restrict the user to only writing this packet to the board. However, this packet may be the only packet that should really be synchronized. Again, combined with SIGNAL ON mode, this creates a powerful way to asynchronously use the BTFP.

Example
See INTCONN for an example of using PACKET.

2.5.4 DEBUG

Command
DEBUG

Purpose
Turn on or off internal driver debugging information.

Inputs

Argument **(bit #)**	Description
1	Turns on internal debug locator statements.
2	Turns on internal function entry-exit statements.
3	Turns on internal data value statements.
4	Controls interrupt mode.
5-8	Same as bits 1-3 for interrupt mode.

Outputs
Only returns OK.

Description
This command controls the amount of debugging information the driver outputs. The driver is capable of outputting a tremendous amount of debugging information, however, it is recommended that only the necessary output be used because the VxWorks logTask will throw away debugging messages if too many are printed.

The first bit controls the locator statement print. This is used for locating the last thing the driver did. Typically, this sort of statement is used to follow the flow of the driver just before a crash occurred. It is only useful with driver source code.

The second bit controls the drivers entry and exit prints. This is a more general locator type statement and is typically used the same way a locator statement is used, but with much coarser granularity. Any bug can be tracked to a function with this type of debugging output.

The third, and probably most useful, bit controls the internal data printout. This information contains the values of internal data structures used by the driver. This is typically a good check to make sure user data is being passed to the driver correctly.

The fourth bit controls the interrupt mode. When this bit is set, bits 5-8 are used the same as bits 1-3. The difference is that bits 1-3 are ignored and are set to 0 to insure that only interrupt level debugging statements are printed. This means that any function called by the driver's interrupt handler that prints debug information will print its debug info if the right bits 5-8 are on. This was implemented because VxWorks logTask discards messages when too many are output. This helps track a problem to the regular driver, or the driver's interrupt handling.

Think of all of the above as debugging levels. As more levels are requested, more data is output.

Example
See INTCONN for an example of using DEBUG.

2.5.5 ROUTFIFO

Command
ROUTFIFO

Purpose
Read one byte from outfifo.

Inputs

Argument	Description
pointer to char	pointer to location where byte read will be stored.

RETURNS
OK only.

Description
This function will read one byte from the outfifo and put it into the specified byte buffer. This function can also be accomplished with the READREG ioctl discussed below, however, for a few bytes, this command is simpler.

NOTE: This is a way to get directly to the fifo. However, this is not recommended. The fastest way to read the fifo is through the read command in packet mode. This will also accomplish all of the packet stripping necessary, and performs all of the fifo empty checks.

Example
See WINFIFO for an example of using ROUTFIFO.

2.5.6 WINFIFO

Command
WINFIFO

Purpose
Write one byte to the input fifo.

Inputs

Argument	Description
char	Character to be written to the fifo.

RETURNS
OK only.

Description:
This function takes the byte provided by argument and writes it to the infifo.

NOTE: This is a way to get directly to the fifo, however, this is not recommended. The simplest, fastest way to write to the fifo is by using the write command in packet mode. This takes care of all of the acknowledges and stuffs the packet if necessary.

Example
The following function will write a packet to the infifo and check for an acknowledgment from the BTFP. It will then set the BTFP in action on the sent packet. Finally it will read a byte from the outfifo. This example assumes that btfpDevCreate has already been called and that the device has been successfully opened.

```
/* much of this code doesn’t check return status. But it should be done. */
#include “bctfp.app.h”

test()
{
        unsigned char infifo;
        args ioctlArgs;
        unsigned short regval;
        int dev;
        .
        .
        .
        /* set the BTFP into diagnostic mode */
        /* start by writing packet ‘A7’ to board */
        infifo = SOH;
        ioctl(dev,WINFIFO, infifo);
        infifo = ‘A’
        ioctl(dev,WINFIFO, infifo);
        infifo = ‘7’;
        ioctl(dev,WINFIFO, infifo);
        infifo = ETB;
        ioctl(dev,WINFIFO, infifo);
        /* check to make sure board ack’d packet */
        /* make sure that we can check the
         packet acknowledged bit */
        ioctlArgs.reg_id = R_ACK;
        ioctlArgs.reg_val = ACK_INACT | ACK_INFIFO;
        ioctl(dev,WRITEREG, (int)&ioctlArgs);
        /* wait for packet to process */
        do {
        ioctl(dev,READREG, (int)&ioctlArgs);
        } while((ioctlArgs.reg_val | ACK_INFIFO) == 0)

        /* need to acknowledge any previous response
         packets if we expect a response. In this case
         no response was expected, but it was done
         anyway. This is actually in the wrong place,
         but we can leave it here because if you were
         to process a second packet for which you expected
         a response, you’d need this here.*/
        ioctlArgs.reg_id = R_ACK;
        ioctlArgs.reg_val = ACK_DPA;
        ioctl(dev, WRITEREG, (int)&ioctlArgs);

        /* read an outfifo byte */
        ioctl(dev, ROUTFIFO, (int)&infifo);
        printf(“Character read was: 0x%x\n”,infifo);
        close(dev);
}
```

2.5.7 READREG

Command
READREG

Purpose
Read a BTFP register.

Inputs

Argument	Description
args *	Pointer to an args ioctl struct.

Outputs

Returns	Description
unsigned short	The value of the register is returned in the reg_val element of the args structure.

RETURNS
OK only.

Description
This function takes the value stored in the reg_id element of the args structure and uses this as a pointer to the correct register to read. It reads the register and stores the value into the reg_val element. If a pointer to an array of 30 unsigned shorts is used and the first element is assigned the value R_ALL, then the values of each of the registers will be read and stored into the array. Certain registers, namely the fifo register, will be skipped as reading this register inadvertently could lead to loss of data.

Example
The following example will open the BTFP and read the INTSTAT register, then all of the registers. A typical use of this command would be to poll the INTSTAT register. This example assumes btfpDevCreate has already been called.

```
#include “bctfp.app.h”

test()
{
        args ioctlArgs;
        unsigned short regvals[30];
        int stat;
        int dev, i;
        .
        .
        .
        dev = open(“btfp0”,0,0);
        if(dev < 0)
        {
         printf(“Error opening BTFP device.\n”);
         return;
        }
        /* example of reading just the INTSTAT register */
        ioctlArgs.reg_id = R_INTSTAT;
        stat = ioctl(dev,READREG,(int)&ioctlArgs);
        if(stat == ERROR)
        {
         printf(“Error reading INTSTAT.\n”);
         close(dev);
         return ERROR;
        }
        printf(“INSTAT reg = 0x%x\n”,ioctlArgs.reg_val);

        /* example of reading all registers at once */
        regvals[0] = R_ALL;
        stat = ioctl(dev,READREG,(int)&regvals[0]);
        if(stat == ERROR)
        {
         printf(“Error reading all registers.\n”);
         close(dev);
         return ERROR;
        }
        /* there are really only 23 registers */
        for(i = 0; i < 24; i++)
        {
         printf(“Register %d = 0x%x\n”,i,regvals[i]);
        }
        .
        .
        .
        close(dev);
}
```

2.5.8 WRITEREG

Command
WRITEREG

Purpose
Write a value to the specified register.

Inputs

Argument	Description
args *	Pointer to an args structure.

Returns
OK only.

Description
This function will take the value in the reg_id element of the args structure and use it as a pointer to the BTFP register to write the value in the reg_val element to. If the R_ALL value is used in the reg_id element, it will be ignored.

Example
See INTCONN for an example of using WRITEREG.

2.5.9 P_MODE_SEL

Command
P_MODE_SEL

Purpose
Change the TFPs mode.

