bc635VME/bc350VXI
TIME AND FREQUENCY PROCESSOR

HP-RT v1.1x DRIVER MANUAL

User’s Guide
(August 2000)

bc635VME/bc350VXI
TIME AND FREQUENCY PROCESSOR
HP-RT v1.1x DRIVER
User’s Guide
August 2000

TABLE OF CONTENTS

SECTION **PAGE**

CHAPTER 1
INSTALLATION INSTRUCTIONS

1.0 INTRODUCTION........1-1
1.1 RELATED DOCUMENTS........1-1
1.2 HARDWARE INSTALLATION........1-1

CHAPTER 2
SOFTWARE INSTALLATION

2.0 PROCEDURE........2-1

CHAPTER 3
MANPAGE BTFP(7)

3.0 DESCRIPTION........3-1
3.1 READ/WRITE DEFAULT MODE ("PACKETOFF")........3-1
3.2 READ/WRITE PACKET MODE ("PACKETON")........3-2
3.3 IOCTL CALLS........3-3
3.4 NOTES........3-8

CHAPTER 4
BTFPTEST OUTPUT

4.0 LISTING........4-1

CHAPTER 5
BTFPTEST SOURCE CODE

5.0 LISTING........5-1

TABLE OF CONTENTS (continued)

SECTION **PAGE**

CHAPTER 6
DRIVER/USER HEADER FILE

6.0 LISTING..6-1

CHAPTER 7
DRIVER HEADER FILE

7.0 LISTING..7-1

CHAPTER 8
DRIVER SOURCE CODE

8.0 LISTING..8-1

CHAPTER 1
INSTALLATION INSTRUCTIONS

bc635/635VME-bc350/357VXI
Time and Frequency Processor Module Driver
for
HP-RT v1.1x Systems

1.0 Introduction

The Datum BTFP is a time and frequency processor module hardware device with an interface for VME/VXI bus computer systems. It is used for decoding industry standard, serialized time code formats. It may be used for time-stamping events, initiating external events at known times, etc. When accompanied by a GPS Satellite Receiver module, it is referred to as a bc637VME or bc357VXI module. In this document, references to BTFP include the standalone and GPS-integrated versions of the product, and the same driver supports both versions.

This document describes how to install the Datum BTFP time and frequency processor module on a VME-bus based computer system, and provides some general discussion that may be helpful for installing the HP-RT v1.1x driver on a HP system.

The driver supplied was implemented and tested on a HP747i industrial workstation running HP-RT v1.11. It is intended that this driver be installed by someone familiar with system administration on HP-UX based systems. Note that a driver is also available for the HP-UX (Hewlett-Packard's flavor of UNIX).

1.1 Related Documents

The hardware manual for the module: bc635/bc350VXI Time and Frequency Processor, User’s Guide (document/part number 8500-0019). Datum, Timing, Test & Measurement, 6741 Via del Oro, San Jose, CA 95119-1360, 408-578-4161. Describes how to install the module.

The UNIX manual pages for the 'BTFP'(BANCOMM time & frequency processor) device driver. Describes application process calls to the driver.

Driver Writing in the HP-RT Environment. Hewlett-Packard Co., Manual Part No. B3127-90006. Describes VME driver installation and resource configuration for HP-RT systems.

Also, the bc637VME GPS Satellite Receiver Addendum. Datum, Timing, Test & Measurement, 6741 Via del Oro, San Jose, CA 95119-1360, 408-578-4161. Discusses the installation and programming of the optional Satellite Receiver module.

1.2 Hardware Installation

Refer to the BTFP operation and technical manual for installation instructions for the module. Your system will likely reserve some part of the VME16 address space 0x0000-0xFFFF for use by user-supplied devices. You must examine the /HP-RT/etc/conf/cfg/CONFIG.TBL file to determine which, if any, addresses in this range are in use on your system. Then pick an address not in use, keeping in mind that the BTFP uses 64 bytes of address space. The example configuration file supplied is named btfpdrvr.cfg and has a BTFP device at 0xD000. Set your address on the BTFP's address switches and install the module. Verify that DIP switches SW2-3 & SW2-4 are in the VME (closed or ON) Position.

CHAPTER 2
SOFTWARE INSTALLATION

2.0 Procedure

WARNING: **BACK UP HOST AND TARGET BEFORE CONTINUING!**

This procedure assumes you know how to edit and copy files, become superuser, move between directories on a UNIX system, etc. Execute the following procedure as superuser.

On the target machine:

```
/etc/showreboot
```

to find the CPU number of the target machine.

On the host machine:

```
cd /HP-RT/etc/conf/cfg
view sysdev.h
```

Look for the following lines:

```
VME_1_CPU  0
VME_2_CPU  0
```

To determine which VME interrupts are available for your target platform. In the above example VME interrupts 1 and 2 are serviced by CPU 0.

NOTE: **If you change the interrupts serviced by the various processors on your system*, BE SURE* all systems have the same configuration(including the HP-UX systems).**

If more than 1 (of any type) of device will be interrupting on the same VME interrupt level, the HI_VME_x(i.e. HI_VME_2) must be defined as the highest value returned by the VME vectors associated with the interrupting devices.

1) On the host machine, untar the driver files and select the directory containing the driver files.

2) Edit the Makefile and change DRIVER_SRC to reflect the location of the driver files.

3) Edit btfpinfo.h and change the following:

   Modify BTFPCOUNT to reflect the number of btfp devices installed in the target system.
   Modify the btfp entries for devices installed in the system. Be sure that the
   #define BTFPx_VME_VEC VME_x_VEC entries match the BTFPx_VME_LEVEL entries chosen for the respective devices.

4) Edit the btfpdrvr.cfg file and change the following:

   Modify the swsm declaration:
   O:btfp0swsm:A16+DATA_ACESS:0xD000:0x0040:3:SYSTEM_ONLY
   to reflect the location of the installed btfp device. (In the above example, the btfp device is located at 0xD000)

5) If there is more than one device installed in the target system, uncomment a node and swsm declaration for each device and modify the address as detailed above.

6) Edit /HP-RT/etc/conf/cfg/CONFIG.TBL and change as follows:

   At the bottom of the file, add the following:
   I:btfpdrvr.cfg

   This will include the driver information in the new kernel. Be sure to add the line at the end of the file as location in the file determines the major device number assigned to the driver.

7) Change to the directory containing the driver source files and make the driver.

8) Now to make the new target kernel:

```
cd /HP-RT/etc/conf
make -f hp-rt.make
```

   Move the new kernel into place on the target machine and boot the new kernel.

9) Check that the driver is installed on the target machine.

```
ll -g /dev/btfp*
```

After logging in, do a quick test of the driver and module by typing 'cat /dev/btfp0'. This will read the time from the btfp device. Download a new time value using one of the following variations:

'date +%j%H%M%S > /dev/btfp0' or:
'echo "123112233" > /dev/btfp0'

In the latter case, the time is set to day 123, 11:22:33. In the former case, the workstation's time is downloaded to the btfp.

Run the btfptest example program from the target system. Modify the btfptest program on the host and rebuild it there with:

```
/HP-RT/hpux/bin/ccrt ./btfptest.c -lp -o/btfptest
```

The comprehensive standalone test suite is run by typing**: 'btfptest -a'**
For more information, type the **btfptest** command with no arguments.

CHAPTER 3
MANPAGE BTFP(7)

NAME

btfp - time and frequency processor interface

3.0 DESCRIPTION

The 'btfp' driver supports the Datum bc635VME/bc350VXI VME/VXIbus time and frequency processor (hereafter BTFP), as well as the bc637VME/bc357VXI GPS Satellite Receiver module (including a GPS receiver core module). It decodes a number of industry-standard time-code formats, provides notification of external and time events, etc.

The driver supports access to multiple BTFP modules (see NOTES). A single interrupt vector per module is supported for all 5 of the BTFP's interrupt sources. The calling process can choose to be blocked, waiting for an interrupt, or to receive a signal when a BTFP interrupt occurs.

The open(2), close(2), read(2), write(2), and ioctl(2) system calls are supported.

The open(2) call disables the BTFP's interrupt sources and puts the driver into SIGNALOFF mode (see below). The driver provides shared access to each BTFP (see NOTES).

The read(2) and write(2) calls behave differently depending on the mode of the driver. The default mode of the driver is "packet-mode off", meaning that read/write calls refer to the real time clock. Using the ioctl(2) calls PACKETON and PACKETOFF, the user can toggle to the "packet-mode on" mode, in which read/write calls refer to the OUTFIFO and INFIFO, respectively. Note that certain modes are affected by SIGNALON and SIGNALOFF modes (see below). Note also that each mode stays in effect until the device is closed, or another ioctl is issued to alter the mode.

3.1 Read/Write Default Mode ("PACKETOFF")

The read(2) call freezes the current time, and returns it from the module's TIMEn registers to the specified application buffer in ASCII format of up to 32 bytes(there are 3 spaces between each of the time fields, and the string is terminated by a new line and a zero), as follows:

365	24	60	60	999	999	9'\n''\0'
day	hr	min	sec	ms	us	ns

This feature allows you, for example, to use a cat(1) command to obtain the decoded time from the module. See section 3.3.1 of the <u>bc635VME User's Guide</u> for more details. Note that SIGNALON/SIGNALOFF have no effect on the read(2) call in this mode.

The write(2) call decodes the time sent in a specific format (see below), waits for the 1PPS pulse (unless SIGNALON mode is in effect), and issues a Packet "B" to the BTFP to set the major time.

To prepare a write(2) call in SIGNALON/PACKETOFF mode, the application must issue the following ioctl(2) commands:

> PACKETOFF (changes the following write(2) call into packet mode)
> [Note: this is only necessary if a PACKETON ioctl has been issued since opening the device, which is the default mode]
>
> SIGNALON (sets mode to non-blocking signal handling mode)
>
> WMASK (to clear/enable the 1PPS interrupt, and send a SIGUSR1 signal to the user when the 1PPS pulse occurs)
>
> The (non-blocking) write call is issued from the signal handler, once the SIGUSR1 signal has been received in response to the 1PPS interrupt. If both PACKETOFF and SIGNALOFF modes are in effect, then the write(2) call will immediately write the data packet (containing the date and time) to the INFIFO.

to load the major time using the write(2) call in default PACKETOFF mode, the date and time must be supplied in a form similar to that provided by the default read(2) call (see above). If the ms, us, and ns fields are present, they will be stripped off by the driver. Also, any intervening white space will be stripped out. Thus, the output of a default read(2) call can be issued as input to a default write(2) call. Another convenient way to use the default write(2) mode is shown in the following example (where '#' is the shell prompt), which loads the system's date and time into the BTFP:

```
# date '%j/H%M%S' > /dev/btfp0
```

3.2 Read/Write Packet Mode ("PACKETON")

Note that PACKETON mode provides the fastest means of downloading and uploading packets to and from the BTFP. However, for short packets, the ROUTFIFO and WINFIFO ioctl(2) commands can be used for reading and writing single bytes at a time, regardless of the PACKETON/PACKETOFF mode.

To prepare for making a read(2) call in SIGNALON mode, the application must issue the following ioctl(2) commands, followed by a non-blocking read(2) call:

> PACKETON (changes the following read(2) call into packet mode)
>
> SIGNALON (sets mode to non-blocking signal handling mode)
>
> WMASK (to clear/enable the data packet available interrupt, and send a SIGUSR1 signal to the user when a packet is available from the OUTFIFO)

The (non-blocking) read call is issued from the signal handler, once the SIGUSR1 signal has been received in response to the data packet available (DPA) interrupt. If in SIGNALOFF (blocking) mode, only the PACKETON ioctl call, followed by a (blocking) read(2) call is needed.

Next, the read(2) call will return the complete raw data packet, which can be as large as 512 bytes. If the first read(2) does not return the complete data packet, then subsequent reads can be issued, after placing the driver into SIGNALON mode (to ensure that the driver does not try to block for another DPA interrupt).

A write(2) call will write to all the bytes in the referenced buffer, up to but not including a (optional) terminating NULL character to the BTFP's INFIFO. This buffer should not exceed 512 bytes (not including the NULL character, if present).

***Note**:* The write(2) call in PACKETON mode is not affected by either SIGNALON or SIGNALOFF modes.

All other BTFP access (including also reading the time) is accomplished by ioctl(2) calls.

3.3 Ioctl Calls

The ioctl(2) arg is a pointer to an int for all ioctl(2) commands. The 16 and 8 bit data read from and written to the registers are right-justified in the least significant bits of the int variables. Definitions for all of these data/control fields as well as for the following ioctl(2) requests can be found in the BTFP Operation Manual. The driver does not interpret or modify with any of the data/control values except as noted below.

RIDR

The driver reads from the BTFP's ID register (ID), and returns them to the least significant 16 bits of the int pointed to by arg. This register contains Datum's VXIbus manufacturer's ID number, device class (register based). and address space (A16 only). Note that writes to this register are not allowed (see section 3.1.1 of the bc635VME User's Guide for more details).

RDTR

The driver reads from the BTFP's Device Type register (DEVICE), and returns them to the least significant 16 bits of the int pointed to by arg. This register contains the bc350VXI device type identifier. Note that writes to this register are not allowed (see section 3.1.1 of the bc635VME User's Guide for more details).

RSR

The driver reads from the BTFP's Status register (STATUS),and returns them to the least significant 16 bits of the int pointed to by arg. This register always returns 0xFFFF when read (see section 3.1.1 of the bc635VME User's Guide for more details).

WCR

The driver writes to the BTFP's Control register (CONTROL), the least significant 16 bits of the int pointed to by arg. Only bit 0 is acted on;setting bit 0 to 1 causes all pending interrupts to be unasserted and clearsall the following registers: UNLOCK, ACK, CR0, MASK, INTSTAT, VECTOR, and LEVEL. Setting bit 0 to 0 has no effect. See section 3.1.1 of the bc635VME User's Guide for more details.

RTIMEREQ

The driver reads the contents of the BTFP's capture register (TIMEREQ) and returns them to the least significant 16 bits of the int pointed to by arg. This causes the BTFP to freeze the current time. Note that the content of TIMEREQ is not intended to be meaningful.

RTIME0
RTIME1
RTIME2
RTIME3
RTIME4

Each of these commands returns one of the BTFP's 5 16-bit time registers, to the least significant 16 bits of the memory location addressed by arg. Table 3-3 on page 3-4 of the bc635VME User's Guide explains the register contents. See section 3.1.2 of the bc635VME User's Guide for more details.

REVENT0
REVENT1
REVENT2
REVENT3
REVENT4

Each of these commands returns one of the BTFP's 4 16bit event registers, to the least significant 16 bits of the memory location addressed by arg. Table 3-3 on page 3-4 of the bc635VME User's Guide explains the register contents. See section 3.1.3 of the bc635VME User's Guide for more details.

WSTROBE1
WSTROBE2
WSTROBE3

Each of these commands writes to the specified strobe register, the least significant 16 bits of the int pointed to by arg. Note: thereare only 3 strobe registers. Table 3-3 on page 3-4 of the BTFPOperations Manual explains the register contents.

RUNLOCK

The driver reads the contents of the BTFP's release time capture lockout register (UNLOCK) and returns them to the least significant 16 bits of the int pointed to by arg. This causes the BTFP to release time capture lockout. Note that the content of UNLOCK is not intended to be meaningful.

RACK

Read the BTFP's ACK register and transfer its contents to the least significant 8 bits of the memory location addressed by arg. See section 3.3 of the bc635VME User's Guide for more details.

WACK

Write the BTFP's ACK register with the least significant 8 bits of the contents of the int pointed to by arg. See section 3.3 of the bc635VME User's Guide for more details.

WCR0

Write the BTFP's control register 0 with the least significant 8 bits of the contents of the int pointed to by arg. Table 3-4 on page 3-6 of the bc635VME User's Guide explains the register contents. See section 3.1.7 of the bc635VME User's Guide for more details.

RCR0

Read the BTFP's control register 0 and transfer its contents to the least significant 8 bits of the memory location addressed by arg. Table 3-4 on page 3-6 of the bc635VME User's Guide explains the register contents. See section 3.1.7 of the bc635VME User's Guide for more details.

ROUTFIFO

Read one byte from the BTFP's OUTFIFO, and transfer its contents to the least significant 8 bits of the memory location addressed by arg. See section 3.3 of the bc635VME User's Guide for more details.

WINFIFO

Write one byte to the BTFP's INFIFO, from the least significant 8 bits of the int pointed to by arg. See section 3.3 of the bc635VME User's Guide for more details.

RMASK

Read from the BTFP's MASK register, and transfer its contents to the least significant 8 bits of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.8 of the bc635VME User's Guide for more details.

WMASK

Write the BTFP's MASK register, using the least significant 16 bits of the contents of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.8 of the bc635VME User's Guide for more details. If you set a bit such as the DPA interrupt bit, then the driver will either block until the interrupt occurs if in SIGNALOFF mode or return control to the application at once and then later generate a SIGUSR1 signal if in SIGNALON mode.

NOTE: WINTSTAT need not be used just prior to a WMASK ioctl call, since the driver clears the relevant Interrupt Status bit prior to writing to the MASK register.

RINTSTAT

Read from the BTFP's INTSTAT register, and transfer its contents to the least significant 8 bits of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.9 of the bc635VME User's Guide for more details.

WINTSTAT

Write to the BTFP's INTSTAT register, from the least significant 16 bits of the contents of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.9 of the bc635VME User's Guide for more details. NOTE: WINTSTAT need not be used just prior to a WMASK ioctl call, since the driver clears the relevant Interrupt Status bit prior to writing to the MASK register.

RVECTOR

Read from the BTFP's VECTOR register, and transfer its contents to the least significant 8 bits of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.9 of the bc635VME User's Guide for more details. This call is provided for completeness, for the driver does not allow callers to write to this register. The driver determines the interrupt vector from the header configuration information for this device. This call is used for VMEbus models only.

RLEVEL

Read from the BTFP's LEVEL register, and transfer its contents to the least significant 8 bits of the int pointed to by arg. Table 3-5 on page 3-7 of the bc635VME User's Guide explains the register contents. See section 3.1.9 of the bc635VME User's Guide for more details. This call is provided for completeness, for the driver does not allow callers to write to this register. The driver determines the interrupt level from the header configuration information for this device.

READTIME

Capture the current time, and return the contents of all 5 BTFP time registers into a structure of type btfp_time, defined in btfpu.h, which is pointed to by arg. This single ioctl(2) call is the fastest way the driver provides to obtain the time from the module.

READEVENT

Return the contents of all 5 BTFP event registers into a structure of type btfp_event, defined in btfpu.h, which is pointed to by arg. This single ioctl(2) call is thefastest way the driver provides to obtain the event time from the module. See section 3.1.3 of the bc635VME User's Guide for more details.

WRITESTROBE

First, disable strobe output by writing to CR0 (STRDIS). Next, from a structure of type btfp_strobe, defined in btfpu.h, pointed to by arg, write the contents of all 3 of the BTFP's strobe words. A subsequent WCR0 call can then enable strobe output (STREN).

READALLREGS

Read all readable registers on the BTFP, and place them into the caller-supplied structure addressed by arg. The structure is of type btfp_device as defined in btfpu.h provided with the driver package. The structure is an image of the BTFP registers. The registers are read in order from the base to the highest. Note that TIMEREQ and UNLOCK are accessed by this call, while OUTFIFO is not read.

RPACKET

Enable a Data Packet Available interrupt, wait for it, then read a data packet from the OUTFIFO into the caller-supplied structure pointed to by arg. The structure is of type btfp_packet as defined in btfpu.h provided with the driver package. The structure is provided "raw", that is exactly as presented by the BTFP. See section 3.3 of the <u>bc635VME User's Guide</u>, or the GPS Operations Manual, for more details.

WPACKET

Write a data packet to the INFIFO from the caller-supplied structure pointed to by arg. The structure is of type btfp_packet as defined in btfpu.h provided with the driver package. The structure will be transferred to the BTFP "raw", that is exactly as provided by the caller. See section 3.3 of the <u>bc635VME User's Guide</u>, or the GPS Operations Manual, for more details.

SIGNALON

Tells the driver to send a signal of type SIGUSR1 to the calling process upon receipt of any device interrupt from the BTFP. Puts the driver into non-blocking mode on receipt of a MASK command to enable any of the BTFP interrupt sources via the setting of any of the INTEN0, INTEN1, INTEN2, INTEN3, or INTEN4 MASK bits. Signal sending will continue until the process sends a SIGNALOFF ioctl(2) call to the driver, or the device is closed via a close(2) call.

SIGNALOFF

Tells the driver to stop sending signals of type SIGUSR1 to the calling application upon receipt of any device interrupt from the BTFP. Also disables all 5 of the BTFP's interrupt sources by clearing MASK_INT0, MASK_INT1, MASK_INT2, MASK_INT3, MASK_INT4 of the MASK register.

PACKETON

Tells the driver to interpret subsequent read(2) and write(2) commands as requests to read data packets to/from the OUT/IN FIFO's. See above for more detailed discussion of the read(2) and write(2) calls in this mode. This mode stays in effect until a close(2) call, or a subsequent ioctl(2) call specifying PACKETOFF.

PACKETOFF

Tells the driver to revert to the power-up default, in which read(2) and write(2) calls are interpreted as accesses to the BTFP's real time clock. See above for more detailed discussion of the read(2) and write(2) calls in this mode.

