Vector CAN Driver

Technical Reference

Texas Instruments

TMS470

DCAN

Authors	Georg Pflügel, Karol Kostolny, Sebastian Gaertner, Arthur Jendrusch
Versions:	1.05.00
Status:	Released

Contents

History

Author	Date	Version	Remarks
Georg Pflügel	12.06.2007	1.00	creation
Karol Kostolny	28.08.2007	1.01	Low level message transmit feature added
Sebastian Gärtner	28.07.2009	1.02	Support new derivative TMS470MSF542
Georg Pflügel	25.10.2010	1.03	Add description for DCAN Issue#22 workaround Support new derivative TMS570PSFC66
Georg Pflügel	09.12.2010	1.03.01	Add description for the already supported derivatives TMS570PSFC61, TMS570LS1x and TMS570LS2x
Georg Pflügel	29.09.2011	1.03.02	Add description for the already supported derivatives TMS470MSF542, TMS470MF03107, TMS470MF04207 and TMS470MF06607.
Arthur Jendrusch	20.12.2011	1.03.03	CAN Driver Version changed to V1.14.01 Support new derivative TMS570LS30316U
Arthur Jendrusch	16.04.2012	1.03.04	Support new derivative TMS570LS12004U
Georg Pflügel	04.06.2012	1.04.00	Support of local dower down mode Support of wakeup polling
Georg Pflügel	06.12.2012	1.05.00	Support new derivatives TMS570LS0322 and TMS470PSF764

1 Introduction

The concept of the CAN driver and the standardized interface between the CAN driver and the application is described in the document **TechnicalReference_CANDriver.pdf**. The CAN driver interface to the hardware is designed in a way that capabilities of the special CAN chips can be utilized optimally. The interface to the application was made identical for the different CAN chips, so that the "higher" layers such as network management, transport protocols and especially the application would essentially be independent of the particular CAN chip used.

This document describes the hardware dependent special features and implementation specifics of the CAN Chip D-CAN on the microcontrollers TMS470 and TMS570.

2 Important References

The following table summarizes information about the CAN Driver. It gives you detailed information about the versions, derivatives and compilers. As a very important information the documentations of the hardware manufacturers are listed. The CAN Driver is based upon these documents in the given version.

Drivers	RI	Derivative	Compiler	Hardware Manufacturer Document Name	Version
1.14.01	1.5	TMS470PSF761	Texas Instruments	TMS470PSF761 DesignSpec.pdf	Rev 0.8
		TMS570PSF762	ARM	TMS570PSF762_1.5.pdf	Rev 1.5
		TMS470PSF764		TMS470PSF764 Delphinus datasheet .pdf	SPNS146
		TMS470MSF542		TMS470MSF54x TRM (Draft)	02/2009
				TMS470MSF542PZ DesignSpec.pdf	Rev 1.01
		TMS570PSFC66		TMS570PSFC66_design_specification_22.pdf	Rev 2.2
				TMS570PSFC66_device_datasheet.pdf	SPNS141
		TMS570PSFC61		TMS570PSFC61_Specification_044.pdf	Rev 0.44
		TMS570LS30316U		Gladiator_design_specification_GM_Auto.pdf	V2.5.1
		TMS570LS12004U			
		TMS570LS0322		SPNS186_TMS570LS0x32_DataSheet.pdf	SPNS186
				DCAN_reference_guide_v0_23.pdf	V 0.23

Drivers: This is the current version of the CAN Driver
RI: Shows the version of the Reference Implementation and therefore the functional scope of the CAN Driver
Derivative: This can be a single information or a list of derivatives, the CAN Driver can be used on.
Compiler: List of Compilers the CAN Driver is working with
Hardware Manufacturer Document Name: List of hardware documentation the CAN Driver is based on.
Version: To be able to reference to this hardware documentation its version is very important.

2.1 Known Compatible Derivatives

Texas Instruments has established a new name space for the TMS570 derivatives. With this name space it is possible to rename existing derivatives. Future planed derivatives will named up to now with this name space too.

