GE863-PRO^3 Linux CAN Package Software User Guide

For GE863-PRO^3 with Linux OS

1vv0300866 Rev. 0 – 2009-11-27

Making machines talk.

Disclaimer

Applicable Products

PRODUCT
GE863-PRO^3 with Linux OS

Contents

1. Introduction

1.1. Scope

This user guide serves the following purpose:

- Introduces briefly the CAN specification
- Illustrates the CAN package that exports on Linux operative system the main CAN features (such as CAN frames reading/writing on a CAN bus) and to provide detailed examples in order to show the correct use of the APIs.

1.2. Audience

This User Guide is intended for software developers who develop applications on the ARM processor of the module and need to use the GE863-PRO^3 connected to a CAN bus.

1.3. Contact Information, Support

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.
For general contact, technical support, report documentation errors and to order manuals, contact Telit Technical Support Center at:
TS-EMEA@telit.com or
http://www.telit.com/en/products/technical-support-center/contact.php
Telit appreciates feedback from the users of our information.

1.4. GNU General Public License

The CAN package code embedded into the module is licensed with the GNU General Public License as follows:

Please refer to the following web page for the full text of the license:
http://git.denx.de/?p=u-boot.git;a=blob;f=COPYING;h=f616ab96cc773d1719761d511a8649c9aa6eb473;hb=f4eb54529bb3664c3a562e488b460fe075f79d67

1.5. Product Overview

The GE863-PRO^3 is an innovation to the quad-band, RoHS compliant GE863 product family which includes a powerful ARM9^{TM} processor core exclusively dedicated to customer applications. The concept of collocating a powerful processor core with the GSM/GPRS engine allows developers to host their application directly. The PRO^3 incorporates much of the necessary hardware for communicating microcontroller solutions, including the critical element of memory, significant simplification of the bill of material, vendor management, and logistics effort are achieved.

1.6. Document Organization

This manual contains the following chapters:

- "Chapter 1, Introduction" provides a scope for this manual, target audience, technical contact information, and text conventions.
- "Chapter 2, Controller Area Network" introduces the CAN version 2.0 Specification.
- "Chapter 3, CAN Package Description" describes the CAN package with its software architecture.
- "Chapter 4, CAN Package Setup" describes briefly how to load CAN modules and import the CAN package APIs within a project.
- "Chapter 5, Functionalities and APIs Summary" provides a brief list and description of the CAN package functionalities and APIs.
- "Chapter 6, APIs Data Types and Structures" provides a description of the types used within the CAN package APIs library.
- "Chapter 7, APIs Description" provides a detailed description and examples of the methods that implement the CAN package APIs library.
- "Chapter 8, Acronyms and Abbreviations" provides definition for all the acronyms and abbreviations used in this guide.

How to Use

If you are new to this product, it is highly recommended to start by reading through TelitGE863PRO3Linux_SW_UserGuide 1vv0300781 [4] and TelitGE863PRO3 Linux Development Environment User Guide 1VV0300780 [1] manuals and this document in their entirety in order to understand the concepts and specific features provided by the built in software of the GE863-PRO3.

1.7. Text Conventions

This section lists the paragraph and font styles used for the various types of information presented in this user guide.

Format	Content
`Courier New`	Shell command examples, C code examples and types definition.

1.8. Related Documents

The following documents are related to this user guide:

1. TelitGE863PRO3 Development Environment User Guide 1vv0300775a
2. TelitGE863PRO3 EVK User Guide 1vv0300776
3. TelitGE863PRO3 Hardware User Guide 1vv0300773a
4. TelitGE863PRO3 Software User Guide
5. TelitGE863PRO3 Product Description 80285ST10036a
6. Atmel AT91SAM 9260 Summary Datasheet: 6221s.pdf on web page link: http://www.atmel.com/dyn/products/datasheets.asp?family_id=605
7. CAN Specification Version 2.0, Robert Bosch GmbH, 1991: can2spec.pdf on web page link: www.semiconductors.bosch.de/pdf/can2spec.pdf
8. Stand-Alone CAN Controller With SPI Interface: 21801d.pdf on web page link: ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf

All documentation can be downloaded from Telit's official web site www.telit.com if not otherwise indicated.

1.9. Document History

Revision	Date	Changes
ISSUE #0	2009-11-27	First revision

2. Controller Area Network

The Controller Area Network (CAN) is a serial communications protocol which efficiently supports distributed real-time control with a very high level of security.
Its domain of application ranges from high speed networks to low cost multiplex wiring. In automotive electronics, engine control units, sensors, anti-skid-systems, etc. are connected using CAN with bitrates up to 1 Mbit/s. At the same time it is cost effective to build into vehicle body electronics, e.g. lamp clusters, electric windows etc. to replace the wiring harness otherwise required.

2.1. CAN version 2.0 specification

Controller Area Network (CAN) was initially created by German automotive system supplier Robert Bosch in the mid-1980s for automotive applications as a method for enabling robust serial communication. The goal was to make automobiles more reliable, safe and fuel-efficient while decreasing wiring harness weight and complexity.
Since its inception, the CAN protocol has gained widespread popularity in industrial automation and automotive/truck applications. Other markets where networked solutions can bring attractive benefits like medical equipment, test equipment and mobile machines are also starting to utilize the benefits of CAN.
Bosch published the CAN version 2.0 specification in 1991 [7]; CAN is also an international standard: ISO 11898.
CAN version 2.0 Specification consists of two parts:

- Part A describing the CAN message format as it is defined in CAN version 1.2 Specification;
- Part B describing both standard and extended message formats.

In order to be compatible with this CAN version 2.0 Specification it is required that a CAN implementation be compatible with either Part A or Part B.
CAN implementations that are designed according to part A and CAN implementations that are designed according to part B can communicate with each other as long as it is not made use of the extended format.

To achieve design transparency and implementation flexibility CAN has been subdivided into different layers according to the ISO/OSI Reference Model:

- the Data Link Layer
  - the Logical Link Control (LLC) sublayer
  - the Medium Access Control (MAC) sublayer
- the Physical Layer

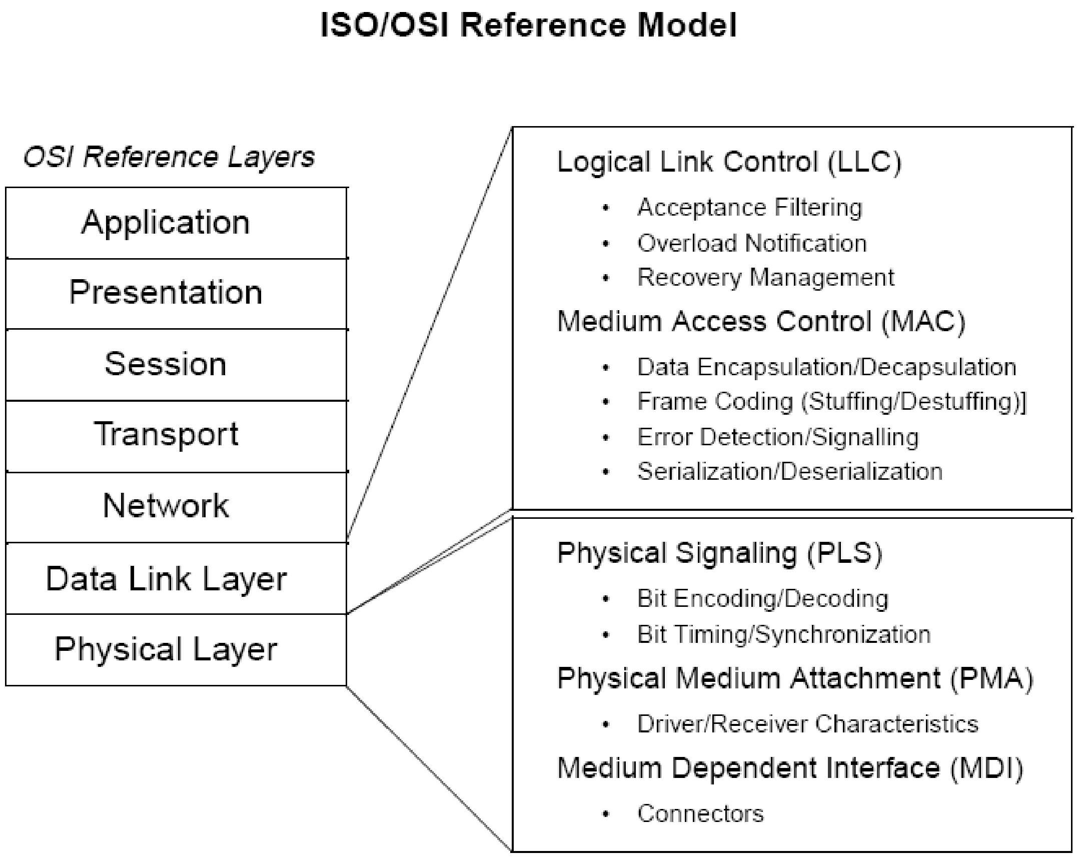

ISO/OSI Reference Model
OSI Reference Layers
Application
Presentation
Session
Transport
Network
Data Link Layer
Physical Layer
Logical Link Control (LLC)
Acceptance Filtering
Overload Notification
Recovery Management
Medium Access Control (MAC)
Data Encapsulation/Decapsulation
Frame Coding (Stuffing/Destuffing)]
Error Detection/Signalling
Serialization/Deserialization
Physical Signaling (PLS)
Bit Encoding/Decoding
Bit Timing/Synchronization
Physical Medium Attachment (PMA)
Driver/Receiver Characteristics
Medium Dependent Interface (MDI)
Connectors

Figure 2.1

The scope of the LLC sublayer is

- to provide services for data transfer and for remote data request,
- to decide which messages received by the LLC sublayer are actually to be accepted,
- to provide means for recovery management and overload notifications.

The scope of the MAC sublayer mainly is the transfer protocol, i.e. controlling the Framing, performing Arbitration, Error Checking, Error Signalling and Fault Confinement. Within the MAC sublayer it is decided whether the bus is free for

starting a new transmission or whether a reception is just starting. Also some general features of the bit timing are regarded as part of the MAC sublayer.
The scope of the physical layer is the actual transfer of the bits between the different nodes with respect to all electrical properties. Within one network the physical layer, of course, has to be the same for all nodes. There may be, however, much freedom in selecting a physical layer.
CAN version 2.0 Specification defines the MAC sublayer and a small part of the LLC sublayer of the Data Link Layer and to describe the consequences of the CAN protocol on the surrounding layers. ISO standard includes the Media Dependant Interface definition such that all of the lower two layers are specified.
CAN does not have device addresses like the MAC addresses by Ethernet. The only thing used for addressing is the CAN ID. Since all messages are broadcasted to the whole CAN network, it is not possible to send a message only to one device.
The rest of the layers of the ISO/OSI protocol stack are left to be implemented by the system software developer. Higher Layer Protocols (HLPs) are generally used to implement the upper five layers of the OSI Reference Model.
HLPs are used to:

1. standardize startup procedures including bit rates used,
2. distribute addresses among participating nodes or types of messages,
3. determine the structure of the messages, and
4. provide system-level error handling routines.

