The Embedded I/O Company

TEWS TECHNOLOGIES

TIP670-SW-42

VxWorks Device Driver

Digital I/O

Version 2.0.x

User Manual

Issue 2.0.1

June 2008

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
25469 Halstenbek, Germany
www.tews.com
Phone: +49 (0) 4101 4058 0
Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com

TEWS TECHNOLOGIES LLC
9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com
Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

TIP670-SW-42

VxWorks Device Driver

Digital I/O

Supported Modules:
TIP670

Issue	Description	Date
1.0	First Issue	April 1996
1.1	General Revision	September 17, 2003
2.0.0	Carrier Support added, New File List, General Revision Initialization functions changed (tip670Drv(), tip670DevCreate())	August 22, 2006
2.0.1	Carrier Driver description added	June 24, 2008

Table of Contents

1 Introduction

1.1 Device Driver

The TIP670-SW-42 VxWorks device driver software allows the operation of the supported IPAC conforming to the VxWorks I/O system specification. This includes a device-independent basic I/O interface with *open*(), *close(), read(), write(),* and *ioctl()* functions.

Special I/O operation that do not fit to the standard I/O calls will be performed by calling the ioctl() function with a specific function code and an optional function dependent argument.

The TIP670-SW-42 device driver supports the following features:

- reading input value
- setting output value
- wait for selectable input events (match, high-, low-, any transition on the input line(s))
- enable, disable and trigger output watchdog
- start, read, and stop timers, which can be controlled by the user or by input lines

The TIP670-SW-42 supports the modules listed below:

TIP670-10	8 Channel Digital I/O	IPAC
TIP670-20	4 Channel Digital I/O	IPAC

To get more information about the features and use of supported devices it is recommended to read the manuals listed below.

TIP670 User manual

TIP670 Engineering Manual

CARRIER-SW-42 IPAC Carrier User Manual

1.2 IPAC Carrier Driver

IndustryPack (IPAC) carrier boards have different implementations of the system to IndustryPack bus bridge logic, different implementations of interrupt and error handling and so on. Also the different byte ordering (big-endian versus little-endian) of CPU boards will cause problems on accessing the IndustryPack I/O and memory spaces.

To simplify the implementation of IPAC device drivers which work with any supported carrier board, TEWS TECHNOLOGIES has designed a so called Carrier Driver that hides all differences of different carrier boards under a well defined interface.

The TEWS TECHNOLOGIES IPAC Carrier Driver CARRIER-SW-42 is part of this TIP670-SW-42 distribution. It is located in directory CARRIER-SW-42 on the corresponding distribution media.

This IPAC Device Driver requires a properly installed IPAC Carrier Driver. Due to the design of the Carrier Driver, it is sufficient to install the IPAC Carrier Driver once, even if multiple IPAC Device Drivers are used.

Please refer to the CARRIER-SW-65 User Manual for a detailed description how to install and setup the CARRIER-SW-42 device driver, and for a description of the TEWS TECHNOLOGIES IPAC Carrier Driver concept.

How to use the carrier driver in the application program is shown in the programming example tip670exa.c.

If the IPAC carrier driver isn't used for the IPAC driver setup, the application software has to setup carrier board hardware, mapping of device memory and interrupt level setup by itself.

2 Installation

Following files are located on the distribution media:

Directory path 'TIP670-SW-42':

tip670drv.c	TIP670 device driver source
tip670def.h	TIP670 driver include file
tip670.h	TIP670 include file for driver and application
tip670exa.c	Example application
include/ipac_carrier.h	Carrier driver interface definitions
TIP670-SW-42-2.0.1.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

2.1 Include device driver in Tornado IDE project

For including the TIP670-SW-42 device driver into a Tornado IDE project follow the steps below:

(1) Copy the files from the distribution media into a subdirectory in your project path.
(For example: ./TIP670)

(2) Add the device drivers C-files to your project.
Make a right click to your project in the 'Workspace' window and use the 'Add Files ...' topic.
A file select box appears, and the driver files can be selected.

(3) Now the driver is included in the project and will be built with the project.

For a more detailed description of the project facility please refer to your Tornado User's Guide.

2.2 Special installation for Intel x86 based targets

The TIP670 device driver is fully adapted for Intel x86 based targets. This is done by conditional compilation directives inside the source code and controlled by the VxWorks global defined macro **CPU_FAMILY**. If the content of this macro is equal to *I80X86* special Intel x86 conforming code and function calls will be included.