Old name supported by Geny:	With this selection this derivatives from new name space will run too:	Comment:
TMS570PSFC66	TMS570LS101xx	1M Flash 128k RAM
	TMS570LS102xx	1M Flash 160k RAM
	TMS570LS202xx	2M Flash 160k RAM
TMS470MSF542	TMS470MF03107	320k Flash 16k RAM
	TMS470MF04207	448k Flash 24k RAM
	TMS470MF06607	640k Flash 64k RAM

3 Usage of Controller Features

3.1 [#hw_comObj] - Communication Objects

Depending of the controller the CAN cells provide a specific number of mailboxes.

Controller	#Objects of CAN cell 1	#Objects of CAN cell 2	#Objects of CAN cell 3
TMS470PSF761	64	32	-
TMS570PSF762	64	32	-
TMS470PSF764	64	32	-
TMS470MSF542	16	32	-
TMS570PSFC61	64	64	32
TMS570PSFC66	64	64	32
TMS570LS30316U	64	64	64
TMS570LS12004U	64	64	64
TMS570LS0322	32	16	-

The generation tool supports a flexible allocation of message buffers. In the following tables the configuration variants of the CAN driver are listed. The message buffers are allocated in the following order for each channel:

Obj number	Obj type	No. of Objects	comment
1 – n	Tx Full CAN	0-n_{msg}	These objects are used by *CanTransmit()* to send a certain message. The user must define statically (Generation Tool) which CAN messages are located in such Tx FullCAN objects. The Generation Tool distributes the messages to the FullCAN objects according to their identifier priority.
m	Tx Normal	1	This object is used by *CanTransmit()* to send several messages. If the transmit message object is busy, the transmit request is stored in a queue
o	Low Level Tx	0-1	This object is used by *CanMsgTransmit()* to send it's messages, if the low level transmit functionality is selected.
p – q	unused	0-n_{msg}	These objects are not used. It depends on the configuration of receive and transmit objects if unused objects are available.

r – x	Rx Full CAN	$0\text{-}n_{msg}$	These objects are used to receive specific CAN messages. The user defines statically (Generation Tool) that a CAN message should be received in a FullCAN message object. The Generation Tool distributes the message to the FullCAN objects.
y – z	Basic CAN	2-4	All other CAN messages (Application, Diagnostics, Network Management) are received via the Basic CAN message object.

n_{msg} = (Max number of objects) – (number of Tx Normal objects) – (number of Basic CAN objects)

Example

For a CAN cell with 32 objects the following values are assumed:

Configurations with Standard Id or Extended Id: x = 30, y = 31, z = 32

Configurations with Mixed Id: x = 28, y = 29, z = 32

If the configuration contains Standard Ids and Extended Ids (configuration with Mixed Id), the Basic CAN use 4 hardware message objects. Two of them will be used for the reception of the Standard Ids and two will be used for the reception of the Extended Ids.

3.2 Miscellaneous

The CAN driver was designed to run in privileged mode only. There is no support for user mode.

4 [#hw_sleep] - SleepMode and WakeUp

The CAN module can be switched into sleep mode by calling the function CanSleep and from sleep into operation mode by calling the function CanWakeUp. There are two power-down modes available, the global power-down mode and the local power-down mode. The first is supported by all TMS570 DCAN derivates, the second only if it is documented in the datasheet. This is because the CAN controller has not initially supported the local power-down mode. It was added to the DCAN cell since documented in the reference guide revision 0.30.

4.1 Global Power down mode

The configuration-bits to set and reset the hardware of the D_CAN into and from global power down mode are not inside the CAN-controller. It is part of the Power-down-management of the CPU. To make the driver independent of access to the configuration bits, there are two callback-functions inside CanSleep and CanWakeUp.

ApplCanGoToSleepModeRequest

This function will be called from CanSleep. The user has to add this function to the application with some code inside to set the CAN-controller into sleep mode. Parameter CAN_CHANNEL_CANTYPE_ONLY is **void** for Single Receive Channels (SRC) and **channel** for Multiple Receive Channel (MRC).