DEBUGON

Enables some limited printfs for driver debugging purposes. Not intended for production use.

DEBUGOFF

Disables some limited printfs for driver debugging purposes. Not intended for production use.

FILES

/dev/btfp*	BTFP special files installed with mknod(8).
btfpu.h	Header required for processes calling the driver.
btfptest.c	Example application showing some driver calling sequences.

EXAMPLES

The BTFP Operation Manual contains example programming sequences. The btfptest.c sample application is provided with the driver. It demonstrates a number of possible calling sequences upon which you may build.

SEE ALSO

The user's guide: bc635VME/bc350VXI Time and Frequency Processor, User's Guide. Datum Timing, Test and Measurement, 6741 Via del Oro, San Jose, CA 95119-1360, 408-578-4161.

Also see the GPS satellite receiver manual, bc637VME/bc357VXI GPS Satellite Receiver Addendum, Operation and Technical Manual, Datum Timing, Test and Measurement, 6741 Via del Oro, San Jose, CA 95119-1360, 408-578-4161.

The open(2), read(2), ioctl(2), and close(2), signal(2) or sigset(2), pause(2) or sigpause(2), and cat(1) system call descriptions in the Programmer's Reference Manual which was supplied with your UNIX-based system.

WARNINGS

If SIGNALON mode is not in effect, the writing to MASK via a WMASK command which enables any of the BTFP's INTENn MASK bits, will cause the driver to sleep awaiting the interrupt. It is possible that the cause of the interrupt may already have happened, so the process could sleep forever. This may be avoided by using signals. To do that, make the signal(2) or sigset(2) call, then issue a SIGNALON command, then issue a WMASK command to enable the relevant INTENn bits, and then finally set up the BTFP so that it will begin generating interrupts (start the heartbeat interrupt, for example).

When using signals, only one process can usefully have a BTFP open at once.

A problem may possibly arise concerning the handling of critical sections of code. The standard method for protecting critical sections of code is to make spl calls to change the system priority level so that no interrupts at the device's interrupt level or below can interrupt the process. The problem with this system is that all VME bus interrupts are processed and then the HP-UX PA-RISC processor is interrupted at processor interrupt level 6. This is a very high priority interrupt level. Among other processes interrupting at this level is the system clock. The current implementation of this driver makes a spl call at the VME interrupt level of the device. If data is corrupted try modifying the spln() routine in the device driver to return spl6(). You can do this by adding the following line above the switch (ipl) statement.

```
return( spl6() );
```

3.4 NOTES

Versions of this driver are available for systems running SunOS version 4.1.x, Solaris 2.X, HP-UX v9.01 and HP-RT v1.1. Also, a 'generic' UNIX System V driver template is available for use in porting to other systems. The HP-RT v1.1x driver version is written to support up to 4 BTFP modules, limited by MAXDEV defined in btfpd.h. This can be changed by recompilation of this driver. The BTFP is considered a special purpose device, and so full, shared access to its features is provided as directly as possible to applications which must, in turn, be cautious to coordinate its use amongst themselves, if necessary. For example, if one process is engaged in taking signals from a BTFP, then if another process opens the same module, that will disable the BTFP's interrupts. On the other hand, multiple processes could have the same BTFP open and use it for timestamping, via the READTIME ioctl(2) call, without interference with each other.

For Heartbeat programming, the best method is the SIGNALON mode. On a standalone system, a heartbeat rate of >4500 heartbeats per second causes the signal handler to miss heartbeats; thus, the test program prevents timer values that calculate to >4500 heartbeats per second.

This driver has been written such that the application can service signals for one interrupt type at a time (e.g., Heartbeat). If the application needs to service more than one interrupt type at a time (e.g., External Interrupt and Strobe), then the following modifications will be necessary to btfp.c: in the btfp_intr() routine in btfp.c, in the signalon section, the driver will need to distinguish between the two interrupt types, and send SIGUSR1 for one type and SIGUSR2 for the other. Then the application can service the signal types separately (either with separate signal handlers, or with a common signal handler that looks at the signal value passed to it). There are apparently only two user signals, and no other mechanism for passing up additional information in a timely way from the driver, so two interrupt types are believed to be the maximum supportable under all the operating systems. The HP-RT v1.1x version of the driver was tested on a Hewlett-Packard HP747i industrial workstation which is equipped with a HP 742rt.

BUGS

DIAGNOSTICS

See **DEBUGON, DEBUGOFF** above.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4
BTFPTEST OUTPUT

4.0 Listing

```
BANCOMM bc63{57}VME/bc350VXI Time and Frequency Processor, Driver Test Program.

 Mon Jan 10 20:29:58 1994


 Open(2) test (standard UNIX open of the device)...
   Time and frequency processor /dev/btfp0 opened successfully.

   READALLREGS- Read all readable bc63{57} registers at once...

         ID =     0xfef4.

         DEVICE=  0xf350.

         STATUS=  0xf250.

         TIMEREQ= 0xfe7e.

         TIME0=   0x0060.
         TIME1=   0x1104.
         TIME2=   0x3048.
         TIME3=   0x4108.

         TIME4=   0x1160.

         EVENT0=  0x0060.
         EVENT1=  0x1020.
         EVENT2=  0x2905.
         EVENT3=  0x0000.

         EVENT4=  0x0010.

         UNLOCK=  0x00ff.

         ACK=     0x0ef6.

         CR0=     0x0e00.
      Skipping IN/OUT FIFOs.
         MASK=    0x0ee0.
         INTSTAT= 0x0efb.
         VECTOR=  0x0e01.
         LEVEL=   0x0efa.

READTIME ioctl call test (read time in one call)...
     Two back-to-back such reads indicate call timing:

      READTIME- Time words 0: 0x0060   0x0060
      READTIME- Time words 1: 0x1104   0x1104
      READTIME- Time words 2: 0x3048   0x3048
      READTIME- Time words 3: 0x4122   0x4122
      READTIME- Time words 4: 0x3800   0x8390
READTIME test completed

READEVENT test
      READEVENT words:  0x0060  0x1104  0x3050  0x0000  0x0010
READEVENT test completed


 Read(2)/Write(2) test (default mode)...
```

```
      3 Time reads before SIGNALOFF write call...

      Time read is: 00   6   011   04   30   50   99


      Time read is: 00   6   011   04   30   50   99
      Time read is: 00   6   011   04   30   52   99
      Loading major time as 010203004
      3 Time reads after SIGNALOFF write call and before SIGNALON write call...
      Time read is: 00   6   011   04   30   52   99
      Time read is: 00   6   010   20   30   05   99
      Time read is: 00   6   010   20   30   05   99
      Loading major time as 010203007
      3 Time reads after SIGNALON write call...
      Time read is: 00   6   010   20   30   07   99
      Time read is: 00   6   010   20   30   07   99
      Time read is: 00   6   010   20   30   09   99


 Ioctl(2) call tests...

      RIDR - Read from the ID register...
      RIDR - ID register is 0xfef4.


      RDTR - Read from the DT register...
      RDTR - DT register is 0xf350.


      RSR - Read from the Status register...
      RSR - Status register is 0xff7f.

      RTIME0- Time word 0 is 0x0060.
      RTIME1- Time word 1 is 0x1020.
      RTIME2- Time word 2 is 0x3009.
      RTIME3- Time word 3 is 0x9947.
      RTIME4- Time word 4 is 0x0540.


NOTE: The content of these registers are meaningless for this test:
      REVENT0- Event word 0 is 0x0060.
      REVENT1- Event word 1 is 0x1104.
      REVENT2- Event word 2 is 0x3050.
      REVENT3- Event word 3 is 0x0000.
      REVENT4- Event word 4 is 0x0010.
      Writing 0x66 to each Strobe register individually.

   Strobe register writes OK

      Reading from UNLOCK register.
      UNLOCK register read OK

      ACK register is 0x00e2.
      Writing the same value back to the ACK register...

      ACK register is 0x00e0.
      ACK register read/write OK

      WCR0 - RCR0- Write/read control register 0...
      RCR0- Read the register's current state...
      RCR0- Control register 0 is 0x0000.

      WCR0- Write all zeros to the register...
      RCR0- Control register 0 is 0x0000.
      WCR0- Write 0x4 to the register...

      RCR0- Control register 0 is 0x0004.

      WCR0- Clear it to all zeros, its hardware initialized state.

      WCR0 - RCR0 Write/read control register 0 tests complete

      IN/OUT FIFO tests

            NOTE: data are meaningless for this test
            Writing zero to the INFIFO...
      Read 0xff from the OUTFIFO...
      IN/OUT FIFO tests completed

Write/Read MASK register tests
      MASK register is now  0x0000.
      Clearing MASK register
      MASK register set to  0x0000.
MASK tests completed

INTSTAT register tests
      Waiting for the 1PPS status bit...
      Got the 1PPS status bit.
INTSTAT tests completed

VECTOR/LEVEL register tests
      VECTOR register contents:  0x0001.
      LEVEL register contents:  0x00fa.
VECTOR/LEVEL tests completed

WRITESTROBE test
      Writing zeros to all the strobe registers.
WRITESTROBE test completed

SIGNALON/OFF test
      Setting Signal Handling Mode (SIGNALON).
      Restoring to Default Blocking Mode (SIGNALOFF).
SIGNALON/OFF test completed

PACKETON/OFF test
      Setting Packet Mode (PACKETON).
      Restoring to Default Non-Packet Mode (PACKETOFF).
PACKETON/OFF test completed

All Ioctl(2) Call Tests Completed.

Load Major Time test (using ioctl calls)
      Waiting for the 1PPS status bit...
      Got the 1PPS status bit.
      Loading major time as 123 11:22:33
      Verify that display is incrementing from 11:22:33...

      Now setting to the system clock's time...
      Loading major time as 010203021
      Waiting for the 1PPS status bit...
      Got the 1PPS status bit.
      Verify that display is incrementing from 20:30:21...
Load Major Time test completed
GPS Read/Write test
      Requesting Current Time from GPS...
      GPS Time packet received: 10 41 48 38 a8 fd 02 db 41 10 10 00 00 10 03

GPS Read/Write test completed
```

```
Interrupt tests...Blocking mode

External Interrupt Test
      READTIME words:  0x0060  0x1104  0x3124  0x7130  0x0730
      Waiting for external interrupt...
      READEVENT words:  0x0060  0x1104  0x3125  0x0000  0x0000
End of External Interrupt Test
Heartbeat Interrupt Test
      READTIME words:  0x0060  0x1104  0x3125  0x0004  0x7600
      Heartbeat will be 10 times per second
      READTIME words:  0x0060  0x1104  0x3127  0x1001  0x3430
      READTIME words:  0x0060  0x1104  0x3128  0x1001  0x2450
      READTIME words:  0x0060  0x1104  0x3129  0x1001  0x3150
      READTIME words:  0x0060  0x1104  0x3130  0x1001  0x2700
      READTIME words:  0x0060  0x1104  0x3131  0x1001  0x2750
      READTIME words:  0x0060  0x1104  0x3131  0x1003  0x5140
End of Heartbeat Interrupt Test
Major/minor Strobe Interrupt Test
      READTIME words:  0x0060  0x1104  0x3131  0x1006  0x0740

      RTIME2- Time word 2 to the strobe is 0x3134.

      Now writing the later time value to the strobe ...
      READTIME words:  0x0060  0x1104  0x3134  0x1002  0x3250
End of Major/minor Strobe Interrupt Test
Minor-only Strobe Interrupt Test
      READTIME words:  0x0060  0x1104  0x3134  0x1005  0x6310

      RTIME2- Time word 2 to the strobe is 0x3137.

      Now writing the later time value to the strobe ...
      READTIME words:  0x0060  0x1104  0x3135  0x1002  0x2740
      READTIME words:  0x0060  0x1104  0x3136  0x1002  0x1470
      READTIME words:  0x0060  0x1104  0x3137  0x1002  0x2950
      READTIME words:  0x0060  0x1104  0x3138  0x1002  0x1740
      READTIME words:  0x0060  0x1104  0x3139  0x1002  0x1480
End of Minor-only Strobe Interrupt Test
1PPS Interrupt Test
      READTIME words:  0x0060  0x1104  0x3140  0x0002  0x1330
      READTIME words:  0x0060  0x1104  0x3141  0x0002  0x1580
      READTIME words:  0x0060  0x1104  0x3142  0x0002  0x2550
      READTIME words:  0x0060  0x1104  0x3143  0x0002  0x1140
      READTIME words:  0x0060  0x1104  0x3144  0x0002  0x1530
End of 1PPS Interrupt Test

End of Interrupt tests...Blocking mode


Interrupt tests...Signaling mode

External Interrupt Test
      READTIME words:  0x0060  0x1104  0x3144  0x0005  0x9100
      Waiting for external interrupt...
      READEVENT words:  0x0060  0x1104  0x3145  0x0000  0x0000
End of External Interrupt Test
Heartbeat Interrupt Test
      READTIME words:  0x0060  0x1104  0x3145  0x0007  0x2950
      Heartbeat will be 10 times per second
      READTIME words:  0x0060  0x1104  0x3147  0x1002  0x4660
      READTIME words:  0x0060  0x1104  0x3148  0x1002  0x3110
      READTIME words:  0x0060  0x1104  0x3149  0x1002  0x3590
      READTIME words:  0x0060  0x1104  0x3150  0x1002  0x3510
      READTIME words:  0x0060  0x1104  0x3151  0x1002  0x3580
End of Heartbeat Interrupt Test
```

```
Major/minor Strobe Interrupt Test
      READTIME words:  0x0060  0x1104  0x3151  0x1006  0x4400

      RTIME2- Time word 2 to the strobe is 0x3154.

      Now writing the later time value to the strobe ...
      READTIME words:  0x0060  0x1104  0x3154  0x1003  0x4140
End of Major/minor Strobe Interrupt Test
Minor-only Strobe Interrupt Test
      READTIME words:  0x0060  0x1104  0x3154  0x1006  0x9810

      RTIME2- Time word 2 to the strobe is 0x3157.

      Now writing the later time value to the strobe ...
      READTIME words:  0x0060  0x1104  0x3155  0x1003  0x5590
      READTIME words:  0x0060  0x1104  0x3156  0x1003  0x3410
      READTIME words:  0x0060  0x1104  0x3157  0x1003  0x4920
      READTIME words:  0x0060  0x1104  0x3158  0x1003  0x3380
      READTIME words:  0x0060  0x1104  0x3159  0x1003  0x3710
End of Minor-only Strobe Interrupt Test
1PPS Interrupt Test
      READTIME words:  0x0060  0x1104  0x3200  0x0003  0x3470
      READTIME words:  0x0060  0x1104  0x3201  0x0003  0x3620
      READTIME words:  0x0060  0x1104  0x3202  0x0003  0x4560
      READTIME words:  0x0060  0x1104  0x3203  0x0003  0x4120
      READTIME words:  0x0060  0x1104  0x3204  0x0003  0x3310
End of 1PPS Interrupt Test

End of Interrupt tests...Signaling mode

All tests completed!
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5
BTFPTEST SOURCE CODE

5.0 Listing

```
#ident "@(#)btfptest.c          1.7"

/************************************************************************/
/*
                BANCOMM bc63{57}VME/ bc350VXI Time and Frequency Processor
                                        Test Application for UNIX Systems.

                Copyright (C) 1992, BANCOMM, San Jose, CA.  All rights reserved.

                Author:         Greg Dowd
*/
/************************************************************************/
/*
                Description:  This program is used to test the bc63{57} driver functions.
                                                        It is not specifically designed to test
                                                        the bc63{57} module.
*/
/************************************************************************/
/*
                Notes:          #defines are used to select the OS version
                #define BSDUNIX - Use this if on SunOS or other Berkely type OS
                                        which does not support sigset/sigpause.
*/
/************************************************************************/
/************************************************************************/
/*
                Command line options select various tests to be run, as well as other
                        test options. Options are processed left to right, so the
                        rightmost option takes precedence.

                btfptest -vSaALRTEDrilp -b <block test #> -s <signal test #> -d <display #>

                where:
                -a : Run all tests, excluding tests g, b1, and s1.
                -A : Run ALL tests (assumes external interrupt generator and GPS attached)
                -v : verbose mode (prints diagnostic displays to stdout)
                -S : Slow terminal (defaults to fast terminal)
                -L : Loop through all selected tests, as appropriate (default: no loop)
                -R : READALLREGS test. Reads all bc63{57} registers.
                -T : READTIME test.
                -E : READEVENT test.
                -I : Interactive mode (affects heartbeat/external-interrupt tests only)
                -D : enable driver debugging
                -r : read(2)/write(2) test (default mode).
                -i : enable simple ioctl(2) tests
                        (excludes READTIME, READEVENT, READALLREGS)
                -l : load major time test. Uses current time from system clock.
                -g : select GPS test. Writes to INFIFO, reads from OUTFIFO.
                -b : select block-until-interrupt test
                        0 : no block-until-interrupt tests
                        1 : external interrupt test
                        2 : heartbeat test
                        3 : major/minor strobe test (one-time)
                        4 : minor-only strobe test (once per second)
                        5 : 1PPS test
                        6 : all block-until-interrupt tests
                -s : select signal handling test
                        0 : no signal handling tests
                        1 : external interrupt test
                        2 : heartbeat test
                        3 : major/minor strobe test (one-time)
                        4 : minor-only strobe test (once per second)
                        5 : 1PPS test
                        6 : all signal handling tests
                -d : select display mode when heartbeat signal received
```

```
                              0 : no heartbeat status display
                              1 : display time of heartbeats (DEFAULT)
                              2 : display count of heartbeats
                              3 : display time and count of heartbeats
*/
/***********************************************************************/
/*          Headers containing definitions used in this program.
*/
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <ctype.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include "btfpu.h"

#define     BLOCK    1          /* Blocking mode */
#define     SIGNAL   2          /* Signaling mode */

extern int signalhandler();
extern int hbsighandler();
extern int wtsighandler();
extern int rpsignalhandler();
extern int optarg;
/***********************************************************************/
struct btfp_device ds;                          /* Device structure for use with    */
                                                                        /* READALL                    */
struct btfp_time ts;                                    /* 16bit time words. */
                                                                        /* Used with READTIME ioctl call.*/
struct btfp_time t2s;                           /* Same thing for the machine speed test.*/
struct btfp_event event;                                /* Event words */
                                                                        /* Used with READEVENT ioctl call.*/
struct btfp_strobe strobe;                              /* Strobe words */
                                                                        /* Used with WRITESTROBE ioctl call.*/

struct sigvec myhandler;                        /* Used for signal handler          */
struct sigvec myhandlerhb;
struct sigvec myhandlerrp;
struct sigvec myhandlerwt;

int signalcount;                                        /* Used for counting received signals.*/
int         printcount;                                         /* Limits printing of signal count.*/
int         prints_done;                                        /* Limits looping of heartbeats.    */
long per_second;                                        /* Heartbeats per second                        */
int bytes_read;                                                 /* Packet bytes read                    */

/***********************************************************************/
  int fildes;                   /* Return value from open.     */
  char* bc635name = "/dev/btfp0";       /* Name of the bc63{57} device.   */
  char    buf[ 32 ];              /* Place to put time read in.   */
  char    time_buf[ 12 ];           /* Place to put jjjHHMMSS date. */
        int                 num_chars;                                  /* Number of chars in time_buf */
  unsigned    nbytes;             /* Number of bytes to read.     */
  unsigned    numread;             /* # bytes actually read.       */
  int       regvalue, newvalue;       /* Register value returned.     */
  int       t;                  /* Temporary time holder.      */
  int       time0;                /* Time values for general use. */
  int       time1;
  int       time2;
  int       time3;
  int       i;                  /* Loop index thanks to K&R.    */
  unsigned int    uitemp;            /* Temporary unsigned integer.  */
  unsigned int    uihold;            /* Temporary unsigned integer.  */
  time_t        unixtime;          /* Returned by 'time'.         */
  int                 err;
        unsigned char wpacket[MAXPACKET];                       /* for write to INFIFO */
        unsigned char rpacket[MAXPACKET];                               /* for read from INFIFO */
        char delimiters[2]={' ','\n'};                          /* for build_packet */
        int time_requested;                                                     /* GPS test globals */
        int time_received;
        int setup_done;
        unsigned char packet_21[7]={                    /* "Request Current Time" packet */
                SOH, 'J', ETB, DLE, 0x21, DLE, ETX}; /* for GPS Satellite Receiver */

  void usage(),
                do_readall_test(),
                do_readtime_test(),
                do_readevent_test(),
                do_read_test(),
                do_ioctl_test(),
                do_loadtime_test(),
                do_GPS_test(),
                do_int_tests ();
        void set_heartbeat(),
                set_strobe(),
                write_time(),
                send_packet(),
                write_packet(),
                read_packet(),
                build_packet(),
                disp_packet();

  /* Set up default flags... */
  int   verbose = NO;
  int   fast_terminal = YES;
  int   all_tests = NO;
  int   ALL_tests = 0;
  int   test_picked = NO;
  int   loop_tests = NO;
  int   readall_test = NO; /* READALLREGS */
  int   readtime_test = NO; /* READTIME */
  int   readevent_test = NO; /* READEVENT */
  int   read_test = NO; /* read(2) */
  int   ioctl_tests = NO;
  int   loadtime_test = NO;