2.3 System resource requirement

The table gives an overview over the system resources that will be needed by the driver.

Resource	Driver requirement	Devices requirement
Memory	< 1 KB	< 1 KB
Stack	< 1 KB	---
Semaphores	TIP670_MAX_JOBS	---

Memory and Stack usage may differ from system to system, depending on the used compiler and its setup.

The following formula shows the way to calculate the common requirements of the driver and devices.

*<total requirement> = <driver requirement> + (<number of devices> * <device requirement>)*

The maximum usage of some resources is limited by adjustable parameters. If the application and driver exceed these limits, increase the according values in your project.

2.4 Device driver configuration

To configure the device driver behavior look for the definitions listed below. The definition is part of the device driver header file tip670.h.

TIP670_MAX_JOBS

This definition defines the maximum number of available input event jobs for all TIP670 devices. It's used by a global job list that is shared across all installed devices. The default setting is 10.

3 I/O system functions

This chapter describes the driver-level interface to the I/O system. The purpose of these functions is to install the driver in the I/O system, add and initialize devices.

3.1 tip670Drv()

NAME

tip670Drv() - installs the TIP670 driver in the I/O system

SYNOPSIS

#include “tip670.h”

```
STATUS tip670Drv
(
      void
)
```

DESCRIPTION

This function installs the TIP670 driver in the I/O system and allocates resources for job handling.

A call to this function is the first thing the user has to do before adding any device to the system or performing any I/O request.

EXAMPLE

```
#include "tip670.h”

STATUS             result;
…

/*-------------------
  Initialize Driver
  -------------------*/
result = tip670Drv();
if (result == ERROR)
{
     /* Error handling */
}
```

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

Error codes are only set by system functions. The error codes are stored in *errno* and can be read with the function *errnoGet()*.

SEE ALSO

VxWorks Programmer's Guide: I/O System

3.2 tip670DevCreate()

NAME

tip670DevCreate() – Add a TIP670 device to the VxWorks system

SYNOPSIS

```
#include “tip670.h”

STATUS tip670DevCreate
(
        char        *name,
        int         devIdx,
        int         funcType,
        void        *pParam
)
```

DESCRIPTION

This routine adds the selected device to the VxWorks system. The device hardware will be setup and prepared for use.

This function must be called before performing any I/O request to this device.

PARAMETER

name

This string specifies the name of the device that will be used to identify the device, for example for *open()* calls.

devIdx

This index number specifies the device to add to the system.
The index number depends on the search priority of the modules. The modules will be searched in the following order:

- TIP670-10
- TIP670-20

If modules of the same type are installed the channel numbers will be assigned in the order the CARRIER driver function will find the devices.

Example: (A system with 2 TIP670-20, and 1 TIP670-10) will assign the following device indices:

Module	Device Index
TIP670-10	0
TIP670-20 (1st)	1
TIP670-20 (2nd)	2

funcType

This parameter is unused and should be set to *0*.

pParam

This parameter points to a structure (*TIP670_DEVCONFIG*) containing the default configuration of the device.

```
typedef struct
{
    struct ipac_resource    *ipac;
    unsigned char           inputPol;
    unsigned char           outputPol;
    unsigned short          watchDogTime;
} TIP670_DEVCONFIG;
```

ipac

Pointer to TIP670 module resource descriptor, retrieved by CARRIER Driver ipFindDevice() function

inputPol

Specifies the input polarity of the input lines. Inverted inputs may be useful to select the active level of an input line. For example a low active input pin will be recognized as a 1, if the specified input pin is connected to a low level, and it will be read as 0, if the input is pulled to a high level. A bit set to '0' will be inverted, set to '1' will not invert the input. Bit 0 specifies the input polarity of input 1, bit 1 specifies the input polarity of input 2, and so on.

outputPol

Specifies the output polarity of the output lines. Inverted output may be useful to select the active output level of a line. For example a low active pin shall be set to active, if inverted output is set. A bit set to '1' will be inverted, set to '0' will not invert the output. Bit 0 specifies the output polarity of output 1, bit 1 specifies the output polarity of output 2, and so on.

watchDogTime

Specifies the output watchdog time. If there are no accesses to the TIP670 registers, during the specified watchdog time the output will be disabled until a new access is made. The time is set in 0.5µsec units. If the watchdog time is set to 0 the watchdog is disabled.