Example:

```
vuint8 ApplCanGoToSleepModeRequest(CAN_CHANNEL_CANTYPE_ONLY)
{
  /* Quadrants are 256 bytes each, so DCAN1 is QUAD0 and QUAD1, DCAN2 is QUAD2 
and QUAD3 */
  if (channel == 0)
  {
    *((vuint32 *)0xFFFFE084 /* Peripheral Power-Down Set Register 1 */) = 0x00000003; /* 
QUAD0 and QUAD1 */
  }
  else if (channel == 1)
  {
    *((vuint32 *)0xFFFFE084 /* Peripheral Power-Down Set Register 1 */) = 0x0000000C; /* 
QUAD2 and QUAD3 */
  }
  return kCanOk;
}
```

ApplCanWakeUpFromSleepModeRequest

This function will be called from CanWakeUp. The user has to add this function to the application with some code inside to reset the CAN-controller from sleep mode. Parameter CAN_CHANNEL_CANTYPE_ONLY is **void** for Single channel and **channel** for Multiple channel.

Example:

```
vuint8 ApplCanWakeUpFromSleepModeRequest(CAN_CHANNEL_CANTYPE_ONLY)
{
  /* Quadrants are 256 bytes each, so DCAN1 is QUAD0 and QUAD1, DCAN2 is QUAD2
and QUAD3 */
  if (channel == 0)
  {
    *((vuint32 *)0xFFFFE0A4 /* Peripheral Power-Down Clear Register 1 */) = 0x00000003;
/* QUAD0 and QUAD1 */
  }
  else if (channel == 1)
  {
    *((vuint32 *)0xFFFFE0A4 /* Peripheral Power-Down Clear Register 1 */) = 0x0000000C;
/* QUAD2 and QUAD3 */
  }

  return kCanOk;
}
```

4.2 Local Power down mode

If the local power down mode is selected, the PDR bit inside the CAN cell will be used to set the CAN cell into the sleep mode. With this there are no callback functions necessary and the application has not to handle some additional hardware register.

5 [#hw_loop] - Hardware Loop Check

For the feature Hardware Loop Check (see TechnicalReference_CANDriver in the chapter Hardware Loop Check) this CAN Driver provides the following timer identifications:

KCanLoopIrqReq
Where is the loop implemented?
CAN Interrupt service routine.
What is the loop for?
Loop over all pending interrupts (Rx, Tx).
Is the loop channel dependent? Can this timer identification be called reentrant?
One loop for each channel, no reentrant call.
How often is `ApplCanTimerLoop` called?
Once with every pending interrupt request or permanently until loop exit.
Maximum expected duration of the loop or maximum expected calls of the loop
Depend on interrupt occurrence.
Reasons for a delay - why is the maximum expected duration exceeded?
Defect in hardware (what leads to a longer duration than the maximum expected time)
If the loop does not end and the application has to terminate the loop, what has to be done then?
Exit loop and retrigger interrupt after application action is done.

kCanLoopBusyReq
Where is the loop implemented?
CanCopyDataAndStartTransmission, CanBasicCanMsgReceived, CanFullCanMsgReceived CanHL_TxConfirmation, CanMsgTransmit
What is the loop for?
Check that the CAN-cell leaves the busy state.
Is the loop channel dependent? Can this timer identification be called reentrant?
One loop for each channel, reentrant call.
How often is `ApplCanTimerLoop` called?
Permanently until CAN-cell leaves the busy state.
Maximum expected duration of the loop or maximum expected calls of the loop
The busy state need 3-6 CAN_CLK periods.
Reasons for a delay - why is the maximum expected duration exceeded?
Defect in hardware (what leads to a longer duration than the maximum expected time)
If the loop does not end and the application has to terminate the loop, what has to be done then?
If the loop does not end and the application has to terminate the loop, CanInit has to be called.