  /* ...and default modes */
  int   GPS_test = 0;
  int   block_tests = 0;
  int   signal_tests = 0;
  int   display_mode = 1;                               /* Default to display time */
  int   interact_mode = 0;

/************************************************************************/
main (argc, argv)
int argc;
char **argv;
{
        int dummy;
        printf( "\n\n\n" );
        printf( "BANCOMM bc63{57}VME/bc350VXI Time and Frequency Processor, Driver Test Program.\n" );
        time( &unixtime );                                              /* Get current 'unix-format' time.*/
        printf( "\n %s", ctime( &unixtime ) );          /* Log out the current date/time.*/

        /* Check arguments */
        if (argc == 1)
        {
                usage(argv[0]);
                exit (-1);
        }

/******************************************************************************/
        printf( "\n\n Open(2) test (standard UNIX open of the device)...\n" );
        fildes = open( bc635name, O_RDWR );
                                                                                                      /* Open the bc63{57}
device.         */
        if( fildes == -1 )
        {
                printf( "   UNABLE TO OPEN bc63{57} DEVICE %s!\n\n", bc635name );
                exit( 0 );                                                          /* This ends the test.          */
        }
        else
        {
                printf( "   Time and frequency processor %s opened successfully.\n", bc635name );
                if( ioctl( fildes, RUNLOCK, &dummy ) ) pioerr(__LINE__);
                                                                                                      /* Release capture lockout
in      */
                                                                                                      /* case it was set from
before. */
                if( ioctl( fildes, WCR0, 0 ) ) pioerr(__LINE__); /* Clean up from kill on       */
        } /* end of successful open case */

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

        while ((err = getopt (argc, argv, "vSaALRTEIDrilgb:s:h:d:"))
                        != -1)
                switch (err)
                {
                case 'v':
                        verbose = YES;
                        printf ("Enabling verbose mode\n");
                        break;

                case 'S':
                        if (verbose)
                                printf ("Slow Terminal Mode\n");
                        fast_terminal = NO;
                        break;

                case 'a':
                        if (verbose)
                                printf ("All Tests Mode (except for interrupt tests)\n");
                        all_tests = YES;
                        test_picked = YES;
                        readall_test = YES; /* READALLREGS */
                        readtime_test = YES; /* READTIME */
                        readevent_test = YES; /* READEVENT */
                        read_test = YES; /* read(2) */
                        ioctl_tests = YES;
                        loadtime_test = YES;
                        block_tests = 6;        /* Won't do external interrupt test */
                        signal_tests = 6;       /* Won't do external interrupt test */
                        break;

                case 'A':
                        if (verbose)
                                printf ("ALL TESTS Mode (including external tests)\n");
                        ALL_tests = YES;
                        test_picked = YES;
                        readall_test = YES; /* READALLREGS */
                        readtime_test = YES; /* READTIME */
                        readevent_test = YES; /* READEVENT */
                        read_test = YES; /* read(2) */
                        ioctl_tests = YES;
                        loadtime_test = YES;
                        GPS_test = YES;
                        block_tests = 6;
                        signal_tests = 6;
                        /* Note that display option still defaults to OFF */
                        break;
```

```
case 'L':
        if (verbose)
                printf ("Infinite Test-Loop Mode\n");
        loop_tests = YES;
        break;

case 'R':
        if (verbose)
                printf ("Read All Registers Test\n");
        readall_test = YES;
        test_picked = YES;
        break;

case 'T':
        if (verbose)
                printf ("READTIME Test\n");
        readtime_test = YES;
        test_picked = YES;
        break;

case 'E':
        if (verbose)
                printf ("READEVENT Test\n");
        readevent_test = YES;
        test_picked = YES;
        break;

case 'I':
        interact_mode = YES;
        break;

case 'D':
        if (verbose)
                printf ("Enable Driver debugging\n");
        if( ioctl( fildes, DEBUGON, 0 ) ) pioerr(__LINE__); /* Set driver debug flag */
        break;

case 'r':
        if (verbose)
                printf ("read(2) Test\n");
        read_test = YES;
        test_picked = YES;
        break;

case 'i':
        if (verbose)
                printf ("ioctl(2) Test\n");
        ioctl_tests = YES;
        test_picked = YES;
        break;

case 'l':
        if (verbose)
                printf ("Load Major Time Test\n");
        loadtime_test = YES;
        test_picked = YES;
        break;

case 'g':
        if (verbose)
                printf ("GPS Test\n");
        GPS_test = YES;
        test_picked = YES;
        break;

case 'b':
        block_tests = atoi (optarg);
        switch (block_tests)
```

```
                {
                        case 0:
                                if (verbose)
                                        printf ("No blocking tests\n");
                                break;
                        case 1:
                                if (verbose)
                                        printf ("External Interrupt blocking test\n");
                                break;
                        case 2:
                                if (verbose)
                                        printf ("Heartbeat blocking test\n");
                                break;
                        case 3:
                                if (verbose)
                                        printf ("Major/Minor Strobe blocking test\n");
                                break;
                        case 4:
                                if (verbose)
                                        printf ("Minor-only Strobe blocking test\n");
                                break;
                        case 5:
                                if (verbose)
                                        printf ("1PPS blocking test\n");
                                break;
                        case 6:
                                if (verbose)
                                        printf ("ALL blocking tests\n");
                                break;
                        default:
                                fprintf (stderr, "Invalid Block Test Option: %d\n",
                                        block_tests);
                                exit (1);
                } /* end of "switch (optarg)" */
                test_picked = YES;
                break;

        case 's':
                signal_tests = atoi (optarg);
                switch (signal_tests)
                {
                        case 0:
                                if (verbose)
                                        printf ("No signaling tests\n");
                                break;
                        case 1:
                                if (verbose)
                                        printf ("External Interrupt signaling test\n");
                                break;
                        case 2:
                                if (verbose)
                                        printf ("Heartbeat signaling test\n");
                                break;
                        case 3:
                                if (verbose)
                                        printf ("Major/minor Strobe signaling test\n");
                                break;
                        case 4:
                                if (verbose)
                                        printf ("Minor-only Strobe signaling test\n");
                                break;
                        case 5:
                                if (verbose)
                                        printf ("1PPS signaling test\n");
                                break;
                        case 6:
                                if (verbose)
                                        printf ("ALL signaling tests\n");
                                        break;
                                default:
                                        fprintf (stderr, "Invalid Signal Test Option: %d\n",
                                                signal_tests);
                                        exit (1);
                        } /* end of "switch (optarg)" */
                        test_picked = YES;
                        break;

                case 'd':
                        display_mode = atoi (optarg);
                        switch (display_mode)
                        {
                                case 0:
                                        if (verbose)
                                                printf ("No display for heartbeat tests\n");
                                        break;
                                case 1:
                                        if (verbose)
                                                printf ("Display time for heartbeat tests\n");
                                        break;
                                case 2:
                                        if (verbose)
                                                printf ("Display count for heartbeat tests\n");
                                        break;
                                case 3:
                                        if (verbose)
                                                printf ("Display time and count for heartbeat tests\n");
                                        break;
                                default:
                                        fprintf (stderr, "Invalid Display Mode: %d\n",
                                                display_mode);
                                        exit (1);
                        } /* end of "switch (optarg)" */
                        break;

                default:
                                fprintf (stderr, "Invalid option\n");
                case '?':
                                (void) usage (argv[0]);
                                exit (-1);

                } /* end of "switch (err)" */

                if (test_picked == NO)
                {
                        printf ("No test selected.\n");
                        exit (1);
                }

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

                /* Main section of test */

        send_packet("A1");
        send_packet("P05");
        do
        {

                if (readall_test)
                        (void) do_readall_test();

                if (readtime_test)
                        (void) do_readtime_test();

                if (readevent_test)
                        (void) do_readevent_test();
                if (read_test)
                        (void) do_read_test();

                if (ioctl_tests)
                        (void) do_ioctl_test();

                if (loadtime_test)
                        (void) do_loadtime_test();

                if (GPS_test)
                        (void) do_GPS_test();

                if (block_tests)
                        (void) do_int_tests (block_tests, BLOCK);

                if (signal_tests)
                        (void) do_int_tests (signal_tests, SIGNAL);

        } /* end of Big Loop */
        while (loop_tests == YES);

        if (all_tests == YES)
                printf ("\nAll standalone tests completed!\n");

        if (ALL_tests == YES)
                printf ("\nAll tests completed!\n");

} /* end of main routine */
/****************************************************************************/
void
do_readall_test()
{

        printf( "\n   READALLREGS- Read all readable bc63{57} registers at once...\n\n" );
        if( ioctl( fildes, READALLREGS, &ds ) ) pioerr(__LINE__);
        printregs();                                                    /* Print out what they all are.    */

} /* end of do_readall_test routine */
/****************************************************************************/
void
do_readtime_test()
{

        printf( "\nREADTIME ioctl call test (read time in one call)...\n" );
        printf( "    Two back-to-back such reads indicate call timing:\n\n" );

        if( ioctl( fildes, READTIME, &ts ) ) pioerr(__LINE__);
                                                                                        /* Read all 5
time registers.*/
        if( ioctl( fildes, READTIME, &t2s ) ) pioerr(__LINE__);

        for (i=0; i < 5; i++)
        {
                printf( "     READTIME- Time words %d: 0x%04x   0x%04x\n",
                                                        i, ts.btfp_time[ i ] , t2s.btfp_time[ i ]);
        }

        printf ("READTIME test completed\n");

} /* end of do_readtime_test routine */
/****************************************************************************/
void
do_readevent_test()
{
        int dummy;

        /* READEVENT test */
```

```
		printf ("\nREADEVENT test\n");

		/* first disable interrupts */
		if( ioctl( fildes, WMASK, 0 ) ) pioerr(__LINE__);

		/* clear any existing interrupt status bits */
		if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
		if( ioctl( fildes, WINTSTAT, regvalue ) ) pioerr(__LINE__);
		/* clear any time capture lockout */
		if( ioctl( fildes, RUNLOCK, &regvalue ) ) pioerr(__LINE__);

		/* enable heartbeat time capture, and time capture lockout */
		if( ioctl( fildes, WCR0, LOCKEN | HBEN ) ) pioerr(__LINE__);

		/* set up the heartbeat signal to be 1 count per second, */
		/* synchronized with the 1 PPS */

		set_heartbeat( 10000, 1000);

		/* clear heartbeat interrupt */
		if( ioctl( fildes, WINTSTAT, INTSTAT_HEART ) ) pioerr(__LINE__);

		/* wait for heartbeat to occur */
		regvalue = 0;
		do
		{
			if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
		}
		while (!(regvalue & INTSTAT_HEART));

		/* clear the heartbeat status */
		if( ioctl( fildes, WINTSTAT, INTSTAT_HEART ) ) pioerr(__LINE__);

		/* Release capture lockout. */
		if( ioctl( fildes, RUNLOCK, &dummy  ) ) pioerr(__LINE__);

		/* Event registers should now have the captured time */
		get_event();

		/* Clear the CR0 register */
		if( ioctl( fildes, WCR0, 0 ) ) pioerr(__LINE__);

		printf ("READEVENT test completed\n");
} /* end of do_readevent_test routine */

/***************************************************************************/
void
do_read_test()
{
		int rtn;
		time_t timeval;

		/* Note: this routine exercises the "default" read/write mode, */
		/* which accesses the time in Free Running Mode (mode 1) */
		/* This mode corresponds to PACKETOFF */

		printf( "\n\n Read(2)/Write(2) test (default mode)...\n" );

		/* Ensure that we are in default mode for reads and writes */
		if( ioctl( fildes, PACKETOFF, 0 ) ) pioerr(__LINE__);

		printf ("\t3 Time reads before SIGNALOFF write call...\n");

		for( i = 0; i < 3; i++ )				/* Read the time and display it.*/
		{
			sleep( 1 );							/* Wait a second and go again.*/
			nbytes = 32;							/* Ask for all 32 bytes.*/
			numread = read( fildes, buf, nbytes );		/* Read time from bc63{57} device.*/
```

```
        if( numread == 0 )                                      /* Then we are at 'end of file'.*/
        {
            lseek( fildes, 0L, 0 );                         /* 'Rewind' the device.          */
            numread = read( fildes, buf, nbytes );      /* Read time from bc63{57}. */
        }
        else if( numread == -1 )
        {
            printf( "      ERROR READING TIME FROM %s!\n",  bc635name );
        }
        printf( "      Time read is: %s\n", buf );      /* Show the time read.          */
    } /* end of for loop */

    /* Now write to the BTFP real time clock from the PC's system clock */
    /* First in SIGNALOFF mode */
    write_time (SIGNALOFF);

    if (verbose)
        printf ("Wrote current time from PC system clock to BTFP (SIGNALOFF mode)\n");
    printf ("\t3 Time reads after SIGNALOFF write call and before SIGNALON write call...\n");

    /* Now read the time again */

    for( i = 0; i < 3; i++ )                            /* Read the time and display it.*/
    {
        sleep( 1 );                                             /* Wait a second and go again.*/
        nbytes = 32;                                                /* Ask for all 32 bytes.*/
        numread = read( fildes, buf, nbytes );      /* Read time from bc63{57} device.*/
        if( numread == 0 )                                      /* Then we are at 'end of file'.*/
        {
            lseek( fildes, 0L, 0 );                         /* 'Rewind' the device.          */
            numread = read( fildes, buf, nbytes );      /* Read time from bc63{57}. */
        }
        else if( numread == -1 )
        {
            printf( "      ERROR READING TIME FROM %s!\n",  bc635name );
        }
        printf( "      Time read is: %s\n", buf );      /* Show the time read.          */
    } /* end of for loop */

    /*  Write again to the BTFP real time clock from the PC's system clock */

    /* Next in SIGNALON mode */
    write_time (SIGNALON);

    if (verbose)
        printf ("Wrote current time from PC system clock to BTFP (SIGNALON mode)\n");
    printf ("\t3 Time reads after SIGNALON write call...\n");
    /* Now read the time again */

    for( i = 0; i < 3; i++ )                            /* Read the time and display it.*/
    {
        sleep( 1 );                                             /* Wait a second and go again.*/
        nbytes = 32;                                                /* Ask for all 32 bytes.*/
        numread = read( fildes, buf, nbytes );      /* Read time from bc63{57} device.*/
        if( numread == 0 )                                      /* Then we are at 'end of file'.*/
        {
            lseek( fildes, 0L, 0 );                         /* 'Rewind' the device.          */
            numread = read( fildes, buf, nbytes );      /* Read time from bc63{57}. */
        }
        else if( numread == -1 )
        {
            printf( "      ERROR READING TIME FROM %s!\n",  bc635name );
        }
        printf( "      Time read is: %s\n", buf );      /* Show the time read.          */
    } /* end of for loop */

} /* end of do_read_test routine */
```

```
/***************************************************************************/
void
do_ioctl_test()
{
        printf( "\n\n Ioctl(2) call tests...\n" );

        /* Configuration Registers */

        printf( "\n\tRIDR - Read from the ID register...\n");

        if( ioctl( fildes, RIDR, &regvalue ) ) pioerr(__LINE__); /* Read ID register */

        printf ( "\tRIDR - ID register is 0x%04x.\n\n", regvalue);

        printf( "\n\tRDTR - Read from the DT register...\n");

        if( ioctl( fildes, RDTR, &regvalue ) ) pioerr(__LINE__);        /* Read DT register */

        printf ( "\tRDTR - DT register is 0x%04x.\n\n", regvalue);

        printf( "\n\tRSR - Read from the Status register...\n");

        if( ioctl( fildes, RSR, &regvalue ) ) pioerr(__LINE__);  /* Read Status register */

        printf ( "\tRSR - Status register is 0x%04x.\n\n", regvalue);

        /* General Registers */

        /* Write to Control Register here? */

        /* NOTE: this would then require setting up LEVEL/VECTOR, which are */
        /* currently set up only by the driver */

        /* Time Request Registers */

        if( ioctl( fildes, RTIMEREQ, &regvalue ) ) pioerr(__LINE__);    /* Read the time req reg.*/
                                                                                                /* Ignore the read value */


        if( ioctl( fildes, RTIME0, &regvalue ) ) pioerr(__LINE__);      /* Read the 1st time reg.*/
        printf( "    RTIME0- Time word 0 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, RTIME1, &regvalue ) ) pioerr(__LINE__);      /* Read the 2nd time reg.*/
        printf( "    RTIME1- Time word 1 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, RTIME2, &regvalue ) ) pioerr(__LINE__);      /* Read the 3rd time reg.*/
        printf( "    RTIME2- Time word 2 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, RTIME3, &regvalue ) ) pioerr(__LINE__);      /* Read the 4th time reg.*/
        printf( "    RTIME3- Time word 3 is 0x%04x.\n", regvalue );

        if( ioctl( fildes, RTIME4, &regvalue ) ) pioerr(__LINE__);      /* Read the 5th time reg.*/
        printf( "    RTIME4- Time word 4 is 0x%04x.\n", regvalue );

        if( ioctl( fildes, REVENT0, &regvalue ) ) pioerr(__LINE__);     /* Read the 1st event reg.*/
        /* Event Registers */

        printf ("\n\nNOTE: The content of these registers are meaningless for this test:\n");
        printf( "    REVENT0- Event word 0 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, REVENT1, &regvalue ) ) pioerr(__LINE__);     /* Read the 2nd event reg.*/
        printf( "    REVENT1- Event word 1 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, REVENT2, &regvalue ) ) pioerr(__LINE__);     /* Read the 3rd event reg.*/
        printf( "    REVENT2- Event word 2 is 0x%04x.\n", regvalue );
        if( ioctl( fildes, REVENT3, &regvalue ) ) pioerr(__LINE__);     /* Read the 4th event reg.*/
        printf( "    REVENT3- Event word 3 is 0x%04x.\n", regvalue );

        if( ioctl( fildes, REVENT4, &regvalue ) ) pioerr(__LINE__);     /* Read the 5th event reg.*/
        printf( "    REVENT4- Event word 4 is 0x%04x.\n", regvalue );

        /* Strobe Registers */
        printf ("\tWriting 0x66 to each Strobe register individually.\n");
        regvalue = 0x66;
        if( ioctl( fildes, WSTROBE1, regvalue ) ) pioerr(__LINE__);         /* Write the 1st strobe reg.*/

        if( ioctl( fildes, WSTROBE2, regvalue ) ) pioerr(__LINE__);         /* Write the 2nd strobe reg.*/
        if( ioctl( fildes, WSTROBE3, regvalue ) ) pioerr(__LINE__);         /* Write the 3rd strobe reg.*/
        printf ("\n   Strobe register writes OK \n");

/****************************************************************************/
        /* UNLOCK, ACK registers */
        printf ("\n\tReading from UNLOCK register.\n");
        if( ioctl( fildes, RUNLOCK, &regvalue ) ) pioerr(__LINE__);         /* Read the UNLOCK reg. */
                                                                                        /* Ignore value read */
        printf ("\tUNLOCK register read OK \n");

        if( ioctl( fildes, RACK, &regvalue ) ) pioerr(__LINE__);            /* Read the ACK reg. */
        printf( " \n\tACK register is 0x%04x.\n", regvalue );

        /* write the same value back */
        if( ioctl( fildes, WACK, regvalue ) ) pioerr(__LINE__); /* Write the ACK reg. */

        printf ("\tWriting the same value back to the ACK register...\n");
        if( ioctl( fildes, RACK, &regvalue ) ) pioerr(__LINE__);            /* Read the ACK reg. */
        printf( " \n\tACK register is 0x%04x.\n", regvalue );
        printf ("\tACK register read/write OK \n");

/****************************************************************************/
        printf( "\n\tWCR0 - RCR0- Write/read control register 0...\n" );

        printf( "      RCR0- Read the register's current state...\n" );
        if( ioctl( fildes, RCR0, &regvalue ) ) pioerr(__LINE__);
        printf(   "      RCR0- Control register 0 is 0x%04x.\n", regvalue );
        uihold = regvalue;                                                  /* Save original state.  */

        printf( "\n      WCR0- Write all zeros to the register...\n" );
        if( ioctl( fildes, WCR0, 0 ) ) pioerr(__LINE__);
        if( ioctl( fildes, RCR0, &regvalue ) ) pioerr(__LINE__);
        printf(   "      RCR0- Control register 0 is 0x%04x.\n", regvalue );

        uitemp = 0x04;                                                              /* Set a new value.           */
        printf( "\n      WCR0- Write 0x%x to the register...\n", uitemp );
        if( ioctl( fildes, WCR0, uitemp ) ) pioerr(__LINE__);
        if( ioctl( fildes, RCR0, &regvalue ) ) pioerr(__LINE__);
        printf(   "      RCR0- Control register 0 is 0x%04x.\n", regvalue );

        printf( "\n      WCR0- Clear it to all zeros, its hardware initialized state.\n" );
        if( ioctl( fildes, WCR0, 0 ) ) pioerr(__LINE__);

        printf( "\n\tWCR0 - RCR0 Write/read control register 0 tests complete\n");
/****************************************************************************/
        /* FIFO basic tests; the GPS test actually exchanges real data */

        printf ("\n\tIN/OUT FIFO tests\n");
        printf ("\n\t\tNOTE: data are meaningless for this test\n");

        printf ("\t\tWriting zero to the INFIFO...\n");
        regvalue = 0; /* init to zero */
        if( ioctl( fildes, WINFIFO, regvalue ) ) pioerr(__LINE__);
        if( ioctl( fildes, ROUTFIFO, &regvalue ) ) pioerr(__LINE__);
        printf ("\tRead 0x%0.2x from the OUTFIFO...\n", regvalue);
        printf ("\tIN/OUT FIFO tests completed\n");