Inputs

Argument	Description
char *	Pointer to a mode character. Valid mode characters are: A_TIMECODE_DEC A_FREE_RUNNING A_EXT_1PPS A_REAL_TIME_CLK A_DIGITAL_SYNC A_GPS_ONBOARD A_GPS_ANTENNA A_DIAGNOSTIC See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘A’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘A’ packet to the TFP hardware.

Example
See the example program later in the manual.

2.5.10 P_CMD_INP

Command
P_CMD_INP

Purpose
Direct TFP to take appropriate action.

Inputs

Argument	Description
char *	Pointer to a command character. Valid command characters are: C_SOFT_RESET C_JAMSYNC C_BUF_RTC See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘C’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘C’ packet to the TFP hardware.

Example
See the example program later in the manual.

2.5.11 P_SEL_CLK

Command
P_SEL_CLK

Purpose
Select TFP clock source.

Inputs

Argument	Description
char *	Pointer to a mode character. Valid mode characters are: I_EXT I_INT See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘I’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘I’ packet to the TFP hardware.

Example
See the example program later in the manual.

2.5.12 P_SEL_GCODE

Command
P_SEL_GCODE

Purpose
Selects either IRIGB amplitude modulated or IRIGH DC level shift only mode.

Inputs

Argument	Description
char *	Pointer to a mode character. Valid mode characters are: K_IRIGB K_IRIGH See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘K’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘K’ packet to the TFP hardware.

Example
See the example program later in the manual.

2.5.13 P_DATA_REQ

Command
P_DATA_REQ

Purpose
Sends a data request packet to TFP. Response returned in FIFO.

Inputs

Argument	Description
char *	Pointer to a format character. Valid format characters are: O_FMT0 O_FMT1 O_FMT2 O_FMT3 O_FMT4 See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘O’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘O’ packet to the TFP hardware.

Example
See the example program later in the manual.

2.5.14 P_MJTM_SET

Command
P_MJTM_SET

Purpose
Set major time for modes 1 and 2 of the TFP.

Inputs

Argument	Description
int *	Pointer to an array of four integers. The integers are converted to the proper format for the TFP. Integers in order are: Days (001-365) Hours (00-23) Minutes (00-59) Seconds (00-59) See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet 'B'.

Returns
ERROR if packet couldn't be sent.

Description
This function sends a properly formatted 'B' packet to the TFP hardware using the supplied integers. Four integers must be supplied.

Example
See the example program later in the manual.

2.5.15 P_DAC_LD

Command
P_DAC_LD

Purpose
Direct TFP to take appropriate action.

Inputs

Argument	Description
int *	Pointer to an integer. The integer is correctly formatted according to packet ‘D’ specs in the hardware manual. High bits are first. For example: 0x1234. The ‘1’ in this number represents the high bits. See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘D’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘D’ packet to the TFP hardware using the supplied integer.

Example
See the example program later in the manual.

2.5.16 P_HBT_CTL

Command
P_HBT_CTL

Purpose
Establishes frequency of TFP output periodics.

Inputs

Argument	Description
int *	Pointer to two integers. The first integer must be either 2 or 5. The second integer is four nibbles long. The first two nibbles represent the value m1 discussed in the hardware manual. The second two nibbles represent m2, also discussed in the hardware manual. The formula for picking m1 and m2 can be found in the hardware manual under packet ‘F’. High nibbles are towards the left. For example: 0x12345678 sets m1=0x1234 and m2=0x5678. See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘F’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘F’ packet to the TFP hardware using the supplied integers.

Example
See the example program later in the manual.

2.5.17 P_TC_FMT

Command
P_TC_FMT

Purpose
Set the format for TFP mode 0.

Inputs

Argument	Description
char *	Pointer to two characters. The first character is one of the following: H_IRIGA H_IRIGB H_2137 H_NASA36 H_XR3 The second must be one of these: 'M' - amplitude modulated sine wave 'D' - pulse code modulation (DC level shift) See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet 'H'.

Returns
ERROR if packet couldn't be sent.

Description
This function sends a properly formatted 'H' packet to the TFP hardware using the supplied characters.

Example

```
int status;
char dataVal[2];
.
.
.
dataVal[0] = H_IRIGB; /* set IRIGB */
dataVal[1] = 'D';          /* DC level shift */
status = ioctl(fd,P_TC_FMT,(INT)dataVal);
/* check status == ERROR here */
.
.
```

2.5.18 P_SET_RCLK

Command
P_SET_RCLK

Purpose
Set the real time clock.

Inputs

Argument	Description
int *	Pointer to six integers. Integers are entered as follows: Years (00-99) Month (01-12) Day of month (00-31) /* do your own checking */ Hours (00-23) Minutes (00-59) Seconds (00-59) See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet 'L'.

Returns
ERROR if packet couldn't be sent.

Description
This function sends a properly formatted 'L' packet to the TFP hardware using the supplied integers.

Example
See the example program later in the manual.

2.5.19 P_TIME_OFF

Command
P_TIME_OFF

Purpose
Set the local time offset.

Inputs

Argument	Description
int *	Pointer to one signed integer. Integer represents hours offset. A positive value is a positive offset, a negative value a negative offset. See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘M’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘M’ packet to the TFP hardware using the supplied integer.

Example
See the example program later in the manual.

2.5.20 P_OFF_CTL

Command
P_OFF_CTL

Purpose
Set the local time offset.

Inputs

Argument	Description
int *	Pointer to one signed and two unsigned integers. Integers are as follows: Milliseconds (-999-999) Microseconds (000-999) Nanoseconds (0-9) /* represents hundreds */ See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet 'M'.

Returns
ERROR if packet couldn't be sent.

Description
This function sends a properly formatted 'M' packet to the TFP hardware using the supplied integer.

Example
See the example program later in the manual.

2.5.21 P_PATH_SEL

Command
P_PATH_SEL

Purpose
Set up the processing path inside the TFP.

Inputs

Argument	Description
int *	Pointer to one integer. The integer is divided into two nibbles which are described in the hardware manual. Highest nibble comes first. See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘P’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘P’ packet to the TFP hardware using the supplied integer.

Example

```
int status;
int dataVal;
.
.
.
/* u = UPPER nibble, l = LOWER nibble */
/*             ul                          */
dataVal = 0x12; /* FIFO echo on, Leap Year on */
status = ioctl(fd,P_PATH_SEL,(INT)dataVal);
/* check status == ERROR here */
.
.
```

2.5.22 P_SET_YEAR

Command
P_SET_YEAR

Purpose
Set the TFP year for modes 0, 1, and 2.

Inputs

Argument	Description
int *	Pointer to one integer. The integer is formatted into a packet ‘S’ and sent to the hardware. The integer represents: Year (00-99) See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘S’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘S’ packet to the TFP hardware using the supplied integer.

Example

```
int status;
int dataVal;
.
.
.
dataVal = 95; /* Year ‘95 */
status = ioctl(fd,P_PATH_SEL,(INT)dataVal);
/* check status == ERROR here */
.
.
```

2.5.23 P_SET_GAIN

Command
P_SET_GAIN

Purpose
Set the disciplining gain.

Inputs

Argument	Description
int *	Pointer to two integers. Integers are entered as follows: Gain /* Most significant nibble first */ 0 or 1 /* 1=positive gain, 0=negative gain */ See the bc635VME/bc350VXI TIME AND FREQUENCY PROCESSOR manual for more information regarding packet ‘Q’.

Returns
ERROR if packet couldn’t be sent.

Description
This function sends a properly formatted ‘Q’ packet to the TFP hardware using the supplied integers.

Example
See the example program later in the manual.

2.5.24 LATCHTIME

Command
LATCHTIME

Purpose
Cause the BTFP hardware to latch the current time into the time registers.