6 [#hw_busoff] - Bus off

The DCAN CAN-controller contains an Auto-Bus-On mode. Using this mode, the DCAN automatically starts a bus-off-recovery sequence by resetting bit Init to *zero* after a delay defined by register „Auto Bus On Time“, when DCAN is getting bus-off.

The feature Auto-Bus-On is deactivated by the CAN-driver and the software has to decide, whether to leave DCAN in bus-off state or to start the bus-off-recovery sequence by resetting the Init bit. The CAN-driver supports the application with the call-back function ApplCanBusOff and the macro CanResetBusStart, that is defined to CanInit.

The application has to call CanResetBusStart as soon as possible after the CAN driver has made a busoff notification by calling ApplCanBusOff. After this the CAN-controller will be initialized again and the Init bit will be reset.

7 CAN Driver Features

7.1 [#hw_feature] - Feature List

CAN Driver Functionality

		Standard	HighEnd
		Texas Instruments / ARM	Texas Instruments / ARM
Initialization			
Power-On Initialization		■	■
Re-Initialization		■	■
Transmission			
Transmit Request		■	■
Transmit Request Queue		■	■
Internal data copy mechanism		■	■
Pretransmit functions		■	■
Common confirmation function		■	■
Confirmation flag		■	■
Confirmation function		■	■
Offline Mode		■	■
Partial Offline Mode		■	■
Passive Mode		■	■
Tx Observe mode		■	■
Dynamic TxObjects	ID	■	■
	DLC	■	■
	Data-Ptr	■	■
Full CAN Tx Objects		■	■
Cancellation in Hardware		■	■
Low Level Message Transmit			■
Reception			
Receive function		■	■
Search algorithms	Linear	■	■
	Table		
	Index		
	Hash	■	■
Range specific precopy functions (min. 2, typ.4)		4	4
DLC check		■	■
Internal data copy mechanism		■	■
Generic precopy function		■	■
Precopy function		■	■
Indication flag		■	■
Indication function		■	■

Message not matched function		■	■
Overrun Notification		■	■
FullCAN overrun notification		■	■
Multiple BasicCAN			■
Rx Queue			■
Bus off			
Notification function		■	■
Nested Recovery functions		■	■
Sleep Mode			
Mode Change		■	■
Preparation		■	■
Notification function		■	■
Special Feature			
Status		■	■
Security Level		■	■
Assertions		■	■
Hardware loop check		■	■
Stop Mode		■	■
Support of OSEK operating system		■	■
Polling Mode	Tx	■	■
	Rx (FullCAN objects)	■	■
	Rx (BasicCAN objects)	■	■
	Error	■	■
	Wakeup	■	■
Individual Polling			■
Multi-channel		■	■
Support extended ID addressing mode		■	■
Support mixed ID addressing mode		■	■
Support access to error counters		■	■
Copy functions		■	■
CAN RAM check		■	■
Interrupt-lock-level			

7.2 Description of Hardware related features

7.2.1 [#hw_status] – Status

If a status is not supported, the related macro returns always false.

CanHwIsOk(state)	■
CanHwIsWarning(state)	■
CanHwIsPassive(state)	■
CanHwIsBusOff(state)	■
CanHwIsWakeup(state)	■
CanHwIsSleep(state)	■
CanHwIsStart(state)	■
CanHwIsStop(state)	■
CanIsOnline(state)	■
CanIsOffline(state)	■

7.2.2 [#hw_stop] - Stop Mode

The function CanStop can be called to switch the CAN driver into stop mode. Then the CAN module will enter the listen only mode. In this mode the CAN interface does not communicate, i.e. no acknowledge and no active error flags are driven to the CAN bus. The error counters stay at the current value.

The function CanStart has to be called to leave the stop mode and to switch the CAN hardware back into operation mode.

CanOffline must be called before calling CanStop. CanOnline must be called after CanStart to enable message transmission.

