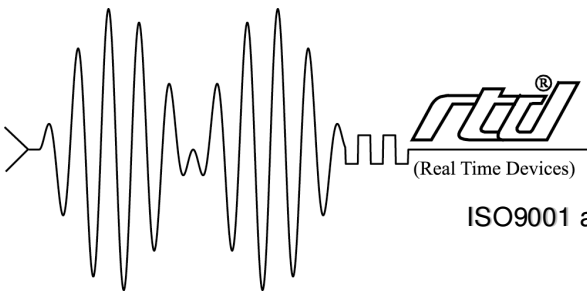
ECAN527/ECAN1000/CMK6486DX
CAN Bus Interface Driver for Linux

Driver Version 3.0.0 User's Manual



RTD Embedded Technologies, Inc.
(Real Time Devices) *"Accessing the Analog World"®*
ISO9001 and AS9100 Certified

SWM-640020007
rev C

RTD Embedded Technologies, INC.
103 Innovation Blvd.
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

02/12/2004	Revision A issued Documented for ISO9000
09/30/2004	Revision B issued Cleaned up trademark references Added hardware/software diagram to “Introduction” section Added “Notational Conventions” section Rewrote “Extracting the Software” section Updated all driver versions from 2.0 to 2.1.x Documented need to have kernel source installed to build driver Mentioned warnings generated by insmod when driver loaded Removed obsolete example program descriptions Added description of new example programs Added documentation for bus bit rate and ECAN1000 filter functions Updated rtd_ecan_get_driver_version() documentation to reflect new driver version encoding Added “Message Filtering” section
04/16/2010	Updating for API changes in version 3.0.0

ECAN 527/1000 Driver for Linux
Published by:

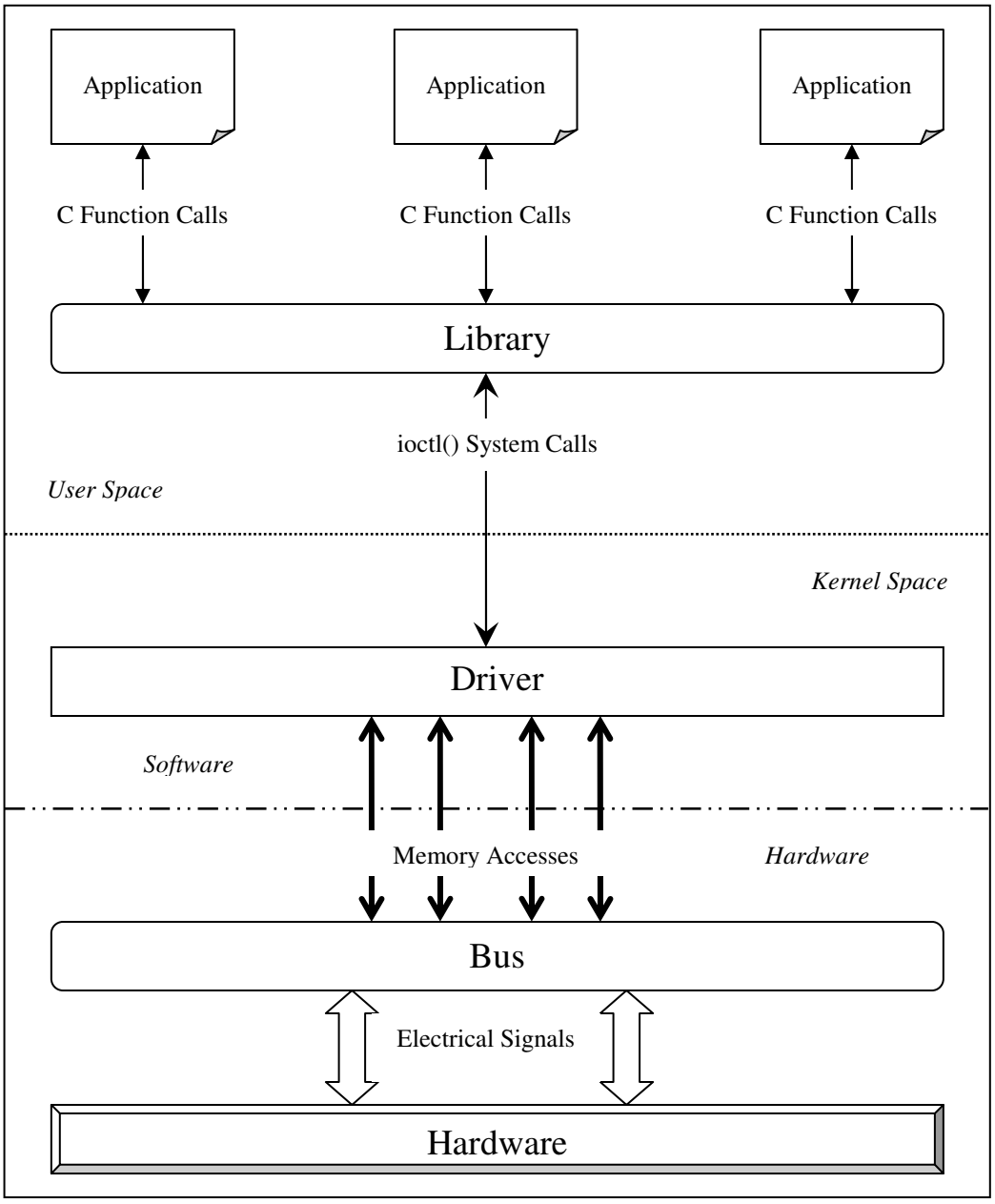
RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College, PA 16803-0906

Table of Contents

Introduction

This document targets anyone wishing to write Linux applications for an RTD ECAN527 or ECAN1000 board. It provides information on building the software and using the Application Programming Interface to communicate with the hardware and drivers. Each high-level library function is described as well as any low-level ioctl() system call interface it may make use of.

The diagram below 1) provides a general overview of what hardware and software entities are involved in device access, 2) shows which units communicate with each other, and 3) illustrates the methods used to transfer data and control information.



Notational Conventions

RTD Linux drivers are assigned version numbers. These version numbers take the form "A.B.C" where:

* A is the major release number. This will be incremented whenever major changes are made to the software. Changing the major release number requires updating the software manual.
* B is the minor release number. This will be incremented whenever minor, yet significant, changes are made to the software. Changing the minor release number requires updating the software manual.
* C is the patch level number. This will be incremented whenever very minor changes are made to the software. Changing the patch level number does not require updating the software manual.

Occasionally you will notice text placed within the < and > characters, for example <installation path>. This indicates that the text represents something which depends upon choices you have made or upon your specific system configuration.