Inputs

Argument	Description
Void	Takes no argument. Use zero for dummy value.

Returns
ERROR if time couldn't be latched.

Description
This function causes the BTFP hardware to latch the current time.

Example
See the example program later in the manual.

2.5.25 LATCHEVENT

Command
LATCHEVENT

Purpose
Cause the BTFP hardware to latch the current time into the event registers.

Inputs

Argument	Description
Void	Takes no argument. Use zero for dummy value.

Returns
ERROR if time couldn't be latched.

Description
This function causes the BTFP hardware to latch the current time into the event registers.

Example
See the example program later in the manual.

2.5.26 READTIME

Command
READTIME

Purpose
Read the time from the time registers.

Inputs

Argument	Description
BCDStruct	Pointer to the structure BCDStruct which contains fields for each of the time fields in the time and event registers.

Returns
ERROR if time couldn't be read.

Description
This function formats the time found in the time registers and generates the actual values for each element in the structure BCDStruct. The user can then use these values in regular comparisons or math functions.

Example
See the example program later in the manual.

2.5.27 READEVENT

Command
READEVENT

Purpose
Read the time from the event registers.

Inputs

Argument	Description
BCDStruct	Pointer to the structure BCDStruct which contains fields for each of the time fields in the time and event registers.

Description
This function formats the time found in the event registers and generates the actual values for each element in the structure BCDStruct. The user can then use these values in regular comparisons or math functions.

Example
See the example program later in the manual.

2.6 STATUS RETURN CODES

In all cases, if the driver detects some type of failure, an ERROR return code will result. In all other cases it depends on the operation being performed. For example, read returns the number of bytes read from the device unless an error occurs. However, many of the ioctl commands only return OK on success. Refer to the sections of this manual concerning the relevant function for more information on return codes.

2.7 READING AND WRITING

Reading and writing are the two main operations the driver performs. Writing to the BTFP device is always done using the infifo of the BTFP. The format of the packets written to the infifo are described in the Datum Inc. Manuals. Reading on the other hand may involve either the time registers, or the outfifo depending on the driver mode. Before going into reading and writing to the BTFP, a short explanation of the driver modes is in order.

There are two ioctl commands related to the driver modes. The two commands are PACKET and SIGNAL. The PACKET command changes the drivers interpretation of reading and writing to the BTFP. When PACKET mode is ON, the driver deals only with the in and out fifos of the BTFP board. When PACKET mode is OFF, the driver may read certain BTFP registers. It works as follows:

- With PACKET mode ON, reading from the BTFP will cause the driver to read “packets” from the outfifo. It will read packets from the fifo until either the fifo is empty, or the user supplied buffer is full. Writing to the BTFP with PACKET mode ON causes the driver to write packets to the BTFP’s infifo as soon as a packet is received. Packets are discussed in greater detail in the Datum Inc. Manuals.

- With PACKET mode OFF, reading from the BTFP will cause the driver to latch the time registers by reading the TIMEREQ register. Any read to this register causes the TIME# (# is 0-4) registers to be latched. The driver will read each register and format a time string for use by the user. Writing to the BTFP with PACKET mode OFF will cause the driver to wait for the 1PPS interrupt, thereby synchronizing the packet write on the second. Any packet can be written on the interrupt, but writing a packet ‘B’ on this interrupt will set the major time and is generally how this operation is used.

SIGNAL mode is used to tell the driver when a user supplied interrupt handler needs to be notified of a BTFP interrupt. This is useful for operating the BTOF in an asynchronous fashion. If CR SIGNAL mode is OFF, the user is never notified of events from the BTFP. However, this doesn’t shut off interrupts. The driver still continues to process interrupts received from the BTFP. This was done to facilitate continuous fifo updates. When SIGNAL mode is OFF and a DPA interrupt is received, the driver’s interrupt handler will read the fifo, thereby clearing it (this lets the user ignore packets and let the driver “drain” the outfifo independently). It is therefore recommended that the user connect an interrupt handler to the driver BEFORE enabling interrupts and setting the SIGNAL mode to ON. This way no loss of data occurs.

When SIGNAL mode is ON, a user supplied interrupt handler is called and passed the INSTAT BTFP register value.

Reading returns the number of bytes read in packet mode on. It returns the user supplied value in packet mode off. It returns error if an error condition is detected such as if the BTFP times out.

Writing always returns the number of bytes actually written unless an error occurs. Then ERROR is returned.

2.7.1 READING

With PACKET mode OFF, the time registers are read as described above. It must be noted, however, that the full time string is always returned and the user supplied length is ignored except as a return value. Therefore, at this time, space must be allocated for enough bytes to enclose the entire string, not just the part specified in the number of bytes. If this is not done, many errors in user code could result.

With PACKET mode ON, packets are stripped according to the Datum technical manuals. Therefore, the user doesn't have to manually strip the packets.

NOTE: When reading packets from the buffer, the BTFP operates much as a queued response system. This means the outfifo queues response packets generated from some packets entered into the infifo. For example, if two 'O' packets were written to the BTFP board, both responses would appear in the outfifo, and one read might read both of them. Therefore, the user supplied buffer might contain both response packets. The user therefore must scan the supplied buffer for the appropriate response. This is especially true of GPS packets as one written packet may generate more than one packet read. A good example of this is the 0x37 GPS packet. This packet requests a position fix. The response to a position fix generates many packets. If the user is only interested in the actual position packet for example, the buffer must be scanned for it. It is also possible to let the driver read many packets from the outfifo before looking at the user supplied buffer. In this case it is especially important to scan the buffer for the correct position of the desired packet. Note that if more than one of a particular packet occur in the user supplied buffer, the ones occurring later are the newer ones. So if you are interested in the latest position fix, it may be necessary to read the entire buffer to find the latest position response packet.

2.7.2 WRITING

When writing to the buffer, it is important to clear the ACK_DPA bit and the INT_DPA bit of the ACK and INSTAT registers respectively. It should be done before the write. This is just in case a packet was already in the outfifo that hadn't been acknowledged. This might happen if packets are requested and not read right away. If this is done, make sure NOT to clear the ACK_MORE bit of the ACK register. This is the bit that indicates that more data still exists in the outfifo. The driver uses this bit to determine whether to keep reading the fifo or not. If this bit is cleared, not only will none of the data be read, but the outfifo is cleared by the BTFP so the data will be lost.

Also note that writing to the BTFP in PACKET mode OFF, it is important to clear the INT_1PPS bit of the INSTAT register. This is so the write operation will actually be synched with the 1 PPS interrupt.

CHAPTER THREE
GPS RESPONSE SYSTEM

3.0 GPS RESPONSE SYSTEM

The GPS Response System is an application software provided for those users of the Datum Time Frequency Processor that have the GPS upgrade. This system is provided as a quick way to get GPS applications off the ground. An example program, btfpGPStest.c, is provided to users writing their own applications. The Response System only provides for those GPS packets with either determined responses or no response. This means that a GPS position request is not provided because the possible responses depend on the users board I/O configuration. The packets the response system supports are as follows:

- 0x1f - request software version
- 0x21 - request current time
- 0x22 - mode select
- 0x23 - initial position (ECEF XYZ)
- 0x24 - request position fix mode
- 0x25 - Initiate reset/self test
- 0x26 - Request health
- 0x27 - Request signal levels
- 0x2C - Set/Req operating parameters
- 0x35 - Set/Req I/O parameters

Most of the supported packets also have example usage in the btfpGPStest.c. To use the response system you must include bctfp.gpsapp.h in your program. This contains the type definitions of the responses returned by the Response System. Please note that the response system must be loaded after the driver is loaded to use it. Follow these steps to use the response system (these steps assume that you have changed directories to the correct one for the driver software).