7.2.3 [#hw_int] - Control of CAN Interrupts

The application has to initialize the CAN I/O port pins and the CAN interrupt request register before calling function CanInitPowerOn.

7.2.4 [#hw_cancel] - Cancel in Hardware

With the feature “cancel in hardware” it is possible to clear a transmit request direct inside the CAN-controller hardware. This feature can be used to clear the pending transmit request of a CAN-message that can not be send out of the hardware, because the CAN-message can not arbitrate the bus.

If the feature cancel in Hardware is used, it is necessary to call the Tx-Task cyclic.

	Yes	No
Has the CanTxTask() to be called by the application to handle the canceled transmit request in the hardware?	■	

Cancelling transmission of messages via `CanCancelTransmit` or `CanCancelMessageTransmit`:

In some cases the callback function `ApplCanTxConfirmation` is called for an already cancelled message. This is how this CAN Driver reacts:

	Yes	No
ApplCanConfirmation() is only called for transmitted messages. Successfully cancelled messages are not notified. That means the CAN Driver is able to detect whether is message is transmitted even if the application has tried to cancel the message.	■	

After a message has finished the arbitration of the bus, it is no more possible to cancel this message. Because of this, after cancel in Hardware was used, it is necessary to wait a security delay time to be sure that a message, that was not able to cancel, was send. This delay time must be the maximal length of a CAN-message (132 Bittimings). So the wait time depends from the used Baudrate and is:

wait time [sec] = 132Bit * (1 / Baudrate [Bit/sec])

Example

Time for 100 kBaud:

wait time [sec] = 132Bit * (1 / 100000 [Bit/sec]) = 0,00132 sec

This wait time can be produced with the two call back functions ApplCanTxCancelInHwStart and ApplCanTxCancelInHwConfirmed.

```
void ApplCanTxCancelInHwStart(CanObjectHandle txHwObject)
{
}
```

This call back function will be called once time after the transmit request is cleared from the hardware. It can be used from the application to start a wait time. This wait time depends on the used baudrate.

```
vuint8 ApplCanTxCancelInHwConfirmed(CanObjectHandle txHwObject)
{
}
```

This call back function will be called from the CanTxTask. If the CanTxTask is called cyclic, ApplCanTxCancelInHwConfirmed can be used from the application to count a wait time down. The return value of this call back function has to be False after the delay time is over, and True during the delay time.

7.2.5 Polling Mode

The driver supports Rx Full-CAN Polling, Rx Basic-CAN Polling, Tx Polling and Error Polling. Wake-up polling is not supported. If the hardware wakes up, a Status Interrupt will be generated. It is not possible to notify a Wakeup with a polling mode. Because of this, it is not allowed to activate the feature Sleep-Wakeup if Error Polling is configured.

8 [#hw_assert] - Assertions

In case of a user assertion:	
kErrorInitObjectHdlTooLarge	`CanInit()` called with parameter too large
kErrorTxHdlTooLarge	`CanTransmit()` called with transmit handle too large
kErrorIntRestoreTooOften	`CanInterruptRestore()` called too often
kErrorIntDisableTooOften	`CanInterruptDisable()` called too often
kErrorAccessedInvalidDynObj	`CanGetDynTxObj()`, `CanReleaseDynTxObj()` or `CanDynTxObjSet...()` is called with wrong transmit handle (transmit handle too large)
kErrorAccessedStatObjAsDyn	`CanGetDynTxObj()`, `CanReleaseDynTxObj()` or `CanDynTxObjSet...()` is called with wrong transmit handle (transmit handle depends on a static object)
kErrorDynObjReleased	UserConfirmation() or UserPreTransmit() is called for a dynamic object which is already released.
In case of a generation assertion:	
kErrorToManyFullCanObjects	The generated number of Full-CAN Objects is too big.
In case of a hardware assertion:	
kErrorTxBufferBusy	Hardware transmit object is busy, but this is not expected.
kErrorRxBufferBusy	Hardware receive object is busy, but this is not expected.
kErrorHwObjNotInPolling	
In case of a internal assertion:	
kErrorIllIrptNumber	A CAN-Interrupt occurs with a not valid Interruptnumber.
kErrorHwObjNotInPolling	A Hardwareobject that is configured to Pollingmode generates an unexpected CAN-Interrupt.