The main features of CAN version 2.0 Specification are:

- Carrier Sense Multiple Access with Collision Detection (CSMA/CD)
- message-based communication
- prioritization of messages
- guarantee of latency times
- configuration flexibility
- multicast reception with time synchronization
- system wide data consistency
- multimaster
- error detection and signaling
- automatic retransmission of corrupted messages as soon as the bus is idle again
- distinction between temporary errors and permanent failures of nodes
- autonomous switching off of defect nodes

Further detailed information can be found in the document [7].

3. CAN Package Description

CAN Package is a software solution for Telit GE863-Pro^3 that supports CAN version 2.0 Specification part B. This package is based on a CAN controller connected to GE863-Pro^3 by SPI port (**Microchip MCP2515**).

3.1. Hardware

The Microchip MCP2515 CAN controller shall be connected with GE863-PRO³ through SPI bus 1. CAN package needs one GE863-PRO³ GPIO as interrupt signal. As default SPI chip select 0 and GPIO PA29 are selected. Different chip select and/or GPIO can be selected when the driver module is loaded (see Section 4.1).

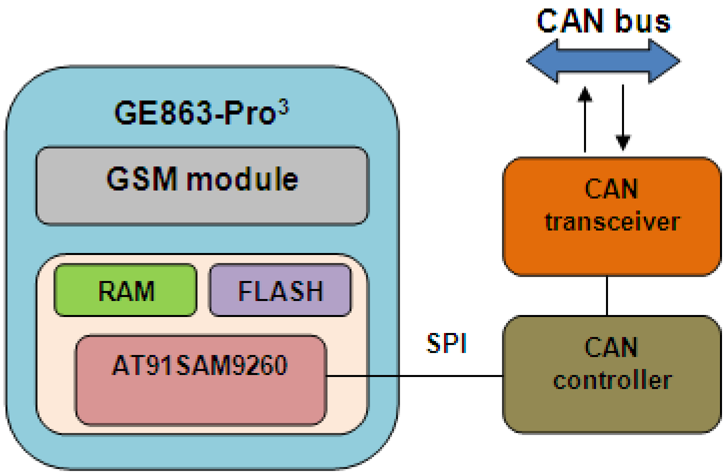


Figure 3.1

3.2. Software

Figure 3.2 shows software architecture of the **CAN Package** for GE863-Pro[3]:

- Linux OS SPI driver for MCP2515 CAN controller
- SocketCAN framework modules
- CAN Package APIs library
- User Application

CAN Package includes Linux controller driver, SocketCAN framework modules and CAN APIs library. User applications use the functionalities provided by CAN APIs library to manage CAN bus.

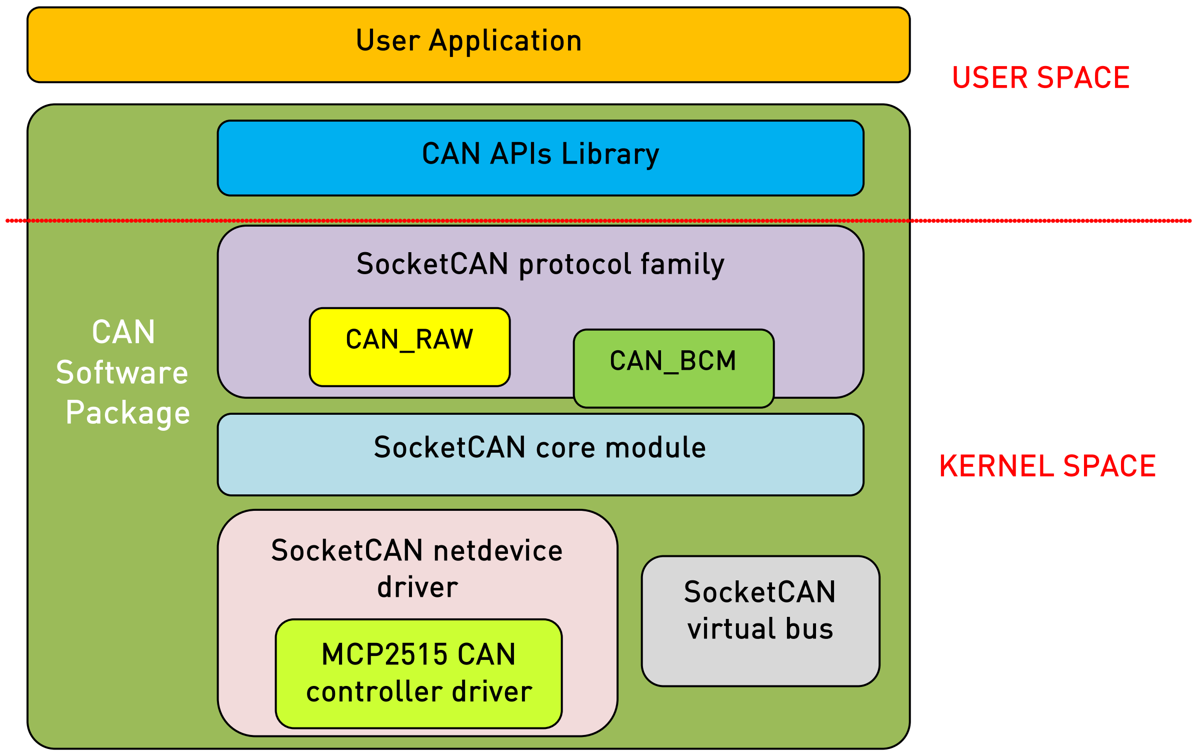


Figure 3.2

3.2.1. SocketCAN

SocketCAN is a set of open source CAN drivers and a networking stack contributed by Volkswagen Research to the Linux kernel. Formerly it is known as `Low Level CAN Framework` (LLCF).

Established CAN drivers are based on the model of character devices. Typically they only allow sending to and receiving from the CAN controller. Most of the implementations for this device class only allow a single process on the device which means that all other processes are blocked in the meantime as known from accessing a device via the serial interface. The SocketCAN concept on the other hand uses the model of network devices, which allows multiple applications to access to one CAN device simultaneously. Equally single applications are able to access multiple CAN networks in parallel.
The SocketCAN concept introduces a new protocol family PF_CAN that coexists with other protocol families like PF_INET for the Internet Protocol. The communication with the CAN bus is done analogue to the use of the Internet Protocol via Sockets. Fundamental components of SocketCAN are the network device drivers for different CAN controllers and the implementation of the CAN protocol family.
SocketCAN does not touch higher level protocols like CANopen or DeviceNet: these HLPs can be put on top of one CAN socket.

3.2.2. Modules

CAN Package is made up of the following SocketCAN modules:

- **can.ko (SocketCAN core module):** allows to use the SocketCAN features;
- **can-dev.ko (SocketCAN netdevice driver):** manages controller driver like network driver;
- **mcp251x.ko (MCP2515 CAN controller driver):** Linux OS SPI CAN controller driver;
- **can-raw.ko (CAN_RAW protocol):** allows direct access (R/W) to the CAN bus;
- **can-bcm.ko (CAN_BCM protocol):** provides data message filtering in cyclic frames;
- **vcan.ko (virtual CAN bus):** provides a virtual CAN interface.

3.2.3. CAN connections

CAN package allows multiple connections to access at the same CAN device simultaneously.

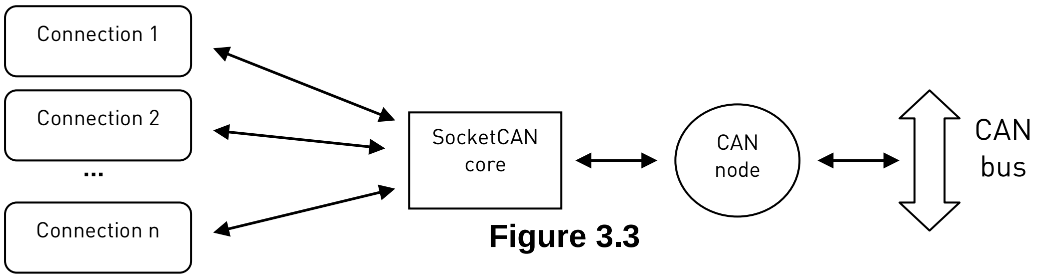


Figure 3.3

Communication between CAN connections and CAN node happens according to following modes:

- When loopback hardware control mode is active, no frames are received or sent on CAN bus. Every frame sent to CAN node is returned to every CAN connection active in the same CAN node.
- When listen-only mode is active, no frames are sent on CAN bus, while the CAN node is able to receive frame from CAN bus.
- When local loopback mode is active, all the CAN frames sent from a CAN connection are looped back to the other CAN connections active on the same CAN node.
- When own frame receiving is active, every CAN connection receives the frame sent by itself.
- When error signaling is active, special error signaling frames are sent to the other CAN connections active on the same CAN node. These special frames are not sent on CAN bus.

These operating modes are not incompatible between themselves. For example, when loopback hardware control mode and local loopback are active, if a CAN connection sends a frame:

- the CAN connection that has sent this frame is able to receive it;
- the other CAN connections active in the same CAN node receive twice this frame;
- this frame is not transmitted on CAN bus.

4. CAN Package Setup

4.1. Modules loading

In order to use CAN package features, **can.ko** module must be loaded by this shell command:

```
# modprobe can.ko
```

Once can.ko module has been loaded, controller driver can be loaded by typing **can-dev.ko** and **mcp251x.ko**:

```
# modprobe can-dev.ko
# modprobe mcp251x.ko
```

When the controller is probed, a CAN interface is created with name "can*x*", where *x* is a positive integer (the first CAN interface is named "can0"). This network interface can be shown by:

```
# ifconfig -a
```

mcp251x.ko probing sets as default chip select 0 and GPIO PA29 respectively for SPI bus 1 communication and interrupt signal. In order to change these settings, it is possible to use optional `cs` and/or `irq_gpio` parameters:

```
# modprobe mcp251x.ko cs=2 irq_gpio=90
```

GPIO pins are numbered from 0 to 95. The first 32 pins refer to the PIO A controller, the second 32 pins refer to the PIO B controller and the others refer to the PIO C controller. In the example chip select 2 and interrupt on PC26 are set.