Installation Instructions

Extracting the Software

All software comes packaged in a gzip'd tar file named Ecan_Linux_V03.00.00.tar.gz. First, decide where you would like to place the software. Next, change your current directory to the directory in which you have chosen to install the software by issuing the command "cd installation path>". Then, extract the software by issuing the "tar -xvzf <path to tar file>/Ecan_Linux_V03.00.00.tar.gz" command; this will create a directory Ecan_Linux_V03.00.00/ that contains all files comprising the software package.

Contents of Installation Directory

Once the tar file is extracted, you should see the following files and directories within Ecan_Linux_V03.00.00/:

driver/
examples/
include/
lib/
CHANGES.TXT
LICENSE.TXT
README.TXT

The file CHANGES.TXT describes the changes made to the software for this release, as well as for previous releases. The file LICENSE.TXT provides details about the RTD end user license agreement which must be agreed to and accepted before using this software. The file README.TXT contains a general overview of the software and contact information should you experience problems, have questions, or need information. The directory driver/ contains the source code and Makefile for the drivers. The directory examples/ holds the source code and Makefile for the example programs. The directory include/ contains all header files used by the driver, example programs, library, and your application programs. Library source code and Makefile reside in the directory lib/.

Building the Driver

Driver source code uses files located in the kernel source tree. Therefore, you must have the full kernel source tree available in order to build the driver. The development system, which provides a full compilation environment, must be running the exact same version of the kernel as your production machine(s); otherwise the kernel module may not load or may load improperly. After the code is built, you can then move the resulting object files, libraries, and executables to the production system(s).

Building the driver consists of several steps: 1) compiling the source code, 2) loading the resulting kernel module into the kernel, and 3) creating hardware device files in the /dev directory. To perform any of the above steps, you must change your current directory to driver/. The file Makefile contains rules to assist you.

To compile the source code, you need to know which driver you want to build. If the ECAN527 driver is desired, issue the command “make MODEL=527”. Use the command “make MODEL=1000” to compile the ECAN1000 driver. Simply issuing the “make” command will result in usage information. The GNU C compiler gcc is used to build the driver code.

Before the driver can be used, it must be loaded into the currently running kernel. Using the command “make insmod527” will load the ECAN527 driver into the kernel using the board default I/O address and IRQ jumper values for an ECAN527D (0xD0000, 0xD0100 and IRQ5,IRQ11). If you have special requirements, you may need to manually issue an insmod command. The command “make insmod1000” will load the ECAN1000 driver into the kernel; this target assumes that a single ECAN1000 is installed and that its base I/O address and IRQ are set to the factory defaults (0x300 and IRQ5 respectively). You may need to edit the Makefile and change this rule to reflect your board configuration or manually issue an appropriate insmod command.

The final step is to create /dev entries for the hardware. Driver versions prior to 2.0 did this automatically whenever the driver was loaded into the kernel. Due to changes in the Linux kernel, this is no longer possible. Instead, the device files must be created manually. Use the command “make devices527” to create ECAN527 device files; this generates ten files in /dev named rtd-ecan527-0 through rtd-ecan527-9. Type the command “make devices1000” to make ECAN1000 device entries; this creates ten files in /dev name rtd-ecan1000-0 through rtd-ecan1000-9. The driver object file for the ECAN527 is named rtd-ecan527.o or rtd-ecan527.ko for 2.4 kernel and 2.6 kernel respectively. The ECAN1000 it is called rtd-ecan1000.o or rtd-ecan1000.ko for 2.4 kernel and 2.6 kernel respectively. The ECAN527 driver will also work with the CMK6486DX board.

When you load either kernel driver in 2.4 kernel, the insmod command may print the following message:

"Warning: loading ./<driver>.o will taint the kernel: no license"

where <driver> is either rtd-ecan1000 or rtd-ecan527. You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

Building the Library

The example programs and your application use the ECAN library, so it must be built before any of these can be compiled. To build the library, change your current directory to lib/ and issue the command “make”. The GNU C++ compiler g++ is used to compile the library source code. To prevent compatibility problems, any source code which makes use of library functions should also be built with g++.

The ECAN library is statically linked and is created in the file librtd-ecan.a.

Building the Example Programs

The example programs may be compiled by changing your current directory to examples/ and giving the command “make”, which builds all the example programs. If you wish to compile a subset of example programs, there are targets in Makefile to do so. For example, the command “make rtd-ecan-read rtd-ecan-write” will compile and link the source files rtd-ecan-read.cc and rtd-ecan-write.cc. The GNU C++ compiler g++ is used to compile the example program code.

Message Filtering

Utilizing message filters can be one of the most complex facets of CAN bus operation. Both the ECAN527 and ECAN1000 implement hardware filtering. The following sections shed some light on using filters with both boards.

ECAN527 Message Filtering

The ECAN527 contains fifteen message objects. An object can be thought of as a mailbox for receiving or transmitting messages. Message objects one through fourteen can be configured to send or receive messages. Message object fifteen can only receive messages. Within each message object exists a bit pattern which will be compared to the message ID of any message. If the bit patterns match, the object will receive a message or, if configured to reply to a remote frame, the object will transmit a message.

Three masks indicate which message ID bits in each message object are considered or ignored when making message comparisons. These masks are the 1) Global Mask - Standard Register, 2) Global Mask - Extended Register, and 3) Message 15 Mask Register.

The Global Mask - Standard Register applies to standard frame messages and to objects configured to receive or send such messages. The Global Mask - Extended Register applies to extended frame messages and to objects configured to receive or send such messages. The Message 15 Mask Register applies to both standard and extended frame messages; in addition, it applies only to message object fifteen.

As stated previously, mask bits control which message object ID bits affect comparisons. A 0 bit causes any bit value in the corresponding bit position of the incoming message to be accepted. A 1 bit indicates that the bit value in the corresponding bit position of the incoming message must exactly match the object's message ID bit.

Consider the following filtering example. Suppose you wish to have message object one accept odd-numbered message IDs and message object fifteen accept even-numbered message IDs for standard frame messages. Using one of the message object configuration functions, set message object one to receive standard frames and set its message ID to 1. Set message object fifteen to receive standard frames and set its message ID to 0. Make sure all other objects are disabled. The following code fragment will set up filters to implement this scheme:

```
ECAN_FILTER_STRUCTURE filter;

/*
 * Bit zero in message object one must match exactly.  All other bits are
 * ignored.  In this case, bit zero must be 1.
 */

filter.SetAcceptMask(0x1, false);

/*
 * Bit zero in message object fifteen must match exactly.  All other bits
 * are ignored.  In this case, bit zero must be 0.
 */

filter.SetMessage15Mask(0x1, false);

Ecan_SetFilter(handle, &filter);
```

As another example, suppose you wish to have message object two accept message ID 100 and message object fifteen accept message ID 333 for extended frames. Using one of the message object configuration functions, set message object two to receive extended frames and set its message ID to 100. Set message object fifteen to receive extended frames and set its message ID to 333. Make sure all other objects are disabled. The following code fragment will set up filters to implement this scheme:

```
ECAN_FILTER_STRUCTURE filter;

/*
 * All bits in message object two must match exactly.  In this case, the
 * message ID must be 100.
 */

filter.SetExtended(0x1FFFFFFF);

/*
 * All bits in message object fifteen must match exactly.  In this case,
 * message ID must be 333.
 */

filter.SetMessage15Mask(0x1FFFFFFF, true);

Ecan_SetFilter(handle, &filter);
```

For more information regarding message objects and message filtering, please see the Intel Corporation document 82527 Serial Communications Controller Architectural Overview.