EXAMPLE

```
#include <tip670.h>

…

STATUS             result;
TIP670_DEVCONFIG   tip670Conf;

…

/*-------------------------------------------------------
  Create the device "/tip670/0" for the first TIP670 device
  Device specific parameters must be set up:
         do not invert input and output
         disable output watchdog
  -------------------------------------------------------*/
tip670Conf.inputPol =       0x00;
tip670Conf.outputPol =      0x00;
tip670Conf.watchDogTime =   0;

result = tip670DevCreate(   "/tip670/0",
                            0,
                            0,
                            (void*)&tip670Conf);
if (result == OK)
{
     /* Device successfully created */
}
else
{
     /* Error occurred when creating the device */
}

…
```

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno* and can be read with the function *errnoGet()*.

Error code	Description
S_ioLib_NO_DRIVER	Driver has not been installed (*tip670Drv*() not called)
EINVAL	Illegal parameter value
EISCONN	The device has been created before
ENOMEM	Can't allocate device memory

SEE ALSO

VxWorks Programmer's Guide: I/O System

4 I/O Functions

4.1 open()

NAME

open() - open a device or file.

SYNOPSIS

```
int open
(
        const char  *name,
        int         flags,
        int         mode
)
```

DESCRIPTION

Before I/O can be performed to the TIP670 device, a file descriptor must be opened by invoking the basic I/O function *open().*

PARAMETER

name

Specifies the device which shall be opened, the name specified in tip670DevCreate() must be used

flags

Not used

mode

Not used

EXAMPLE

```
int       fd;

…

/*----------------------------------------
  Open the device named "/tip670/0" for I/O
  ----------------------------------------*/
fd = open("/tip670/0", 0, 0);
if (fd == ERROR)
{
     /* handle error */
}

…
```

RETURNS

A device descriptor number or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual).

SEE ALSO

ioLib, basic I/O routine - *open()*

4.2 close()

NAME

close() – close a device or file

SYNOPSIS

```
STATUS close
(
      int           fd
)
```

DESCRIPTION

This function closes opened devices.

PARAMETER

fd

This file descriptor specifies the device to be closed. The file descriptor has been returned by the *open()* function.

EXAMPLE

```
int       fd;
STATUS    retval;

…

/*----------------
  close the device
  ----------------*/
retval = close(fd);
if (retval == ERROR)
{
     /* handle error */
}

…
```

RETURNS

OK or ERROR. If the function fails, an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual)

SEE ALSO

ioLib, basic I/O routine - close()

4.3 read()

NAME

read() – reads input value of the specified TIP670 device.

SYNOPSIS

```
int read
(
        int             fd,
        char            *buffer,
        size_t          maxbytes
)
```

DESCRIPTION

This function reads the input value of the specified module and places them in *buffer*. The value returned in *buffer* corresponds to the signals at the input lines

Remember the configuration, there may be inverted signals.

PARAMETER

fd

This file descriptor specifies the device to be used. The file descriptor has been returned by the *open()* function.

buffer

This argument points to a user supplied buffer. The returned input value will be filled into this buffer.

maxbytes

This parameter specifies the maximum number of read bytes. The function always returns a buffer of 1 byte.

EXAMPLE

```
int           fd;
unsigned char inVal;
int           retval;

…

/*---------------------------------
  Read input data from TIP670 device
  ---------------------------------*/
retval = read(fd, &inVal, 1);
if (retval == 1)
{
     printf("INPUT: %02Xh\n", inVal);
}
else
{
     /* handle the read error */
}

…
```

RETURNS

Number of bytes read or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual) or a driver set error code described below.

Error code	Description
S_t670_Drv_SM_BUFFER	Specified buffer to small

SEE ALSO

ioLib, basic I/O routine - read()

4.4 write()

NAME

write() – write data to the output of a specified device.

SYNOPSIS

```
int write
(
        int             fd,
        char            *buffer,
        size_t          nbytes
)
```

DESCRIPTION

This function sets the output value of the specified device.

PARAMETER

fd

This file descriptor specifies the device to be used. The file descriptor has been returned by the *open()* function.

buffer

This argument points to a user supplied buffer. The buffer value must be filled with the output value.

nbytes

This parameter specifies the maximum number of write bytes. The buffer must have a size of 1 byte.

EXAMPLE

```
int           fd;
unsigned char outVal = 0x12;
int           retval;

…

/*-------------------------------------
  Set output value of the TIP670 device
  -------------------------------------*/
retval = write(fd, &outVal, 1);
if (retval == 1)
{
     printf("output succesfully set \n");
}
else
{
     /* handle the write error */
}

…
```

RETURNS

Number of bytes written (1) or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual).