/****************************************************************************/
        /* MASK register */

        printf ("\nWrite/Read MASK register tests\n");

        if( ioctl( fildes, RMASK, &regvalue ) ) pioerr(__LINE__);
        /* Note: only bits 0-4 are used */
        printf(  "     MASK register is now  0x%04x.\n", regvalue & 0x1f);

        printf(  "     Clearing MASK register\n");

        regvalue = 0; /* init to zero */

        if( ioctl( fildes, WMASK, regvalue ) ) pioerr(__LINE__);
        if( ioctl( fildes, RMASK, &newvalue ) ) pioerr(__LINE__);
        /* Note: only bits 0-4 are used */
        if (regvalue != (newvalue & 0x1f))
        {
                printf (" Error: MASK register mismatch (old/new): 0x%04x/0x%04x\n",
                        regvalue, (newvalue & 0x1f));
        }
        else
        {
                printf(  "     MASK register set to  0x%04x.\n", (newvalue & 0x1f) );
        }

        printf ("MASK tests completed\n");

/*****************************************************************************/
        /* INTSTAT register test */

        printf ("\nINTSTAT register tests\n");

        /* clear any existing interrupt status bits */
        if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
        if( ioctl( fildes, WINTSTAT, regvalue ) ) pioerr(__LINE__);

        printf ("\tWaiting for the 1PPS status bit...\n");

        /* now look for the 1 PPS status bit */
        regvalue = 0;
        do {
                if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
                if( ioctl( fildes, WINTSTAT, regvalue ) ) pioerr(__LINE__);
        }
        while ((regvalue & INTSTAT_1PPS) == 0);

        printf ("\tGot the 1PPS status bit.\n");

        /* Clear the 1PPS interrupt status */
        if( ioctl( fildes, WINTSTAT, regvalue ) ) pioerr(__LINE__);

        /* clear the 1PPS ACK bit */
        if( ioctl( fildes, WACK, ACK_1PPS ) ) pioerr(__LINE__);

        printf ("INTSTAT tests completed\n");

/*****************************************************************************/
        /* VECTOR/LEVEL registers */

        printf ("\nVECTOR/LEVEL register tests\n");
        if( ioctl( fildes, RVECTOR, &regvalue ) ) pioerr(__LINE__);
        printf("\tVECTOR register contents:  0x%04x.\n", regvalue );
        if( ioctl( fildes, RLEVEL, &regvalue ) ) pioerr(__LINE__);
        printf("\tLEVEL register contents:  0x%04x.\n", regvalue );
        printf ("VECTOR/LEVEL tests completed\n");

/*****************************************************************************/
        /* READTIME done separately */
/*****************************************************************************/
        /* READEVENT done separately */

/*****************************************************************************/
        /* WRITESTROBE test */
        printf ("\nWRITESTROBE test\n");

        /* just clear to zeroes */
        for (i=0; i < 5; i++)
        {
                strobe.btfp_strobe [i] = 0;
        }
        printf ("\tWriting zeros to all the strobe registers.\n");
        if( ioctl( fildes, WRITESTROBE, &strobe ) ) pioerr(__LINE__);
        printf ("WRITESTROBE test completed\n");
/****************************************************************************/
        /* READALLREGS done separately */
/****************************************************************************/
        /* SIGNALON/OFF tests */
        printf ("\nSIGNALON/OFF test\n");
        printf ("\tSetting Signal Handling Mode (SIGNALON).\n");
        if( ioctl( fildes, SIGNALON, 0 ) ) pioerr(__LINE__);
        printf ("\tRestoring to Default Blocking Mode (SIGNALOFF).\n");
        if( ioctl( fildes, SIGNALOFF, 0 ) ) pioerr(__LINE__);
        printf ("SIGNALON/OFF test completed\n");

/****************************************************************************/
        /* PACKETON/OFF tests */
        printf ("\nPACKETON/OFF test\n");
        printf ("\tSetting Packet Mode (PACKETON).\n");
        if( ioctl( fildes, PACKETON, 0 ) ) pioerr(__LINE__);
        printf ("\tRestoring to Default Non-Packet Mode (PACKETOFF).\n");
        if( ioctl( fildes, PACKETOFF, 0 ) ) pioerr(__LINE__);
        printf ("PACKETON/OFF test completed\n");

        printf ("\nAll Ioctl(2) Call Tests Completed.\n");

} /* end of do_ioctl_test routine */
/****************************************************************************/
void
do_loadtime_test()
{
        int bytes_written;
        time_t timeval;

        printf ("\nLoad Major Time test (using ioctl calls)\n");
        /* set mode 1 (Free running mode) */
        send_packet ("A1");
        sleep(3);

        /* clear 1PPS interrupt status bit */
        if( ioctl( fildes, WINTSTAT, INTSTAT_1PPS ) ) pioerr(__LINE__);

        printf ("\tWaiting for the 1PPS status bit...\n");

        regvalue = 0;
        do
        {
                if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
        }
        while (!(regvalue & INTSTAT_1PPS));

        printf ("\tGot the 1PPS status bit.\n");

        /* Clear the 1PPS status bit */
        if( ioctl( fildes, WINTSTAT, INTSTAT_1PPS ) ) pioerr(__LINE__);

        printf ("\tLoading major time as 123 11:22:33\n");

        /* Send 123 11:22:33 as major time */
        send_packet ("B123112233");
        printf ("\tVerify that display is incrementing from 11:22:33...\n");

        sleep (10);

        printf ("\n\tNow setting to the system clock's time...\n");

        /* Get the current date and time in jjjHHMMSS format */
        timeval = time(0);

        time_buf[0] = 'B';
        num_chars = strftime (&time_buf[1], sizeof(time_buf) - 1, "%j%H%M%S", localtime(&timeval));
        if (num_chars <= 0) {
                perror ("do_loadtime_test: strftime");
                exit (1);
        }

        time_buf[num_chars + 1] = '\0';

        if (verbose)
                printf ("\tPC time buffer: %s\n", time_buf);

        printf ("\tLoading major time as %s\n", &time_buf[1]);

        /* clear 1PPS interrupt status bit */
        if( ioctl( fildes, WINTSTAT, INTSTAT_1PPS ) ) pioerr(__LINE__);

        printf ("\tWaiting for the 1PPS status bit...\n");

        regvalue = 0;
        do
        {
                if( ioctl( fildes, RINTSTAT, &regvalue ) ) pioerr(__LINE__);
        }
        while (!(regvalue & INTSTAT_1PPS));

        printf ("\tGot the 1PPS status bit.\n");

        /* Clear the 1PPS status bit */
        if( ioctl( fildes, WINTSTAT, INTSTAT_1PPS ) ) pioerr(__LINE__);

        send_packet (time_buf);

        printf ("\tVerify that display is incrementing from %.1s%.1s:%.1s%.1s:%.1s%.1s...\n",
                &time_buf[4], &time_buf[5], &time_buf[6], &time_buf[7],
                &time_buf[8], &time_buf[9]);

        sleep (10);

        printf ("Load Major Time test completed\n");
} /* end of do_loadtime_test routine */
/*****************************************************************************/
/*
 *      do_GPS_test - this test sets up a loop to read the GPS/TANS packets
 *                      being broadcast continuously, then injects a "Request Current
 *                      Time" packet (packet ID 21) to the GPS, then filters the
 *                      sugsequent packets until the response is received, namely a
 *                      GPS Time packet (packet ID 41).
 *
 *                      NOTE: uses globals time_requested and time_received to communicate
 *                              with rpsignalhandler
/*****************************************************************************/
void
do_GPS_test()
{
        int packets_received;

        printf ("GPS Read/Write test\n");
```

```
    packets_received = 0;
    time_requested = NO;
    time_received = NO;
    setup_done = NO;

    /* set GPS mode 6 */
    send_packet ("A6");

    /* Enable signal handling. */
    if( ioctl( fildes, SIGNALON, 0 ) ) pioerr(__LINE__);

    /* Make sure we are in packet mode for the read/write calls */
    if( ioctl( fildes, PACKETON, 0 ) ) pioerr(__LINE__); /* Enable packet mode. */

    if (verbose)
    {
        printf ("Request Current Time Packet to be sent: ");
        disp_packet (packet_21);
    }

    do
    {

        /* Receive a packet */
        read_packet();

        /* Let the read side get going before writing... */
        if (packets_received++ > 5)
        {

            /* Now inject the Request Current Time packet */
            /* unless we have already done so                                  */
            if (time_requested == NO)
            {
                time_requested = YES;
                printf ("\tRequesting Current Time from GPS...\n");
                /* Write the packet to the INFIFO */
                write_packet (packet_21, sizeof(packet_21));
            }

        } /* end of "time to send Request Current Time packet */

        /* Signal handler (rpsignalhandler) will look for GPS Time packet */
        /* and tell us via (global) time_received. Keep looping until we  */
        /* find it. */

    } /* end of do loop */

    while (time_received == NO);

    if (verbose)
        printf ("Disabling DPA interrupts with WMASK...\n");
    if( ioctl( fildes, WMASK, 0 ) ) pioerr(__LINE__);

    printf ("\tGPS Time packet received: ");
    disp_packet (rpacket);

    printf ("GPS Read/Write test completed\n");

    /* set mode 1 */
    send_packet ("A1");
    sleep(3);

} /* end of do_GPS_test routine */
/*****************************************************************************/
void
do_int_tests(test_select, mode)
int test_select;
int mode;
{
        int mask, cr0, timer1, timer2, ext_int_type;
        int all_int_tests=NO;

        if (test_select == 6)
                printf( "\n\nInterrupt tests...%s mode\n\n",
                        (mode == BLOCK) ? "Blocking" : "Signaling");
        else
                printf( "\n\nInterrupt test...%s mode\n\n",
                        (mode == BLOCK) ? "Blocking" : "Signaling");

next_test:

        /* Clear CR0 */
        cr0 = 0;
  /* set mode 1 */
        /*send_packet ("A1");*/

        switch (test_select)
        {
                case 0: /* No tests */
                        if (verbose)
                                printf ("No interrupt tests selected\n");
                        return;

                case 1: /* External */
external:
                        printf ("External Interrupt Test\n");
                        if (interact_mode == YES)
                        {
                                printf ("Real (0), HB (1), or Strobe(2) external interrupt?\n");
                                fscanf (stdin, "%d", &ext_int_type);
                                switch (ext_int_type)
                                {
                                        case 0: /* No test fixture */
                                                printf ("Real External Interrupt Mode\n");
                                                break;
                                        case 1: /* Heartbeat */
                                                /* NOTE: interrupt will be via loopback fixture */
                                                printf ("Heartbeat Loopback Test Interrupt Mode\n");
                                                timer1 = 1000;
                                                timer2 = 1000;
                                                per_second = 10000000 / (timer1 * timer2);
                                                printf ("\tHeartbeat will be %ld times per second\n", per_second);
                                                if (verbose)
                                                        printf ("\tSetting heartbeat to %d x %d\n", timer1, timer2);
                                                set_heartbeat(timer1,timer2);
                                                cr0 = HBEN;
                                                break;
                                        case 2: /* Strobe */
                                                printf ("Strobe Loopback Test Interrupt Mode\n");
                                                /* set strobe as about 3 seconds beyond current time...*/
                                                set_strobe ();
                                                /* major/minor mode, which will happen exactly once */
                                                cr0 = STREN | STRMODE_MAJ;
                                                break;
                                }
                        }
                        get_time();
                        mask =  MASK_EXT;
                        cr0 |=  LOCKEN | EVENTEN;
                        printf ("\tWaiting for external interrupt...\n");
                        break;

                case 6: /* all tests */
                        all_int_tests = YES;
                        if (ALL_tests == YES)
		{ /* "-A" option */
			/* Include external interrupt test */
			test_select = 1;
			goto external;
		}
		else { /* "-a" option */
			/* skip external interrupt test, */
			/* start with heartbeat test */
			test_select = 2;
			/* fall into test 2 */
		}

	case 2: /* Heartbeat */
		printf ("Heartbeat Interrupt Test\n");
		get_time();
		if (interact_mode == YES)
		{
			interact_mode = NO; /* Turn off in case of looping */
			printf ("Please enter timer 1 value: ");
			fscanf (stdin, "%d", &timer1);
			printf ("Please enter timer 2 value: ");
			fscanf (stdin, "%d", &timer2);
		}
		else
		{
			/* Default timers to be ten per second */
			timer1 = 1000;
			timer2 = 1000;
		}
		per_second = 10000000 / (timer1 * timer2);
		printf ("\tHeartbeat will be %ld times per second\n", per_second);
		if (per_second > 4500)
		{
			printf ("Heartbeat too fast...");
			printf ("Signal handler cannot keep up!\n");
			printf ("Aborting test.\n");
			return;
		}
		if (verbose)
			printf ("\tSetting heartbeat to %d x %d\n", timer1, timer2);
		set_heartbeat(timer1,timer2);
		mask =  MASK_HEART;
		cr0 = LOCKEN | HBEN;
		break;

	case 3: /* Major/minor Strobe */
		printf ("Major/minor Strobe Interrupt Test\n");
		sleep(1);
		/* set strobe as about 3 seconds beyond current time...*/
		set_strobe ();
		mask =  MASK_STROBE;
		/* major/minor mode, which will happen exactly once */
		cr0 = LOCKEN | STREN | STRMODE_MAJ;
		break;

	case 4: /* Minor-only Strobe */
		printf ("Minor-only Strobe Interrupt Test\n");
		/* set strobe as about 3 seconds beyond current time...*/
		set_strobe ();
		mask =  MASK_STROBE;
		/* minor-only, which will happen once a second */
		cr0 = LOCKEN | STREN | STRMODE_MIN;
		break;

	case 5: /* 1PPS */
		printf ("1PPS Interrupt Test\n");
		mask =  MASK_1PPS;
		cr0 = LOCKEN;
```

```
                        break;

                default: /* none of the above */
                        fprintf (stderr, "Invalid test_select: %d\n", test_select);
                        return;

        } /* end of test_select switch */

        if ( verbose && (fast_terminal == YES) )
        {
                printf( "\n    About to make the WMASK interrupt enable call... \n" );
        }

        if (mode == SIGNAL)
        {

                switch (test_select)
                {

                case 0: /* No tests */
                        if (verbose)
                                printf ("No interrupt tests selected\n");
                        return;

                case 1: /* External */
                        if (ext_int_type == 1)
                        {
                                if (verbose)
                                        printf ("Acting like Heartbeat test\n");
                                goto heartbeat1;
                        }
                        /* Otherwise, act like other tests */
                case 3: /* Major/minor Strobe */
                case 4: /* Minor-only Strobe */
                case 5: /* 1PPS */
                        prints_done  = 0;       /* Clear the print counter.                     */

#ifndef BSDUNIX
                        myhandler.sv_handler = signalhandler;
                        myhandler.sv_mask = sigmask(SIGUSR1);
                        myhandler.sv_onstack = 0;
                        sigvec( SIGUSR1, myhandler, NULL);

/*                      sigset( SIGUSR1, signalhandler );*/ /* Link the signal to the handler.*/
#else
                        signal( SIGUSR1, signalhandler ); /* Link the signal to the handler.*/
#endif
                        break;

                case 2: /* Heartbeat */
heartbeat1:
                        signalcount = 0;        /* Clear the signal counter.            */
                        printcount  = 0;        /* Ditto the print counter.                     */
                        prints_done  = 0;       /* Ditto the print counter.                     */

#ifndef BSDUNIX
                        myhandlerhb.sv_handler = hbsighandler;
                        myhandler.sv_mask = sigmask(SIGUSR1);
                        myhandler.sv_onstack = 0;

                        sigvec( SIGUSR1, myhandlerhb, NULL);

/*                      sigset( SIGUSR1, hbsighandler );*/ /* Link signal handler to signal.        */
#else
                        signal( SIGUSR1, hbsighandler ); /* Link signal handler to signal.  */
#endif
                        break;
```

```
		case 6: /* "all of the above" not valid here */
		default:
			fprintf (stderr, "Invalid test_select: %d\n", test_select);
			return;

		} /* end of test_select switch */

		/* Enable signal handling. */
		if( ioctl( fildes, SIGNALON, 0 ) ) pioerr(__LINE__);
	} /* end of "if (mode == SIGNAL)" */

	/* Wait for the interrupt.*/

	/* Enable the interrupt */
	if( ioctl( fildes, WCR0, cr0 ) ) perror(__LINE__);
block_wait:

	/* Clear and enable the interrupt status */
	/* The WMASK call will clear the interrupt status */
	/* both before and after it occurs */

	if( ioctl( fildes, WMASK, mask ) ) perror(__LINE__);

	if (mode == SIGNAL)
	{
signal_pause:
		if (verbose)
			printf( "\n    Non-blocked waiting...\n" );

#ifndef BSDUNIX
		sigpause( SIGUSR1 );		/* Wait for signal to come in (UNIX SV).*/
#else
		pause( SIGUSR1 );			/* Wait for signal to come in (BSD UNIX).*/
#endif

		if( ioctl( fildes, RUNLOCK, &regvalue ) ) pioerr(__LINE__);	/* Read the UNLOCK reg. */

		/* loop 5 times for repeating tests */
		switch (test_select)
		{
			case 1: /* External Interrupt */
				if (ext_int_type == 1)
				{
					if (verbose)
						printf ("Acting like Heartbeat test\n");
					goto heartbeat2;
				}
				else
				{
					/* Otherwise, just get the event registers */
					get_event();
					break;
				}
			case 3: /* Major/minor strobe */
				get_time();
				break;
			case 2: /* Heartbeat */
heartbeat2:
				if (++prints_done < 5)
				{
					goto signal_pause;
				}
				break;
			case 4: /* Minor-only Strobe */
			case 5: /* 1PPS */
				get_time();
				if (++prints_done < 5)
				{
```

```
                                        goto signal_pause;
                                }
                                break;
                        default:
                                /* No looping in other tests */
                                break;
                } /* end of switch on test_select */

                /* Disable signal handling. */
                if( ioctl( fildes, SIGNALOFF, 0 ) ) pioerr(__LINE__);

        } /* end of "if (mode == SIGNAL)" */
        else
        { /* blocking mode */


                if ( (test_select == 2) /* Internal Heartbeat */
                        || ((test_select == 1) && (ext_int_type == 1)) ) /* Loopback Heartbeat */
                {
                        printcount += 1;
                        if (verbose)
                        {
                                printf ("%d\n", printcount);

                        }

                        if( printcount == per_second )
                        {

                                get_time();

                                printcount = 0;                         /* Clear the print counter.     */
                                prints_done += 1;               /* Add one to prints done               */

                                if (verbose)
                                        printf ("\tPrints_done = %d\n", prints_done);
                        }
                } /* end of heartbeat section */
                else
                {
                        /* Read the UNLOCK reg. */
                        if( ioctl( fildes, RUNLOCK, &regvalue ) ) pioerr(__LINE__);
                }

                /* loop 5 times for repeating tests */
                switch (test_select)
                {
                        case 1: /* External Interrupt */
                                if (ext_int_type == 1)
                                {
                                        if (verbose)
                                                printf ("Acting like Heartbeat test\n");
                                        goto heartbeat3;
                                }
                                else
                                {
                                        /* Otherwise, just get the event registers */
                                        get_event();
                                        break;
                                }
                        case 3: /* Major/minor strobe */
                                get_time();
                                break;
                        case 2: /* Heartbeat */
heartbeat3:
                                if (prints_done < 5)
                                {
                                        goto block_wait;
                                }
                                if (verbose)
                                        printf( "\n    Time after all prints done is: \n" );
                                get_time();
                                break;
                        case 4: /* Minor-only Strobe */
                        case 5: /* 1PPS */
                                get_time();
                                if (++prints_done < 5)
                                {
                                        goto block_wait;
                                }
                                break;
                        default:
                                /* No looping in other tests */
                                break;
                } /* end of switch on test_select */


        } /* end of blocking mode section */
        switch (test_select)
        {
                case 0: /* No tests */
                        printf ("No interrupt tests selected\n");

                case 1: /* External */
                        printf ("End of External Interrupt Test\n");
                        break;

                case 2: /* Heartbeat */
                        printf ("End of Heartbeat Interrupt Test\n");
                        break;

                case 3: /* Major/minor Strobe */
                        printf ("End of Major/minor Strobe Interrupt Test\n");
                        break;

                case 4: /* Minor-only Strobe */
                        printf ("End of Minor-only Strobe Interrupt Test\n");
                        break;

                case 5: /* 1PPS */
                        printf ("End of 1PPS Interrupt Test\n");
                        break;

                case 6:
                default: /* none of the above */
                        fprintf (stderr, "Invalid test_select: %d\n", test_select);
                        return;

        } /* end of test_select switch */

        if (verbose)
                printf ("Disabling all interrupts with WMASK...\n");
        if( ioctl( fildes, WMASK, 0 ) ) pioerr(__LINE__);

        /* Clear the CR0 register */
        if( ioctl( fildes, WCR0, 0 ) ) pioerr(__LINE__);