If a virtual CAN device must be used instead of a physical controller, **vcan.ko** must be loaded in place of **mcp251x.ko**:

```
# modprobe vcan.ko
```

In this case, system is ready to create one virtual can interface called "vcan0" when this will be enabled.

After SPI driver controller or virtual can bus is loaded, protocol modules must be loaded:

```
# modprobe can-raw.ko
# modprobe can-bcm.ko
```

After loading these modules, user will be able to use CAN package features.

4.2. Modules removing

The following instructions remove CAN modules for the use of MCP2515 CAN controller:

```
# rmomod can-bcm
# rmmod can-raw
# rmmod mcp251x
# rmmod can-dev
# rmmod can
```

The following instructions remove CAN modules for the use of a virtual CAN device:

```
# rmomod can-bcm
# rmmod can-raw
# rmmod vcan
# rmmod can-dev
# rmmod can
```

4.3. Library setup

In order to include or update the library version on your development environment, user has to copy the header file and the library respectively into the `/opt/crosstools/telit/include/` and `/opt/crosstools/telit/lib/` directories:

- Start the Linux console (Windows **Start Menu → All Programs → Telit Development Platform → Console**).
- Copy the library typing

cp /mnt/windows/<PATH>/libCAN.a /opt/crosstools/telit/lib/

- Copy the header file typing
  cp /mnt/windows/<PATH>/CANlib.h /opt/crosstools/telit/include/

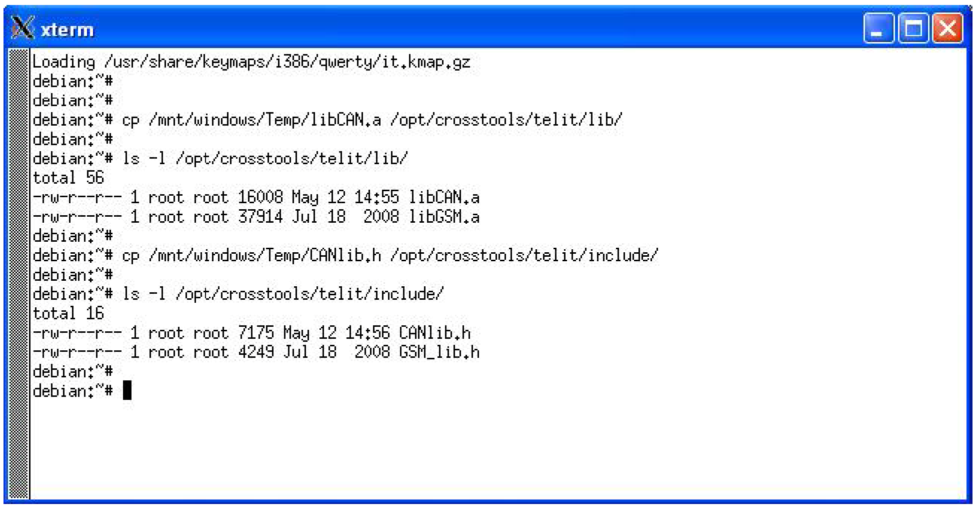
where **<PATH>** is the windows folder where user has stored the new version of the library files. For example (see Figure 4.1), if user stores them within C:\Temp, he has to digit

cp /mnt/windows/Temp/libCAN.a /opt/crosstools/telit/lib/

and

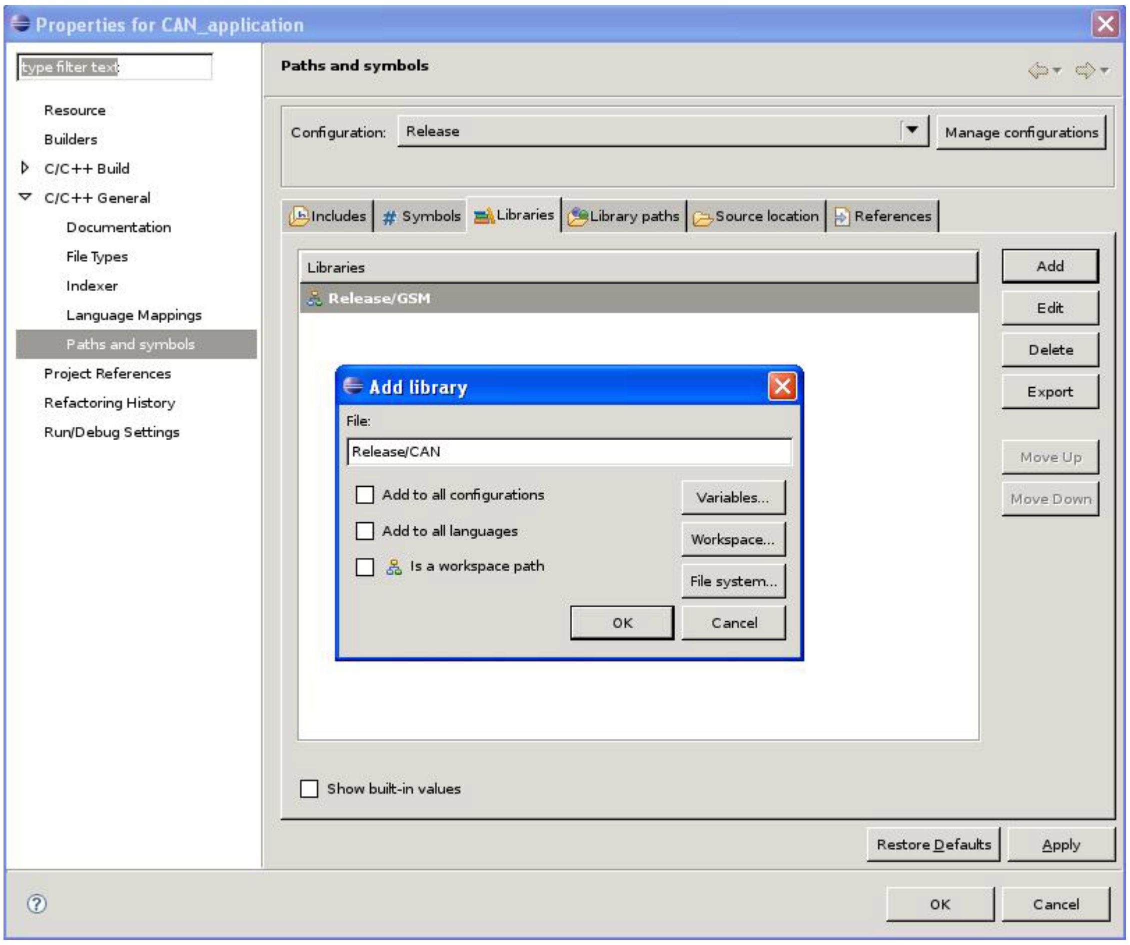
```
cp   /mnt/windows/Temp/CANlib.h   /opt/crosstools/telit/include/
```

In order to link the CAN library in a project, in the development environment under **Project → Properties → C/C++ General → Path and symbols → Libraries** user has to click **Add**, insert the file `Release/CAN` and click **OK** (see Figure 4.2).



```
Loading /usr/share/keymaps/i386/qwerty/it.kmap.gz
debian:~#
debian:~#
debian:~# cp /mnt/windows/Temp/libCAN.a /opt/crosstools/telit/lib/
debian:~#
debian:~# ls -l /opt/crosstools/telit/lib/
total 56
-rw-r--r-- 1 root root 16008 May 12 14:55 libCAN.a
-rw-r--r-- 1 root root 37914 Jul 18  2008 libGSM.a
debian:~#
debian:~# cp /mnt/windows/Temp/CANlib.h /opt/crosstools/telit/include/
debian:~#
debian:~# ls -l /opt/crosstools/telit/include/
total 16
-rw-r--r-- 1 root root 7175 May 12 14:56 CANlib.h
-rw-r--r-- 1 root root 4249 Jul 18  2008 GSM_lib.h
debian:~#
debian:~# 
```

Figure 4.1



Configuration: Release

Libraries
Release/GSM

File:
Release/CAN

☐ Add to all configurations
☐ Add to all languages
☐ Is a workspace path

☐ Show built-in values

Figure 4.2

5. Functionalities and APIs Summary

CAN package provides to user applications also the following main functionalities:

Functionalities	Notes
MCP2515 CAN controller supported	See [8]
CAN ver 2.0 Specification part B supported	See Chapter 2
CAN bus sharing	CAN bus can be shared by different applications in the same moment
CAN frames semantics	CAN package does not define semantics of CAN frames. Semantics is only defined by user applications

In the following table a summary of the APIs is shown.

Functionality Group	APIs	Notes
CAN bus access	CAN_Enable()	Enables CAN bus access on a real or virtual controller
	CAN_Disable()	Disables CAN bus access on a real or virtual controller
CAN connection	CAN_Open()	Opens a connection with the CAN node
	CAN_Close()	Closes a connection with the CAN node
CAN node settings	CAN_SetBitrate()	Sets CAN bus bitrate according CIA recommendations
	CAN_GetBitrate()	Gets CAN bus actual bitrate
	CAN_SetTimings()	Sets CAN node timings: time quantum, propagation segment, phase segments, synchronization jump width
	CAN_GetTimings()	Gets CAN node timings: time quantum, propagation segment, phase segments, synchronization jump width
CAN node settings	CAN_SetCtrlMode()	Enables normal, hardware loopback or listen-only CAN node control mode
	CAN_GetCtrlMode()	Reads the CAN node actual control

		mode (normal, hardware loopback or listen-only)
	CAN_GetState()	Reads the CAN node actual state (active, bus warning, bus passive, bus off, stopped)
CAN operations	CAN_Read()	Reads a CAN frame on a CAN connection
	CAN_Write()	Sends a CAN frame on a CAN connection
	CAN_CyclicSending()	Sends a cyclic CAN frame on a CAN connection
	CAN_ErrorSignaling()	Enables or disables error states signaling through CAN frames reading on a CAN connection
	CAN_LocalLoopback()	Enables or disables the receiving of the CAN frame sent by other CAN connections on the same CAN node
	CAN_ReceiveOwnFrames()	Enables or disables the receiving of the CAN frame sent by the same CAN connection
CAN filtering	CAN_SetHardwareFilters()	Sets hardware filters on a CAN node
	CAN_GetHardwareFilters()	Gets hardware filters on a CAN node
	CAN_AddSoftwareFilter()	Adds a new CAN ID software filter on a CAN connection
	CAN_DelSoftwareFilter()	Deletes a CAN ID software filter on a CAN connection
	CAN_SoftwareFiltersList()	Shows the active software filters on a CAN connection
	CAN_SetContentFilter()	Enables or disables monitoring for CAN data change according to a frame mask