SEE ALSO

ioLib, basic I/O routine - write()

4.5 ioctl()

NAME

ioctl() - performs an I/O control function.

SYNOPSIS

#include “tip670.h”

```
int ioctl
(
        int     fd,
        int     request,
        int     arg
)
```

DESCRIPTION

Special I/O operation that do not fit to the standard basic I/O calls (read, write) will be performed by calling the ioctl() function.

PARAMETER

fd

This file descriptor specifies the device to be used. The file descriptor has been returned by the *open()* function.

request

This argument specifies the function that shall be executed. Following functions are defined:

Function	Description
t670_EVREAD	wait for event and read the input lines
t670_WDENABLE	enable watchdog
t670_WDDISABLE	disable watchdog
t670_WDTRIGGER	trigger watchdog
t670_CTREAD	read the value of a timer
t670_CTSTART	start C/T with special options
t670_CTSTOP	stop C/T in continuous mode

arg

This parameter depends on the selected function (request). How to use this parameter is described below with the function.

RETURNS

OK or ERROR. If the function fails an error code will be stored in *errno*.

ERROR CODES

The error code can be read with the function *errnoGet()*.

The error code is a standard error code set by the I/O system (see VxWorks Reference Manual).

SEE ALSO

ioLib, basic I/O routine - ioctl()

4.5.1 t670_EVREAD

This I/O control function reads the value of the input lines after a special event is detected by the driver. The function specific control parameter **arg** is a pointer on a *t670_EVRD_PAR* structure.

A special characteristic of the CIO Z8536 which is used on the TIP670 is the functionality of the input port. In addition to the normal port read function, the driver can wait for a special event, before the input port is read. The event can be specified by a special bit pattern of the input port or by any transition of an input line. The input port is used for the purpose as bit port in OR mode (see also the Technical Manual Z8536 CIO which is part of the TIP670-ED Engineering Documentation).

```
typedef struct
{
        int             mode;
        unsigned short  mask;
        unsigned short  match;
        unsigned short  value;
        unsigned short  timeout;
} t670_EVRD_PAR;
```

mode

This parameter specifies the kind of event to wait for. The following events are defined an valid.

Event	Description
t670_MATCH	The I/O request terminates, if all bits which are masked in the call parameter *mask* have the state defined in the parameter *match*. **Note: Match events may get lost if the matching state is only valid for a short while.**
t670_HIGH_TR	The I/O request terminates, if a single input which is specified in the call parameter *mask* has a low to high transition. The parameter *match* is don't care. **Note: Only one bit shall be selected in this mode.**
t670_LOW_TR	In this mode the I/O request terminates, if a single input which is specified in the call parameter *mask* has a high to low transition. The parameter *match* is don't care. **Note: Only one bit shall be selected in this mode.**
t670_ANY_TR	In this mode the I/O request terminates if a single input which is specified in the call parameter *mask* has any transition. The parameter *match* is don't care. **Note: Only one bit shall be selected in this mode.**

mask

This parameter masks the input lines that shall be used to detect the event. Bit 0 masks input line 1, bit 1 masks input line 2 and so on.
Note: masks set for transition events shall only mask one bit.

match

If the masked bits match to the actual input value the function will terminate. Bit 0 specifies input line 1, bit 1 specifies input line 2 and so on.
This parameter is used for match events, for other events it is not used.

value

This value returns the state of input lines.
Note: The value is read in the interrupt service function and may not represent the value at the moment the event has occurred.

timeout

This value specifies the maximum time to wait for the event. The unit of this parameter is milliseconds.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
t670_EVRD_PAR      evBuf;
int                retval;

…

/*-------------------------------------------------------
  Wait until the input port is set to >>>1000 XXXX<<< or
  timeout after 10 seconds
  -------------------------------------------------------*/
evBuf.mode =       t670_MATCH;
evBuf.mask =       0xF0;
evBuf.match =      0x80;
evBuf.timeout =    10000;

retval = ioctl(fd, t670_EVREAD, (int)&evBuf);
if (retval != ERROR)
{
     /* function succeeded, event occurred */
}
else
{
     /* handle the error, or timeout */
}

…
```

```
…

/*-----------------------------------------
  Wait until bit 3 (input 4) has a low to high
  transition or timeout after 5 seconds
  -----------------------------------------*/
evBuf.mode =       t670_HIGH_TR;
evBuf.mask =       (1<<3);
evBuf.timeout =    5000;

retval = ioctl(fd, t670_EVREAD, (int)&evBuf);
if (retval != ERROR)
{
     /* function succeeded, event occurred */
}
else
{
     /* handle the error, or timeout */
}