ECAN1000 Message Filtering

The ECAN1000 implements four filtering strategies: 1) dual filter mode on extended frame messages, 2) dual filter mode on standard frame messages, 3) single filter mode on extended frame messages, and 4) single filter mode on standard frame messages.

Filters control which message bits are considered or ignored when making comparisons. A 1 bit causes any bit value in the appropriate bit position of the incoming message to be accepted. A 0 bit indicates that the bit value in the appropriate bit position of the incoming message must exactly match.

The library function Ecan_SetDualFilterExtended() is used to set a dual mode filter for extended frames. Consider the following filtering example. Suppose you wish to accept messages with the most significant 16 bits of the message ID equal to 0xC000 or 0x3000; this is equivalent to accepting messages with IDs in the range [0x18000000 .. 0x18001FFF] or in the range [0x06000000 .. 0x06001FFF]. The corresponding function call to set this filter would be:

```
Ecan_SetDualFilterExtended(handle, 0xC000, 0x0, 0x3000, 0x0);
```

The library function Ecan_SetDualFilterStandard() is used to set a dual mode filter for standard frames. Consider the following filtering example. Suppose you wish to accept messages having a message ID of 10 or 20 but the Remote Transmission Request bit and the first data byte don't matter. The corresponding function call to set this filter would be:

```
Ecan_SetDualFilterStandard(
       handle, 0xA, 0x0, 0x14, 0x0, 0x1, 0x1, 0x1, 0x1, 0xFF, 0xFF
);
```

The library function Ecan_SetSingleFilterExtended() is used to set a single mode filter for extended frames. Consider the following filtering example. Suppose you wish to accept messages that have an odd-numbered message ID and a Remote Transmission Request bit of 0. The corresponding function call to set this filter would be

```
Ecan_SetSingleFilterExtended(handle, 0x1, 0x1FFFFFFE, 0x0, 0x0);
```

The library function Ecan_SetSingleFilterStandard() is used to set a single mode filter for standard frames. Consider the following filtering example. Suppose you wish to accept messages that have any message ID, any Remote Transmission Request bit, and whose first two data bytes are 0xFFFF. The corresponding function call to set this filter would be

```
Ecan_SetSingleFilterStandard(
        handle, 0x0, 0x7FF, 0x0, 0x1, 0xFFFF, 0x0
);
```

For more information regarding message filtering, please see the Phillips Semiconductors document SJA1000 Stand-alone CAN Controller Product Specification.

Using the API Functions

ECAN hardware and the associated driver functionality can be accessed through the library API (Application Programming Interface) functions. Applications wishing to use library functions must include the include/ecanlib.h header file and be statically linked with the lib/librtd-ecan.a library file.

The following function reference provides for each library routine a prototype, description, explanation of parameters, and return value or error code. By looking at a function's entry, you should gain an idea of: 1) why it would be used, 2) what it does, 3) what information is passed into it, 4) what information it passes back, 5) how to interpret error conditions that may arise, and 6) the ioctl() system call interface if the function makes use of a single ioctl() call. To obtain more information about the structures used in the library functions, please consult the files include/ecanbaseioctl.h and include/ecanioctl.h.

Note that errno codes other than the ones indicated in the following pages may be set by the library functions. Please see the ioctl(2) man page for more information.

Function Reference

API Function Groups

Board Control

Ecan_BusConfig
Ecan_GetBuffer
Ecan_SendCommand
Ecan_SetBitRate
Ecan_SetBuffer
Ecan_SetDualFilterExtended
Ecan_SetDualFilterStandard
Ecan_SetFilter
Ecan_SetLeds
Ecan_SetSingleFilterExtended
Ecan_SetSingleFilterStandard
Ecan_SetupBoard
Ecan_StartBoard
Ecan_StopBoard

Digital I/O

Ecan_LoadPortBitDir
Ecan_ReadDigitalIO
Ecan_WriteDigitalIO

General

Ecan_CreateHandle
Ecan_GetBoardName
Ecan_TestBoard
Ecan_Clear_Accounts
Ecan_Get_Accounts
Ecan_Set_TX_Queue_Size
Ecan_Set_RX_Queue_Size
EncodeMessageID
Ecan_GetInterrupts
Ecan_GetMessage
Ecan_GetStatus
Ecan_MessageObjectSetup
Ecan_SendMessage

Receive and Transmit Queue

Ecan_AllowBufferOverwrite
Ecan_GetQueuesCounts

Alphabetical Function Listing

Ecan_AllowBufferOverwrite

int Ecan_AllowBufferOverwrite(int handle, bool allow);

Description:

Inform the driver how to process receive queue overruns.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

allow: Flag to indicate whether or not to overwrite receive queue contents on overrun. A value of false means do not overwrite the oldest queue message with a new message and instead try to put an special overrun message in the queue. A value of true means overwrite the oldest queue message with a new message and do not put a special overrun message in the queue.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

IOCTL Interface:

```
int rc;

/*
 * Don't allow receive queue overwrites when buffer is full.  Try to put overrun message in receive
 * queue instead.
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__ALLOW_BUFFER_OVERWRITE, 0);

/*
 * Allow receive queue overwrites when buffer is full.  No overrun message is put in receive
 * queue.  If the buffer is full, the oldest received message will be overwritten.
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__ALLOW_BUFFER_OVERWRITE, 0xff);
```

```
int Ecan_BusConfig(
   int handle,
   u8_t BusTiming0,
   u8_t BusTiming1,
   u8_t ClockOut = 0,
   u8_t BusConfig = 0xff
);
```

Description:

Set CAN timing and bus configuration. This function puts the board in reset mode, so you must start the board afterward.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
BusTiming0:	Device specific value for bus timing register 0.
BusTiming1:	Device specific value for bus timing register 1.
ClockOut:	Device specific value for frequency divider at the external CLKOUT pin relatively to the frequency of the external oscillator. A value of 0 means don't change. The The default value is 0.
BusConfig:	Device specific value for Output Control Register (ECAN1000) or Bus Configuration Register (ECAN527). A value of 0xFF means set the bus to the default configuration. The default value is 0xFF.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
: handle is not a valid file descriptor.

EINVAL
: Configuration value is not valid.

EIO
: The driver was unable to turn on the Reset Mode bit in the Mode Register to reconfigure the board (ECAN1000 only).