9 API

9.1 Category

Single Receive Channels (SRC)		■
	A "Single Receive Channel" CAN Driver supports one CAN channel.)	
Multiple Receive Channel (MRC):		■
	A "Single Receive Channel" CAN Driver is typically extended for multiple channels by adding an index to the function parameter list (e.g. CanOnline() becomes to CanOnline(channel)) or by using the handle as a channel indicator (e.g. CanTransmit(txHandle)).	

10 Implementations Hints

These options are highly compiler dependent. The necessary options are described in the installation instructions.

10.1 Important Notes

1.) The following condition will lead to an endless recursion in the CAN Driver:

recursive call of 'CanTransmit' within a confirmation routine, if the CAN Driver has been set into the passive state by `CanSetPassive`.

recommendations =>

- NO CALL OF `CanTransmit` WITHIN CONFIRMATION-ROUTINES
- PLEASE USE `CanSetPassive` ONLY ACCORDING TO THE DESCRIPTION

2.) Only the transmit line of the CAN Driver is blocked by the functions `CanOffline()`. However, messages in the transmit buffer of the CAN-Chip, are still sent. For a reliable prevention of this fact, call function `CanInit` after calling `CanOffline()`. The order of the two function calls is urgently required, due to the fact, that `CanInit()` is only allowed in offline mode.

3.) Resetting indication flags and confirmation flags is done by Read-Modify-Write. The application is responsible for consistence. `CanGlobalInterruptDisable()` and `CanGlobalInterruptRestore()` must be called to avoid interruption by the CAN. Confirmations or indications can be lost otherwise.

4.) [TMS470MSF542 only:] The CAN driver will suspend all interrupts by disabling all interrupts of the same or lower level (i.e. larger priority value). The chosen level must be equal or higher than the highest level of any functionality of the CAN Driver (Wakeup Interrupt, signal access, etc). To allow this the interrupt nesting option has to be disabled during all CAN driver operations.

5.) [TMS470MSF542 only:] All external interrupts (i.e. all interrupts controlled by M3VIM) have to be configured as ISR type. NMI exceptions would pass the global interrupt suspension of the CAN driver.

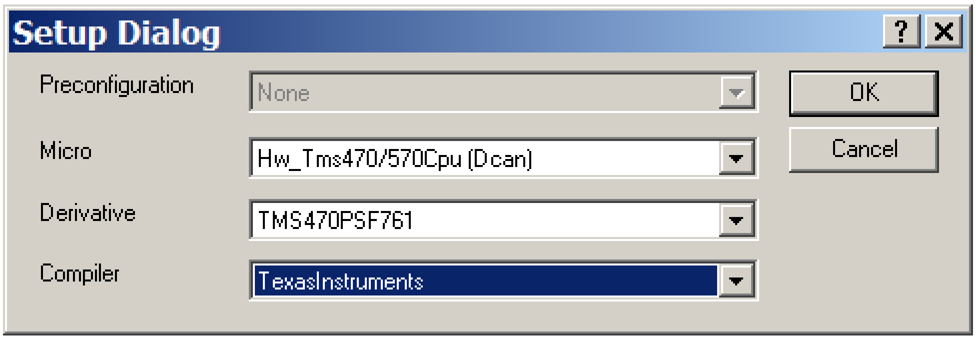
11 Configuration

11.1 Configuration by GENy

Using the Generation Tool the complete configuration can be done by the tool. The configuration options common to all CAN Drivers are described in the CAN Driver manual TechnicalReference_CANDriver.pdf.

	Info To get further information please refer to the Online-Help of the Generation Tool.