        /* Keep looping if multiple tests were selected */
        if (all_int_tests == YES)
        {
                test_select++;
                /* test #5 is the last one... */
                if (test_select < 6)
                {
                        goto next_test;
                }
```

```
	}

	if (all_int_tests == YES)
		printf( "\nEnd of Interrupt tests...%s mode\n",
			(mode == BLOCK) ? "Blocking" : "Signaling");
	else
		printf( "\nEnd of Interrupt test...%s mode\n",
			(mode == BLOCK) ? "Blocking" : "Signaling");

	return;
} /* end of do_int_tests routine */

/***************************************************************************/
int
wtsignalhandler()
{
	int bytes_written;

	if (verbose)
		printf ("Got 1PPS signal\n");

	/* Write out packet */
	/* Note: this should not block */
	bytes_written = write (fildes, time_buf, num_chars+1);
	if (bytes_written <= 0)
	{
		perror ("write in wtsighandler");
		exit (2);
	}

	if (verbose)
		printf ("Wrote Date Packet\n");

	if (verbose)
		printf ("Disabling DPA interrupts with WMASK...\n");
	if( ioctl( fildes, WMASK, 0 ) ) pioerr(__LINE__);

} /* end of wtsignalhandler routine */
/***************************************************************************/
void
write_time (mode)
int mode;
{
	int bytes_written;
	time_t timeval;

	/* Get the current date and time in jjjHHMMSS format */
	timeval = time(0);

	num_chars = strftime (time_buf, sizeof(time_buf), "%j%H%M%S", gmtime(&timeval));
	if (num_chars <= 0) {
		perror ("write_time: strftime");
		exit (1);
	}

	if (verbose)
		printf ("\tPC time buffer: %s\n", time_buf);

	printf ("\tLoading major time as %s\n", time_buf);

	/* Write time to btfp in proper mode */
	if (mode == SIGNALOFF)
	{
		/* Blocking mode */
		if( ioctl( fildes, SIGNALOFF, 0 ) ) pioerr(__LINE__);
		bytes_written = write( fildes, time_buf, num_chars+1);
		printf("bytes_written = %d\n",bytes_written);
		if (bytes_written <= 0)
```

```
                {
                        perror ("write in write_time");
                        exit (2);
                }
        }
        else
        {
                /* Non-blocking mode */
                if( ioctl( fildes, SIGNALON, 0 ) ) pioerr(__LINE__);
#ifndef BSDUNIX
                        myhandlerwt.sv_handler = wtsignalhandler;
                        myhandler.sv_mask = sigmask(SIGUSR1);
                        myhandler.sv_onstack = 0;
                        sigvec( SIGUSR1, myhandlerwt, NULL);

        /*      sigset( SIGUSR1, wtsignalhandler );*/ /* Link the signal to the handler.*/
#else
                signal( SIGUSR1, wtsignalhandler ); /* Link the signal to the handler.*/
#endif
                /* Clear, then enable 1PPS interrupt */
                if( ioctl( fildes, WMASK, MASK_1PPS ) ) pioerr(__LINE__);
                if (verbose)
                        printf ("Waiting for 1PPS signal...\n");
#ifndef BSDUNIX
                sigpause( SIGUSR1 );            /* Wait for signal to come in (UNIX SV).*/
#else
                pause( SIGUSR1 );                       /* Wait for signal to come in (BSD UNIX).*/
#endif
                if (verbose)
                        printf ("Resumed after getting 1PPS signal...\n");
        }
} /* end of write_time routine */

/****************************************************************************/
int
hbsighandler()                                          /* Handle a heartbeat signal.           */
{
        int dummy;

#ifndef BSDUNIX
                        myhandlerhb.sv_handler = hbsighandler;
                        myhandler.sv_mask = sigmask(SIGUSR1);
                        myhandler.sv_onstack = 0;
                        sigvec( SIGUSR1, myhandlerhb, NULL);

/*      sigset( SIGUSR1, hbsighandler );*/ /* Link signal handler to signal.            */
#else
        signal( SIGUSR1, hbsighandler ); /* Link signal handler to signal.  */
#endif

        signalcount += 1;                               /* Add one to the counters.             */
        printcount += 1;

        if (verbose)
        {
                printf ("printcount=%d\n", printcount);
                printf ("per_second=%d\n", per_second);
        }

        if( printcount == per_second )
        {
                if ( (display_mode == 2) || (display_mode == 3) )
                {
                printf( "       Heartbeat test signal handler has counted %d signals.\n",
                                                        signalcount );
                }

                /* Release capture lockout. */
                if( ioctl( fildes, RUNLOCK, &dummy  ) ) pioerr(__LINE__);

                if ( (display_mode == 1) || (display_mode == 3) )
                {
                        get_time();
                }

                printcount = 0;                                 /* Clear the print counter.                     */
                prints_done += 1;                               /* Add one to prints done                       */

        } /* end of "if( printcount == per_second )" */

} /* end of hbsighandler */

/****************************************************************************/
int
signalhandler()                                                 /* Activated when SIGUSR1 is received.      */
{
        if (verbose)
                printf( "    signalhandler(): Signal received!\n" );

}

/****************************************************************************/
get_time()
{
        int dummy, i;

        if( ioctl( fildes, READTIME, &ts ) ) pioerr(__LINE__); /* Read current time.   */

        if (fast_terminal == YES)
        {
                printf ("      READTIME words: ");
                for (i=0; i<=4; i++ )
                {
                        printf( " 0x%04x ", ts.btfp_time[ i ] );
                }
                printf ("\n");
        }

        return;
} /* end of get_time routine */
/****************************************************************************/
get_event()
{
        int dummy, i;

        if( ioctl( fildes, READEVENT, &event ) ) pioerr(__LINE__);

        if (fast_terminal == YES)
        {
                printf ("      READEVENT words: ");
                for (i=0; i<=4; i++ )
                {
                        printf( " 0x%04x ", event.btfp_event[ i ] );
                }
                printf ("\n");
        }

        /* Release capture lockout. */
        if( ioctl( fildes, RUNLOCK, &dummy  ) ) pioerr(__LINE__);

        return;
} /* end of get_event routine */
/****************************************************************************/
pioerr(line_no)                                                                         /* Log ioctl(2) error to terminal.*/
int line_no;
{
    char *s;
    int sys_nerr;
    int errno;

              printf( "     ERROR RETURNED ON IOCTL CALL (%d)!\n          " ,
                                    line_no);
        perror( s );
}
/****************************************************************************/
printregs()                                                              /* Print out all of the registers.*/
{
        printf( "       ID =\t\t0x%04x.\n\n", ds.btfp_id );
        printf( "       DEVICE=\t0x%04x.\n\n", ds.btfp_device );
        printf( "       STATUS=\t0x%04x.\n\n", ds.btfp_status );

        printf( "       TIMEREQ=\t0x%04x.\n\n", ds.btfp_timereq );
        printf( "       TIME0=\t\t0x%04x.\n", ds.btfp_time[ 0 ] );
        printf( "       TIME1=\t\t0x%04x.\n", ds.btfp_time[ 1 ] );
        printf( "       TIME2=\t\t0x%04x.\n", ds.btfp_time[ 2 ] );
        printf( "       TIME3=\t\t0x%04x.\n\n", ds.btfp_time[ 3 ] );
        printf( "       TIME4=\t\t0x%04x.\n\n", ds.btfp_time[ 4 ] );

        printf( "       EVENT0=\t0x%04x.\n", ds.btfp_event[ 0 ] );
        printf( "       EVENT1=\t0x%04x.\n", ds.btfp_event[ 1 ] );
        printf( "       EVENT2=\t0x%04x.\n", ds.btfp_event[ 2 ] );
        printf( "       EVENT3=\t0x%04x.\n\n", ds.btfp_event[ 3 ] );
        printf( "       EVENT4=\t0x%04x.\n\n", ds.btfp_event[ 4 ] );

        printf( "       UNLOCK=\t0x%04x.\n\n", ds.btfp_unlock );
        printf( "       ACK=\t\t0x%04x.\n\n", ds.btfp_ack );

        printf( "       CR0=\t\t0x%04x.\n", ds.btfp_cr0 );

        printf( "\tSkipping IN/OUT FIFOs.\n");

        printf( "       MASK=\t\t0x%04x.\n", ds.btfp_mask );
        printf( "       INTSTAT=\t0x%04x.\n", ds.btfp_intstat);
        printf( "       VECTOR=\t0x%04x.\n", ds.btfp_vector );
        printf( "       LEVEL=\t\t0x%04x.\n", ds.btfp_level );

} /* end of printregs routine */
/****************************************************************************/
void
write_packet (byte_ptr, num_bytes)
unsigned char *byte_ptr;
int num_bytes;
{
        int bytes_written;

        /* Note: this routine uses the PACKETON write(2) method of */
        /* sending a data packet */
        /* The send_packet() routine uses the byte-by-byte method */

        if( ioctl( fildes, PACKETON, 0 ) ) pioerr(__LINE__); /* Enable packet mode. */
        bytes_written = write (fildes, byte_ptr, num_bytes);
        if (bytes_written < num_bytes)
        {
                fprintf (stderr, "Error from write in write_packet: %d\n",
                                bytes_written);
                exit (2);
        }
        if (verbose)
                printf ("Returned from packeton write with %d bytes!\n",
                                bytes_written);

} /* end of write_packet routine */
/****************************************************************************/
void
```

```
send_packet (byte_ptr)
unsigned char *byte_ptr;
{
        register int loop_count;

        /* Note: this routine uses the byte-by-byte method of sending a packet */
        /* The write_packet() routine uses the PACKETON write(2) method */

        /* clear the INFIFO ACK bit */
        regvalue = ACK_INFIFO;
        if( ioctl( fildes, WACK, regvalue ) ) pioerr(__LINE__);

        /* write the SOH */
        regvalue = SOH;
        if( ioctl( fildes, WINFIFO, regvalue ) ) pioerr(__LINE__);

        /* write the null-terminated packet */
        while (*byte_ptr)
        {
                regvalue = *byte_ptr;
                if( ioctl( fildes, WINFIFO, regvalue ) ) pioerr(__LINE__);
                byte_ptr++;
        }

        /* write the ETB */
        regvalue = ETB;
        if( ioctl( fildes, WINFIFO, regvalue ) ) pioerr(__LINE__);

        /* command the TFP to take action on this packet */
        regvalue = ACK_INACT;
        if( ioctl( fildes, WACK, regvalue ) ) pioerr(__LINE__);

        /* wait for FIFO to ACK back */

        loop_count = 0;
        regvalue = 0;
        while (loop_count <= MAX_SPIN)
        {
                if( ioctl( fildes, RACK, &regvalue ) ) pioerr(__LINE__);
                if ( regvalue & ACK_INFIFO ) /* Break if action taken */
                {
                        break;
                }
                else
                {
                        loop_count++;
                }
        }

        if (verbose)
                printf ("Looped %d times\n", loop_count);

        /* clear the INFIFO ACK bit */
        if( ioctl( fildes, WACK, ACK_INFIFO ) ) pioerr(__LINE__);

        if (loop_count >= MAX_SPIN)
        {
                printf ("TIMEOUT (%d) waiting for INFIFO action\n", MAX_SPIN);
        }

} /* end of send_packet routine */
/*****************************************************************************/
void
read_packet()
{
        /* Enable DPA interrupts, then leave them alone */

        if (setup_done == NO)
        {
        setup_done = YES;

#ifndef BSDUNIX
                        myhandlerrp.sv_handler = rpsignalhandler;
                        myhandlerrp.sv_mask = sigmask(SIGUSR1);
                        myhandlerrp.sv_onstack = 0;
                        sigvec( SIGUSR1, myhandlerrp, NULL);

        /*sigset( SIGUSR1, rpsignalhandler );*/ /* Link the signal to the handler.*/
#else
        signal( SIGUSR1, rpsignalhandler ); /* Link the signal to the handler.*/
#endif
        /* Clear and enable the Data Packet Available interrupt */
        /* Note: this should not block */
        if (verbose)
                printf ("Entering non-blocking WMASK...\n");
        if( ioctl( fildes, WMASK, MASK_DPA ) ) pioerr(__LINE__);
        if (verbose)
                printf ("Returned from non-blocking WMASK...\n");

        } /* end of initial setup section */

        /* Now wait for the packet */
#ifndef BSDUNIX
        sigpause( SIGUSR1 );                    /* Wait for signal to come in (UNIX SV).*/
#else
        pause( SIGUSR1 );                               /* Wait for signal to come in (BSD UNIX).*/
#endif
} /* end of read_packet routine */
/*****************************************************************************/
int
rpsignalhandler()
{
        /* bytes_read is a global, so disp_packet can decode properly */

        if (verbose)
                printf ("Got DPA signal\n");

        /* Read packet in */
        /* Note: this should not block */
        bytes_read = read (fildes, rpacket, sizeof(rpacket));
        if (bytes_read <= 0)
        {
                fprintf (stderr, "Error from read in rpsignalhandler: %d\n",
                                bytes_read);
                exit (2);
        }

        if (time_requested)
        {
                /* Scan for GPS Time packet */
                /* Coming from GPS, format is: DLE ID .... DLE ETX */
                if (rpacket[1] == (unsigned char) 0x41)
                {
                        if (verbose)
                                printf ("GPS Time packet received!\n");
                        time_received = YES;
                }
                else
                {
                        if (verbose)
                                printf ("%.2x \n", rpacket[1]);
                }
        }

        if (verbose)
                printf ("Read Data Packet!\n");
```

```
} /* end of rpsignalhandler routine */
/***************************************************************************/
/*
 *          NOTE: build_packet is not currently used. It is provided as a template for
 *  an application which may want to read in packets to be sent to the GPS
 *  Sateliite Receiver from stdin. The do_GPS_test routine would have to
 *  be modified to use this read-in packet, and the rpsignalhandler routine
 *  would have to know which packet to look for.
 */
/***************************************************************************/
void
build_packet(byte_ptr)
unsigned char *byte_ptr;
{
        int index, length, c;
        unsigned char pack_buff[2*MAXPACKET], *item_ptr, byte_value;
        long hex_value;

        /* Read in character stream from stdin */
        /* Expect space or newline-separated characters of three types: */
        /*          single characters: ASCII characters */
        /*   double characters: hex values */
        /*   triple characters: mnemonic values, such as SOH */

        index = 0;
        while ( (c = getc (stdin)) != EOF )
        {
                pack_buff[index++] = (unsigned char) c;
        }

        index = 0;

        /* Expect space-separated stuff, maybe on multiple lines */
        item_ptr = (unsigned char *) strtok (pack_buff, delimiters);
        do
        {

                length = strlen (item_ptr);
                if (verbose)
                        printf ("%s: Length is %d\n", item_ptr, length);

                switch (length)
                {
                        case 1: /* Single Letters */
                                /* just treat it as an ASCII character */
                                if ( isascii (*item_ptr) )
                                {
                                        byte_value = *item_ptr;
                                }
                                else
                                {
                                        printf ("Not an ASCII character: 0x%x\n", *item_ptr);
                                        exit (2);
                                }
                                break;

                        case 2: /* Hex Values */
                                if ( (isxdigit (*item_ptr)) && (isxdigit (*(item_ptr+1))) )
                                {
                                        hex_value = strtol (item_ptr, (char **) NULL, 16);
                                        byte_value = (unsigned char) hex_value;
                                }
                                else
                                {
                                        printf ("Not hex digits: %s\n", item_ptr);
                                        exit (2);
                                }
                                break;
                        case 3: /* SOH, ETB, etc. */
                                if (strncmp (item_ptr, "SOH", 3) == 0)
                                {
                                        byte_value = 0x01;
                                }
                                else if (strncmp (item_ptr, "ETB", 3) == 0)
                                {
                                        byte_value = 0x17;
                                }
                                else if (strncmp (item_ptr, "DLE", 3) == 0)
                                {
                                        byte_value = 0x10;
                                }
                                else if (strncmp (item_ptr, "ETX", 3) == 0)
                                {
                                        byte_value = 0x03;
                                }
                                else
                                {
                                        printf ("Unknown keyword: %s\n", item_ptr);
                                        exit (1);
                                }
                                break;

                        default:
                                printf ("Illegal length (%d) of item: %s\n",
                                        length, item_ptr);
                                exit (1);
                } /* end of "switch (length)" */

                if (verbose)
                        printf ("byte_value is 0x%.2x\n", byte_value);

                *byte_ptr++ = byte_value;

                item_ptr = (unsigned char *) strtok (NULL, delimiters);

        } /* end of do loop for finding new tokens */
        while (item_ptr != (unsigned char *) NULL);

        /* Ensure that packet is NULL-terminated */
        *byte_ptr = (unsigned char) NULL;

    } /* end of build_packet routine */
    /****************************************************************************/
    void
    disp_packet(byte_ptr)
    unsigned char *byte_ptr;
    {
        int index, column;
        unsigned char byte_value;

        column = 0;
        byte_value = *byte_ptr++;
        for (index = 0; index < bytes_read; index++)
        {
                printf ("%.2x ", byte_value);

                if (++column == 16)
                {
                        printf ("\n");
                        column = 0;
                }
                byte_value = *byte_ptr++;
        } /* end of for loop to print out packet buffer */

        printf ("\n");
    } /* end of disp_packet routine */
```

```
/***************************************************************************/
void
set_heartbeat (n1, n2)
int n1, n2;
{
        unsigned char packet[12];

        packet[0] = 'F';
        packet[1] = '5'; /* synchronized mode */
        sprintf (&packet[2], "%04x", n1-1);
        sprintf (&packet[6], "%04x", n2-1);
        packet[10] = (unsigned char)NULL;
        send_packet (packet);
} /* end of set_heartbeat routine */

/***************************************************************************/
void
set_strobe ()
{

tryagain:;

        /* Reads time into global ts structure */
        get_time ();

        t = ts.btfp_time[ 2 ] & 0x00ff;                         /* Isolate the seconds field.*/
        ts.btfp_time[ 2 ] &= 0xff00;                            /* Clear the seconds field.     */

        if( t >= 0x07 && t <= 0x09 )
                ts.btfp_time[ 2 ] |= 0x0012;
        else if( t >= 0x17 && t <= 0x19 )
                ts.btfp_time[ 2 ] |= 0x0022;
        else if( t >= 0x27 && t <= 0x29 )
                ts.btfp_time[ 2 ] |= 0x0032;
        else if( t >= 0x37 && t <= 0x39 )
                ts.btfp_time[ 2 ] |= 0x0042;
        else if( t >= 0x47 && t <= 0x49 )
                ts.btfp_time[ 2 ] |= 0x0052;
        else if( t >= 0x56 )
        {
                sleep( 4 );
                goto tryagain;
        }
        else
        {
                t += 0x0003;                                    /* Just add 3 seconds to what we had.*/

                ts.btfp_time[ 2 ] |= t;
                                                                                /* That is what we'll wait for.     */
        }

        if (fast_terminal == YES)
        {
                printf( "\n     RTIME2- Time word 2 to the strobe is 0x%04x.\n",
                                                                ts.btfp_time[ 2 ] );
                printf( "\n     Now writing the later time value to the strobe ... \n" );
        }

        /* Note: the driver will disable strobe output before writing */
        /* to the Strobe registers */
        if( ioctl( fildes, WRITESTROBE, &ts.btfp_time[1] ) ) perror(__LINE__); /* Write to strobe.     */

} /* end of set_strobe routine */
/***************************************************************************/
void
usage(name)
char    *name;
{
        printf ("usage: %s [-aAvSLRTEIDrilpbsd]\n", name);

        printf ("\ta : Run all tests, excluding tests g, b1, and s1\n");
        printf ("\tA : Run ALL tests\n");-
        printf ("\t\tNOTE: external interrupt source and GPS Satellite Receiver\n");
        printf ("\t\t      must be connected and operational\n");
        printf ("\tv : verbose mode (prints diagnostic displays to stdout)\n");
        printf ("\tS : Slow terminal (defaults to fast terminal)\n");
        printf ("\tL : Loop through all selected tests (default: no loop)\n");
        printf ("\tR : READALLREGS test. Reads all bc63{57} registers.\n");
        printf ("\tT : READTIME test.\n");
        printf ("\tE : READEVENT test.\n");
        printf ("\tI : make heartbeat/external-interrupt tests interactive\n");
        printf ("\tD : enable driver debugging\n");
        printf ("\tr : read(2)/write(2) test.\n");
        printf ("\ti : enable most ioctl(2) tests (except for interrupts)\n");
        printf ("\tl : load major time test. Uses current time from system clock.\n");
        printf ("\tg : select GPS test. Writes to INFIFO, reads from OUTFIFO.\n");
        printf ("\t\tNOTE: GPS Satellite Receiver must be connected and operational.\n");
        printf ("\tb : select block-until-interrupt test\n");
        printf ("\t\t0 : no blocking tests (DEFAULT)\n");
        printf ("\t\t1 : external interrupt test\n");
        printf ("\t\t2 : heartbeat test\n");
        printf ("\t\t3 : strobe test (major/minor)\n");
        printf ("\t\t4 : strobe test (minor-only)\n");
        printf ("\t\t5 : 1PPS test\n");
        printf ("\t\t6 : all of the above blocking tests\n");
        printf ("\ts : select signal handling test\n");
        printf ("\t\t0 : no signal handling tests (DEFAULT)\n");
        printf ("\t\t1 : external interrupt test\n");
        printf ("\t\t2 : heartbeat test\n");
        printf ("\t\t3 : strobe test (major/minor)\n");
        printf ("\t\t4 : strobe test (minor-only)\n");
        printf ("\t\t5 : 1PPS test\n");
        printf ("\t\t6 : all of the above signal handling tests\n");
        printf ("\td : select display mode when heartbeat signal received\n");
        printf ("\t\t0 : no display\n");
        printf ("\t\t1 : display time of heartbeats (DEFAULT)\n");
        printf ("\t\t2 : display count of heartbeats\n");
        printf ("\t\t3 : display time and count of heartbeats\n");

        return;
} /* end of usage routine */
```

CHAPTER 6
DRIVER/USER HEADER FILE

6.0 Listing

```
/****************************************************************************/
/*
            The include file for applications calling the

                        bc635VME/bc650VXI HP-RT device driver.