1. Load in the driver software.

   -> *ld < btfpdrv.o*

2. Load in the response software.

   -> *ld < bctfp.gpstmplt.o*

3. Load in your application software.

```
-> ld < btfpGPStest.o
```

If these steps are not followed, then the response system will be unavailable. Examine the bctfp.gpsapp.h to determine what structure elements are available when using a response from the system.

CHAPTER FOUR

SOFTWARE INSTALLATION

4.0 SOFTWARE INSTALLATION

If this is a first time installation of the driver software, refer to the VxWorks User's Guide for instructions on installing driver software. The BTFP driver software will consist of 3 files. They are listed in the following table.

File Name	Description
btfpdrv.o	Installed in the ".../obj" directory. This is the driver itself.
bctfp.app.h	Installed in the ".../src" directory. This is the header file that should be used in any VxWorks task that will access the Bancomm BTFP board.
btfpTFPtest.c	Installed in the ".../src" directory. This is a test routine which will exercise the BTFP and driver to verify its operation.

CHAPTER FIVE
HARDWARE CONFIGURATION

5.0 HARDWARE CONFIGURATION

The Datum board occupies 64 bytes of VME A16 Address Space (D16). The device address is selected by setting DIP switches S-1 and S-2. Set these DIP switches to A15 through A6 of the desired device address.

For more information related to jumpers, and this switch, see the Datum bc635VME/bc350VXI Time and Frequency Processor Operation and Technical Manual.

CHAPTER SIX

EXAMPLE PROGRAM

6.0 EXAMPLE PROGRAM

6.1 OVERVIEW

The sample program listed in this chapter that is executed under VxWorks fully exercises the driver's capabilities. Please read the program carefully along with its respective comments. The program should do the following things:

Step 1: Calls btfpDevCreate to make the device.

Step 2: Opens the device with the open command.

Step 3: Tests most of the ioctl calls. Namely:

- DEBUG
- PACKET
- SIGNAL
- INTCONN
- READREG
- WINFIFO
- most P_ ioctls

Step 4: Reads the time from the board's time registers.

Step 5: Installs the user interrupt handler "myhandler."

Step 6: Sets signal mode to ON and enables the DPA and 1PPS interrupts.

Step 7: Tests the boards response to packet 'O' (multiple times).

Step 8: Reads all registers into regData and prints them.

Step 9: Closes the device.

6.2 RUNNING THE SAMPLE PROGRAM

Once the hardware is installed, boot the target CPU as documented in the standard VxWorks User Manuals. Make sure the test program are compiled and configured for the VxWorks CPU to be used.

At the VxWorks console terminal issue the following commands:

1. Change directories to the location on the host where the driver is located.

```
-> cd "/usr/vw/drivers/obj"
```

2. Load in the driver software.

```
-> ld < btfpdrv.o
```

3. Load in the TFP test. This will test only TFP functions for those users who don't have GPS functionality. For the users that do, please check the btfpGPStest.o program. (There are other tests provided, however due to some driver changes, they should be used with caution. Feel free to examine them, however.) For examples of the response system application for GPS see the btfpGPStest.c program.

```
-> ld < btfpTFPtest.o
```

4. To run *btfpTFPtest*, type

```
-> go (sysClkRateGet()*2,1)
```

Note the parameters. sysClkRateGet() gets the current VxWorks system clock rate. Multiplying this by some value gives a value in seconds. Here two seconds is used. This is the length of time between some of the software tests. This time was inserted for the user's benefit so the LCD display could be examined while running the test. Otherwise, much of the test would fly by without the user knowing if a failure occurred. After the initial LCD testing is done, other tests occur involving some printout. An example printout is given here to compare. The output may differ slightly, but the general format is the same.

```
##############################################################################
#           Sample btfpTFPtest.o output                         #
#           Your output may differ slightly                     #
#           In otherwords, your milage may vary                      #
##############################################################################
-> ld < /home/helios/bpw/projs/bancomm/obj/btfpTFPtest.o
value = 3957576 = 0x3c6348
-> go(sysClkRateGet()*2,1)
0x3d2c80 (tShell): ----- Bancomm Time Frequency Processor Driver Test -----
0x3d2c80 (tShell): Starting test...
0x3d2c80 (tShell): Step 1: Calling btfpDevCreate...
```

```
0x3d2c80 (tShell): Starting btfpDevCreate...
0x3d2c80 (tShell): Value of drvParams->regs is 0xfeffff00
0x3d2c80 (tShell): Starting btfpDrvInstall...
0x3d2c80 (tShell): Leaving btfpDrvInstall....
0x3d2c80 (tShell): Leaving btfpDevCreate....
0x3d2c80 (tShell): Starting btfpOpen...
0x3d2c80 (tShell): Leaving btfpOpen....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Step 2: Testing new ioctls....
0x3d2c80 (tShell): Setting time...
0x3d2c80 (tShell): Setting time...
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x44.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x31.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x32.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x33.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x34.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe3
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x46.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x32.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x30.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x30.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x43.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x38.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x32.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x37.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x31.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x30.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
```

```
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe3
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x47.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x2d.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x31.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x32.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x33.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x34.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x35.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x36.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x37.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe3
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x51.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x32.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x31.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x31.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe3
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x4d.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x2d.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x30.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x35.
```

```
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe3
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Starting sendPacket...
0x3d2c80 (tShell): Value of fifo register is 0xfeffff27
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x1.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x4f.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x30.
0x3d2c80 (tShell): Value of send:packet[i] in state 0 is 0x17.
0x3d2c80 (tShell): Value of send:packet[i] in state 1 is 0x0.
0x3d2c80 (tShell): Value of regs->btfpAck is 0xffe2
0x3d2c80 (tShell): Value of regs->btfpAck is 0xfff7
0x3d2c80 (tShell): Leaving sendPacket....
0x3d2c80 (tShell): Leaving btfpIoctl....
0x3d2c80 (tShell): Starting btfpIoctl...
0x3d2c80 (tShell): Packet ‘O’ response:
0x01 0x6f 0x30 0x39 0x35 0x31 0x32 0x30 0x38 0x31 0x36 0x34 0x38 0x30 0x36 0x17
interrupt: *** called myhandler ***
interrupt: *** called myhandler ***
0x3d2c80 (tShell): Packet ‘O’ response:
0x01 0x6f 0x30 0x39 0x35 0x31 0x32 0x30 0x38 0x31 0x36 0x34 0x38 0x30 0x36 0x17
Time registers: 123   40   56   05   413   121   0
0x3d2c80 (tShell): Packet ‘O’ response:
0x01 0x6f 0x30 0x39 0x35 0x31 0x32 0x30 0x38 0x31 0x36 0x34 0x38 0x31 0x36 0x17
Register 0x00 = 0xffff
Register 0x01 = 0xffff
Register 0x02 = 0xffff
Register 0x03 = 0xffff
Register 0x04 = 0xffff
Register 0x05 = 0xffff
Register 0x06 = 0x0061
Register 0x07 = 0x2340
Register 0x08 = 0x5610
Register 0x09 = 0x2253
Register 0x0a = 0x7970
Register 0x0b = 0x00ff
Register 0x0c = 0xffff
Register 0x0d = 0xffff
Register 0x0e = 0xffff
Register 0x0f = 0xfff0
Register 0x10 = 0xffff
Register 0x11 = 0xffe3
Register 0x12 = 0xff00
Register 0x13 = 0xff00
Register 0x14 = 0xffe0
Register 0x15 = 0xfffa
Register 0x16 = 0xff19
Register 0x17 = 0xfffd
Vector=0x19, Level=0x05
value = 0 = 0x0
->
```

In case a board sanity check is required, type

```
-> sP("A7",2,0xXXXX)
```

This should put the board into diagnostic mode. The Xs represent the sixteen bit hexidecimal address of your TFP board. When the board is in diagnostic mode, the front panel LED's will show a static display of 123456.