11.1.1 Compiler and Chip Selection

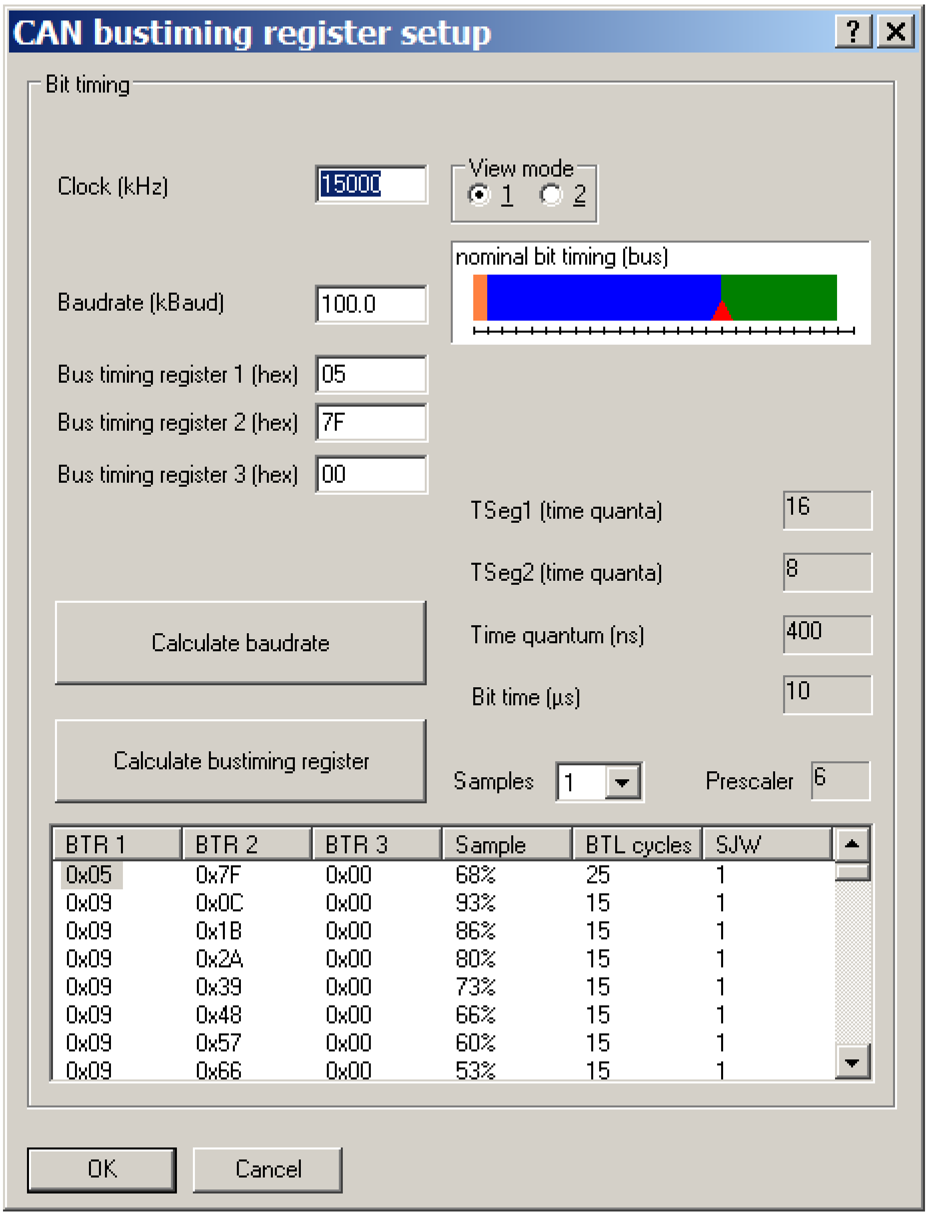


Target system	Hw_Tms470/570Cpu (Dcan)
Compiler	Texas Instruments ARM
Derivative	TMS470PSF761 TMS570PSF762 TMS470PSF764 TMS470MSF542 TMS570PSFC66 TMS570LS30316U TMS570LS12004U TMS570LS0322

11.1.2 Bus Timing

In the Bus Timing dialog it is possible to select a Clock frequency and a Baudrate for the calculation of the Bus timing register 1-3. The calculated values will be generated and used by the can-driver.