IOCTL Interface:

```
int rc;
struct __rtd_ecan_ioctl_busconfig bus_config;
```

```
memset((void *) &bus_config, 0, sizeof(struct __rtd_ecan_ioctl_busconfig));

/*
 * Set CAN bus to default configuration
 */

bus_config.BusConfig = 0xff;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_BUS_CONFIG, &bus_config);
```

Ecan_CreateHandle

---

int Ecan_CreateHandle(size_t DevNum = 0, bool Ecan1000 = false);

Description:

Open an ECAN device so that other functions may be called for it.

Parameters:

DevNum:	Board minor number. The default value is 0.
Ecan1000:	Selects board type. The value false indicates that the device is an ECAN527. The value true indicates that the device is an ECAN1000. The default value is false.

Return Value:

>= 0:	Success. The integer returned is the file descriptor from open() system call.
-1:	Failure. Please see the open(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

None.

Ecan_GetBoardName

---

int Ecan_GetBoardName(int handle, unsigned long *board_type);

Description:

Get an ECAN board's type.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

board_type: Address in which to store the board type value.

Return Value:

0: Success.

-1 Failure with errno set as follows:
EBADF
handle is not a valid file descriptor.

IOCTL Interface:

```
int rc;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__GET_BOARD_NAME);
```

Ecan_GetBuffer

```
size_t Ecan_GetBuffer(
   int handle,
   size_t StartAddress,
   size_t Count,
   void *Buffer_p,
   size_t BuffSize,
   uint8_t *BytesRead
);
```

Description:

Read CAN controller's RAM area into user buffer. This can be used to examine particular controller registers or see how a board is configured.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

StartAddress: Offset from beginning of RAM area.

Count: Number of bytes to read from RAM area.

Buffer_p: Address of buffer in which to place RAM data.

BuffSize: Size of buffer in bytes pointed to by Buffer_p.

BytesRead: The integer returned is the number of bytes read from board's RAM area.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
buffer_p is not a valid user address.

EFAULT
The buffer is not big enough to store the data read.

EINVAL
ram_offset is outside the board's RAM area.

EINVAL
ram_len is outside the board's RAM area.

EINVAL
ram_offset plus ram_len is outside the board's RAM area.

IOCTL Interface:

```
char ram_buffer[1];
int rc;
struct __rtd_ecan_ioctl_get_ram get_arguments;

/*
 * Read contents of Control Register on ECAN1000
 */

get_arguments.ram_offset = 0x00;
get_arguments.ram_length = 1;
get_arguments.user_buffer = (void *) &ram_buffer;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__GET_RAM, &get_arguments);
```

Ecan_GetInterrupts

```
uint Ecan_GetInterrupts(
   int handle,
   ulong *QueueSize_p = NULL,
   bool DontQueueUse = false
);
```

Description:

Prepare a received message for subsequent library calls. This function can operate either on the driver's receive queue or on the board directly.

NOTE: If this function is used to operate on the receive queue, it will remove the first available message from that queue. In addition, the queue count stored in the memory address referred to by QueueSize_p represents the number of entries remaining in the receive queue after removing the message.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
QueueSize_p:	Address where to store number of messages in driver's receive queue. If this is NULL, do not return this information. The default value is NULL.
DontQueueUse:	Flag to indicate whether or not the receive queue should be bypassed. A value of true means go to the board for a message. A value of false means use the driver's receive queue. The default value is false.

Return Value:

0:	Failure. Please see the description of rtd_ecan_prepare_received_message() for information on possible values errno may have in this case.
> 0:	Success. The unsigned integer returned is a mask of pending events.

IOCTL Interface:

```
int rc;
struct __rtd_ecan_ioctl_prepare_received_message message;

/*
 * Prepare message using driver's receive queue
 *
 * Note: if no message is waiting, both message.rx_queue_count and message.events are set to 0
 */

rc = ioctl(
   descriptor,
   __RTD_ECAN_IOCTL__PREPARE_RECEIVED_MESSAGE,
   &message
);

/*
 * Prepare message by going directly to the board
 *
 * Note: on the ECAN527, message.rx_queue_count is always set to 0
 */

rc = ioctl(
   descriptor,
   (
         __RTD_ECAN_IOCTL__PREPARE_RECEIVED_MESSAGE
         |
         __RTD_ECAN_DONT_USE_QUEUE
   ),
   &message
```

);

Ecan_GetMessage

int Ecan_GetMessage(int handle, ECAN_MESSAGE_STRUCTURE *message_p);

Description:

Retrieve a received message. This function can operate either on the driver's receive queue or on the board directly.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

message_p: Address of structure where message should be written. The DontQueueUse member is used to indicate whether or not the receive queue should be bypassed. A value of true in DontQueueUse means go to the board. A value of false in DontQueueUse means use the receive queue.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EBADSLT
The Channel member in the structure pointed to by message_p is zero and dont_use_queue is nonzero (ECAN527 only).

EFAULT
message_p is not a valid user address.

EINVAL
The Channel member in the structure pointed to by message_p is not valid and dont_use_queue is nonzero (ECAN527 only).

EPERM
The Reset Mode bit is set in the Mode Register and dont_use_queue is nonzero (ECAN1000 only).

EPERM
The Initialization bit is set in the Control Register and dont_use_queue is nonzero (ECAN527 only).

IOCTL Interface:

```
int rc;
struct rtd_ecan_message message;

/*
 * Get message out of driver's receive queue
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__RECV_MESSAGE, &message);

/*
 * Get message by going directly to the board
 */

rc = ioctl(
   descriptor,
   (__RTD_ECAN_IOCTL__RECV_MESSAGE | __RTD_ECAN_DONT_USE_QUEUE),
   &message
);
```

Ecan_GetQueuesCounts

```
int Ecan_GetQueuesCounts(
   int handle,
   ulong *TX_Count_p,
   ulong *RX_Count_p,
   bool ClearRX = false,
   bool ClearTX = false
);
```

Description:

Optionally get current driver receive and transmit queue message counts. Optionally clear driver receive and transmit queues.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
TX_Count_p:	Address where to store the number of messages in the driver's transmit queue.
RX_Count_p:	Address where to store the number of messages in the driver's receive queue.

A value of NULL for either address above indicates that the corresponding information should not be returned.

ClearRX:	Flag to indicate whether or not the receive queue should be cleared.
ClearTX:	Flag to indicate whether or not the transmit queue should be

cleared.

A value of false for either flag above signifies that the corresponding queue should not be cleared. A value of true means that the corresponding queue should be cleared. The default value for each flag is false.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
rx_count_p is not a valid user address.

EFAULT
tx_count_p is not a valid user address.

Ecan_GetStatus

---

int Ecan_GetStatus(int handle, ECAN_STATUS_STRUCTURE *status_p);

Description:

Get status of current message. This function can operate either on the driver's receive queue or on the board directly.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

status_p: Address of structure where status information should be written. The DontQueueUse member is used to indicate whether or not the receive queue should be bypassed. A value of true in DontQueueUse means go to the board. A value of false in DontQueueUse means use the receive queue.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
status_p is not a valid user address.