            The device name is 'btfp' for 'BANCOMM time and frequency processor'.

            Copyright (C) 1991, BANCOMM, San Jose, CA.  All rights reserved.

            Author:     Greg Dowd
*/
/****************************************************************************/
/*#include <sys/ioconfig.h>*/                          /* Required for ioctl cmd definitions.*/
#include <sys/ioctl.h>

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

struct btfp_device                                     /* bc635 module register definitions.       */
                                                                               /* Note that unsigned short must be used!*/
{
            unsigned short          btfp_id;                /* ID register.                                    */

            unsigned short          btfp_device;            /* Device type.                          */

            unsigned short          btfp_status;            /* Status register.                      */
#define btfp_control   btfp_status /* Control register.                  */

            unsigned short          btfp_gap1[ 2 ];         /* Unused address space.            */

            unsigned short          btfp_timereq;           /* Time Request.                         */

            unsigned short          btfp_time [5];          /* Requested Time registers.     */
                                                                               /* XX, XX, Stat, DH */
                                                                               /* DT, DU, HT, HU */
                                                                               /* MT, MU, ST, SU */
                                                                               /* MSH, MST, MSU, USH */
                                                                               /* UST, USU, NSH, XX */
            unsigned short          btfp_event [5];         /* Requested Event registers.    */
                                                                               /* XX, XX, Stat, DH */
                                                                               /* MT, MU, ST, SU */
                                                                               /* MSH, MST, MSU, USH */
                                                                               /* UST, USU, NSH, XX */

#define btfp_strobe1   btfp_event[1] /* Strobe part 1 */
                                                                               /* XX, XX, HT, HU */
#define btfp_strobe2   btfp_event[2] /* Strobe part 2 */
                                                                               /* MT, MU, ST, SU */
#define btfp_strobe3   btfp_event[3] /* Strobe part 3 */
                                                                               /* MSH, MST, MSU, XX */

            unsigned short          btfp_unlock;            /* Release Capture Lockout       */

            unsigned short          btfp_ack;               /* Data Acknowledge              */

            unsigned short          btfp_cr0;               /* Control Register 0            */

            unsigned char           btfp_fifo[2];                   /* I/O FIFO Data                 */
#define btfp_outfifo   btfp_fifo[1]            /* Output FIFO (from module) */
#define btfp_infifo            btfp_fifo[1]            /* Input FIFO (to module) */

            unsigned short          btfp_mask;                      /* Interrupt Mask                */
```

```
        unsigned short      btfp_intstat;               /* Interrupt Status             */

        unsigned short      btfp_vector;                /* Interrupt Vector             */

        unsigned short      btfp_level;                 /* Interrupt Level              */

        unsigned short      btfp_gap2[ 6 ];             /* Unused address space.*/

}; /* end of btfp_device structure */

/****************************************************************************/
struct btfp_time                                                /* Structure for reading 5 time words       */
                                                                            /* in one ioctl(2) operation.       */
{
        unsigned short      btfp_time[ 5 ];     /* Time words 0, 1, 2, 3, and 4. (16bit)*/
};

/****************************************************************************/
struct btfp_event                                               /* Structure for reading 5 event words      */
                                                                            /* in one ioctl(2) operation.       */
{
        unsigned short      btfp_event[ 5 ];    /* Event words 0, 1, 2, 3, and 4. (16bit)*/
};
/****************************************************************************/

struct btfp_strobe                                              /* Structure for writing 3 strobe words */
                                                                            /* in one ioctl(2) operation. */
{
        unsigned short      btfp_strobe[ 3 ];   /* Strobe words 1, 2, and 3. (16bit)*/
};
/****************************************************************************/
                                /* Ioctl commands to the btfp driver.           */

/***** Simple ioctl commands *****/


#define RIDR     1                              /* Read the ID register. */
#define RDTR     2                              /* Read the Device Type register. */

#define RSR     3                               /* Read Status register. */
#define WCR     4                                       /* Control register. */

#define RTIMEREQ     5                          /* Read Time Request register. */
#define RTIME0     6                                    /* Read time word zero. */
#define RTIME1     7                                    /* Read time word one. */
#define RTIME2     8                                    /* Read time word two. */
#define RTIME3     9                                    /* Read time word three. */
#define RTIME4     10                           /* Read time word four. */

#define REVENT0     11                          /* Read event word zero. */
#define REVENT1     12                          /* Read event word one. */
#define REVENT2     13                          /* Read event word two. */
#define REVENT3     14                          /* Read event word three. */
#define REVENT4     15                          /* Read event word four. */

#define WSTROBE1     16                         /* Read strobe word one. */
#define WSTROBE2     17                         /* Read strobe word two. */
#define WSTROBE3     18                         /* Read strobe word three. */

#define RUNLOCK     19                          /* Release Capture Lockout */

#define RACK     20                                     /* Read Acknowledge */
#define WACK     21                                     /* Write Acknowledge. */

#define RCR0     22                             /* Read control register zero.              */
#define     WCR0            23                                  /* Write control register zero.             */
#define     ROUTFIFO        24                                  /* Read from OUTFIFO */
```

```
#define     WINFIFO             25                              /* Write to INFIFO */
#define RMASK       26                              /* Read Interrupt Mask. */
#define WMASK       27                              /* Write Interrupt Mask. */

#define RINTSTAT    28                      /* Read Interrupt Status. */
#define WINTSTAT    29                      /* Write Interrupt Status. */

#define RVECTOR     30                      /* Interrupt Vector. */
                                                                        /* Only driver can write to it */

#define RLEVEL      31                      /* Interrupt Level. */
                                                                        /* Only driver can write to it */

/***** Compound ioctl commands *****/

#define READTIME  32                                /* Read all 5 time words in one call.        */

#define READEVENT           33                              /* Read all 5 event words in one call.        */

#define WRITESTROBE 34                              /* Write all 3 strobe words in one call.*/

#define     READALLREGS      35                     /* Read all btfp regs.   */

#define     RPACKET          36                             /* Read packet from OUTFIFO */

#define     WPACKET          37                             /* Write packet to INFIFO */

#define SIGNALON  38                                /* turn on  signal sending from driver.*/
                                                                        /* Also turns off blocking on interrupts.*/
#define SIGNALOFF 39                                /* turn off signal sending from driver.*/
                                                                        /* Also turns on blocking on interrupts.*/
#define PACKETON  40                                /* turn on packet-mode for driver */
#define PACKETOFF41                                 /* turn off packet-mode for driver */
#define DEBUGON             42                              /* turn on debug-mode for driver */
#define DEBUGOFF  43                                /* turn off debug-mode for driver */


/****************************************************************************/
                                        /* bc635 Requested Time/Event register masks.*/

#define     TIME_REF            0x0000          /* Reference source present */
#define     EVENT_REF           0x0000          /* Reference source present */

#define     TIME_FLY            0x0010          /* Flywheeling; time code lost */
#define     EVENT_FLY           0x0010          /* Flywheeling; time code lost */

#define TIME_OSCIN              0x0000          /* Oscillator in range */
#define EVENT_OSCIN             0x0000          /* Oscillator in range */

#define TIME_OSCOUT             0x0040          /* Oscillator out of range */
#define EVENT_OSCOUT 0x0040                     /* Oscillator out of range */

/****************************************************************************/
                                        /* bc635 ACK register masks */

#define ACK_INFIFO              0x0001          /* Input FIFO ACK Bit */
#define ACK_1PPS    0x0002              /* 1 Pulse Per Second ACK Bit */
#define ACK_DPA                 0x0004          /* Output FIFO Data Packet Available Bit */
#define ACK_MORE 0x0010                 /* Output FIFO "More in OUTFIFO" Flag */
                                                /* 0: empty, 1: more bytes */
#define ACK_CLEAR               0x0010          /* Output FIFO Clear Bit */
#define ACK_INACT0x0080                 /* Input FIFO "Take Action" Bit */
/****************************************************************************/
                                        /* bc635 CR0 control register masks.*/

#define     LOCKEN              0x0001          /* Enable Capture Lockout */
#define     LOCKDIS             0x0000          /* Disable Capture Lockout */
```

```
#define HBEN                0x0002          /* Enable Periodic Time Capture */
#define HBDIS               0x0000          /* Disable Periodic Time Capture */
#define EVSENSE_FALL        0x0004          /* Event Input Active Edge Select */
                                            /* (Falling Edge)
#define EVSENSE_RISE        0x0000          /* Event Input Active Edge Select */
                                            /* (Rising Edge) */

#define EVENTEN             0x0008          /* Event Input Time Capture Enable */
#define EVENTDIS    0x0000          /* Event Input Time Capture Disable */

#define STREN               0x0010          /* Time Coincidence Strobe Output Enable */
#define STRDIS              0x0000          /* Time Coincidence Strobe Output Disable */

#define STRMODE_MIN         0x0020          /* Time Coincidence Strobe Mode Select */
                                            /* (Minor-only mode) */
#define STRMODE_MAJ         0x0000          /* Time Coincidence Strobe Mode Select */
                                            /* (Major/Minor mode) */

#define FREQSEL_10          0x0000          /* Output Clock 10 MHz */
#define FREQSEL_5  0x0040           /* Output Clock 5 MHz */
#define FREQSEL_1  0x0080           /* Output Clock 1 MHz */

/****************************************************************************/
                        /* bc635 Interrupt Mask register masks.*/

#define MASK_INT0 0x0001            /* Interrupt Source 0 */
#define MASK_EXT  0x0001            /* External Event Input */

#define MASK_INT1 0x0002            /* Interrupt Source 1 */
#define MASK_HEART          0x0002          /* Periodic Pulse Output */

#define MASK_INT2 0x0004            /* Interrupt Source 2 */
#define MASK_STROBE         0x0004          /* Time Coincidence Strobe */

#define MASK_INT3 0x0008            /* Interrupt Source 3 */
#define MASK_1PPS 0x0008            /* 1 Pulse per Second (1PPS) */

#define MASK_INT4 0x0010            /* Interrupt Source 4 */
#define MASK_DPA  0x0010            /* Data Packet Available */

/****************************************************************************/
                        /* bc635 Interrupt Status register masks.*/

#define INTSTAT_INT0        0x0001          /* Interrupt Source 0 */
#define INTSTAT_EXT         0x0001          /* External Event Input */

#define INTSTAT_INT1        0x0002          /* Interrupt Source 1 */
#define INTSTAT_HEART       0x0002          /* Periodic Pulse Output */

#define INTSTAT_INT2        0x0004          /* Interrupt Source 2 */
#define INTSTAT_STROBE      0x0004          /* Time Coincidence Strobe */

#define INTSTAT_INT3        0x0008          /* Interrupt Source 3 */
#define INTSTAT_1PPS        0x0008          /* 1 Pulse per Second (1PPS) */

#define INTSTAT_INT4        0x0010          /* Interrupt Source 4 */
#define INTSTAT_DPA                 0x0010          /* Data Packet Available */

/****************************************************************************/
                /* bc635 Interrupt Level register masks */

#define LEVEL_DIS  0x0000           /* Interrupts disabled */
#define LEVEL_IRQ1          0x0001          /* IRQ1 */
#define LEVEL_IRQ2          0x0002          /* IRQ2 */
#define LEVEL_IRQ3          0x0003          /* IRQ3 */
#define LEVEL_IRQ4          0x0004          /* IRQ4 */
#define LEVEL_IRQ5          0x0005          /* IRQ5 */
#define LEVEL_IRQ6          0x0006          /* IRQ6 */
```

```
#define LEVEL_IRQ7              0x0007                  /* IRQ7 */

/***************************************************************************/
                        /* General defines */
#define     YES         1
#define NO              0
#define SOH             0x01
#define ETB             0x17
#define DLE 0x10
#define ETX 0x03
#define MAXPACKET               512                             /* Maximum bytes in FIFO packet */
#define MAX_SPIN        1000000                         /* maximum loops to check for INFIFO action */

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

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7
DRIVER HEADER FILE

7.0 Listing

```
/* btfpdrvr.h */
/***************************************************************************/
/*
        The include file for the bc635VME/bc350VXI HP-RT v1.1x device driver.

        The device name is 'btfp' for 'BANCOMM time and frequency processor'.

        Copyright (C) 1992, BANCOMM, San Jose, CA.  All rights reserved.

        Author:    Greg Dowd
*/
/***************************************************************************/


/* This file defines the driver data that is passed to most of the driver
   entry points by the system when they are called.  The structure defined
   below is what is passed to the driver.  Note that the order of the
   elements within the structure is very specific.  Driverhead and
   invector must be first.  After that, everything is driver definable. */

#define VME
#include <machine/kernel.h>
#include <machine/dvrio.h>
#include <sys/proc.h>
#include <machine/mem.h>
#include <machine/sysdev.h>
#include <errno.h>
#include <file.h>
#include <shmmap.h>
#include <sys/sysshmem.h>
#include <machine/asm.h>
#include <machine/psl.h>
#include <string.h>
#include "./btfpinfo.h"
#include "./btfpd.h"
#include "./btfpu.h"

struct card_info {
        struct driverhead int_info;
        struct intvector *vector;
        int bcdrvr_sem;
        unsigned short *pBC;
        int user_sem;
        char btfp_level;
        char btfp_vector;
        };

struct bc_data {
        struct card_info card_data[BTFPCOUNT];
};

typedef struct bc_data bcd;
```

```
/* btfpd.h */
/****************************************************************************/
/*
	The include file for the bc635VME/bc350VXI HP-RT v1.1x device driver.

	The device name is 'btfp' for 'BANCOMM time and frequency processor'.

	Copyright (C) 1992, BANCOMM, San Jose, CA.  All rights reserved.

	Author:	Greg Dowd
*/
/****************************************************************************/
#define MAXDEV	4

#define BTFPVOLATILE volatile struct btfp_device
```

CHAPTER 8
DRIVER SOURCE CODE

8.0 Listing

```
/***************************************************************************/
/*
BANCOMM bc63{57}VME/ bc35{07}VXI Time and Frequency Processor Driver

for HP-RT v1.1X Systems

Copyright (C) 1994, Datum Inc, Bancomm Div. All Rights Reserved.

Author: Greg Dowd
Datum Inc., Bancomm Div.
6541 Via Del Oro
San Jose, CA 95119-1294
(408)578-4161
*/
/***************************************************************************/
#include "./btfpdrvr.h"

#define MAXBUF 1024
static char hold[MAXBUF];
static void btfp_int();
int send_packet();

/******************************************************************************/
/*		btfp device state information structures.  MAXDEV is defined in
		btfpd.h as the max number of bc637's this driver can support.
*/

static struct btfp_status {
				int intwait;						/* Copy of MASK interrupt enable bits.*/
				int		signalon;					/* True if signal sending is enabled.*/
				int		packeton;					/* True if packet mode is enabled.*/
				int		debugon;					/* True if debug mode is enabled.*/
				int procp;						/* Pointer to calling process */
				int gps_synch;						/* True if synched with GPS packets */
				int rpacki;						/* Read Packet buffer index.			*/
				int wpacki;					/* Write Packet buffer index.		*/
				unsigned char rpack_buffer[MAXPACKET+1];		/* used for read calls */
				unsigned char wpack_buffer[MAXPACKET+1];		/* used for write calls */
				unsigned char work_buffer[MAXPACKET+1];		/* used to build packet */
} btfp_status[ MAXDEV ];							/* One for each bc63{57} supported.		*/

/***************************************************************************/
/*		btfpinstall - called during power-up

Try to access each btfp defined, up to
MAXDEV, which is our local definition
of how many btfp's this driver can support.
If an access succeeds, mark the board
on-line and announce it is there, else mark it
off-line and announce it is not there.  'Opens' to
boards either not there or beyond MAXDEV will fail.

For boards that are there, their interrupt registers
are loaded so that a subsequent interrupt enable (via
a WMASK ioctl(2) call which turns on any bits) will
allow an interrupt to come in.  The vector and
the interrupt level are obtained from the btfpinfo.h
initialized during kernel configuration.

*/
```

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

/* Allocate memory for static information saved by each device */
/*  Return an error if the call is not successful */
/* Start the system thread to handle theinterrupts */
/* The interrupt handler should be set up here. */
/* Turn on interrupts for this particular interrupt bit */
bcd *btfpinstall(info)
struct bc_info *info;
{
        bcd *DrvrData;                          /* struct passed to driver entry points */
        int ps;                                 /* rtn value from iointset call       */
        unsigned short *BCva;                   /* pointer to board virtual address*/
        register BTFPVOLATILE *dp;              /* Pointer to board base reg.         */
        register struct card_info *cp;          /* Pointer to card_data init          */
        register num_dev;                       /* # devices to initialize.           */
        register i;                             /* Index to a btfp & strucs.          */
        register j;                             /* Loop index for inits.              */
        ushort  temp;                           /* Temp place to access board.        */

        go_away_ifnot_this_cpu(BTFP0_VME_VEC);
        if ((DrvrData = (bcd *)sysbrkequiv ((long)(sizeof (bcd)),
          DEF_SBRK_ALIGN)) == NULL )
        {                                       /* allocate shared mem for drvr data struct */
                pseterr(ENXIO) ;
                return((bcd *)SYSERR) ;
        }

        if( BTFPCOUNT > MAXDEV )                /* Support only MAXDEV devices.*/
        {
                num_dev = MAXDEV;               /* Truncate count to the max.         */
                kprintf("btfpinit(): Driver supports only the first ",
                                        "%d btfp configured.\n", MAXDEV );
        }
        else                                    /* Else we can support all of         */
        {
                num_dev = BTFPCOUNT;            /* the btfp's configured.             */
        }

        for (i = 0; i < num_dev; i++)           /* their structures.                  */
        {
                dp = (BTFPVOLATILE *) sysshmem_to_va(A16+DATA_ACCESS,
                        (char *)info[i].btfp_physaddr);   /* Pointer to base register.          */

                cp = &DrvrData->card_data[i];             /* Pointer to card_info struct        */

                dp->btfp_mask   = 0x00;
                dp->btfp_vector = info[i].btfp_vector;
                dp->btfp_level  = info[i].btfp_level;
                btfp_status[ i ].intwait  = 0;                          /* NOT waiting for an int.                      */
                btfp_status[ i ].signalon = 0;                          /* Signal mechanism is turned OFF.              */
                btfp_status[ i ].packeton = 0;                          /* Packet mechanism is turned OFF.              */
                btfp_status[ i ].debugon  = 0;                          /* Debug mode is turned OFF.                    */
                btfp_status[ i ].gps_synch= 0;                          /* NOT synched with GPS packets.                */
                cp->vector = &info[i].vector;                           /* define driver interrupt vector               */
                cp->int_info.interrupt_action = SIGNAL_ONLY ;           /* define driver interrupt action               */
                                                                        /* as SIGNAL_ONLY.  This lets the               */
                                                                        /* int handler run as a normal                  */
                                                                        /* kernel thread.          */
                cp->int_info.sem = &DrvrData->card_data[i].bcdrvr_sem ;         /* set pointer to semaphore which */
                                                                                /* kernel will ssignal() on int      */
                cp->btfp_level  = info[i].btfp_level;                           /* define btfp interrupt level       */
                DrvrData->card_data[i].btfp_vector = info[i].btfp_vector;       /* define btfp interrupt vector      */
                DrvrData->card_data[i].bcdrvr_sem = 0 ;                 /* initialize semaphore to 0                    */
                DrvrData->card_data[i].pBC = dp;                        /* set ptr to board virt addr                   */
                DrvrData->card_data[i].user_sem = 0 ;                   /* initialize user semaphore                    */
        if(i == 0)
        {
            /* Create the interrupt handler thread */
            if(ststart(btfp_int,MINSTK,51,"btfp_int",1,
                (char *)&DrvrData->card_data[i]) == SYSERR)
            {
                pseterr(EIO);
                kprintf("\nCouldn't start int thread\n") ;
                return((bcd *)SYSERR) ;
            }
        }
        /* Set the interrupt  handler */
        if((ps = iointset(DrvrData->card_data[i].vector,
                        (char *)NULL ,&DrvrData->card_data[i])) == SYSERR)
        {
            kprintf("\nCouldn't setup the interrupt\n") ;
            return((bcd *)SYSERR) ;
        }

        /* Enable our interrupt */
        dvrenable(DrvrData->card_data[i].vector->interrupt_bit) ;