CAN filtering	CAN_ReadChangedContent()	Reads a frame only when content change is detected according to CAN_SetContentFilter() settings
CAN statistics	CAN_GetDeviceStats()	Shows CAN controller statistics

6. APIs Data Types and Structures

6.1. CAN_ERROR_CODE_E

This type is an enum containing codes for all errors that may occur during CAN operations. Each method described within chapter 7 returns an error code.

```
typedef enum
{
       CAN_OK,
       CAN_ERROR,
       CAN_UNKNOWN_IF,
       CAN_ALREADY_ENABLED_IF,
       CAN_ALREADY_DISABLED_IF,
       CAN_DISABLED_IF,
       CAN_CONNECTION_ERROR,
       CAN_VIRTUAL_IF,
       CAN_BITRATE_ERROR,
       CAN_BIT_TIMINGS_ERROR,
       CAN_TYPE_ERROR,
       CAN_ERROR_SIGNAL,
       CAN_FILTER_ERROR,
       CAN_TOO_ENABLED_FILTERS,
       CAN_ALREADY_DEFINED_FILTER,
       CAN_NON_EXISTENT_FILTER,
       CAN_STOP_CYCLIC,
       CAN_TIMEOUT_EXPIRED
} CAN_ERROR_CODE_E;
```

6.2. CAN_CONNECTION_T

This struct contains the interface name of CAN node and the sockets implementing CAN connection.

```
typedef struct CAN_CONNECTION_TAG
 {
       INT8         interface[6];
       INT32        rawSocket

       INT32        bcmSocket;
} CAN_CONNECTION_T;
```

6.3. CAN_FRAME_T

This struct contains the CAN frame bit fields and its attributes (standard or extended format, remote frame).

```
typedef struct CAN_FRAME_TAG
{
       UINT32 canId;
       UINT8        canDlc;
       UINT8        data[8];      /* data[0] indicates the most significant
byte */
       UINT8        attributes;
} CAN_FRAME_T;
```

6.4. CAN_SOFTWARE_FILTER_T

This struct contains the CAN software filter specification and attributes.

```
typedef struct CAN_SOFTWARE_FILTER_TAG
{
       UINT32 filterId;
       UINT32 maskId;
       UINT8        attributes;
} CAN_SOFTWARE_FILTER_T;
```

6.5. CAN_HARDWARE_MASK_T

This struct contains the CAN hardware filters specification and attributes with a specified mask.

```
typedef struct CAN_HARDWARE_MASK_TAG
{
       UINT32 maskId;
       UINT16 maskData;
       UINT8        attributes;
       UINT32 filterId[CAN_MAX_HARDWARE_FILTERS_PER_MASK];
       UINT16 filterData[CAN_MAX_HARDWARE_FILTERS_PER_MASK];
} CAN_HARDWARE_MASK_T;
```

6.6. CAN_HARDWARE_FILTERS_T

This struct contains the CAN hardware filters specification and attributes as a set of CAN_HARDWARE_MASK_T structs.

```
typedef struct CAN_HARDWARE_FILTERS_TAG
{
       CAN_HARDWARE_MASK_T        mask[CAN_MAX_HARDWARE_MASKS];
} CAN_HARDWARE_FILTERS_T;
```

6.7. CAN_BIT_TIMINGS_T

This struct contains the CAN timing settings (time quantum, propagation and phase segments, synchronization jump width, sample point mode, oscillator frequency).

```
typedef struct CAN_BIT_TIMINIGS_TAG
{
       UINT32       tq;
       UINT8        propSeg:4;
       UINT8        phaseSeg1:4;
       UINT8        phaseSeg2:4;
       UINT8        sjw:3;
       UINT8        sam3:1;
       UINT32       clock;
} CAN_BIT_TIMINGS_T;
```

6.8. CAN_CTRL_MODE_E

This type is an enum containing the hardware control mode supported by CAN controller.

```
typedef enum
{
       CAN_NORMAL,
       CAN_LOOPBACK,
       CAN_LISTENONLY,
} CAN_CTRL_MODE_E;
```

6.9. CAN_STATE_E

This type is an enum containing the actual state of the CAN node.

```
typedef enum
{
       CAN_ACTIVE,
       CAN_BUS_WARNING,
       CAN_BUS_PASSIVE,
       CAN_BUS_OFF,
       CAN_STOPPED
} CAN_STATE_E;
```

6.10. CAN_DEVICE_STATS_T

This struct contains the statistics about the CAN device.

```
typedef struct CAN_DEVICE_STATS_TAG
 {
UINT8        errorWarning;
       UINT8        dataOverrun;
       UINT8        wakeUp;
       UINT8        busError;
       UINT8        errorPassive;
       UINT8        arbitrationLost;
       UINT8        restarts;
       UINT8        busErrorAtInit;
} CAN_DEVICE_STATS_T;
```

7. APIs Description

If the CAN library is integrated into the development environment (see Section 4.2), in order to use the CAN library user has only to include the header file into his application:

```
#include <CANlib.h>
```

7.1. CAN_Enable()

This function enables CAN bus access on a real or virtual controller. This function must be always called before opening the first CAN connection.
On a virtual node, this function creates and enables one virtual CAN interface called "vcan0". This network interface can be shown by:

```
# ifconfig -a
```

On a real controller, besides interface enabling (network interface "can0" is already created by mcp251x.ko module loading), this function:

- sets default bitrate (125 kbps)
- sets default controller clock oscillator (16MHz)
- sets timings according CIA recommendations
- resets and starts the controller in normal mode
- hardware filters are reset in order that controller receives all the frames

Prototype

```
CAN_ERROR_CODE_E CAN_Enable(INT8 *interface)
```

Parameters

`<interface>` It's the name of the CAN interface node to be enable. It can be both a real device (hardware controller if the SPI device module has been loaded) and a virtual one (if the virtual can device module has been loaded). Interface name is defined during modules loading (see Section 4.1).

Return values

`CAN_OK`	CAN interface has been successfully enabled
`CAN_UNKNOWN_IF`	CAN interface name is wrong
`CAN_ALREADY_ENABLED_IF`	CAN interface is already enabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_ERROR`	Unspecified error

Example

`CAN_Enable (“can0”);`
initializes the CAN interface “can0” to 125 kbps and starts the CAN controller

7.2. CAN_Disable()

This function disables CAN bus access on a real or virtual controller. This function must be always called after closing of the active CAN connections.
On a real controller, besides disabling interface, this function stops the controller.

Prototype
```
CAN_ERROR_CODE_E CAN_Disable(INT8 *interface)
```

Parameters
`<interface>` It’s the name of the CAN interface node to be disable. It can be both a real device (hardware controller if the SPI device module has been loaded) and a virtual one (if the virtual can device module has been loaded). Interface name is defined during modules loading (see Section 4.1).

Return values

`CAN_OK`	CAN interface has been successfully disabled
`CAN_UNKNOWN_IF`	CAN interface name is wrong
`CAN_ALREADY_DISABLED_IF`	CAN interface is already disabled
`CAN_ERROR`	Unspecified error

Example
`CAN_Disable (“can0”);`
disables the CAN interface “can0” and stops the CAN controller

7.3. CAN_Open()

This function opens a connection with a CAN interface (real or virtual). More than one connection can be opened on the same CAN node.
This function sets the CAN_CONNECTION_T that must be passed in all the functions performing activities on CAN bus. When a new connection is opened:

- local loopback is active;
- receiving of own frames is disabled;
- error signalling is disabled;
- no software filter is enabled and the frames are received only according the enabled hardware filters;
- no content filter is enabled.

Prototype

```
CAN_ERROR_CODE_E CAN_Open(CAN_CONNECTION_T *can , INT8 *interface)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T to be opened.
`<interface>` It's the name of the CAN interface on which a CAN connection must be opened. It can be both a real device and a virtual one already enabled.

Return values

`CAN_OK`	CAN connection has been successfully opened
`CAN_UNKNOWN_IF`	CAN interface name is wrong
`CAN_DISABLED_IF`	CAN interface is disabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;

CAN_Enable(“can0”);
CAN_Open (&can, “can0”);
```

creates a new connection on “can0” enabled interface

7.4. CAN_Close()

This function closes one specified connection with a CAN interface (real or virtual). If the CAN interface is a real controller, this function does not stop it.

Prototype

```
CAN_ERROR_CODE_E CAN_Close(CAN_CONNECTION_T *can)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T to be closed.

Return values

`CAN_OK`	CAN connection has been successfully closed
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
…
/* open connection on an enabled interface */
…
```

```
/* CAN operations on CAN connection */
...
CAN_Close (&can);
```

closes a CAN connection previously opened.

7.5. CAN_SetBitrate()

This function sets bitrate on a real CAN interface already enabled. CAN controller timings are set according to CIA recommendations. In order to set custom bit timings, CAN_SetTimings() must be used.
Only the following bitrates are supported (according to CIA recommendations): 10, 20, 50, 125, 250, 500, 800 and 1000 kbps.
Bitrate settings are applied on all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_SetBitrate(CAN_CONNECTION_T *can, UINT32 bitrate,
                  UINT32 clock)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which bitrate must be set.
`<bitrate>`	Bitrate (bps) to be set.
`<clock>`	Controller oscillator clock frequency (Hz).

Return values

`CAN_OK`	CAN bus bitrate has been successfully set
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_BITRATE_ERROR`	Unable to set the specified bitrate
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
…
/* open connection on an enabled interface */
…
CAN_SetBitrate(&can, 500000, 16000000);
```

sets CAN bus bitrate at 500 kbps and CAN controller timings according to CIA recommendations (oscillator clock frequency is 16 MHz)

7.6. CAN_GetBitrate()

This function reads the actual bitrate on a real CAN interface already enabled. Bitrate is the same in all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_GetBitrate(CAN_CONNECTION_T *can, UINT32 *bitrate)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which bitrate is read.
`<bitrate>`	Pointer to the bitrate (bps) to be read.

Return values

`CAN_OK`	CAN bus bitrate has been successfully read
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
UINT32 bitrate;
…
/* open connection on an enabled interface */
…
CAN_GetBitrate(&can, &bitrate);
```

reads the actual CAN bus baudrate and writes its value in bps in the `bitrate` variable

7.7. CAN_SetTimings()

This function sets custom bit timings on a real CAN interface already enabled. Bit timings to be set are:

- Time quantum (nsec);
- Propagation segment (from 1 to 8 time quanta), phase buffer segments 1 (from 1 to 8 time quanta) and 2 (from 2 to 8 time quanta);
- Synchronization jump width (from 1 to 4 time quanta);
- Oscillator frequency clock (Hz);
- Sampling point mode (the bus line can be sampled in the sample point once or three times).