EPERM
The Initialization bit is set in the Control Register and dont_use_queue is nonzero (ECAN527 only).

IOCTL Interface:

```
int rc;
struct rtd_ecan_status status;

/*
 * Get message status using driver's receive queue
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__GET_STATUS, &status);

/*
 * Get message status by going directly to the board
 */

rc = ioctl(
   descriptor,
   (__RTD_ECAN_IOCTL__GET_STATUS | __RTD_ECAN_DONT_USE_QUEUE),
   &status
);
```

Ecan_LoadPortBitDir

```
int Ecan_LoadPortBitDir(
   int handle,
   bool bit7 = false,
   bool bit6 = false,
   bool bit5 = false,
   bool bit4 = false,
   bool bit3 = false,
   bool bit2 = false,
   bool bit1 = false,
   bool bit0 = false
);
```

Description:

Program the direction (input or output) of each bit in the digital I/O port. ECAN527 only.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
bit7:	Direction for bit 7.
bit6:	Direction for bit 6.

bit5: Direction for bit 5.

bit4: Direction for bit 4.

bit3: Direction for bit 3.

bit2: Direction for bit 2.

bit1: Direction for bit 1.

bit0: Direction for bit 0.

For bit7 through bit0, a value of false means input and a value of true means output. By default, each bit is set to input.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
direction_p is not a valid user address.

ENOTSUP
Operation is not supported (ECAN1000 only).

IOCTL Interface:

```
int rc;
struct rtd_ecan_load_port_bit_dir bit_mask;
memset((void *) &bit_mask, 0, sizeof(struct rtd_ecan_load_port_bit_dir));

/*
 * Set bits 7, 2, 1, and 0 to output; all others are input
 */

bit_mask.bit7 = 1;
bit_mask.bit2 = 1;
bit_mask.bit1 = 1;
bit_mask.bit0 = 1;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__LOAD_PORT_BIT_DIR, &bit_mask);
```

Ecan_MessageObjectSetup

```
int Ecan_MessageObjectSetup(
   int handle,
   ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE *object_p
```

);

Description:

Set up a message object on an interface. This function can instruct the driver to not process the transmit queue when a message object issues a Transmit Message Successfully interrupt. ECAN527 only.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

object_p: Address of structure where object initialization data is stored. The DontQueueUse member indicates whether or not to process the transmit queue when an Transmit Message Successfully interrupt is generated by a message object. A DontQueueUse value of 0 means process the transmit queue after such an interrupt. Any other DontQueueUse value means do not process the transmit queue after such an interrupt.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

ECHNRG
The Channel member in the structure pointed to by object_p does not refer to a valid object.

EFAULT
object_p is not a valid user address.

EINVAL
The State member in the structure pointed to by object_p does not contain a valid value.

EINVAL
In the structure pointed to by object_p, the Channel member has the value 15, the MakeDefault member is nonzero, and the State member is set to either RTD_ECAN_MO_TRANSMIT or RTD_ECAN_MO_REMOTE_TRANSMIT.

EINVAL
In the structure pointed to by object_p, the Channel member has the value 0 and the State member is set to

ENOTSUP
Operation is not supported (ECAN1000 only).

IOCTL Interface:

```
int rc;
struct rtd_ecan_msg_obj_setup object;

memset((void *) &object, 0, sizeof(struct rtd_ecan_msg_obj_setup));

/*
 * Object can be used to receive messages
 */

object.status = RTD_ECAN_MO_RECEIVE;

/*
 * Enable receive message interrupt for object
 */

object.RXIE = 1;

/*
 * Target of operation is message object 6
 */

object.Channel = 6;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SETUP_MESSAGE_OBJECT, &object);
```

Ecan_ReadDigitalIO

---

int Ecan_ReadDigitalIO(int handle, unsigned char *digital_data_p);

Description:

Read whatever value happens to be currently available on an interface's digital I/O port. ECAN527 only.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
digital_data_p:	Address of user buffer in which to store data.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
digital_data_p is not a valid user address.

ENOTSUP
Operation is not supported (ECAN1000 only).

IOCTL Interface:

```
int rc;
unsigned char digital_data;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__READ_DIGITAL_IO, &digital_data);
```

Ecan_SendCommand

```
int Ecan_SendCommand(
   int handle,
   bool TR = false,
   bool RRB = false,
   bool AT = false,
   bool CDO = false,
   bool SRR = false
);
```

Description:

Send a command to an ECAN device. ECAN1000 only.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
TR:	Flag to send Transmission Request command.
RRB:	Flag to send Release Receive Buffer command.
AT:	Flag to send Abort Transmission command.
CDO:	Flag to send Clear Data Overrun command.
SRR:	Flag to send Self Reception Request command.

A value of false for any of the above command flags indicates that the corresponding command should not be sent. A value of true for any of the above command flags indicates that the corresponding command should be sent. By default, none of the commands are sent.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
command_p is not a valid user address.

ENOTSUP
Operation is not supported (ECAN527 only).

IOCTL Interface:

```
int rc;
struct rtd_ecan_send_command command;

/*
 * Tell device to clear the data overrun status bit
 */

command.TR = 0;
command.RRB = 0;
command.AT = 0;
command.CDO = 1;
command.SRR = 0;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SEND_COMMAND, &command);
```

Ecan_SendMessage

---

int Ecan_SendMessage(int handle, ECAN_MESSAGE_STRUCTURE *message_p);

Description:

Send a message either using the driver's transmit queue or by going directly to the board.

NOTE: Regardless of whether or not message transmission succeeds, the driver removes the message from the transmit queue. If message send fails, you are responsible for retrying the send.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

message_p: Address of structure where message is stored. The DontQueueUse member is used to indicate whether or not the transmit queue should be bypassed. A value of true in DontQueueUse means go to the board. A value of false in DontQueueUse means use the transmit queue.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EBADSLT
The Channel member in the structure pointed to by message_p has the value 15 (ECAN527 only).

EBUSY
The Transmission Buffer Status bit in the Control Register is cleared and dont_use_queue is nonzero (ECAN1000 only).

EFAULT
message_p is not a valid user address.

EINVAL
The Channel member in the structure pointed to by message_p is not valid (ECAN527 only).

ENOBUFS
dont_use_queue is 0 and there is no free slot in the driver's transmit queue.

EPERM
The Reset Mode bit is set in the Mode Register and dont_use_queue is nonzero (ECAN1000 only).

EPERM
The Initialization bit is set in the Control Register and dont_use_queue is nonzero (ECAN527 only).