    }


    return(DrvrData) ;
}

btfpopen(DrvrData, mmdev, fd)
bcd *DrvrData;
int mmdev;
struct file *fd;
{
    int ps,ps2;
    register BTFPVOLATILE *dp = (BTFPVOLATILE *)
    DrvrData->card_data[ minor(mmdev) ].pBC;                        /* Init pointer to base register    */
    register struct btfp_status *sp = &btfp_status[minor(mmdev)];   /* Init pointer to status register. */
    ushort  ustemp, usmask;                                         /* For use turning off interrupts.  */
    int         s;                                                  /* Interrupt level from spl().      */

    if(( minor(mmdev) > MAXDEV) || (dp == (BTFPVOLATILE *)0))
    {
        pseterr(ENXIO);
        return SYSERR;                                              /* Set no such device or addr error */
    }
    /* Do a successful, normal OPEN */

    dvrdisable(ps,DrvrData->card_data[0].vector->interrupt_bit);    /* disable board interrupt          */
    sp->intwait = 0;                                                /* Indicate not waiting for int.    */
    sp->signalon = 0;                                               /* Signal mechanism turned off      */
    sp->packeton = 0;                                               /* Packet mode is turned off.       */
    sp->debugon  = 0;                                               /* Debug mode is turned off.        */

    dp->btfp_mask = 0;                                              /* Turn off interrupt               */

    usmask = LOCKDIS | HBDIS | EVENTDIS | STRDIS;                   /* Disable lockout, heartbeat       */
                                                                    /* event & strobe interrupts.       */
    dp->btfp_cr0 = usmask;
    sp->procp = currpptr;

    dvrrestore(ps,DrvrData->card_data[0].vector->interrupt_bit);    /* reenable board interrupt.        */
    return(0);                                                      /* Return open successful           */
}

btfpclose(DrvrData, fd)
bcd *DrvrData;
struct file *fd;
```

```
{
        int mmdev = fd->dev;

        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
        DrvrData->card_data[ minor(mmdev) ].pBC;                       /* Init pointer to base register      */
        register struct btfp_status *sp = &btfp_status[minor(mmdev)];  /* Init pointer to status register.   */
        ushort     ustemp, usmask;                                     /* For use turning off interrupts.    */
        int                ps;                                         /* Interrupt restore value.           */

        if(( minor(mmdev) > MAXDEV) || (dp == (BTFPVOLATILE *)0))
        {
                pseterr(ENXIO);
                return SYSERR;                                         /* Set no such device or addr error */
        }
        /* Do a successful, normal CLOSE */

        dvrdisable(ps,DrvrData->card_data[0].vector->interrupt_bit);   /* disable board interrupt            */
        sp->intwait  = 0;                                              /* Indicate not waiting for int.      */
        sp->signalon = 0;                                              /* Signal mechanism turned off        */
        sp->packeton = 0;                                              /* Packet mode is turned off.         */
        sp->debugon  = 0;                                              /* Debug mode is turned off           */

        dp->btfp_mask = 0;                                             /* Turn off interrupt                 */

        usmask = LOCKDIS | HBDIS | EVENTDIS | STRDIS;                  /* Disable lockout, heartbeat         */
                                                                       /* event & strobe interrupts.         */
        dp->btfp_cr0 = usmask;

        sp->procp = 0;                                                 /* Pointer to user process for        */
                                                                       /* sending signals                    */

        dvrrestore(ps,DrvrData->card_data[0].vector->interrupt_bit);   /* reenable board interrupt.   */
        return(0);                                                     /* Return close successful.    */
}

/************************************************************************/
/*
        btfpread()- Called when an application issues a read(2) call.


        (default) PACKETOFF mode:

        Set to Free Running Mode (mode 1), then makes a time request.

        Access the bc63{57}'s capture register to freeze the time,
        then return (converting BCD to ASCII) bytes of the time
        to the caller, until the number of bytes he requested is
        reached or until we run out of bytes to return.

        Sends 34 bytes maximum, separating each of the 16 bytes
        of time data with 3 spaces, and new-line and null
        terminating the string.

        This routine exits when 34 bytes have been sent in response
        to a read, regardless of what u.u_count is.  If it was not
        zero, then presumably the caller (e.g. cat()) will call
        again, but with u.u_offset = 33, which is our indication to
        just return without any transfer, which will result in the
        caller seeing 0 bytes transferred, which means EOF, by
        UNIX definition.

        SIGNALON/SIGNALOFF modes have no effect on this mode.

        PACKETON mode:

        Block until a data packet is available from the OUTFIFO (unless
        SIGNALON mode), then fill the buffer supplied by the read call
        with the data from the OUTFIFO.
```

```
                SIGNALON notes:

                If both PACKETON and SIGNALON modes are in effect, then
                the user must issue the ioctl calls to clear the
                DPA interrupt status (WINTSTAT) and enable DPA interrupts
                in the MASK register in a SIGNALON (non-blocking) WMASK ioctl call.
                When the signal is received to indicate the DPA interrupt has
                occurred, the signal handler should issue the read(2) call,
                still leaving the driver in the SIGNALON mode. This will
                cause the btfpread() routine to immediately read the data
                packet from the OUTFIFO.

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

#define MAXTBYTES               49
#define MAXABYTES               22

btfpread(DrvrData, fd, buff, count)
bcd *DrvrData;
struct file *fd;
char *buff;
register int count;

{
        int mmdev = fd->dev;                    /* get dev nbr from fd                  */
        register int pass_count = count;        /* set up counter                       */
        register j = 0;                         /* Loop index                           */

        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
        DrvrData->card_data[ minor(mmdev) ].pBC;

        register struct btfp_status *sp = &btfp_status[ minor(mmdev) ];         /* Init pointer to status structure.  */
        unsigned short dummy;                   /* Dummy for use accessing captr*/
        unsigned short ustemp, ustemp2; /* Temps for use accessing regs.  */
        char timearray[ MAXABYTES ];/* Place to put converted time.     */
        register i = 0;                         /* Loop index thanks to K&R.    */
        register twi;                           /* Time word index                      */
        register tai;                           /* Time array index                     */
        unsigned char lochar;                   /* Place to build the ASCII             */
        unsigned char hichar;                   /* Ditto                                */
        int rtn;                                /* catcher for return code              */
        int ps;                                 /* previous state of int*/
        int s;                                  /* return from spl()                    */
        int length;                             /* Used for PACKETON read               */
        unsigned char *user_buffer;             /* pointer into user buffer             */
union char_union                /* For converting short to 'chars'. */
        {
                unsigned char   two_chars[ 2 ];         /* Two characters holder.               */
                unsigned short  time_word;              /* One of the time words.               */
        } ch_union;

        if( minor(mmdev) >= MAXDEV )                    /* Beyond supportable range             */
                {
                pseterr(ENXIO);                         /* Return error indication              */
                return(SYSERR);
                }

        if (!sp->packeton)
        {
        /* PACKETOFF mode (default) */
dummy = dp->btfp_unlock;                                /* Release capture lock if set.         */
        dummy = dp->btfp_timereq;                       /* Freeze the time via access to the timereq register */
                                                        /* Unpack bytes from btfp..             */

        timearray[ MAXABYTES - 1 ] = '\0';              /* Set up zero termination.             */
        timearray[ MAXABYTES - 2 ] = '\n';              /* And new-line termination             */
        tai = MAXABYTES - 3;                    /* Fill time array back to front      */
        for( twi = 4; twi >= 0; twi--)          /* Read time; BCD->ASCII      */
        {
                ch_union.time_word = dp->btfp_time[ twi ]; /* Get a time word from module   */

                lochar = ch_union.two_chars[ 1 ];               /* Get two digits to process       */
                hichar = ch_union.two_chars[ 1 ];               /* And into another var.       */

                lochar = ( lochar >> 4 ) | 0x30;        /* Convert a nibble to ASCII byte*/
                hichar = ( hichar & 0xf) | 0x30;        /* Convert the other nibble.       */

                timearray[ tai ] = hichar;              /* Move into the time array.       */
                tai--;                                  /* Decrement the time array indx.*/
                timearray[ tai ] = lochar;              /* Move the other byte.       */
                tai--;                                  /* Decrement the time array indx.*/

                lochar = ch_union.two_chars[ 0 ];               /* Get two digits to process       */
                hichar = ch_union.two_chars[ 0 ];               /* And into another var       */

                lochar = ( lochar >> 4 ) | 0x30;        /* Convert a nibble to ASCII byte*/
                hichar = ( hichar & 0xf) | 0x30;        /* Convert the other nibble.       */

                timearray[ tai ] = hichar;              /* Move into the time array       */
                tai--;                                  /* Decrement the time array indx.*/
                timearray[ tai ] = lochar;              /* Move the other byte       */
                tai--;                                  /* Decrement the time array indx.*/
                if( tai == 0 ) break;                   /* End the loop after MAXTBYTES.*/
        }
        /* Now send time string to caller.*/
        i = 0;                          /* Initialize byte index.       */
        while(pass_count)
        {
                if( (i <= MAXABYTES - 1) && (fd->position < MAXTBYTES) ) /* Deliver up to max bytes...       */
                                                                        /* And stop on a later read.       */
                {
                        if( i == 2 ||           /* Between the time fields...       */
                                i == 3 ||       /* Day, hour, minute..etc...       */
                                i == 6 ||
                                i == 8 ||
                                i == 10 ||
                                i == 12 ||
                                i == 15 ||
                                i == 18 ||
                                i == 19)        /* Insert spaces for readability.*/
                        {
                        for(j=0;j<3;j++)
                        {
                                *buff++ = ' ';
                                fd->position++;
                                pass_count--;
                        }
                        }
                        *buff++ = timearray[i];         /* Move time byte to user buff     */
                        fd->position++;                 /* increment logical file marker   */
                        pass_count--;                   /* decrement read request count   */
                        i++;                            /* Move on to the next byte.       */
                }
                else                    /* We have no more to give...      */
                {                       /* Or he doesn't want any more.   */
                        break;          /* Exit the while and the read.      */
                }
        }
        return(count - pass_count);     /* Return number of bytes passed */
}       /* end of PACKETOFF mode */
```

```
else /* start of PACKETON mode */
{
        if(pass_count >= sizeof(sp->rpack_buffer) )             /* Beyond pkt buffer size.        */
        {
                pseterr(ENXIO);
                return(SYSERR);
        }
        if (! (sp->signalon) )                                  /* Only block in SIGNALOFF mode*/
        {

                dvrdisable(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);  /* disable board interrupt         */
                                                                                          /* Wait for Data Packet Available */
                        if (! (sp->gps_synch) )                         /* Are we in synch with GPS?    */
                        {
                                dp->btfp_ack = ACK_CLEAR;       /* Clear OUTFIFO        */
                                                                /* When we get the next DPA, the interrupt routine      */
                                                                /* will get us synched up with GPS and set the flag     */
                        }
                        dp->btfp_ack = ACK_DPA;
                        sp->intwait = MASK_DPA;                         /* indicate waiting for a DPA int.        */
                        dp->btfp_mask = MASK_DPA;                       /* Set MASK reg. for a DPA int.           */

                        dvrrestore(ps,
                                        DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);

                        while ( sp->intwait )                           /* btfpint() will clear this on int. */
                        {
                        /* block, waiting for 637 interrupt. */
                                swait(&DrvrData->card_data[minor(mmdev)].user_sem,0);          /* Wait till our interrupt occurs. */
                        }

        }       /* end of SIGNALOFF section */

        /* Interrupt routine has synched us up with the TANS and read in    */
        /* the next packet (rpacki bytes) from the OUTFIFO into rpack_buffer */

        if (sp->rpacki <= pass_count)                   /* If less bytes than requested       */
        {
                rtn = memcpy (buff,
                                                        sp->rpack_buffer,
                                                        sp->rpacki);
                if (rtn != buff)
                {
                        /* Error */
                        kprintf("copyout (partial) error in btfpread\n");
                        return(SYSERR);
                }

        return(sp->rpacki);
        } /* end of "emptied our buffer first" case */
        else
        {
                /* user didn't give us a big enough buffer!     */

                rtn = memcpy(buff,
                                                sp->rpack_buffer,
                                                pass_count);
                if (rtn != buff)
                {       /* Error! */
                        kprintf("copyout (full) error in btfpread\n");
                        return(SYSERR);
                }
        return(pass_count);
        } /* end of "filled user buffer" case */

} /* end of PACKETON mode */

} /* end of btfpread routine */
```

```
/************************************************************************/
/*
        btfpwrite() - Called when an application does a write(2) call.

        (default) PACKETOFF mode:

        Sets to Free Running Mode (mode 1), waits for the 1PPS pulse (unless
        in SIGNALON mode), then loads time from user buffer.

        Accepts a buffer in UNIX's "date '%j%H%M%S'" format, namely Julian date,
        hours, minutes, then seconds (with or without intervening spaces). Prepends
        this buffer with a 'B' (to indicate Load Major Time). Note that this routine
        can accept the same format output by the default read(2) call, by
        stripping off the ms, us, and ns fields.

                        SIGNALON notes:

                        If both PACKETOFF and SIGNALON modes are in effect, then
                        the user must issue the ioctl calls to clear the
                        1PPS interrupt status (WINTSTAT) and enable 1PPS interrupts
                        in the MASK register in a SIGNALON (non-blocking) WMASK ioctl call.
                        When the signal is received to indicate the 1PPS interrupt has
                        occurred, the signal handler should issue the write(2) call,
                        still leaving the driver in the SIGNALON mode. This will
                        cause the btfpwrite() routine to immediately write the data
                        packet (containing the date and time) to the INFIFO.

PACKETON mode:

Accepts a buffer from the user and transmits it directly to the INFIFO.
SIGNALON/SIGNALOFF modes have no effect on this mode.

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

btfpwrite(DrvrData, fd, buff, count)
bcd *DrvrData;
struct file *fd;
char *buff;
register int count;
{
        int mmdev = fd->dev;                    /* get dev nbr from fd          */
        register int pass_count = count;        /* set up counter               */

        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
        DrvrData->card_data[ minor(mmdev) ].pBC;        /* Init pointer to base register.*/
                                                        /* Pointer to btfp registers.   */
        register struct btfp_status *sp = &btfp_status[ minor(mmdev) ];     /* Init pointer to status structure.*/
int rtn;                /* catcher for return code */
        int s;                          /* spl() rtn code */
        register int ps;                /* interrupt state */
        unsigned char *user_buffer;     /* pointer into user buffer */
        register worki;                 /* buffer index */
        int num_bytes;                  /* Number of user bytes */
        unsigned short usmask;          /* Used for register access */

        mmdev = 0;

        if( minor(mmdev) >= MAXDEV )            /* Beyond supportable range.    */
        {
                pseterr(ENXIO);                 /* Return error indication.     */
                return(SYSERR);
        }
        if (!sp->packeton)
        {
          /* PACKETOFF mode(default)    */

                    num_bytes = count;              /* Save number of user bytes */
                                                    /* Do this now to save time later */
                    rtn = memcpy(sp->work_buffer, buff, count);
                    if (rtn != sp->work_buffer)
                    {
                               /* Error !    */
                               kprintf("memcpy (packetoff) Error in btfpwrite\n");
                               pseterr(ENXIO);
                               return(SYSERR);
                    }

          /* Leave room for 'B'  */
                    sp->wpack_buffer[0] = SOH;                           /* Make into null-terminated B packet*/
                    sp->wpack_buffer[1] = 'B';
                    sp->wpacki = 2;
                    for (worki = 0; worki <=num_bytes; worki++ )         /* Eliminate white space, plus */
                                                                         /* ms, us, and ns fields*/
                    {
                               if ( (sp->work_buffer[worki] != ' ')
                                          && (sp->work_buffer[worki] != '\n')
                                          && (sp->wpacki <= 10) )        /* (dddhhmmss takes 9 bytes) */
                               {
                                          sp->wpack_buffer[sp->wpacki++] = sp->work_buffer[worki];
                               }
                    }
                    sp->wpack_buffer[sp->wpacki++] = ETB;                /* Finish it off.           */
                    sp->wpack_buffer[sp->wpacki] = (char) NULL;          /* with ETB and NULL*/

                    if (! sp->signalon)                                  /* Only block in signaloff mode*/
                    {
                               dvrdisable(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);     /* Disable interrupt */
                               if (sp->debugon)
                                          kprintf("T");
                               sp->intwait = MASK_1PPS;                  /* Indicate waiting for 1PPS int. */
                               dp->btfp_intstat = INTSTAT_1PPS;          /* Clear 1PPS int. status */
                               usmask = dp->btfp_intstat;                /* Clear ALL int. status  */
                               dp->btfp_intstat = usmask;                /* ... */
                               dp->btfp_mask = MASK_1PPS;                /* Set MASK reg. for 1PPS int.*/
                               dvrenable(DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);

                               while ( sp->intwait )           /* btfpint() will clear this on int. */
                               {
                                                                                                        /* block, waiting for 637 interrupt. */
                                          swait(&DrvrData->card_data[minor(mmdev)].user_sem,0);
                                                                                                        /* Wait till our interrupt occurs. */
                               }

                               /* Interrupt routine should send the packet in wpackbuffer              */

                    } /* end of (blocking) SIGNALOFF mode */

                               dvrdisable(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                                                    /*      block
interrupt */
                               if ( (send_packet (DrvrData, mmdev, sp->wpack_buffer)) == 1)
                                                                                                                    /* Load time
buffer */
                               {
                               dvrrestore(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                                                    /*    restore
interrupt */
                               kprintf("btfpwrite: Error from send_packet of wpack_buffer\n");
                               pseterr(EIO);
                               return(-3);
                               }
                               dvrrestore(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
```

```
                                                                                                                    /*    restore
interrupt */

                        return(count);                                                      /* Return no problem */
                }
                else
                {
                /* PACKETON mode */

                        if ( count >= sizeof(sp->wpack_buffer) ) /* beyond pkt buffer size*/
                        {
                                pseterr(ENXIO);
                                return(-4);
                        }
                        num_bytes = count;                                          /* Save number of user bytes */

                        rtn = memcpy(sp->wpack_buffer,
                                                buff,
                                                count);
                        if ( rtn != sp->wpack_buffer )
                        {
                                /* Error ! */
                                kprintf("memcpy (packeton) error in btfpwrite\n");
                                pseterr(EIO);
                                return(-5);
                        }
                        sp->wpack_buffer[num_bytes] = (unsigned char) NULL;
                                                                                                            /* Make null terminated
string */
                        dvrdisable(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                                                    /*      block
interrupt */
                        if ( (send_packet (DrvrData, mmdev, sp->wpack_buffer)) == 1)
                                                                                                                    /* Send raw
packet */
                        {
                        dvrrestore(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                                                    /*    restore
interrupt */
                        pseterr(EIO);
                        return(-6);
                        }
                        dvrrestore(ps,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                                                    /*    restore
interrupt */
                        return(count);                                                      /* Return # of bytes written */

                }/* end of PACKETON mode */

} /* end of btfpwrite routine */
/**************************************************************************/
/*
        btfpioctl()- Called when an application does an ioctl(2) call.


                                Generally, the bc63{57}'s registers are written or read
                                by these routines.

                                If we are in SIGNALON mode, we do not block when
                                an external interrupt enable (to MASK) is received,
                                else we do block.