To leave again, type:

```
-> sP("A0",2,0xXXXX)
```

You can use any number 0-7 (this actually sends the 'A' packet from the Bancomm manual. You can also send other packets. Just be sure the number after the packet represents the correct packet length) in this command. Again, the Xs represent the sixteen bit hexidecimal address of your TFP board.

NOTE: When you first install the board, this command will help determine if your installation is correct. Do not attempt to use the driver without making this determination first. Otherwise you could spend a great deal of time debugging bug-free code! However, because this function works doesn't mean the board installation is completely correct. Some jumpers of the board have been known to cause unpredictable results if the installation manual isn't followed carefully.

CHAPTER SEVEN
SAMPLE PROGRAM

7.0 SAMPLE PROGRAM

A source code listing of btftTFPtest.c, an example VxWorks task which exercises the BTFP device driver, follows.

```
/*
********************************************************************************
**Copyright © 1995 Datum  Inc. (Datum Inc.)
**All rights reserved.
**
**This file and its contents are a product of Datum Inc..
**All software distributed by Datum Inc. is done so under a software license.
**This file may not be copied or modified without execution of the
**appropriate software license agreement with Datum Inc. or explicit written
**permission of Datum Inc..
**
**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.
**
**Datum Inc. 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 Datum Inc. be liable for any lost revenue or profits
**or other special, indirect and consequential damages, even if Datum Inc.
**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.
********************************************************************************
*/

/*
-------------------------------------------------------

File Name:  btfp Test.c

Description:
Runs the Bancomm TFP driver test.
```

```
Modification History:
Who Date What
- - -  - - - -  - - - -

bpw 08/28/95 created
--------------------------------------------------------
*/

/*
* Include files
*/
#include “vxWorks.h”
#include “stdio.h”
#include “stdlib.h”
#include “sting.h”
#include “logLib.h”
#include “sysLib.h”
#include “taskLib.h”
#include “stdDef.h”
#include “bctlp.app.h”

/*
*Defines
*/
#define NOCODE(x)
#define PRNT(x) logMsg((CHAR*)x,0,0,0,0,0,0)
#define HVAL(x,y) logMsg(“Value of %s is 0x%x.\n”(INT)x,(UINT)y,0,0,0,0)
#define DVAL(x,y) logMsg(“Value of %s is %d, \n”,(INT)x,(UINT)y,0,0,0,0)
#define CVAL(x,y) logMsg(“Value of %s is %c, \n”,(INT)x,(UINT)y,0,0,0,0)
#define SVAL(x,y) logMsg(“% is %s,\n”,(INT)x,(INT)y,0,0,0,0)
#define ERRP(x,y) logMsg(“*** Error: %s at %s ***\n”,(INT)x,(INT)y,0,0,0,0)
#define YOUR_VME_ADDRESS(CHAR*)0xff00
#define YOUR_INTERRUPT_LEVEL 0x05
#define YOUR_INTERRUPT_VECTOR 0x19
/*
* typedefs
*/

/*
*Imports
*/
```

```
/*
* Exports
*/

/*
* Locals
*/


int flag = 0
max = 0
int fd = 0
char buf[1024];
/*
*Forward declarations
*/
STATUS go();
VOID myhandler(int);

VOID
myhandler(int val)
{

char tmp[30];
int status;
args ioctlArgs;

PRNT("***called myhandler***\n");
if (val & INT_DPA)
{
        max+=read(fd,(char*) &buff[max], 512 - max);
        flat = 1;
}
if(val & INT_1PPS)
{
        ioctl(fd, PACKET, OFF);
        sprintf(tmp,"%c%s%c", SOH, "B123123456", ETB);
        status = write(fd, tmp, 12);
        if(status<12)
        ERRP("noted", "interrupt write");
        ioctl(fd, PACKET, ON);
        ioctlArgs.reg_id = R_MASK;
        icotlArgs.reg_val = INT_DPA;
        ioctl(fd, WRITEREG, (int) &ioctlArgs);
```

```
}
}

STATUS
go(int input, int yesno)
{
STATUS status;
CHAR mode;
INT dataVals[10],i,j;
INT16 regData[25];


CHAR packetData[550];
args ioctlArgs;
PRNT("- - - - - Bancomm Time Frequency Processor Driver Test - - - - -\n");
PRNT("Starting test…\n");
PRNT("Step 1: Calling btfpDevCreate…\n");
if(yesno)
{
        /* NOTE: The interrupt vector and level set here
         * aren't normally recommended.  However a Force
         * 2CE board seems to work with only a few vectors
         * and interrupt levels.  If you have a 680x0 based
         * board, check what interrupt levels you should use.
         * It is a good bet however, that numbers such as
         * 7 and 249 for level and vector will work quite
         * well on the Motorola boards and are recommended.
         * If not, then find check the user documentation that
         * came with the board.
         */
        status = btfpDevCreate("btfp0",
                YOUR_VME_ADDRESS,
                YOUR_INTERRUPT_LEVEL,
                YOUR_INTERRUPT_VECTOR;
        if (status == ERROR)
        {
        ERRP("Couldn't create device","btfpDevCreate");
        return ERROR;
        }
}
fd = open("btfp0",0,0);
if(fd<0)
```

```
{
        ERRP("Can't open time Frequency Processor","open");
        return ERROR;
}
for(i = 0, i<513; i++)
packetData[i] = 0;
/*DEBUG turned off to limit output.  Change
*to nonzero value if output required.
*Default DEBUG is FULL ON.
*/
ioctl(fd,DEBUG,0x00);

PRNT("Step 2: Testing new ioctls….\n");

/*test MODE SELECT*/

/*NOTE: Each mode change requires a POINTER to
*the mode.  The defined values for modes
*DO NOT constitute a pointer.  If you want
*to be cheesy about it, you can use DOUBLE
*quoted characters for the mode.  This creates