Bitrate is calculated by timing settings. It can be shown calling CAN_GetBitrate() function.
Timing settings are applied on all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_SetTimings(CAN_CONNECTION_T *can,
CAN_BIT_TIMINGS_T *timings)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which bit timings must be set.
`<timings>` Pointer to the CAN_BIT_TIMINGS_T struct that contains the timings to be set:

`<timings->tq>`	time quantum (nsec)
`<timings->propSeg>`	propagation segment (from 1 to 8)
`<timings->phaseSeg1>`	phase buffer segment 1 (from 1 to 8)
`<timings->phaseSeg2>`	phase buffer segment 2 (from 2 to 8)
`<timings->sjw>`	synchronization jump width (from 1 to 4)
`<timings->sam3>`	three times sampling mode (0: disabled, 1: enabled)
`<timings->clock>`	oscillator frequency clock (Hz)

Return values

`CAN_OK`	CAN bit timings have been successfully set
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_BIT_TIMINGS_ERROR`	Unable to set the specified bit timings
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_BIT_TIMINGS_T *bt;
UINT32 bitrate;

bt = (CAN_BIT_TIMINGS_T*)malloc(sizeof(CAN_BIT_TIMINGS_T));
bt->tq = 125;
bt->propSeg = 6;
bt->phaseSeg1 = 7;
bt->phaseSeg2 = 2;
bt->sjw = 1;
bt->sam3 = 1;
bt->clock = 16000000;
…
/* open connection on an enabled interface */
…
CAN_SetTimings(&can, bt);
CAN_GetBitrate(&can, &bitrate);
free(bt);
```

sets CAN bit timings with time quantum equal to 125 nsec, propagation segment equal to 6 time quanta, phase buffer segment 1 equal to 7 time quanta, phase buffer segment 2 equal to 2 time quanta, synchronization jump width equal to 1 time quantum, bus line sampled three times at the sample point, oscillator clock frequency equal to 16 MHz; bitrate has been successfully set at 500 kbps

7.8. CAN_GetTimings()

This function reads the actual bit timings on a real CAN interface already enabled. Timing settings are the same on all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_GetTimings(CAN_CONNECTION_T *can,
CAN_BIT_TIMINGS_T *timings)
```

Parameters

`<can>` It's the name of the CAN_CONNECTIO_T on which bit timings are read.
`<timings>` Pointer to the CAN_BIT_TIMINGS_T struct that contains the timings to be read:

`<timings->tq>`	time quantum (nsec)
`<timings->propSeg>`	propagation segment (from 1 to 8)
`<timings->phaseSeg1>`	phase buffer segment 1 (from 1 to 8)
`<timings->phaseSeg2>`	phase buffer segment 2 (from 2 to 8)
`<timings->sjw>`	synchronization jump width (from 1 to 4)
`<timings->sam3>`	three times sampling mode (0: disabled, 1: enabled)
`<timings->clock>`	oscillator frequency clock (Hz)

Return values

`CAN_OK`	CAN bit timings have been successfully read
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_BIT_TIMINGS_T *bt;
UINT32 bitrate;

bitrate = 250000;
bt = (CAN_BIT_TIMINGS_T*)malloc(sizeof(CAN_BIT_TIMINGS_T));
```

```
…
/* open connection on an enabled interface */
…
CAN_SetBitrate(&can, bitrate, 16000000);
CAN_GetTimings(&can, bt);
free(bt);
```

reads CAN bit timings with bitrate set at 250 kbps. In this case bit timings are: time quantum equal to 250 nsec, propagation segment equal to 6 time quanta, phase buffer segment 1 equal to 7 time quanta, phase buffer segment 2 equal to 2 time quanta, synchronization jump width equal to 1 time quantum, bus line sampled three times at the sample point, oscillator clock frequency equal to 16 MHz.

7.9. CAN_SetCtrlMode()

This function enables normal, hardware loopback or listen-only mode on a real CAN interface already enabled.
Loopback hardware control mode allows internal transmission of messages from the transmit buffers to the receive buffers without actually transmitting messages on the CAN bus. In this mode no frames are received or sent on CAN bus.
Listen-only mode is a silent mode, meaning no messages will be transmitted while it is in this state but controller will be able to receive frames from CAN bus.
In normal mode, neither loopback nor listen-only mode is active. Normal mode is active after a controller reset.
Control mode changing is applied on all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_SetCtrlMode(CAN_CONNECTION_T *can,
CAN_CTRL_MODE_E ctrlMode)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which control mode must be set.
`<ctrlMode>` Control mode to be set:

CAN_NORMAL	disables hardware loopback and listen-only modes
CAN_LOOPBACK	actives hardware loopback control mode
CAN_LISTENONLY	actives listen-only control mode

Return values

CAN_OK	CAN control mode has been successfully set
CAN_CONNECTION_ERROR	CAN connection error
CAN_DISABLED_IF	CAN interface of the specified connection is disabled

`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_ERROR`	Unspecified error

Example
```
CAN_CONNECTION_T can;
…
/* open connection on an enabled interface */
…
CAN_SetCtrlMode(&can, CAN_LOOPBACK);
/* CAN controller was in normal control mode
and now it is in hardware loopback mode*/
…
CAN_SetCtrlMode(&can, CAN_LISTENONLY);
/* CAN controller is in listen-only mode */
…
CAN_SetCtrlMode(&can, CAN_NORMAL);
/* CAN controller is in normal control mode */
```
changes CAN controller control mode

7.10. CAN_GetCtrlMode()

This function reads the actual control mode on a real CAN interface already enabled.
Control mode is the same on all the CAN connections opened on the same CAN interface.

Prototype
```
CAN_ERROR_CODE_E CAN_GetCtrlMode(CAN_CONNECTION_T *can,
                                 CAN_CTRL_MODE_E *ctrlMode)
```

Parameters
`<can>` It's the name of the CAN_CONNECTION_T on which control mode is read.
`<ctrlMode>` Pointer to control mode to be read:

`CAN_NORMAL`	hardware loopback and listen-only are not active
`CAN_LOOPBACK`	hardware loopback control mode
`CAN_LISTENONLY`	active listen-only control mode

Return values

`CAN_OK`	CAN operating mode has been successfully read
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_ERROR`	Unspecified error

Example
```
CAN_CONNECTION_T can;
```

```
CAN_CTRL_MODE_E ctrlMode;
…
/* open connection on an enabled interface */
CAN_GetCtrlMode(&can, &ctrlMode);
```

reads the actual CAN controller control mode and stores its value in `ctrlMode` variable

7.11. CAN_GetState()

This function reads the actual state of a real CAN interface already enabled. CAN node state can be active, bus warning, bus passive, bus off or stopped.
State is the same on all the CAN connections opened on the same CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_GetState(CAN_CONNECTION_T *can,
CAN_STATE_E *state)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which state is read.
`<state>`	Pointer to the state to be read:

`CAN_ACTIVE`	CAN node is error active
`CAN_BUS_WARNING`	CAN node is bus warning
`CAN_BUS_PASSIVE`	CAN node is error passive
`CAN_BUS_OFF`	CAN node is bus off
`CAN_BUS_STOPPED`	CAN node is stopped

Return values

`CAN_OK`	CAN state has been successfully read
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_STATE_E state;
…
/* open connection on an enabled interface */
…
CAN_GetState(&can, &state);
```

reads CAN node state and stores its value in `state` variable

7.12. CAN_Read()

This function reads a frame from a CAN connection, according to filters and error signalling settings.
This function returns when a frame is received or when timeout expires. If timeout is equal to 0, this function waits until it reads a CAN frame.
Real and virtual CAN interfaces are supported.
Standard and extended format, remote and error signalling frames are supported.

Prototype

```
CAN_ERROR_CODE_E CAN_Read(CAN_CONNECTION_T *can,
CAN_FRAME_T *frame, UINT32 timeout)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which frame is read.
`<frame>`	Pointer to the frame to be read:
`<frame->canId>`	frame ID: low level 11-bits for standard frames and low level 29-bits for extended frames must be taken into consideration
`<frame->canDlc>`	frame data length code (from 0 to 8)
`<frame->data>`	frame data field: low level `canDlc`-bytes must be taken into consideration
`<frame->attributes>`	frame type; CAN attributes are:
`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format
`CAN_DATA_FRAME`	data frame
`CAN_REMOTE_FRAME`	remote frame
`CAN_ERROR_FRAME`	error signaling frame
`<timeout>`	Timeout in msec

Return values

`CAN_OK`	CAN data or request frame has been successfully read
`CAN_TIMEOUT_EXPIRED`	Timeout is expired
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_TYPE_ERROR`	CAN frame type error
`CAN_ERROR_SIGNAL`	Error signaling frame has been received (see Section 7.16)
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FRAME_T *frame;
CAN_ERROR_CODE_E code;
int i;

frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
…
/* open connection on an enabled interface */
code = CAN_Read(&can, frame, 0);
if (code == CAN_OK) {
       if (frame->attributes & CAN_DATA_FRAME) {
              if (frame->attributes & CAN_STANDARD_FRAME)
                     printf(“\nCAN ID = %x”, frame->canId & 0x7FF);
                            else
                     printf(“\nCAN ID = %x”, frame->canId & 0x1FFFFF);
              printf(“\nCAN DLC = %d”, frame->canDlc);

              for(i=0; i<frame->canDlc; i++ )
                     printf(“\nCAN data[%d] = %x”, i, frame->data[i]);
       }
}
free(frame);
```

waits and reads a CAN frame

7.13. CAN_Write()

This function writes a frame into a CAN connection.
Real and virtual CAN interfaces are supported.
Standard and extended format and remote frames are supported.