                                /// We are enabling the interrupt after the cause
                                        of it may have happened, e.g. strobe may already
                                        have gone off.  Hence an application would sleep
                                        forever.
*/
/**************************************************************************/
```

```
btfpioctl(DrvrData, fd, cmd,arg)
bcd *DrvrData;
struct file *fd;
int cmd;
char *arg;
{
        int mmdev = fd->dev;                                /* define mmdev for btfpioctl */
        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
                                                                        DrvrData->card_data[ minor(mmdev) ].pBC;
                                                                                        /* Init pointer to base
register.*/
        /* Pointer to btfp registers.   */

        register struct btfp_status *sp = &btfp_status[ minor(mmdev) ];
                                                                                /* Init pointer to status structure.*/
        register struct btfp_device *up;
                                                                                /*   READALL   receives   such   a
pointer.*/
        register struct btfp_time *tp;
                                                                                /* Pointer to time register structure.  */
        register struct btfp_event *ep;
                                                                                /* Pointer to event register structure. */
        register struct btfp_strobe *strp;
                                                                                /* Pointer to strobe register structure.
 */
        struct btfp_device btfpdev;
        struct btfp_time btfptime;
        struct btfp_event btfpevent;
        struct btfp_strobe btfpstrobe;
        int     s;              /* Interrupt level from spl().          */
        volatile unsigned short ustemp, ustemp2; /* Temps for use accessing regs          */
        volatile unsigned int uitemp;           /* Temp for use in returning values.    */
        unsigned short  usmask = MASK_INT0 | MASK_INT1 | MASK_INT2 | MASK_INT3 | MASK_INT4;
                                                                        /* Mask for turning off device interrupts.*/

        up = &btfpdev;
        tp = &btfptime;
        ep = &btfpevent;
        strp = &btfpstrobe;

        switch (cmd) {                  /* Handle specific incoming commands.   */

  case RIDR:                    /* Read the ID register.        */
                ustemp = dp->btfp_id;
                uitemp = 0x0 | ustemp;      /* Right justify value in an int.       */
                memcpy (arg, &uitemp, sizeof(int));             /* Move to caller's space */
                break;

        case RDTR:                      /* Read the Device Type register.       */
                ustemp = dp->btfp_device;
                uitemp = 0x0 | ustemp;      /* Right justify value in an int.       */
                memcpy (arg, &uitemp, sizeof(int));             /* Move to caller's space */
                break;

        case RSR:                       /* Read the Status register.        */
                ustemp = dp->btfp_status;
                uitemp = 0x0 | ustemp;      /* Right justify value in an int.       */
                memcpy (arg, &uitemp, sizeof(int));    /* Move to caller's space */
                break;

        case WCR:                       /* Write to the Control register.   */
                dp->btfp_control = (ushort)arg;/* Directly from the user's argument. */
                break;

        case RTIMEREQ:                                          /* Read the Time request register.                              */
                ustemp = dp->btfp_timereq;
                uitemp = 0x0 | ustemp;                  /* Right justify value in an int.                       */
```

```
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case RTIME0:                                           /* Read time word zero.                  */
            ustemp = dp->btfp_time[ 0 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case RTIME1:                                           /* Read time word one.                   */
            ustemp = dp->btfp_time[ 1 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case RTIME2:                                           /* Read time word two.                   */
            ustemp = dp->btfp_time[ 2 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case RTIME3:                                           /* Read time word three.                 */
            ustemp = dp->btfp_time[ 3 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;
    case RTIME4:                                           /* Read time word four.                  */
            ustemp = dp->btfp_time[ 4 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case REVENT0:                                          /* Read event word zero.                 */
            ustemp = dp->btfp_event[ 0 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case REVENT1:                                          /* Read event word one.                  */
            ustemp = dp->btfp_event[ 1 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case REVENT2:                                          /* Read event word two.                  */
            ustemp = dp->btfp_event[ 2 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case REVENT3:                                          /* Read event word three.                */
            ustemp = dp->btfp_event[ 3 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case REVENT4:                                          /* Read event word four.                 */
            ustemp = dp->btfp_event[ 4 ];
            uitemp = 0x0 | ustemp;                     /* Right justify value in an int.       */
            memcpy (arg, &uitemp, sizeof(int));        /* Move to caller's space */
            break;

    case WSTROBE1:                                         /* Write strobe word one.                */
            dp->btfp_strobe1 = (ushort)arg;/* Directly from the user's argument.       */
            break;

    case WSTROBE2:                                         /* Write strobe word two.                */
            dp->btfp_strobe2 = (ushort)arg;/* Directly from the user's argument.       */
```

```
                break;

        case WSTROBE3:                                  /* Write strobe word three.                     */
                dp->btfp_strobe3 = (ushort)arg;/* Directly from the user's argument.      */
                break;

        case RUNLOCK:                                           /* Read Unlock register.        */
                ustemp = dp->btfp_unlock;               /* Contents of the register matter not.     */
                uitemp = 0x0 | ustemp;                  /* Right justify value in an int.           */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case RACK:                                                      /* Read the ACK register.       */
                ustemp = dp->btfp_ack;
                uitemp = 0x00ff & ustemp;       /* Preserve byte value only                         */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case WACK:                                              /* Write to the ACK register.   */
                dp->btfp_ack = (ushort)arg;/* Directly from the user's argument.  */
                break;

        case WCR0:                                              /* Write control register zero.                 */
                dp->btfp_cr0 = (ushort)arg;     /* Directly from the user's argument.*/
                break;

        case RCR0:                                              /* Read control register zero.                  */
                ustemp = dp->btfp_cr0;
                uitemp = 0x00ff & ustemp;       /* Preserve byte value only                         */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case ROUTFIFO:                                  /* Read the OUTFIFO register.           */
                ustemp = dp->btfp_outfifo;
                uitemp = 0x0 | ustemp;                  /* Right justify value in an int.           */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case WINFIFO:                                   /* Write to the INFIFO register.   */
                dp->btfp_infifo = (ushort)arg;/* Directly from the user's argument. */
                break;

        case RMASK:                                             /* Read the MASK register.                      */
                ustemp = dp->btfp_mask;
                uitemp = 0x00ff & ustemp;       /* Preserve byte value only                         */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case WMASK:                                             /* Write to the MASK register.  */
                if( !( (int)arg & ( (int) usmask ) ) )
                {
                        dp->btfp_mask = (ushort)arg; /* Directly from the user's argument.*/
                                                                        /* No interrupt enable requested.  */
                }
                else if( !sp->signalon )        /* Signal sending is NOT enabled.                   */
                {
                        dvrdisable(s, DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                        /* Block interrupts */


                        if ( (usmask & MASK_DPA)                /* DPA already? */
                                && (dp->btfp_mask & (ushort) arg) ) /* and user wants DPA */
                        {
                                if ( !(sp->gps_synch) )                         /* Are we in synch with GPS? */
                                {
                                        dp->btfp_ack = ACK_CLEAR;               /*  Clear OUTFIFO   */
                                                                                                /*Whenweget
the next */
```

```
                                                                                                /* DPA, the
interrupt   */
                                                                                                /* routine will
get us  */
                                                                                                /*synch-ed up
with GPS */
                                                                                                /* and set the
flag     */
                                }
                                dp->btfp_ack = ACK_DPA;                                 /* Clear DPA status             */
                        } /* end of "DPA-already" section */

                        sp->intwait = (ushort)arg;                      /* Indicate we're waiting for an int.           */
                        dp->btfp_intstat = (ushort)arg;     /* Clear any existing interrupt */
                                                                                /* Save the interrupts we're waiting for */
                        usmask = dp->btfp_intstat;                                              /* Clear ALL int. status */
                        dp->btfp_intstat = usmask;                                              /* ... */
                        dp->btfp_intstat = 0xFF;
                        dp->btfp_mask = (ushort)arg; /* Enable bc63{57} interrupt(s).                   */

                        dvrrestore(s,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);

                        while ( sp->intwait )                   /* btfpint() will clear this on int. */
                        {
                                                                                        /* block, waiting for 637 interrupt. */
                                swait(&DrvrData->card_data[minor(mmdev)].user_sem,0);
                                                                                        /* Wait till our interrupt occurs. */
                        }
                }
                else                                                            /* Signal sending IS enabled.                   */
                {
                        dvrdisable(s,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                /* Block interrupts */
                        if (usmask & MASK_DPA)                                  /* DPA already? */
                        {
                                if ( !(sp->gps_synch) )                                 /* Are we in synch with GPS? */
                                {
                                        dp->btfp_ack = ACK_CLEAR;                       /*  Clear OUTFIFO       */
                                                                                                /*When we get
the next */
                                                                                                /* DPA, the
interrupt   */
                                                                                                /* routine will
get us  */
                                                                                                /*synch-ed up
with GPS */
                                                                                                /* and set the
flag     */
                                }
                                dp->btfp_ack = ACK_DPA;                                 /* Clear DPA status             */
                        } /* end of "DPA-already" section */
                        sp->intwait = (ushort)arg;                      /* We will not be blocking*/
                        dp->btfp_intstat = (ushort)arg;                 /* Clear any existing interrupt */
                        usmask = dp->btfp_intstat;                                              /* Clear ALL int. status */
                        dp->btfp_intstat = usmask;                                              /* ... */
                        dp->btfp_mask = (ushort)arg; /* Enable bc63{57} interrupt(s) etc.  */
                        dvrrestore(s,DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
                                                                                /* Allow interrupts from our device again.*/
                }
                break;

        case RINTSTAT:                                                  /* Read the INTSTAT register.                   */
                ustemp = dp->btfp_intstat;
                uitemp = 0x00ff & ustemp;          /* Preserve byte value only                          */
                memcpy (arg, &uitemp, sizeof(int));             /* Move to caller's space */
                break;

        case WINTSTAT:                                                  /* Write to the INTSTAT register.               */
```

```
                dp->btfp_intstat = (ushort)arg;/* Directly from the user's argument.*/
                break;

        case RVECTOR:                                           /* Read the VECTOR register.      */
                ustemp = dp->btfp_vector;
                uitemp = 0x00ff & ustemp;       /* Preserve byte value only             */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case RLEVEL:                                            /* Read the LEVEL register.       */
                ustemp = dp->btfp_level;
                uitemp = 0x00ff & ustemp;       /* Preserve byte value only             */
                memcpy (arg, &uitemp, sizeof(int));     /* Move to caller's space */
                break;

        case READTIME:                                  /* Capture and read all time words.       */
                ustemp                          = dp->btfp_timereq;     /* Capture time.          */
                tp->btfp_time[ 0 ]      = dp->btfp_time[ 0 ];  /* Time word 0.          */
                tp->btfp_time[ 1 ]      = dp->btfp_time[ 1 ];  /* Time word 1.          */
                tp->btfp_time[ 2 ]      = dp->btfp_time[ 2 ];  /* Time word 2.          */
                tp->btfp_time[ 3 ]      = dp->btfp_time[ 3 ];  /* Time word 3.          */
                tp->btfp_time[ 4 ]      = dp->btfp_time[ 4 ];  /* Time word 4.          */
                memcpy(arg,tp,sizeof(struct btfp_time));
                break;

        case READEVENT:                                 /* Read all event words.          */
                ep->btfp_event[ 0 ] = dp->btfp_event[ 0 ];     /* Event word 0.        */
                ep->btfp_event[ 1 ] = dp->btfp_event[ 1 ];     /* Event word 1.        */
                ep->btfp_event[ 2 ] = dp->btfp_event[ 2 ];     /* Event word 2.        */
                ep->btfp_event[ 3 ] = dp->btfp_event[ 3 ];     /* Event word 3.        */
                ep->btfp_event[ 4 ] = dp->btfp_event[ 4 ];     /* Event word 4.        */
                memcpy (arg, &btfpevent, sizeof (struct btfp_event));
                break;

        case WRITESTROBE:                                       /* Write to the Strobe registers.  */
                memcpy (strp, arg, sizeof (struct btfp_strobe) );
                dp->btfp_cr0 = STRDIS;                  /* First disable strobe output    */
                dp->btfp_strobe1 = strp->btfp_strobe[0];       /* Strobe word 1        */
                dp->btfp_strobe2 = strp->btfp_strobe[1];       /* Strobe word 2        */
                dp->btfp_strobe3 = strp->btfp_strobe[2];       /* Strobe word 3        */
                break;

        case READALLREGS:                               /* Read all bc63{57} regs.                */
                up->btfp_id                     = dp->btfp_id;          /* ID register.           */
                up->btfp_device = dp->btfp_device;              /* Device Type register.        */
                up->btfp_status         = dp->btfp_status;      /* Status register.             */

                up->btfp_timereq        = dp->btfp_timereq;     /* Capture the time.    */
                up->btfp_time[ 0 ]      = dp->btfp_time[ 0 ];  /* Time word 0.          */
                up->btfp_time[ 1 ]      = dp->btfp_time[ 1 ];  /* Time word 1.          */
                up->btfp_time[ 2 ]      = dp->btfp_time[ 2 ];  /* Time word 2.          */
                up->btfp_time[ 3 ]      = dp->btfp_time[ 3 ];  /* Time word 3.          */
                up->btfp_time[ 4 ]      = dp->btfp_time[ 4 ];  /* Time word 4.          */

                up->btfp_event[ 0 ] = dp->btfp_event[ 0 ];     /* Event word 0.        */
                up->btfp_event[ 1 ] = dp->btfp_event[ 1 ];     /* Event word 1.        */
                up->btfp_event[ 2 ] = dp->btfp_event[ 2 ];     /* Event word 2.        */
                up->btfp_event[ 3 ] = dp->btfp_event[ 3 ];     /* Event word 3.        */
                up->btfp_event[ 4 ] = dp->btfp_event[ 4 ];     /* Event word 4.        */

                up->btfp_unlock         = dp->btfp_unlock;      /* UNLOCK register. */

                up->btfp_ack                    = dp->btfp_ack;         /* ACK register.        */
                up->btfp_cr0                    = dp->btfp_cr0;         /* Control register 0.  */
                up->btfp_outfifo        = dp->btfp_outfifo;     /* OUTFIFO register. */
                up->btfp_mask                   = dp->btfp_mask;        /* MASK register.       */
                up->btfp_intstat        = dp->btfp_intstat;     /* INTSTAT register. */
                up->btfp_vector                 = dp->btfp_vector;      /* VECTOR register.  */
```

```
			up->btfp_level			= dp->btfp_level;			/* LEVEL register.	*/
			memcpy (arg, up, sizeof (struct btfp_device));

			break;

		case SIGNALON:							/* Turn on signal sending on interrupts.*/
			sp->signalon = 1;				/* Indicate we should send signals on ints.*/
			break;

		case SIGNALOFF:							/* Turn off signal sending on interrupts.*/
			dvrdisable(s, DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
												/* Block interrupts */
			sp->signalon = 0;				/* Stop sending signals on interrupts.	*/
			sp->intwait = 0;				/* Indicate we are not awaiting any int.*/
			dp->btfp_mask = 0;				/* Turn off all 5 interrupts.	*/
			dvrrestore(s, DrvrData->card_data[minor(mmdev)].vector->interrupt_bit);
												/* Restore interrupt */
			break;

		case PACKETON:							/* Turn on packet mode for read/write.*/
			sp->packeton = 1;
			break;

		case PACKETOFF:							/* Turn off packet mode for read/write.*/
			sp->packeton = 0;
			break;

		case DEBUGON:
			sp->debugon = 1;				/* Enable driver debugging */
			break;

		case DEBUGOFF:
			sp->debugon = 0;				/* Disable driver debugging */
			break;

		default:
			pseterr(EINVAL);				/* Return this is not a tty type device.*/
			return(SYSERR);
			break;
		}
		return( 0 );						/* Must return OK if no error found.	*/

} /* end of btfpioctl routine */

/*

btfpselect(DrvrData, fd, which ,se)
bcd *DrvrData;
struct file *fd;
int which;
struct sel *se;
{

		kprintf("\nNow in bcdrvselect\n");
		return(0);
}
 */
btfpuninstall(DrvrData)
bcd *DrvrData;
{
/* The Interrupt handler should be disabled here. */


/* deallocate memory used by static device information */

		sysfree (DrvrData, (long)sizeof (bcd));
		return (0);
}
```

```
/* btfp_int */

/* This interrupt routine does not do anything usefull.  It simply waits
   on a system semaphore, and writes a message to the console when an
   interrupt occurs.  The two things it does that are necessary are to
   clear the interrupt bit, and to turn interrupts back on for the device
   we are using. */

static void btfp_int(DrvrData)
register bcd *DrvrData;
{
        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
                                                                        DrvrData->card_data[0].pBC;
                                                                /* Init pointer to base register */
        register struct btfp_status *sp = &btfp_status[0];
                                                                /* Init pointer to status structure */
        unsigned short ustemp,ustemp2;  /* temps for register access */
        unsigned short usmask = MASK_INT0 | MASK_INT1 | MASK_INT2 |MASK_INT3 |MASK_INT4;
                                                                /* Mask of module interrupt sources */

        while (1)
        {
                swait (DrvrData->card_data[0].int_info.sem, 0);
                usmask = MASK_INT0 | MASK_INT1 | MASK_INT2 |MASK_INT3 |MASK_INT4;
                /* Clear interrupt we received */
                dvrclearint (DrvrData->card_data[0].vector->interrupt_bit);
                /* Turn interrupts back on for this device */
                dvrenable (DrvrData->card_data[0].vector->interrupt_bit);
                ustemp = dp->btfp_intstat;              /* Read INTSTAT register */
                if( !(sp->intwait & ustemp) )   /* We weren't waiting for this int! */
                {
                        dp->btfp_mask = 0;              /* Turn off all interrupt sources */
                        dp->btfp_intstat = usmask;      /* Reset INTSTAT register */
                        kprintf("btfpint(): Unexpected interrupt from btfp0\n");
                        return;
                }
                else if( !sp->signalon)                 /* We were waiting for an int.. */
                {                                                               /* and were blocked doing so.   */
                        sp->intwait = 0;                        /* Indicate we received an int. */
                        if (ustemp & INTSTAT_DPA )
                        {
                                /* For DPA interrupt, remember to clear the ACK_DPA bit */
                                dp->btfp_ack = ACK_DPA;
                        }

                        dp->btfp_mask = 0;                      /* Turn off all int sources  */
                        dp->btfp_intstat = usmask;      /* Reset INTSTAT register    */
                        ssignal(&DrvrData->card_data[0].user_sem); /* Send signal to blocked process */
                }
                else                                                    /* Waiting for interrupt but in */
                {                                                               /* SIGNALON mode...*/
                        dp->btfp_intstat = usmask;      /* Reset INTSTAT register  */

                        /* if DPA is here...*/
                        if (ustemp & INTSTAT_DPA)
                        {

                        if (sp->debugon)
                        {
                                if (ustemp & INTSTAT_DPA)
                                        kprintf ("d");
                        }

                        /* ...AND the user wanted to look at it */
                        if (sp->intwait & MASK_DPA)
                        {
```

```
if (sp->debugon)
{
        if (ustemp & INTSTAT_DPA)
                kprintf("D");
}

dp->btfp_ack = ACK_DPA;                 /* Clear DPA bit in ACK reg */

/* NOTE: SIGNALON mode has already waited for the DPA signal.*/
/* thus, its read just reads the available packet */

/* Now fill the packet buffer from OUTFIFO */

sp->rpacki = 0;

/* We must read the first byte to activate the ACK_MORE flag */
/* and to see if we are out of synch with the TANS packets   */
/* If the first two bytes don't look like the start of a     */
/* packet, just clear the OUTFIFO again                      */

if (!sp->gps_synch)
{

sp->rpack_buffer [sp->rpacki++] = dp->btfp_outfifo; /* Get 1rst byte*/
if ( sp->rpack_buffer [sp->rpacki - 1] != (unsigned char)DLE )
{
        /* Bogus packet...clear OUTFIFO */
        dp->btfp_ack = ACK_CLEAR;    /* Clear OUTFIFO */
}
else
{
        /* First byte OK..now look at next byte; is it a packet ID? */
        sp->rpack_buffer [sp->rpacki++] = dp->btfp_outfifo;
        if ( ( sp->rpack_buffer [sp->rpacki - 1] == (unsigned char)DLE )
         || (sp->rpack_buffer [sp->rpacki - 1] == (unsigned char)ETX ) )
        {
                /* Bogus packet...clear OUTFIFO */
                dp->btfp_ack = ACK_CLEAR;    /* Clear OUTFIFO */
        }
        /* else packet is OK....read it in */

} /* end of first byte OK section */

if (sp->debugon)
        kprintf ("S");
sp->gps_synch = 1;                                      /* We're synched up now */

} /* end of "need to synch up with the GPS" section */

/* NOTE: ACK_MORE bit on means "more bytes in OUTFIFO" */

ustemp = dp->btfp_ack;
while ( (ustemp & ACK_MORE)                     /* Until FIFO empty */
  && (sp->rpacki < MAXPACKET) )                 /* or running away */
{
        sp->rpack_buffer [sp->rpacki++] = dp->btfp_outfifo;/*Get byte*/
        ustemp = dp->btfp_ack;                          /* Any more? */

} /* end of OUTFIFO read loop */

if (sp->rpacki == 0)
{
        if (sp->debugon)
                kprintf("0");
        /* Wait for next DPA interrupt */
        return;
}
```

```
                        } /* end of "user wanted to see DPA" section */

                        else
                        {
                                ustemp &= ~INTSTAT_DPA;          /* Any other interrupts happen? */
                                if ( !(ustemp & sp->intwait) ) /*if not our interest, bail  */
                                {
                                        return;
                                }
                                /* else, fall through to send the user a signal */
                        }

                        } /* end of DPA section */

                        /* if no DPA, we must have matched on another INTSTAT bit */
                        pkill(sp->procp,SIGUSR1);

                } /* end of SIGNALON section */

        } /* end of while loop, setup signal wait again */

} /* end of btfpint routine */

int send_packet(DrvrData, mmdev, packet)
bcd *DrvrData;
dev_t mmdev;
unsigned char *packet;
{
        register BTFPVOLATILE *dp = (BTFPVOLATILE *)
                                                        DrvrData->card_data[ minor(mmdev) ].pBC;
                                                /* init pointer to base register */
        register struct btfp_status *sp = &btfp_status[ minor(mmdev) ];
                                                /* init pointer to status register */

        int loop_count;
        unsigned char *byte_ptr=packet;
        unsigned short ustemp;          /* for accessing ACK register */
        int     ps;                                     /* Interrupt level */

        dp->btfp_ack = ACK_INFIFO;    /* Clear ack bit */

        while (*byte_ptr)
        {
                dp->btfp_infifo = *byte_ptr;
                byte_ptr++;
        } /* end of while */


        dp->btfp_ack = ACK_INACT;               /* Take action bit */

                                                        /* Wait for bc63{57} to take action */

                                                        /* NOTE: no interrupt generated */
                                                        /* so we have to spin */

        loop_count = 0;
        while (loop_count < MAX_SPIN)
        {
                ustemp = dp->btfp_ack;
                if ( ustemp & ACK_INFIFO ) /* Break if action taken */
                {
                        break;
                }
                else
                {
                        loop_count++;
                }
        }
        dp->btfp_ack = ACK_INFIFO;              /* Clear ack bit */

        if (loop_count >= MAX_SPIN)
        {
                kprintf("TIMEOUT (%d) waiting for INFIFO action\n",MAX_SPIN);
                return(SYSERR);                                 /* Return error */
        }
        else
        {
                if (sp->debugon)
                        kprintf("%dLoop ", loop_count);
        }

        return(0);                                              /* Return no problem */

} /* end of send_packet routine */
```