IOCTL Interface:

```
unsigned char octet;
int rc;
struct rtd_ecan_message message;

memset((void *) &message, 0, sizeof(struct rtd_ecan_message));

/*
 * Fill message with characters '0' through '7'
 */

for (octet = 0; octet < 8; octet++) {
   message.Data[octet] = (unsigned char) (0x30 + octet);
}

/*
 * Send message, queueing it into driver's transmit queue first
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SEND_MESSAGE, &message);
```

```
/*
 * Send message by going directly to the board
 */

rc = ioctl(
   descriptor,
   (__RTD_ECAN_IOCTL__SEND_MESSAGE | __RTD_ECAN_DONT_USE_QUEUE),
   &message
);
```

Ecan_SetBitRate

---

int Ecan_SetBitRate(int handle, BitRates BitRate);

Description:

Set CAN bus bit rate.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
BitRate:	CAN bus bit rate to set. For a list of valid values, please see the BitRates enumeration in include/ecanioctl.h.

Return Value:

0:	Success.
-1:	Failure with errno set as follows: EINVAL BitRate is not valid. Please see the description of Ecan_BusConfig() for information on other possible values errno may have in this case.

IOCTL Interface:

```
int rc;
struct __rtd_ecan_ioctl_busconfig bus_config;

/*
 * Set CAN bus bit rate to 1 megabits/second
 */

bus_config.BusTiming0 = 0;
bus_config.BusTiming1 = 0x14;

/*
 * Don't change CLKOUT pin frequency
```

```
bus_config.ClockOut = 0;

/*
 * Set CAN bus to default configuration
 */

bus_config.BusConfig = 0xFF;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_BUS_CONFIG, &bus_config);
```

Ecan_SetBuffer

```
size_t Ecan_SetBuffer(
   int handle,
   size_t StartAddress,
   size_t Count,
   const void *Buffer_p,
   size_t BuffSize,
   uint8_t *ByteWritten);
```

Description:

Write into CAN controller's RAM area from user buffer. This can be used to write to controller registers and thus control how the board operates

CAUTION: Use this function with care as writing improper values to the board or writing to an incorrect address may cause erratic behavior or may cause the board to lock up. It is strongly recommended that other library functions be used to control an interface.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
StartAddress:	Offset from beginning of RAM area.
Count:	Number of bytes to write into RAM area.
Buffer_p:	Address of buffer which contains data to write.
BuffSize:	Size of buffer in bytes pointed to by Buffer_p.
BytesWritten:	Number of bytes written to the board's RAM area.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
buffer_p is not a valid user address.

EFAULT
More bytes are to be written than are in the buffer.

EINVAL
ram_offset is outside the board's RAM area.

EINVAL
ram_len is outside the board's RAM area.

EINVAL
ram_offset plus ram_len is outside the board's RAM area.

IOCTL Interface:

```
char ram_buffer[1];
int rc;
struct __rtd_ecan_ioctl_set_ram set_arguments;

/*
 * Set Hardware Reset Status bit in ECAN527 CPU Interface Register
 */

ram_buffer[0] = 0x80;
set_arguments.ram_offset = 0x02;
set_arguments.ram_length = 1;
set_arguments.user_buffer = (void *) &ram_buffer;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_RAM, &set_srguments);
```

Ecan_SetDualFilterExtended

```
int Ecan_SetDualFilterExtended(
   int handle,
   uint ID_ACR1,
   uint ID_AMR1,
   uint ID_ACR2,
   uint ID_AMR2
);
```

Description:

Set up a filter for extended frames in dual filter mode. ECAN1000 only.

NOTE: Unless you feel adventurous enough to undertake determining the exact bit patterns to set in the filter structure, do not use Ecan_SetFilter() to set filters on the ECAN1000. Use this function instead.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

ID_ACR1: 16-bit Acceptance Code 1 for message ID.

ID_AMR1: 16-bit Acceptance Mask 1 for message ID.

ID_ACR2: 16-bit Acceptance Code 2 for message ID.

ID_AMR2: 16-bit Acceptance Mask 2 for message ID.

Return Value:

0: Filter set succeeded.

-1: Filter set failed. Please see the description of Ecan_SetFilter() for information on possible values errno may have in this case.

IOCTL Interface:

None.

Ecan_SetDualFilterStandard

```
int Ecan_SetDualFilterStandard(
   int handle,
   uint ID_ACR1,
   uint ID_AMR1,
   uint ID_ACR2,
   uint ID_AMR2,
   uint RTR_ACR1,
   uint RTR_AMR1,
   uint RTR_ACR2,
   uint RTR_AMR2,
   uint Data_ACR,
   uint Data_AMR
);
```

Description:

Set up a filter for standard frames in dual filter mode. ECAN1000 only.

NOTE: Unless you feel adventurous enough to undertake determining the exact bit patterns to set in the filter structure, do not use Ecan_SetFilter() to set filters on the ECAN1000. Use this function instead.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

ID_ACR1: 11-bit Acceptance Code 1 for message ID.

ID_AMR1:	11-bit Acceptance Mask 1 for message ID.
ID_ACR2:	11-bit Acceptance Code 2 for message ID.
ID_AMR2:	11-bit Acceptance Mask 2 for message ID.
RTR_ACR1:	1-bit Acceptance Code 1 for Remote Transmission Request bit.
RTR_AMR1:	1-bit Acceptance Mask 1 for Remote Transmission Request bit.
RTR_ACR2:	1-bit Acceptance Code 2 for Remote Transmission Request bit.
RTR_AMR2:	1-bit Acceptance Mask 2 for Remote Transmission Request bit.
Data_ACR:	8-bit Acceptance Code for first 8 bits of message data.
Data_AMR:	8-bit Acceptance Mask for first 8 bits of message data.

Return Value:

0:	Filter set succeeded.
-1:	Filter set failed. Please see the description of Ecan_SetFilter() for information on possible values errno may have in this case.

IOCTL Interface:

None.

Ecan_SetFilter

---

int Ecan_SetFilter(int handle, const ECAN_FILTER_STRUCTURE *filter_p);

Description:

Set interface's message filters to mask out certain incoming messages. This function can operate either on the driver's transmit queue or on the board directly. This function puts the board in reset mode, so you must start the board afterward.

NOTE: On the ECAN527, this function no longer sets the message ID values in the default standard and extended frame receive message objects. You must follow this function with two calls to a function which will set up those message objects.

NOTE: Do not use this function to set filters on the ECAN1000; the filter bits are set incorrectly in this case. New functions have been added to set filters on the ECAN1000.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

filter_p: Address of structure where filter data is stored.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EBUSY
dont_use_queue is nonzero and the Transmit Status or Receive Status bit is set in the Status Register (ECAN1000 only).

EFAULT
filter_p is not a valid user address.

EIO
The driver was unable to turn on the Reset Mode bit in the Mode Register to reconfigure the board (ECAN1000 only).

ENOBUFS
dont_use_queue is 0 and there is no free slot in the driver's transmit queue.