*the required pointer to a string.  However,
*this method isn't recommended or employed here.
*Use the enums, Luke!
*/
mode = A_DIAGNOSTIC;
status = ioctl(fd,P_MODE_SEL,(INT)((CHAR*)&mode));
if(status) == ERROR)
{
        ERRP("found","P_MODE_SEL_:A_DIAGNOSTIC");
        close(fd);
        return ERROR;
}
/*taskDelay u sed to allow user to verify output
*on board display.  NOT REQUIRED.
*/
taskDelay(input);

/*test mode EXTERNAL 1PPS*/
mode = A_EXT_1PPS;
status = ioctl(fd,P_MODE_SEL,(INT)((CHAR*)&mode));
if(status == ERROR)
```

```
{
        ERRP(“found”,”P_MODE_SEL:A_EXT_1PPS);
        close(fd);
        return ERROR;
}
/*taskDelay used to allow user to verify output
*on board display.  NOT REQUIRED.
*/
taskDelay(input);

/*test mode TIMECODE DECODE */
mode = A_TIMECODE_DEC;
status = ioctl(fd,P_MODE_SEL,(INT)((CHAR*)&mode));
if(status == ERROR)
{
        ERRP(“found”,”P_MODE_SEL:A_TIMECODE_DEC”);
        close(fd);
        return ERROR;
}

/*taskDelay used to allow user to verify output
*on board display, NOT REQUIRED.
*/
taskDelay(input);

/*test FREE RUNNING*/
mode = A_FREE_RUNNING;
status = ioctl(fd,P_MODE_SEL,(INT)((CHAR*)&mode));
if(status == ERROR)
{
        ERRP(“found”,”P_MODE_SEL:A_FREE_RUNNING”);
        close(fd);
        return ERROR:
}
/*taskDelay used to allow user to verify output
*on board display.  NOT REQUIRED.
*/
taskDelay(input);

/*NOTE:  Don’t test GPS in case user doesn’t have
GPS antenna, or built in GPS*/
/*set mode to REAL TIME CLOCK for next test*/
mode = A_REAL_TIME_CLK;
status = ioctl(fd,P_MODE_SEL<(INT)((CHAR*)&mode));
```

```
if(status == ERROR)
{
        ERRP(“found”,”P_MODE_SEL:A_REAL_TIME_CLK”);
        close(fd);
        return ERROR;
}

/*test REAL TIME CLOCK*/
/*REAL TIME: 12/08/1995, 4:48:00 PM*/
PRNT(“Setting time…\n”);
/*NOTE: below not all numbers contain enough digits.
*The driver will fill in the blanks with zeros.
*However, there MUST be a pointer to six integers
*passed to the driver for this to work correctly.
*/
dataVals[0] = 95;/*year*/
dataVals[1] = 12;/*month*/
dataVals[2] = 8;/*day*/
dataVals[3] = 16;/*hour*/
dataVals[4] = 48:/*minute*/
dataVals[5] = 0;/*second*/
status = ioctl(fd,P_SET_RCLK,(INT)dataVals);
if(status == ERROR)
{
        ERRP(“found”,”Setting Real Time Clock☺’
        close(fd);
        return ERROR;
}
taskDelay(input);

/*Set mode to free running for packet B */
mode = A_FREE_RUNNING;
status = ioctl(fd,P_MODE_SEL,(INT)((CHAR*)&mode));
if(status == ERROR)
{
        ERRP(“found”,”P_MODE_SEL:A_FREE_RUNNING”);
        close(fd);
        returnERROR;
}
/*taskDelay used to allow user to verify output
*on board display.  NOT REQUIRED.
*/
```

```
taskDelay(input);
/*test SET MAJOR TIME*/
/*REAL TIME; 12/08, 4:48:00 PM*/
PRNT("Setting time…\n");
/*NOTE: below not all numbers contain enough digits.
*The driver will fill in the blanks with zeros.
*However, there MUST be a pointer to six integers
*passed to the driver for this to work correctly.
*/
dataVals[0] = 343; /*day 12/08*/
dataVals[1] = 16;/*hour 04 PM*/
dataVals[2] = 48;/*minute 48*/
dataVals[3] = 0:/*second 0*/
status = ioctl(fd,P_MJTM_SET,(INT)dataVals);
if(status == ERROR)
{
        ERRP("found","Setting Real Time Clock");
        close(fd);
        return ERROR;
}

/*Turn on FULL debut again.  Only way to check
*some of this output for correctness (packet check).
*/

ioctl(fd,DEBUG,0x07);

/*Test D/A Converter load */
/*NOTE: No real way to check this working*/
dataVals[0] = 01234;
status = ioctl(fd,P_DAC_LD,(INT)dataVals);
if(status == ERROR)
{
        ERRP("found","Setting DAC value");
        close(fd);
        return ERROR;
}

dataVals[0] = 2;/*synchronous…why not?*/
/*NOTE: Each 4 nibbles of the long word below
*is used to create the 2 short words needed for
*n1 and n2.  It really doesn't matter what order
*you put the values in unless one is divisible by
*two.  Then the order is n1 in the upper 4 nibbles
```

```
*and n2 in the lower 4 nibbles.  n2 would be the
*one checked for divisibility by 2 in the hardware.
*Also NOTE:  The driver doesn’t care about upper and
*lower case hex values.  Use what ever you are
*comfortable with.
*/
dataVals[1] = 0xc82710;/*200&10000 yields 1/5s*/
/*Again: How to check ?*/
status = ioctl(fd,P_HBT_CTL,(INT)dataVals);
if(status == ERROR)
{
        ERRP(“found”,”Setting Heartbeat”);
        close(fd);
        return ERROR;
}

/*test OFFSET CONTROL*/
dataVals[0] = -123;/*milliseconds, Signed, Important!*/
dataVals[1] = 456;/*microseconds, No sign, Important!*/
dataVals[2] = 7;/*nanoseconds.  One digit, no sign.*/
status = ioctl(fd,P_OFF_CTL,(INT)dataVals;
if(status == ERROR)
{
        ERRP(“found”,”Setting OFFSET”);
        close(fd):
        return ERROR;
}

/*test DISIPLINING GAIN*/
/*NOTE: format of the following for gain control is this:
*First two digits are gain value.  DON’T SWAP DIGITS.
*Just put the gain value here.  The last digit is the
*1 or 0 for + or - gain.  This value must be 1 or 0.
*The driver doesn’t check this.  If the gain is 0,
*then at least the + or - must be filled in (1 or 0).
*The example gain shows a postitive 0x12 for gain.
*That is a positive 18 decimal.
*/
dataVals[0] = 0x121;/*Gain in first two digits, +=1,-=0 in last digit*/
status = ioctl(fd,P_SET_GAIN,(INT)dataVals);
if(status == ERROR)
```

```
{
        ERRP(“found”,”Setting GAIN”);
        close(fd);
        return ERROR;
}

/*test TIME OFFSET SELECT*/