Prototype

```
CAN_ERROR_CODE_E CAN_Write(CAN_CONNECTION_T *can,
CAN_FRAME_T *frame)
```

Parameters

`<can>`	It’s the name of the CAN_CONNECTION_T on which frame must be written.
`<frame>`	Pointer to the frame to be written:

`<frame -> canId>`	frame ID: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<frame -> canDlc>`	frame data length code (from 0 to 8)
`<frame -> data>`	frame data field: low level `canDlc`-bytes are taken into consideration
`<frame -> attributes>`	frame type; CAN attributes masks are:

`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format
`CAN_DATA_FRAME`	data frame
`CAN_REMOTE_FRAME`	remote frame

Return values

`CAN_OK`	CAN data or request frame has been successfully written
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_TYPE_ERROR`	CAN frame type error
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FRAME_T *frame;

frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));

frame -> attributes = CAN_EXTENDED_FRAME | CAN_DATA_FRAME;
frame -> canId = 0x65263;
frame -> canDlc = 4;
frame -> data[0] = 0x45;
frame -> data[1] = 0x42;
frame -> data[2] = 0x25;
frame -> data[3] = 0xe5;
…
/* open connection on an enabled interface */
…
CAN_Write(&can, frame);
free(frame);
```

sends a CAN frame

7.14. CAN_CyclicSending()

This function manages cyclic sending of a frame on a CAN connection.
Cyclic sending is specified by a frame (format, data length, data bytes) and by a period.
This function allows to:

- start a new cyclic sending of a frame with the specified period
- modify the content of a frame that is cyclically sent, specifying the same frame format and ID and different data length or data bytes
- modify the period of a frame that is cyclically sent
- stop a cyclic sending of a frame, specifying only the same format and ID.

Real and virtual CAN interfaces are supported.
Standard and extended format and remote frames are supported.

Prototype

```
CAN_ERROR_CODE_E CAN_CyclicSending(CAN_CONNECTION_T *can,
CAN_FRAME_T *frame,  UINT32 timePeriod)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which frame must be sent.

`<frame>` Pointer to the frame to be cyclically sent:

`<frame -> canId>`	frame ID (low level 11-bits for standard frames and low level 29-bits for extended frames must be taken into consideration)
`<frame -> canDlc>`	frame data length code (from 0 to 8)
`<frame -> data>`	frame data field (low level `canDlc`-bytes must be taken into consideration)
`<frame -> attributes>`	frame type; CAN attributes masks are:

CAN_STANDARD_FRAME	standard frame format
CAN_EXTENDED_FRAME	extended frame format
CAN_DATA_FRAME	data frame
CAN_REMOTE_FRAME	remote frame

`<timePeriod>` Frame is sent every `timePeriod` msec; if it is zero, cyclic sending is stopped.

Return values

CAN_OK	CAN frame cyclic sending has been successfully started
CAN_STOP_CYCLIC	CAN frame cyclic sending has been successfully stopped
CAN_CONNECTION_ERROR	CAN connection error
CAN_DISABLED_IF	CAN interface of the specified connection is disabled
CAN_TYPE_ERROR	CAN type error
CAN_ERROR	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FRAME_T *frame;

frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));

frame -> attributes = CAN_EXTENDED_FRAME | CAN_DATA_FRAME;
frame -> canId = 0x65263;
```

```
        frame -> canDlc = 3;
        frame -> data[0] = 0x45;
        frame -> data[1] = 0x42;
        frame -> data[2] = 0x25;
        …
        /* open connection on an enabled interface */
        …
        CAN_CyclicSending(&can, frame, 1250);
        …
        /* other CAN operations */
        …
        frame -> data[0] = 0x41;
        frame -> data[1] = 0x40;
        frame -> data[2] = 0x25;

        CAN_CyclicSending(&can, frame, 1250);
        …
        /* other CAN operations */
        …
        CAN_CyclicSending(&can, frame, 0);
        free(frame);
```

sends a frame every 1250 msec, modifies its content and stops this cyclic sending

7.15. CAN_ErrorSignaling()

This function enables or disables error signalling on a specified CAN connection on a real CAN interface.

When error signalling is enabled, `CAN_Read()` is able to receive special frames with controller error state. These frames are sent by SocketCAN core and are not transmitted on the CAN bus. They have the following fields:

`<frame -> canId>` frame ID of a standard frame; error conditions are indicated by these masks:

```
CAN_ERROR_TX_TIMEOUT        0x00000001U  TX timeout
CAN_ERROR_LOSTARB                  0x00000002U  Lost arbitration
CAN_ERROR_CRTL                     0x00000004U  Controller problems
CAN_ERROR_BUSOFF                   0x00000040U  Bus off
CAN_ERROR_BUSERROR          0x00000080U  Bus error
```

`<frame -> canDlc>` frame data length code: it is always equal to 8

`<frame -> data>` frame data field; reaching of a CAN controller error status is indicated by these masks applied for frame -> data[1] (only if frame -> canId is equal to CAN_ERROR_CRTL):

```
CAN_ERROR_CRTL_RX_OVERFLOW        0x01          RX buffer overflow

CAN_ERROR_CRTL_RX_WARNING         0x04          Warning level for RX errors
CAN_ERROR_CRTL_TX_WARNING         0x08          Warning level for TX errors
CAN_ERROR_CRTL_RX_PASSIVE         0x10          Error passive status RX
CAN_ERROR_CRTL_TX_PASSIVE         0x20          Error passive status TX
```

<frame -> attributes> frame type; it is always equal to:
`CAN_STANDARD_FRAME | CAN_ERROR_FRAME`

When CAN connection is open, error signalling is disabled as default.

Prototype

```
CAN_ERROR_CODE_E CAN_ErrorSignaling(CAN_CONNECTION_T *can,
BOOLEAN enable)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which error signaling must be enabled/disabled.

`<enable>` Enable/disable flag:

`TRUE`	error signaling is enabled
`FALSE`	error signaling is disabled

Return values

`CAN_OK`	CAN error signaling has been successfully enabled / disabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is `virtual`
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FRAME_T *frame;
CAN_ERROR_CODE_E code;
frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
…
/* open connection on an enabled interface */
…
CAN_ErrorSignaling(&can, TRUE);
…
code = CAN_Read(&can, frame, 0);
if (code == CAN_ERROR_SIGNAL) {
if (frame->canId & CAN_ERROR_TX_TIMEOUT)
printf("\nTX timeout\n");
if (frame->canId & CAN_ERROR_LOSTARB)
printf("\nLost arbitration\n");
if (frame->canId & CAN_ERROR_BUSOFF)
printf("\nBus off\n");
if (frame->canId & CAN_ERROR_BUSERROR)
       printf("\nBus error\n");
if (frame->canId & CAN_ERROR_CRTL) {
       printf("\nController error:");

if (frame->data[1] & CAN_ERROR_CRTL_RX_OVERFLOW)
             printf("RX buffer overflow\n");
if (frame->data[1] & CAN_ERROR_CRTL_RX_WARNING)
             printf("Reached warning level for RX errors\n");
if (frame->data[1] & CAN_ERROR_CRTL_TX_WARNING)
             printf("Reached warning level for TX errors\n");
if (frame->data[1] & CAN_ERROR_CRTL_RX_PASSIVE)
             printf("Reached error passive status RX\n");
```

```
if (frame->data[1] & CAN_ERROR_CRTL_TX_PASSIVE)
                printf("Reached error passive status TX\n");
                    }
}
free(frame);
```

enables and shows error signaling

7.16. CAN_LocalLoopback()

This function enables or disables local loopback on CAN connections on the same CAN interface (real or virtual).
When local loopback is enabled on a CAN connection, all the CAN frames sent from this connection are looped back to the other CAN connections active on the same interface.
When CAN connection is open, local loopback is enabled as default.

Prototype

```
CAN_ERROR_CODE_E CAN_LocalLoopback(CAN_CONNECTION_T *can,
BOOLEAN enable)
```

Parameters

<can> It's the name of the CAN_CONNECTION_T on which local loopback must be enabled/disabled.
<enable> Enable/disable flag:

`TRUE`	local loopback is enabled
`FALSE`	local loopback is disabled

Return values

`CAN_OK`	CAN local loopback has been successfully enabled / disabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
…
/* open connection on an enabled interface */
…
CAN_LocalLoopback(&can, TRUE);
```

enables local loopback

7.17. CAN_ReceiveOwnFrames()

This function enables or disables receiving of the frames sent by the same CAN connection on a CAN interface (real or virtual).
When CAN connection is open, receiving of own frames is disabled as default.

Prototype

```
CAN_ERROR_CODE_E CAN_ReceiveOwnFrames(CAN_CONNECTION_T *can,
BOOLEAN enable)
```

Parameters

<can> It's the name of the CAN_CONNECTION_T on which receiving of own frames must be enabled/disabled.
<enable> Enable/disable flag:

`TRUE`	receiving of own frames is enabled
`FALSE`	receiving of own frames is disabled

Return values

`CAN_OK`	Receiving of own frames has been successfully enabled / disabled
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
…
/* open connection on an enabled interface */
…
CAN_ReceiveOwnFrames(&can, TRUE);
```
enables receiving of own frames

7.18. CAN_SetHardwareFilters()

This function defines and enables all the hardware CAN filters on a real CAN interface already enabled. CAN controller is able to receive from CAN bus only when the frame matches with the filters.
Every CAN hardware filter is defined by a `maskId` (used to determine which bits in the CAN ID are examined with the filters), by a `filterId` and by the format frame (standard or extended). A filter matches when:

(<format of the received frame> == <format of the filter>) &&
(<CAN ID of the received frame> & maskId == (filterId & maskId))

This function also supports filtering on high order 16-bits on data in standard format messages (CAN data byte filtering), in order to extend CAN ID hardware filtering. In this case, every filter is defined also by a `maskData` and a `filterData`. A filter matches when:

(<format of the received frame> == <format of the filter>) &&
(<CAN ID of the received frame> & maskId == filterId & maskId) &&
(<high order 16-bits on CAN data of the received frame> & maskData == filterData & maskData)

When frame has data length equal to 0, `maskData` is not applied; when frame has data length equal to 1, `maskData` is applied only for its most significant byte.

MCP2515 CAN controller supports two masks with respectively two and four filters; every mask defines one filter type (standard or extended frame format and data byte filtering).
Hardware filters for standard frames have effect only for standard frames, and vice versa.
All the frames from CAN bus are received if controller has a `maskId` equal to 0 with standard filter type and the the other `maskId` equal to 0 with extended filter type (see Example 2).
When a CAN node is enabled through CAN_Enable() function, hardware filters are reset and all two filter masks have `maskId` equal to 0, the first one with standard filter type and the second one with extended filter type (all the frames are received).
In order to enable one hardware filter 'A', controller must have all the filters equal to 'A' (all the two filter masks must be defined, see Example 3).
Hardware filters are enabled on all the CAN connections opened on the same CAN.

Prototype