IOCTL Interface:

```
int rc;
struct rtd_ecan_filter filter;

/*
 * Set filter, queueing it into driver's transmit queue first
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_FILTER, &filter);

/*
 * Set filter by going directly to the board
 */

rc = ioctl(
   descriptor,
   (__RTD_ECAN_IOCTL__SET_FILTER | __RTD_ECAN_DONT_USE_QUEUE),
   &filter
);
```

int Ecan_SetLeds(int handle, bool RedLed, bool GreenLed);

Description:

Turn on or off an interface's LEDs. ECAN527 only.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

RedLed: Flag to turn on red LED.

GreenLed: Flag to turn on green LED.

A value of false for any of the above LED flags indicates that the corresponding LED should be turned off. A value of true for any of the above LED flags indicates that the corresponding LED should be turned on.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EINVAL
led_mask contains an invalid LED bit.

ENOTSUP
Operation is not supported (ECAN1000 only).

IOCTL Interface:

```
int rc;
unsigned int led_mask;

/*
 * Turns the red LED on and the green LED off
 */

led_mask = RTD_ECAN_LED_RED;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_LEDS, led_mask);

/*
 * Turns the green LED on and the red LED off
 */

led_mask = RTD_ECAN_LED_GREEN;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_LEDS, led_mask);
```

```
/*
 * Turns the green LED on and the red LED on
 */

led_mask = (RTD_ECAN_LED_GREEN | RTD_ECAN_LED_RED);
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_LEDS, led_mask);

/*
 * Turns the green LED off and the red LED off
 */

led_mask = 0;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_LEDS, led_mask);
```

Ecan_SetSingleFilterExtended

```
int Ecan_SetSingleFilterExtended(
   int handle,
   uint ID_ACR,
   uint ID_AMR,
   uint RTR_ACR,
   uint RTR_AMR
);
```

Description:

Set up a filter for extended frames in single filter mode. ECAN1000 only.

NOTE: Unless you feel adventurous enough to undertake determining the exact bit patterns to set in the filter structure, do not use Ecan_SetFilter() to set filters on the ECAN1000. Use this function instead.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
ID_ACR:	29-bit Acceptance Code for message ID.
ID_AMR:	29-bit Acceptance Mask for message ID.
RTR_ACR:	1-bit Acceptance Code for Remote Transmission Request bit.
RTR_AMR:	1-bit Acceptance Mask for Remote Transmission Request bit.

Return Value:

0:	Filter set succeeded.
-1:	Filter set failed. Please see the description of Ecan_SetFilter() for information on possible values errno may have in this case.

IOCTL Interface:

None.

Ecan_SetSingleFilterStandard

```
int Ecan_SetSingleFilterStandard(
   int handle,
   uint ID_ACR,
   uint ID_AMR,
   uint RTR_ACR,
   uint RTR_AMR,
   uint Data_ACR,
   uint Data_AMR
);
```

Description:

Set up a filter for standard frames in single filter mode. ECAN1000 only.

NOTE: Unless you feel adventurous enough to undertake determining the exact bit patterns to set in the filter structure, do not use Ecan_SetFilter() to set filters on the ECAN1000. Use this function instead.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
ID_ACR:	11-bit Acceptance Code for message ID.
ID_AMR:	11-bit Acceptance Mask for message ID.
RTR_ACR:	1-bit Acceptance Code for Remote Transmission Request bit.
RTR_AMR:	1-bit Acceptance Mask for Remote Transmission Request bit.
Data_ACR:	16-bit Acceptance Code for first 16 bits of message data.
Data_AMR:	16-bit Acceptance Mask for first 16 bits of message data.

Return Value:

0:	Filter set succeeded.
-1:	Filter set failed. Please see the description of Ecan_SetFilter() for information on possible values errno may have in this case.

IOCTL Interface:

None.

```
int Ecan_SetupBoard(
   int handle,
   bool ReceiveIntEn = true ,
   bool ErrorIntEn = false,
   bool TransmitIntEn = false,
   bool BusErrorIntEn = false,
   bool DataOverrunIntEn = false,
   bool ArbitrationLostIntEn = false,
   bool ErrorPassiveIntEn = false,
   bool WakeUpIntEn = false,
   unsigned long int RxSize = 0,
   unsigned long int TxSize = 0
);
```

Description:

Set an interface's event mask and default receive/transmit queue sizes. This function puts the board in reset mode, so you must start the board afterward.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
ReceiveIntEn:	Flag to indicate interest in Receive Interrupt.
ErrorIntEn:	Flag to indicate interest in Error Warn Interrupt.
TransmitIntEn:	Flag to indicate interest in Transmit Interrupt.
BusErrorIntEn:	Flag to indicate interest in Bus Error Interrupt.
DataOverrunIntEn:	Flag to indicate interest in Data Overrun Interrupt.
ArbitrationLostIntEn:	Flag to indicate interest in Arbitration Lost Interrupt.
ErrorPassiveIntEn:	Flag to indicate interest in Error Passive Interrupt.

WakeUpIntEn: Flag to indicate interest in Wake Up Interrupt.

A value of false for any of the above interrupt interest flags indicates that the application is not interested in the corresponding interrupt. A value of true for any of the above interrupt interest flags indicates that the application is interested in the corresponding interrupt. By default, only the Receive Interrupt is of interest.

RxSize: Size of driver's receive queue.

TxSize: Size of driver's transmit queue.

A value of 0 for either of the above queue sizes indicates that the corresponding queue size should not be changed. The default value for either queue size is 0.

Return Value:

0: Success.

-1: Failure.

Ecan_StartBoard

---

int Ecan_StartBoard(int handle);

Description:

Put an interface into operating mode.

NOTE: This function overwrites any filters you may have set. You must set up your filters again after calling this function.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EIO
The driver was unable to turn on the Reset Mode bit in the Mode Register to reconfigure the board (ECAN1000 only).

IOCTL Interface:

int rc;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__START);

Ecan_StopBoard

---

int Ecan_StopBoard(int handle);

Description:

Put an interface into reset mode.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EIO
The driver was unable to turn on the Reset Mode bit in the Mode Register to reconfigure the board (ECAN1000 only).

IOCTL Interface:

int rc;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__STOP);

Ecan_TestBoard

---

int Ecan_TestBoard(int handle);

Description:

Test an interface. This function puts the board in reset mode, so you must start the board afterward. On the ECAN1000, this function also will clear the receive and transmit queue contents.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EIO
The driver was unable to turn on the Reset Mode bit in the Mode Register (ECAN1000 only).

EIO
The interface failed the test which writes indicator values into the thirteen bytes of the Transmit Buffer, reads them back, and verifies values read match the values written (ECAN1000 only).

EIO
The interface failed the test which sets the Init bit in the Control Register and then reads it back in to verify it was set (ECAN527 only).