dataVals[0] = -5/*Eastern standard time from UTC*/
status = ioctl(fd,P_TIME_OFF,(INT)dataVals);
if(status == ERROR)
{
        ERRP(“fond”,”Setting TIME OFFSET”);
        close(fd);
        return ERROR;
}

/*Turn on packets and off signals */
status = ioctl(fd,PACKET,ON);
status = ioctl(fd,SIGNAL,OFF);
if(status == ERROR)
{
        ERRP(“found”,”PACKET or SIGNAL”);
        close(fd);
        return ERROR;
}

/*clear the fifo and ACK previously written packets */
ioctlArgs.reg_id = R_ACK;
ioctlArgs.reg_val = ACK_CLEAR | ACK_DPA;
status = ioctl(fd, WRITEREG, (int) &ioctlArgs);
if(status == ERROR)
{


        ERRP(“found”,”WRITEREG”);
        close(fd):
        return ERROR;
}

/*test DATA REQUEST*/
mode = 0_FMT0;/*Packet O format 0 */
status = ioctl(fd,P_DATA_REQ,(INT)((CHAR*)&mode));
if(status == ERROR)
```

```
{
        ERRP("found"."Requesting Data Packet"):
        close(fd);
        return ERROR;
}

/*read the entire fifo to find packet 'O' response */
/*Turn off debugging*/
ioctl(fd,DEBUG,0x00);
status = read(fd,(CHAR*)&packetData,512);
for(i = 0;i<512;i++)
{
        if(((UINT8)packetData[1] == SOH)&&
        ((UINT8)packetData[i+1] == 'o')&&
        ((UINT8)packetData[i+2] == O_FMT0))
        break;
}
taskDelay(300);/*printf doesn't seem to print without this
        *delay.  The data is there however.
        */
if(i>=512)
{
        PRNT("Packet 'O' response not found!\n");
        for(i = 0:i<512;i++)
        printf("%#02x",(UINT8)packetData[i];
        printf("\n\n");
}
else
{
        PRNT("Packet 'O' response:\n");
        for(j = 0;j<16;j++)
        printf("0x%02x",(UINT8)packetData[i+j]);
        printf("\n");
}

/*Time to test the rest of the driver*/

status ioctl(fd,INTCONN,(INT)myhandler);
if(status == ERROR)
{
        ERRP("found","INTCONN");
        close(fd);
        return ERROR;
}
```

```
taskDelay(300);/*printf doesn’t seem to print without this
        *delay.  The data is there however.
        */
if(i>=512)
{
        PRNT(“Packet ‘O’ response not found!\n”);
        for(i = 0;i<512,i++)
        printf(“%#02x”,(UINT8)packetData[i]);
        printf(“n\n\);
}
else
{
        PRNT(“Packet ‘O’ response:\n”)
        for(j = 0;j<16;j++)
        printf(“0x%2x”,(UINT8)packetData[i+j]);
        printf(“\n”);
}

/*Turn off packets and signals off*/
status = ioctl(fd, PACKET,OFF);
status = ioctl(fd,SIGNAL,OFF);
if(status == ERROR)
{
        ERRP(“found”,”PACKET or SIGNAL”);
        close(fd);
        return ERROR;
}

/*clear interrupts*/
ioctlArgs.reg_id = R_MASK;
ioctlArgs.reg_val = INT_DISABLE;
status = ioctl(fd, WRITEREG, (int) &ioctlArgs);
if(status == ERROR)
{
        ERRP(“found”,”WRITEREG”);
        close(fd);
        return ERROR;
}

status = read(fd, (CHAR*)&buf,40);
if(status<0)
```

```
{
        ERRP(“found”,”read PACKET OFF”);
        close(fd);
        return ERROR;
}
printf(“Time registers: %s\n”,buf);
/*Turn on packets*/
status = ioctl(fd,PACKET,ON);
if(status == ERROR)
{
        ERRP(“found”,”PACKET”);
        close(fd);
        return ERROR;
}

/*write a packet ‘O’ to the driver*/
buf[0] = SOH;
buf[1] = P_DATA_REQ;
buf[2] = O_FMT0;
buf[3] = ETB;
buf[4] = 0;
status = write(fd,(CHAR*)&buf,4);
if(status<4)
{
        ERRP(“found”,”write”);
        close(fd);
        return ERROR;
}
status = read(fd,(CHAR *)&packetData,512);
for(i = 0;i<512;i++)
{
        if(((UINT8)packetData[i] == SOH)&&
        ((UINT8)packetData[i+1] == ‘o’)&&
        ((UINT8)packetData[i+2] == O_FMT0)
        break;
}
taskDelay(300);/*printf doesnt’ seem to print without this
        *delay.  The data is there however.
        */
if(i>=512)
```

```
{
        PRNT(“Packet ‘O’ response not found!\n”);
        for(i = 0;i<512;i++)


        printf(“%#02x”,(UINT8)packetData[i]);
        printf(“\n\n”);
}
else
{
        PRNT(“Packet ‘O’ response:\n”);
        for(j = 0;j<16;j++)
        printf(“0x%02x”,(UINT8)packetData[i+j]);
        printf(“\n”);
}

for(i = 0;i<25;i++)
regData[i] = (UINT16)0;
regData[0] = R_ALL;
status = ioctl(fd,READREG,(INT)regData);
if(status == ERROR)
{
        ERRP(“found”,”READREG”);
        close(fd);
        return ERROR;
}
/*NOTE:printout will show fifo area.  Nothing
*is actually readd from the fifo so as not to
*accidentally lose a byte.
*/
for(i = 0;i<=(R_LEVEL/2;i++)
printf(“Register 0x%02x = 0x%04x\n”,(UINT32)i,(UINT16)regData[i]);

/*format the interrupt level and vector for printing*/
printf(“Vector=0x%02x,Level=0x%02x\n”,
        (UINT16)regData[R_VECTOR/2]&0x00ff,
        (UINT16)regData[R_LEVEL/2]&0x0007);

/*This is another methond of setting the board to
*A_DIAGNOSTIC mode.  However, it is lengty and
*NOT recommended.  If you use the driver, either
*use the P_MODE_SEL ioctl or a write in packet
*on mode.
*/
status = ioctl(fd,WINFIFO,SOH);
status = ioctl(fd,WINFIFO,P_MODE_SEL);
status = ioctl(fd,WINFIFO,A_DIAGNOSTIC);
status = ioctl(fd,WINFIFO,ETB);
if(status == ERROR)
{

        ERRP(“found”,”WINFIFO”);
        close(fd);
        return ERROR;
}
ioctlArgs.reg_id = R_ACK;
ioctlArgs.reg_val = ACK_INACT | ACK_INFIFO;
status = ioctl(fd,WRITEREG,(INT)&ioctlArgs);
if(status == ERROR)
{
        ERRP(“found”,”WRITEREG”);
        close(fd);
        return ERROR;
}
/*wait for board to complete packet read */
do {
        regData[0] = R_ACK;
        status = ioctl(fd,READREG,(INT)regData);
} while(!(regData[0]&ACK_INFIFO));

close(fd);
return OK;
}
```