```
CAN_ERROR_CODE_E CAN_SetHardwareFilters(CAN_CONNECTION_T *can,
CAN_HARDWARE_FILTERS_T *filters)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which filters are enabled.
`<filters>` Pointer to the filters to be added:

`<filters -> mask>` indicates one filter mask

`<filters -> mask[i].maskID>`	ID mask number `i`: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration; if maskID is 0, filter is disabled

`<filters -> mask[i].maskData>`	data mask number `i`: it is taken into consideration only if CAN data byte filtering is enabled; if maskData is 0, filter is disabled
`<filters -> mask[i].attributes>`	filter type of the mask number `i`; filter attributes can be:

`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format
`CAN_DATABYTE_FILTER`	CAN data byte filtering enabled (only for standard frame filter)

`<filters -> mask[i].filterId[j]>`	ID filter number `j` of the mask number `i`: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filters -> mask[i].filterData[j]>`	data filter number `j` of the mask number `i`: it is taken into consideration only if CAN data byte filtering is enabled

Return values

`CAN_OK`	CAN hardware filters have been successfully applied
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_FILTER_ERROR`	Filter type error
`CAN_ERROR`	Unspecified error

Example 1

```
CAN_CONNECTION_T can;
CAN_HARDWARE_FILTERS_T *filters;
CAN_FRAME_T *frame;

filters = (CAN_HARDWARE_FILTERS_T *)malloc(sizeof(CAN_HARDWARE_FILTERS_T));

filters->mask[0].attributes = CAN_STANDARD_FRAME | CAN_DATABYTE_FILTER;
filters->mask[0].maskId = 0x0F8;
filters->mask[0].maskData = 0xF0FF;
filters->mask[0].filterId[0] = 0x123;
filters->mask[0].filterId[1] = 0x124;
filters->mask[0].filterData[0] = 0xAABB;
filters->mask[0].filterData[1] = 0x67ABCD;

filters->mask[1].attributes = CAN_EXTENDED_FRAME;
filters->mask[1].maskId = 0x700;
```

```
filters->mask[1].filterId[0] = 0x187;
filters->mask[1].filterId[1] = 0x412;
filters->mask[1].filterId[2] = 0x412;
filters->mask[1].filterId[3] = 0x412;
…
/* open connection on an enabled interface */
…
CAN_SetHardwareFilters(&can, filters);
…
frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
CAN_Read(&can, frame, 0);
…
free(filters);
free(frame);
defines and enables four hardware filters (a frame is received from CAN bus
if it matches with at least one filter):
the first one accepts frames if
(frame -> attributes & CAN_STANDARD_FRAME) &&
(frame -> canId & 0x0F8 == 0x123 & 0x0F8) &&
(frame -> data[0] & 0xF0 == 0xAA & 0xF0) &&
(frame -> data[1] & 0xFF == 0xBB & 0xFF)
the second one accepts frames if
(frame -> attributes & CAN_STANDARD_FRAME) &&
(frame -> canId & 0x0F8 == 0x124 & 0x0F8) &&
(frame -> data[0] & 0xF0 == 0xAB & 0xF0) &&
(frame -> data[1] & 0xFF == 0xCD & 0xFF)

the  third one accepts frame  if
(frame -> attributes & CAN_EXTENDED_FRAME) &&
(frame -> canId & 0x700 == 0x187 & 0x700)
the  fourth one accepts frame  if
(frame -> attributes & CAN_EXTENDED_FRAME) &&
(frame -> canId & 0x700 == 0x412 & 0x700)
```

Example 2

```
CAN_CONNECTION_T can;
CAN_HARDWARE_FILTERS_T *filters;
CAN_FRAME_T *frame;

filters = (CAN_HARDWARE_FILTERS_T *)malloc(sizeof(CAN_HARDWARE_FILTERS_T));

filters->mask[0].attributes = CAN_STANDARD_FRAME;
filters->mask[0].maskId = 0x0;

filters->mask[1].attributes = CAN_EXTENDED_FRAME;
filters->mask[1].maskId = 0x0;
…
/* open connection on an enabled interface */
…
CAN_SetHardwareFilters(&can, filters);
…
frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
CAN_Read(&can, frame, 0);
…
free(filters);
free(frame);
```

All the frames (standard or extended) from CAN bus are received

Example 3

```
CAN_CONNECTION_T can;
CAN_HARDWARE_FILTERS_T *filters;
CAN_FRAME_T *frame;

filters = (CAN_HARDWARE_FILTERS_T *)malloc(sizeof(CAN_HARDWARE_FILTERS_T));

filters->mask[0].attributes = CAN_STANDARD_FRAME;
filters->mask[0].maskId = 0x0F8;
filters->mask[0].filterId[0] = 0x123;
filters->mask[0].filterId[1] = 0x123;

filters->mask[1].attributes = CAN_STANDARD_FRAME;
filters->mask[1].maskId = 0x0F8;
filters->mask[1].filterId[0] = 0x123;
filters->mask[1].filterId[1] = 0x123;
filters->mask[1].filterId[2] = 0x123;
filters->mask[1].filterId[3] = 0x123;
…
/* open connection on an enabled interface */
…
CAN_SetHardwareFilters(&can, filters);
…
frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
CAN_Read(&can, frame, 0);
…
free(filters);
free(frame);
```

defines and enables one hardware filters; a frame is received from CAN bus if:

```
(frame -> attributes & CAN_STANDARD_FRAME) &&
(frame -> canId & 0x0F8 == 0x123 & 0x0F8)
```

7.19. CAN_GetHardwareFilters()

This function reads the active hardware CAN filters on a real CAN interface already enabled.

When a CAN node is enabled through CAN_Enable() function, hardware filters are reset and CAN_GetHardwareFilters() returns an hardware filters structure with all two filter masks equal to 0, the first one with standard filter type and the second one with extended filter type (all the frames are received)
Hardware filters are the same in all the CAN connections opened on the same CAN.

Prototype

```
CAN_ERROR_CODE_E CAN_GetHardwareFilters(CAN_CONNECTION_T *can,
CAN_HARDWARE_FILTERS_T *filters)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which active filters are read.
`<filters>`	Pointer to the active filters to be read:

`<filters -> mask>` indicates one filter mask

`<filters -> mask[i].maskID>`	ID mask number `i`: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration; if maskID is 0, filter is disabled
`<filters -> mask[i].maskData>`	data mask number `i`: it is taken into consideration only if CAN data byte filtering is enabled; if maskData is 0, filter is disabled
`<filters -> mask[i].attributes>`	filter type of the mask number `i`; filter attributes can be:
`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format
`CAN_DATABYTE_FILTER`	CAN data byte filtering enabled (only for standard frame filter)
`<filters -> mask[i].filterId[j]>`	ID filter number `j` of the mask number `i`: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filters -> mask[i].filterData[j]>`	data filter number `j` of the mask number `i`: it is taken into consideration only if CAN data byte filtering is enabled

Return values

`CAN_OK`	CAN hardware filters have been successfully read
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_VIRTUAL_IF`	CAN interface of the specified connection is virtual
`CAN_FILTER_ERROR`	Filter type error
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_HARDWARE_FILTERS_T *filters;
CAN_FRAME_T *frame;

filters = (CAN_HARDWARE_FILTERS_T *)malloc(sizeof(CAN_HARDWARE_FILTERS_T));
…
/* open connection on an enabled interface */
…
CAN_GetHardwareFilters(&can, filters);
```

```
free(filters);
```
reads the active hardware filters

7.20. CAN_AddSoftwareFilter()

This function adds and enables a software filter on a specific CAN connection of a real or virtual CAN interface.
Software filter is defined by a `maskId` (used to determine which bits in the CAN ID are examined with the filters), by a `filterId` and by the format frame (standard or extended). A software filter for standard frame is applied also to extended frames, but not vice versa.
A filter for standard frames matches when:

<CAN ID of the received frame> & maskId == filterId & maskId

A filter for extended frames matches when:

(<format of the received frame> == CAN_EXTENDED_FRAME) &&
(<CAN ID of the received frame> & maskId == filterId & maskId)

If a frame matches with `n` software filters, it is received `n` times.
If no software filter is defined, all the frames are read (default condition on CAN connection opening). It is possible to define up to 30 software filters.

Prototype
```
CAN_ERROR_CODE_E CAN_AddSoftwareFilter(CAN_CONNECTION_T *can,
CAN_SOFTWARE_FILTER_T *filter)
```

Parameters

`<can>`	It's the name of the CAN_CONNECTION_T on which filter is enabled.
`<filter>`	Pointer to the filter to be added:

`<filter -> filterId>`	ID filter: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filter -> maskId>`	ID mask: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filter -> attributes>`	filter type; filter attributes can be:

`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format

Return values

`CAN_OK`	CAN filter has been successfully applied
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_FILTER_ERROR`	Filter type error
`CAN_TOO_ENABLES_FILTERS`	Too many filters have been enabled
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FILTER_T *filter1, *filter2;
CAN_FRAME_T *frame;

filter1 = (CAN_SOFTWARE_FILTER_T *)malloc(sizeof(CAN_SOFTWARE_FILTER_T));
filter1 -> attributes = CAN_EXTENDED_FRAME;
filter1 -> maskId = 0xF0FF;
filter1 -> filterId = 0x5A53;

filter2 = (CAN_SOFTWARE_FILTER_T *)malloc(sizeof(CAN_SOFTWARE_FILTER_T));
filter2 -> attributes = CAN_STANDARD_FRAME;
filter2 -> maskId = 0x70F;
filter2 -> filterId = 0x153;

frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
…
/* open connection on an enabled interface */
…
CAN_AddSoftwareFilter(&can, filter1);
CAN_AddSoftwareFilter(&can, filter2);
CAN_Read(&can, frame, 0);
…
free(filter1);
free(filter2);
free(frame);
```

enables two software filters (a frame is read `n` times if it matches with `n` software filters):

1. the first one accepts a frame if
   ```
   (frame -> attributes & CAN_EXTENDED_FRAME) &&
   (frame -> canId & 0xF0FF == 0x5A53 & 0xF0FF) &&
   ```
2. the second one accepts frames if
   ```
   frame -> canId & 0x70F == 0x153 & 0x70F
   ```

7.21. CAN_DelSoftwareFilter()

This function deletes a software filter previously added and enabled on a specific CAN connection of a real or virtual CAN interface.

Prototype