EIO
The interface failed the test which attempts to write to Bit Timing Register 1 while the Init bit in the Control Register is set but the Change Configuration Enable bit is cleared. When these Control Register bits are so set, a write to the Bit Timing Register 1 should not change its value. In this case, it did (ECAN527 only).

EIO
The interface failed the test which attempts to write to Bit Timing Register 1 while the Init bit in the Control Register is set and the Change Configuration Enable bit is set. When these Control Register bits are so set, a write to the Bit Timing Register 1 should change its value. In this case, it did not (ECAN527 only).

IOCTL Interface:

int rc;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__TEST);

Ecan_WriteDigitalIO

---

int Ecan_WriteDigitalIO(HANDLE handle, unsigned char *digital_data_p);

Description:

Write a value to an interface's digital I/O port. ECAN527 only.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

digital_data_p: Address of user buffer containing data to write.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EFAULT
digital_data_p is not a valid user address.

ENOTSUP
Operation is not supported (ECAN1000 only).

IOCTL Interface:

```
int rc;
unsigned char digital_data;

digital_data = 0xc5;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__WRITE_DIGITAL_IO, &digital_data);
```

Ecan_Clear_Accounts

int Ecan_Clear_Accounts(int handle);

Description:

Clear the statistics the driver keeps internally about device and driver operation.

Parameters:

handle: Device handle from Ecan_CreateHandle() or file descriptor from open() system call.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF

handle is not a valid file descriptor.

IOCTL Interface:

```
int rc;

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__CLEAR_ACCOUNTS);
```

Ecan_Get_Accounts

---

int Ecan_Get_Accounts(int handle, rtd_ecan_accounts_t *accounts_p);

Description:

Clear the specified queue on an interface.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
queue_mask:	Bit mask of queues to clear. Valid bits are RTD_ECAN_RX_QUEUE and RTD_ECAN_TX_QUEUE.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

EINVAL
queue_mask contains an invalid bit.

IOCTL Interface:

```
int rc;
unsigned int queue_mask;

/*
 * Clear the receive queue
 */

queue_mask = RTD_ECAN_RX_QUEUE;
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__CLEAR_QUEUES, queue_mask);

/*
 * Clear the transmit queue
 */

queue_mask = RTD_ECAN_TX_QUEUE;
```

```
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__CLEAR_QUEUES, queue_mask);

/*
 * Clear both the receive and transmit queues
 */

queue_mask = (RTD_ECAN_RX_QUEUE | RTD_ECAN_TX_QUEUE);
rc = ioctl(descriptor, __RTD_ECAN_IOCTL__CLEAR_QUEUES, queue_mask);
```

Ecan_Set_RX_Queue_Size

int Ecan_Set_RX_Queue_Size(int handle, size_t queue_size);

Description:

Set interface's receive queue size. Doing so will also clear the the receive queue contents.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
queue_size:	Size of receive queue in message items.

Return Value:

0: Success.

-1: Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

ENOMEM
No memory available for given number of message items.

IOCTL Interface:

```
int rc;

/*
 * Set receive queue size to 16 message items
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_RX_MAX_QUEUE_SIZE, 16);
```

Ecan_Set_TX_Queue_Size

int Ecan_Set_TX_Queue_Size(int handle, size_t queue_size);

Description:

Set interface's transmit queue size. Doing so will also clear the the transmit queue contents.

Parameters:

handle:	Device handle from Ecan_CreateHandle() or file descriptor from open() system call.
queue_size:	Size of transmit queue in message items.

Return Value:

0:	Success.
-1:	Failure with errno set as follows:

EBADF
handle is not a valid file descriptor.

ENOMEM
No memory available for given number of message items.

IOCTL Interface:

```
int rc;

/*
 * Set receive queue size to 500 message items
 */

rc = ioctl(descriptor, __RTD_ECAN_IOCTL__SET_RX_MAX_QUEUE_SIZE, 500);
```

Example Programs Reference

Name	Remarks
rtd-ecan1000-send-command	Demonstrates how to send a command to an ECAN1000 board by using Ecan_SendCommand().
rtd-ecan527-digital-io	Demonstrates how to read data from and write data to an ECAN527 board's digital I/O port.
rtd-ecan527-test-hardware	Test the ECAN527 board by exercising the message objects both receiving and sending messages. The driver is bypassed as much as possible. Receive and transmit interrupts are not used.
rtd-ecan527-test-hardware-int	Test the ECAN527 board by exercising the message objects both receiving and sending messages. The driver is bypassed as much as possible. Receive interrupts are used but not transmit interrupts.
rtd-ecan527-test-leds	Performs the following operations in the this exact sequence: 1) turns both LEDs off, 2) turns red LED on and green LED off, 3) turns red LED off and green LED on, 4) turns both LEDs on, and 5) turns both LEDs off.
rtd-ecan-clear-accounts	Demonstrates how to clear the statistics kept internally by the driver.
rtd-ecan-print-accounts	Demonstrates how to print the statistics kept internally by the driver.
rtd-ecan-read	Demonstrates reading messages from a CAN interface. A simple file transfer is implemented which receives a file from rtd-ecan-write.
rtd-ecan-test-bit-rates	Test message receive and send at the following CAN bus bit rates: 1) 50,000 bps, 2) 62,500 bps, 3) 100,000 bps, 4) 125,000 bps, 5) 250,000 bps, 6) 500,000 bps, and 7) 1,000,000 bps. The tests are not stress tests to see how fast messages can be sent and received. Rather, they simply test that a single message can be sent and received at each bit rate.
rtd-ecan-test-board	Demonstrates using Ecan_TestBoard().
rtd-ecan-test-filter	Test driver filter operation. This program can test both standard and extended filters on both the ECAN527 and the ECAN1000. Only one type of filter can be tested with each invocation of the executable.
rtd-ecan-test-id-read	Test driver message ID logic. This program receives special messages sent by rtd-ecan-test-id-write. For each message received, the message ID (as set by rtd-ecan-test-id-write) is verified to ensure the driver encoded the ID properly for transmission and then decoded the ID properly upon reception. Both standard and extended frames are tested.
rtd-ecan-test-id-write	Test driver message ID logic. This program sends special messages to rtd-ecan-test-id-read. Each message is given a message ID. The message ID is also sent in the message data so that rtd-ecan-test-id-read can verify what message ID it should be receiving. Both standard and extended frames are tested.
rtd-ecan-test-tx-error-code	Tests driver error code processing on transmit error. This program requires user intervention in the form of installing and removing the CAN cable. The user is prompted when to install and when to remove the cable. A series of 6 messages are sent and the GetStatus error code of each one is verified against expected behavior.
rtd-ecan-throughput	Calculates the throughput rate for a CAN interface. Seven different CAN bus bit rates are supported.
rtd-ecan-write	Demonstrates writing messages to a CAN interface. A simple file transfer is implemented which sends a file to rtd-ecan-read.

Limited Warranty

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DISCLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College PA 16803-0906
USA
Our website: www.rtd.com