…
```

ERROR CODES

Error code	Description
S_t670_Drv_NO_TR_SEL	No input line selected
S_t670_Drv_ILLEVRDMODE	Unknown event specified
S_t670_Drv_NO_FREE_JOB	There is no free job (increase TIP670_MAX_JOBS)

4.5.2 t670_WDENABLE

This I/O control function enables the watchdog of the TIP670 output port. If the TIP670 module is not triggered during the specified time, the output port of the TIP670 module is disabled. It can be enabled again with a write access or a call of *t670_WDTRIGGER* to the device. The function specific control parameter **arg** specifies the value of the watchdog timer in steps of 0,5µs. Allowed values are 0..65535.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
int                retval;

…

/*-----------------------------------------
  enable the watchdog with a timeout of 10 ms
  -----------------------------------------*/
retval = ioctl(fd, t670_WDENABLE, 20000);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}

…
```

4.5.3 t670_WDDISABLE

This I/O control function disables the watchdog of the TIP670 output port. The function specific control parameter **arg** is not used.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
int                retval;

…

/*--------------------
  disable the watchdog
  --------------------*/
retval = ioctl(fd, t670_WDDISABLE, 0);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}

…
```

4.5.4 t670_WDTRIGGER

This I/O control function triggers the watchdog of the TIP670 output port. The function specific control parameter **arg** is not used.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
int                retval;

…

/*--------------------
  trigger the watchdog
  --------------------*/
retval = ioctl(fd, t670_WDTRIGGER, 0);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}

…
```

4.5.5 t670_CTREAD

This I/O control function reads the current value of a counter/timer. The function specific control parameter **arg** is a pointer on a *t670_CTRD_PAR* structure.

```
typedef struct
{
      int               timer;
      unsigned short    value;
} t670_CTRD_PAR;
```

timer

This value selects the counter/timer that will be read. Allowed values are 1 and 2.

value

This parameter returns the current value of the selected counter/timer.

EXAMPLE

```
#include “tip670.h”

…

int               fd;
t670_CTRD_PAR     ctBuf;
int               retval;

…

/*---------------------------------
  Read the value of counter/timer 2
  ---------------------------------*/
ctBuf.timer = 2;

retval = ioctl(fd, t670_CTREAD, (int)&ctBuf);
if (retval != ERROR)
{
     /* function succeeded */
     printf(“CT(%d)-value: %d\n”, ctBuf.timer, ctBuf.value);
}
else
{
     /* handle the error */
}

…
```

ERROR CODES

Error code	Description
S_t670_Drv_BAD_CT_NUM	The specified number of the counter / timer is not valid

4.5.6 t670_CTSTART

This I/O control function configures and starts a counter/timer. The function specific control parameter **arg** is a pointer on a *t670_CTST_PAR* structure. The counter / timer 1 and 2 can operate in different modes which are selected through the parameter structure (see the Technical Manual Z8536 CIO for more details).

```
typedef struct
{
        int             timer;
        int             options;
        int             *softcnt;
        int             count;
        int             timeout;
        SEM_ID          semaphore;
} t670_CTST_PAR;
```

timer

This value selects the counter/timer to start. Allowed values are 1 and 2.

options

This field specifies counter timer mode. The following flags can be ORed to select the mode:

Flag	Description
t670_CONTINUOUS	If this flag is set the specified counter / timer will work in continuous mode. If the counter reaches the value of 1 the next clock pulse will reload the counter with its initial value which is set with the parameter. If this mode is selected and the parameter semaphore is not NULL every reload will send a signal to the specified semaphore. A timer running in continuous mode has to be stopped with the *t670_CTSTOP* function.
t670_SOFTCOUNT	This flag is only valid if the continuous mode is selected. Otherwise this flag will be ignored. If the device works in softcount mode, the driver counts the reloads of the counter/timer. For this mode the parameter softcount must be filled with the address of a counter in memory (int).
t670_EXTCOUNT	This flag selects the clock input of the counter/timer. If the flag is set the clock is taken from an input. Otherwise the counter/timer is clocked internal with a period of 0.5µs. **Note: TIP670-20 supports external counter/timer signals for C/T 2 only.**
t670_EXTTRIGGER	This flag selects an input as trigger for the counter/timer which starts the counter / timer. **Note: TIP670-20 supports external counter/timer signals for C/T 2 only.**
t670_EXTGATE	This flag selects an input as gate for the counter/timer. If the flag is not set the counter/timer isn't gated. **Note: TIP670-20 supports external counter/timer signals for C/T 2 only.**