```
CAN_ERROR_CODE_E CAN_DelSoftwareFilter(CAN_CONNECTION_T *can,
CAN_SOFTWARE_FILTER_T *filter)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which filter must be deleted.
`<filter>` Pointer to the filter to be deleted:

`<filter -> filterid>`	ID filter: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filter -> maskId>`	ID mask: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<filter -> attributes>`	filter type; filter attributes can be:

CAN_STANDARD_FRAME	standard frame format
CAN_EXTENDED_FRAME	extended frame format

Return values

CAN_OK	CAN filter has been successfully deleted
CAN_CONNECTION_ERROR	CAN connection error
CAN_DISABLED_IF	CAN interface of the specified connection is disabled
CAN_NON_EXISTENT_FILTER	Filter not previously added
CAN_FILTER_ERROR	Filter type error
CAN_ERROR	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FILTER_T *filter;
CAN_FRAME_T *frame;

filter = (CAN_SOFTWARE_FILTER_T *)malloc(sizeof(CAN_SOFTWARE_FILTER_T));
filter -> attributes = CAN_EXTENDED_FRAME;
filter -> maskId = 0xF00F;
filter -> filterId = 0x5A53;

frame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));
…
/* open connection on an enabled interface */
…
CAN_AddFilter(&can, filter);
…
CAN_Read(&can, frame, 0);
…
CAN_DelFilter(&can, filter);
```

```
…
CAN_Read(&can, frame, 0);
…
free(filter);
free(frame);
```

defines, adds and delete a software CAN filter: in the first `CAN_Read() filter` is active, in the second `CAN_Read() filter` is not considered.

7.22. CAN_SoftwareFiltersList()

This function reads the added and enabled software CAN filters on a specific CAN connection of a real or virtual CAN interface.
If filter list is NULL, this function returns only the number of the enabled filters.

Prototype

```
CAN_ERROR_CODE_E CAN_SoftwareFiltersList(CAN_CONNECTION_T *can,
CAN_SOFTWARE_FILTER_T **filters, UINT16 *number)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T about which filters are read.
`<filters>` Pointer to the filters list to be shown:

`<*filter -> filterId>`	ID filter: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<*filter -> maskId>`	ID mask: low level 11-bits for standard frames and low level 29-bits for extended frames are taken into consideration
`<*filter -> attributes>`	filter type; filter attributes can be:

`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format

`<*number >` number of enabled filters

Return values

`CAN_OK`	No error
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_SOFTWARE_FILTER_T *filter;
CAN_SOFTWARE_FILTER_T **filtersList;
int activeFilters;
int i = 0;

/* filter definition */
         …
/* open connection on an enabled interface */
…
CAN_AddSoftwareFilter(&can, filter);
…
CAN_SoftwareFiltersList(&can, NULL, &activeFilters);

filtersList = (CAN_SOFTWARE_FILTER_T **)malloc(sizeof(CAN_SOFTWARE_FILTER_T
*) * activeFilters);

for(i=0; i<activeFilters;i++)
filtersList[i] = (CAN_SOFTWARE_FILTER_T *)
malloc(sizeof(CAN_SOFTWARE_FILTER_T));

CAN_SoftwareFiltersList(&can, filtersList, &activeFilters);

for(i=0; i<activeFilters;i++)
printf("\nsoftware filter %d: maskId %x, filterId %x\n", i, filtersList[i]
-> maskId, filtersList[i] -> filterId);


for(i=0; i<activeFilters;i++)
             free(filtersList[i]);
free(filtersList);
```

shows active software filters

7.23. CAN_SetContentFilter()

This function enables or disables monitoring for content change according to a frame mask on a specific CAN connection.
Frame mask indicates the type (standard or extended frame), CAN ID and data mask.
CAN_ReadChangedContent() will read a frame only when content change is detected according to active content filters.
Content is considered changed when the read frame (all the following conditions must be verified):

- has the same type as one of the frame mask;
- has the same CAN ID as one of the frame mask;

- has a different CAN DLC or data field compared to the last received frame with the same CAN ID (only the bits set to '1' in data field frame mask are considered)

It works on real or virtual CAN interface. Standard and extended format frames are supported.
More than one filter can be active at the same time on the same CAN connection.

Prototype

```
CAN_ERROR_CODE_E CAN_SetContentFilter(CAN CONNECTION_T *can,
CAN_FRAME_T *frameMask, BOOLEAN enable)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which content filter is enabled/disabled.
`<frameMask>` Pointer to the mask frame:

`<frameMask -> canId>`	frame mask ID (low level 11-bits for standard frames and low level 29-bits for extended frames must be taken into consideration)
`<frameMask -> canDlc>`	frame mask data length code (from 0 to 8)
`<frame -> data>`	frame mask data field (low level `canDlc`-bytes must be taken into consideration)
`<frame -> attributes>`	frame type; CAN attributes masks are:

CAN_STANDARD_FRAME	standard frame format
CAN_EXTENDED_FRAME	extended frame format

`<enable>` Enable/disable flag:

TRUE	content filter is enabled
FALSE	content filter is disabled

Return values

CAN_OK	Content filter has been successfully enabled/disabled
CAN_CONNECTION_ERROR	CAN connection error
CAN_DISABLED_IF	CAN interface of the specified connection is disabled
CAN_TYPE_ERROR	CAN type error
CAN_ERROR	Unspecified error

Example

See example in Section 7.24

7.24. CAN_ReadChangedContent()

This function reads a frame only when content change is detected according to `CAN_SetContentFilter()` settings (see Section 7.23).
`CAN_ReadChangedContent()` is able to read only the frames with the same IDs specified in the `frameMasks` passed to `CAN_SetContentFilter()` when content filters are enabled. If no content filter is enabled, `CAN_ReadChangedContent()` reads nothing.
`CAN_ReadChangedContent()` does not apply software filters. Besides if a frame with content change is received, it will be read both `CAN_Read()` and `CAN_ReadChangedContent()`: if both these two functions are used, the same frame could be read therefore two times.
This function returns when a frame with changed content is received or when timeout expires. If timeout is equal to 0, this function waits until it reads a frame with changed content.
It works on real or virtual CAN interface. Standard and extended format are supported.

Prototype

```
CAN_ERROR_CODE_E CAN_ReadChangedContent(CAN CONNECTION_T *can,
CAN_FRAME_T *frame, UINT32 timeout)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T on which frame is read.
`<frame>` Pointer to the frame to be read:

`<frame -> canId>`	frame ID (low level 11-bits for standard frames and low level 29-bits for extended frames must be taken into consideration)
`<frame -> canDlc>`	frame data length code (from 0 to 8)
`<frame -> data>`	frame data field (low level `canDlc`-bytes must be taken into consideration)
`<frame -> attributes>`	frame type; CAN attributes masks are:

`CAN_STANDARD_FRAME`	standard frame format
`CAN_EXTENDED_FRAME`	extended frame format

`<timeout>` Timeout in msec

Return values

`CAN_OK`	Changes in data frame have been successfully received
`CAN_TIMEOUT_EXPIRED`	Timeout is expired
`CAN_CONNECTION_ERROR`	CAN connection error
`CAN_DISABLED_IF`	CAN interface of the specified connection is disabled
`CAN_TYPE_ERROR`	CAN type error
`CAN_ERROR`	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_FRAME_T *frameMask;
CAN_FRAME_T *receivedFrame;
CAN_ERROR_CODE_E code;
int i, j;

frameMask = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));

receivedFrame = (CAN_FRAME_T *)malloc(sizeof(CAN_FRAME_T));

frameMask -> attributes = CAN_STANDARD_FRAME;
frameMask -> canId = 0x123;
frameMask -> canDlc = 2;
frameMask -> data[0] = 0x00;
frameMask -> data[1] = 0xFF;
frameMask -> data[2] = 0xFF;
…
/* open connection on an enabled interface */
…
CAN_SetContentFilter(&can, frameMask, TRUE);
…
         for (i=0; i<50; i++) {
code = CAN_ReadChangedContent(&can, receivedFrame, 0);

if(code == CAN_OK) {
       printf(“\nreceived changed data frame with CAN ID 0x123”);
       for(j=0; j<receivedFrame->canDlc; j++)

              printf(“\ndata[i] = %x”, receivedFrame->data[i]);
}
…
free(frameMask);
free(receivedFrame);
```

only the standard frames with ID = 0x123 and with the second data byte or DLC changed compared to the last received frame are shown

7.25. CAN_GetDeviceStats()

This function reads the CAN device statistics. This function is applicable only on real CAN interface.
Device statistics is the same on all the CAN connections opened on the same CAN interface. It is reset when controller module is loaded.

Prototype

```
CAN_ERROR_CODE_E CAN_GetDeviceStats(CAN_CONNECTION_T *can,
                           CAN_DEVICE_STATS_T *devStats)
```

Parameters

`<can>` It's the name of the CAN_CONNECTION_T about which device statistics is read.

`<devStats>` Pointer to the CAN_DEVICE_STATS_T struct that contains the device statistics:

`<devStats -> errorWarning>`	CAN error warnings counter
`<devStats -> dataOverrun>`	CAN data overruns counter
`<devStats -> wakeUp>`	CAN wakeups counter
`<devStats -> busError>`	CAN bus errors counter
`<devStats -> errorPassive>`	CAN error passive states counter
`<devStats -> arbitrationLost>`	CAN arbitration lost counter
`<devStats -> restarts>`	CAN restarts counter
`<devStats -> busErrorAtInit>`	CAN bus error on controller starting counter

Return values

CAN_OK	CAN device statistics has been successfully read
CAN_CONNECTION_ERROR	CAN connection error
CAN_DISABLED_IF	CAN interface of the specified connection is disabled
CAN_VIRTUAL_IF	CAN interface of the specified connection is virtual
CAN_ERROR	Unspecified error

Example

```
CAN_CONNECTION_T can;
CAN_DEVICE_STATS_T *devStats;

devStats = (CAN_DEVICE_STATS_T*)malloc(sizeof(CAN_DEVICE_STATS_T));
…
/* open connection on an enabled interface */
…
CAN_GetDeviceStats(&can, devStats);
…
free(devStats);
```

fills `devStats` with the device statistics

8. Acronyms and Abbreviations

Term	Definition
CAN	Controller Area Network
LLC	Logical Link Control
MAC	Media Access Control
ISO	International Organization for Standardization
OSI	Open Systems Interconnection
HLP	Higher Layer Protocols
OS	Operating System
BCM	Broadcast Manager
CIA	CAN in Automation