CHAPTER EIGHT

APPENDIX A

8.0 APPENDIX A: BTFP Include File

A source code listing of bctfp.app.h, the include file for the VxWorks task which accesses the BTFP device driver, follows.

```
/*
*********************************************************************************
*
** Copyright (C) 1995 Datum  Inc. (Datum Inc.)
** All rights reserved.
**
** This file and its contents are a product of Datum Inc..
** All software distributed by Datum Inc. is done so under a software license.
** This file may not be copied or modified without execution of the
** appropriate software license agreement with Datum Inc. or explicit written
** permission of Datum Inc..
**
** 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.
**
** Datum Inc. 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 Datum Inc. be liable for any lost revenue or profits
** or other special, indirect and consequential damages, even if Datum Inc.
** 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.
*********************************************************************************
*/

/*
----------------------------------------------------------------------------

File Name: bctfp.app.h

Description:
 User includable header for driver interface.

Modification History:
 Who Date What
 --- ---- ----
 bpw 08/28/95 created

----------------------------------------------------------------------------
*/

/*
 * Include files
 */
#include "fcntl.h"
#include "unistd.h"
#include "ioLib.h"

#ifndef __BCTFP_APP_H__
#define __BCTFP_APP_H__

/*
 * Defines
 */

/*
 * typedefs
 */
/*
 The following enum specifies all the allowable IOCTL calls. For
 DEBUG, PACKET, and SIGNAL, they can be turned on or off by using
 an ioctl call like this:

 ioctl(fd,DEBUG,ON); <-- turns on driver DEBUG mode
 ioctl(fd,PACKET,OFF); <-- turns off driver PACKET mode

 For the READREG command the ioctl will look like this:

 unsigned short val = R_INTSTAT;
 ioctl(fd,READREG,&val); <-- should read the INTSTAT board register

 All registers can be read as follows:
 BTFPRegs vals;
 vals.btfpId = R_ALL;
 ioctl(fd,READREG,&vals); <-- should read all board registers

 The driver will automatically determine what type of data should
 have been passed to it based on the value of the argument. Therefore
 it would be an error to do the following:

 int vals=R_ALL;
 ioctl(fd,READREG,&vals); <-- will fill in unallocated memory
 */
typedef
enum _ioctls
{
 INTCONN = 0x00, /* Connect the user specified function to
 * driver interrupt. This is effectively the
 * same as connecting the interrupt to
 * SIGUSR1. The driver interrupt will call
 * the specified function with the INTSTAT
 * register as a parameter. This will allow
 * the user to multiplex their interrupt
 * code. NOTE 1: Any function connected in
 * this way must follow vxWorks interrupt
 * code specs. NOTE 2: All interrupt
 * processing done by the driver will be
 * completed at the time of this call.
 * Therefore, all structures normally filled
 * in by this interrupt can be expected to be
 * correct during the user interrupt call */
 DEBUG = 0x01, /* Set driver DEBUG mode */
 PACKET = 0x02, /* Set driver PACKET mode */
 SIGNAL = 0x03, /* Set driver SIGNAL mode */
 ROUTFIFO = 0x04, /* Read one byte from in fifo */
 WINFIFO = 0x05, /* write one byte to out fifo */
 READREG = 0x06, /* Read specified register or all if 0xFF */
 WRITEREG = 0x07, /* Write specified register or all if 0xFF */
/* Not implemented because above controls handle them all
 READTIME = 0x08, * Read time from time regs *
 READEVENT = 0x09, * Read event regs *
 WRITESTROBE = 0x0A * Write strobe regs *
*/
} Ioctls;

/*
 - The following enums are used instead of defines for two reasons:
 - 1) Debuggers know about enums and they do not know about #defines
 - so making something that would be a define otherwise makes it
 - easier to debug (Name of value instead of having to find number).
 - 2) enums are cast as ints. This at least gives them a type which
 - might assist in debugging as well.
 */

typedef
enum _RegOffs
{
 R_ID = 0x00, /* ID Register */
 R_DEVICE = 0x02, /* VXI Device Register */
 R_STATUS = 0x04, /* VXI Status Register */
 R_CONTROL = 0x04, /* Control Register */
 R_TIMEREQ = 0x0A, /* Time Request Register */
 R_TIME0 = 0x0C, /* Time 0 (days hundreds) register */
 R_TIME1 = 0x0E, /* Time 1 (days tens, units - hours tens,
 * units) */
 R_TIME2 = 0x10, /* Time 2 (minutes tens, units - seconds
 * tens, units) */
 R_TIME3 = 0x12, /* Time 3 (seconds 10^-1 to 10^-4) */
 R_TIME4 = 0x14, /* Time 4 (seconds 10^-5 to 10^-7) */
 R_EVENT0 = 0x16, /* Event 0 */
 R_EVENT1 = 0x18, /* Event 1 */
 R_EVENT2 = 0x1A, /* Event 2 */
 R_EVENT3 = 0x1C, /* Event 3 */
 R_EVENT4 = 0x1E, /* Event 4 */
 R_STROBE1 = 0x18, /* Strobe 1 (Also Event 0) */
 R_STROBE2 = 0x1A, /* Strobe 2 (Also Event 1) */
 R_STROBE3 = 0x1C, /* Strobe 3 (Also Event 2) */
 R_UNLOCK = 0x20, /* Unlock Register */
 R_ACK = 0x22, /* Packet ACK Register */
 R_CMD = 0x24, /* Command Register (TFP functions) */
 R_FIFO = 0x26, /* Fifo Register (input/output) */
 R_MASK = 0x28, /* Interrupt Mask Register */
 R_INTSTAT = 0x2A, /* Interrupt Status Register */
 R_VECTOR = 0x2C, /* Interrupt Vector Register */
 R_LEVEL = 0x2e, /* Interrupt Level Register */
 R_ALL = 0xFF /* ALL REGS (for ioctl) */
} RegOffs;

typedef
enum _Packets
{
 P_MODE_SEL = ‘A’, /* TFP Mode select */
 P_MJTM_SET = ‘B’, /* Set Major Time */
 P_CMD_INP = ‘C’, /* Command Input */
 P_DAC_LD = ‘D’, /* Load D/A Converter */
 P_HBT_CTL = ‘F’, /* Heartbeat control */
 P_OFF_CTL = ‘G’, /* Offset Control */
 P_TC_FMAT = ‘H’, /* Set Timecode format (mode 0) */
 P_SEL_CLK = ‘I’, /* Clock Source Select */
 P_GPS_DATA = ‘J’, /* GPS Receiver Data */
 P_SEL_GCODE = ‘K’, /* Select generator code */
 P_SET_RCLK = ‘L’, /* Set Real time clock */
 P_TIME_OFF = ‘M’, /* Local Time offset select */
 P_DATA_REQ = ‘O’, /* Request data from the TFP */
 P_PATH_SEL = ‘P’, /* Path selection */
 P_SET_GAIN = ‘Q’, /* Set Disciplining gain */
 P_SET_YEAR = ‘S’ /* Set year */
} Packets;

typedef
struct _BTFPioctlData
{
 enum
 {
 TIME = 0, EVENT = 1, STROBE = 2} dataType;
 unsigned short btfpData[5];
} BTFPioctlData;

typedef
enum _PcktHeader
{
 SOH = 0x01,
 DLE = 0x10,
 ETB = 0x17,
 ETX = 0x03
} PcktHeader;

typedef
enum _Bool
{
 ON = 0x01,
 OFF = 0x00
} Bool;
```

```
typedef
enum _masks
{
 TIME_REF = 0x0000,
 EVENT_REF = 0x0000,

 TIME_FLY = 0x0010,
 EVENT_FLY = 0x0010,

 TIME_OSCIN = 0x0000,
 EVENT_OSCIN = 0x0000,
 TIME_OSCOUT = 0x0040,
 EVENT_OSCOUT = 0x0040,
 /* Acknowledge masks */
 ACK_INFIFO = 0x0001,
 ACK_1PPS = 0x0002,
 ACK_DPA = 0x0004,
 ACK_MORE = 0x0010,
 ACK_CLEAR = 0x0010,
 ACK_INACT = 0x0080,
 /* Mode Enables */
 CMD_DISABLE = 0x0000,
 LOCK_EN = 0x0001,
 HBEAT_EN = 0x0002,
 EVSENSE_FALL = 0x0004,
 EVSENSE_RISE = 0x0000,
 EVENT_EN = 0x0008,
 STROBE_EN = 0x0010,
 STRMDE_MIN = 0x0020,
 STRMDE_MAJ = 0x0000,
 FREQSEL_10 = 0x0000,
 FREQSEL_05 = 0x0040,
 FREQSEL_01 = 0x0080,
 /* Interrupt Masks (INTMASK and INTSTAT) */
 INT_EXT = 0x0001,
 INT_HBEAT = 0x0002,
 INT_STROBE = 0x0004,
 INT_1PPS = 0x0008,
 INT_DPA = 0x0010,
 INT_DISABLE = 0x0000
} masks;

typedef
enum _modes
{
 /* Packet A modes */
 A_TIMECODE_DEC = ‘0’,
 A_FREE_RUNNING = ‘1’,
 A_EXT_1PPS = ‘2’,
 A_REAL_TIME_CLK = ‘3’,
 A_DIGITAL_SYNC = ‘4’,
 A_GPS_ONBOARD = ‘5’,
 A_GPS_ANTENNA = ‘6’,
 A_DIAGNOSTIC = ‘7’,
 /* Packet C modes */
 C_SOFT_RESET = ‘2’,
 C_JAMSYNC = ‘3’,
 C_BUF_RTC = ‘5’,
 /* Packet H modes */
 H_IRIGA = ‘A’,
 H_IRIGB = ‘B’,
 H_2137 = ‘C’,
 H_NASA36 = ‘N’,
 H_XR3 = ‘X’,
 /* Packet I modes */
 I_EXT = ‘E’,
 I_INT = ‘I’,
 /* Packet K modes */
 K_IRIGB = ‘B’,
 K_IRIGH = ‘H’,
 /* Packet O modes */
 O_FMT0 = ‘0’,
 O_FMT1 = ‘1’,
 O_FMT2 = ‘2’,
 O_FMT3 = ‘3’,
 O_FMT4 = ‘4’
} modes;

typedef
struct _args
{
 int reg_id;
 unsigned short reg_val;
} args;

typedef
void (*FPTR) (int);

/*
 * Imports
 */

/*
 * Exports
 */

/*
 * Locals
 */

/*
 * Forward declarations
 */
STATUS btfpDevCreate(char *, void *, int);
#endif /* __BCTFP_APP_H__ */
```