The external C/T input assignments:

Timer	Count	Trigger	Gate	Supported at
1	(1<<5)	(1<<6)	(1<<7)	TIP670-10
2	(1<<1)	(1<<2)	(1<<3)	TIP670-10/-20

softcnt

This parameter must point to a value, where counter/timer reloads are counted. This pointer is only used if the option *t670_SOFTCOUNT* is specified.

count

This parameter specifies the initial counter value.

timeout

This parameter selects the time, the I/O request has to wait before it times out. The unit of this parameter is milliseconds.

semaphore

This value specifies the semaphore ID that shall be signalled on counter/timer reloads. If no semaphore is used the value must be set to *NULL*.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
t670_CTST_PAR      ctStBuf;
SEM_ID             sem_x_ID;
int                retval;
int                event_counter;

…

/*-----------------------------------------
  Wait for 5 milliseconds (use counter/timer 1)
  -----------------------------------------*/
ctStBuf.timer =    1;                   /* timer 1 */
ctStBuf.options =  0;                   /* no special function */
ctStBuf.count =    (2000 * 5);          /* 5ms = (2000 *0.5µs) * 5 */
ctStBuf.timeout =  (ctStBuf.count * 2); /* must be greater than C/T time */
ctStBuf.semaphore = NULL;

…
```

```
…

retval = ioctl(fd, t670_CTSTART, (int)&ctStBuf);
if (retval != ERROR)
{
     /* C/T started, function succeeded */
}
else
{
     /* handle the error, or timeout */
}

…

/*-----------------------------------------------------------------
  send a 3ms signal to sem_x, count the events, use counter/timer 2
  -----------------------------------------------------------------*/
event_counter =    0;
sem_x_ID = semBCreate(…);
…

ctStBuf.timer =    2;                 /* timer 2 */
ctStBuf.options =  t670_CONTINUOUS | t670_SOFTCOUNT;
ctStBuf.count =    (2000 * 3);        /* 3ms = (2000 *0.5µs) * 3 */
ctStBuf.semaphore = sem_x_ID;
param.softcount =  &event_counter;    /* set counter address */

retval = ioctl(fd, t670_CTSTART, (int)&ctStBuf);
if (retval != ERROR)
{
     /* C/T started, function succeeded */
}
else
{
     /* handle the error, or timeout */
}

…
```

ERROR CODES

Error code	Description
S_t670_Drv_BAD_CT_NUM	The specified number of the counter/timer is not valid
S_t670_Drv_CT_BUSY	The counter/timer is busy, use t670_CTSTOP first.

4.5.7 t670_CTSTOP

This I/O control function stops the specified counter/timer. The function specific control parameter **arg** specifies the counter/timer. Allowed values are 1 and 2.

EXAMPLE

```
#include “tip670.h”

…

int                fd;
int                retval;

…

/*--------------------
  Stop counter/timer 2
  --------------------*/
retval = ioctl(fd, t670_CTSTOP, 2);
if (retval != ERROR)
{
     /* function succeeded */
}
else
{
     /* handle the error */
}

…
```

ERROR CODES

Error code	Description
S_t670_Drv_BAD_CT_NUM	The specified number of the counter / timer is not valid

5 Appendix

5.1 Additional Error Codes

Error code	Error value	Description
S_t670Drv_NO_END	0x06700001	There is no END marker in the configuration table or the number of entries exceeds the number of MAX_MODULES.
S_t670Drv_SM_BUFFER	0x06700002	The return buffer is too small.
S_t670Drv_ILLREQUEST	0x06700003	The request code is unknown.
S_t670Drv_ILLEVRDMODE	0x06700004	The event read mode is unknown.
S_t670Drv_NO_TR_SEL	0x06700005	No input line is selected for a transition.
S_t670Drv_NO_FREE_JOB	0x06700006	The number of predefined active jobs has been exceeded.
S_t670Drv_BAD_MEM	0x06700007	A driver memory structure has been destroyed. This is a fatal error. The driver has to be installed again.
S_t670Drv_BAD_CT_NUM	0x06700008	The specified number of the counter/timer is not valid.
S_t670Drv_CT_BUSY	0x06700009	The selected counter/timer is busy.