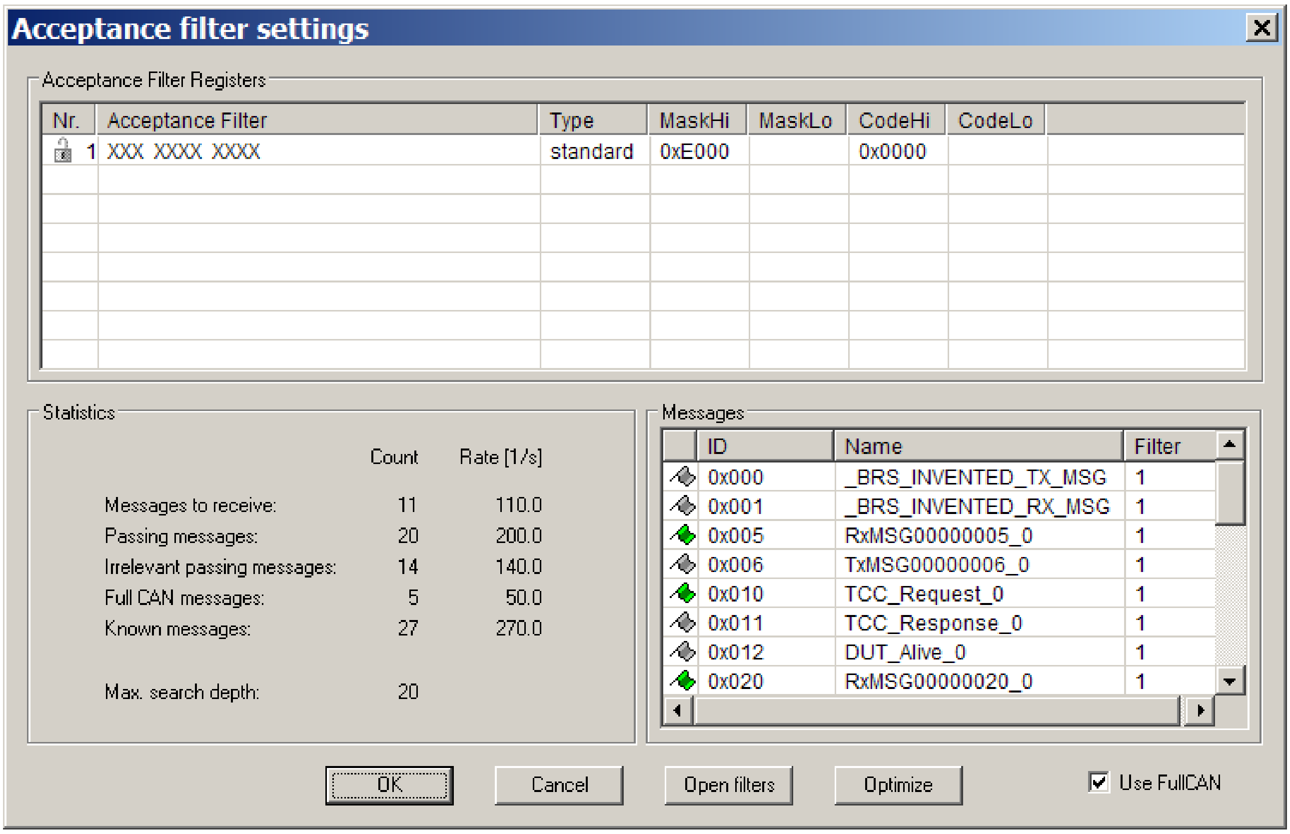
Moreover it is possible to recalculate a Baudrate out of the values of given Bus timing registers or the used Clock frequency of a configuration from values of given Bus timing registers and the Baudrate.

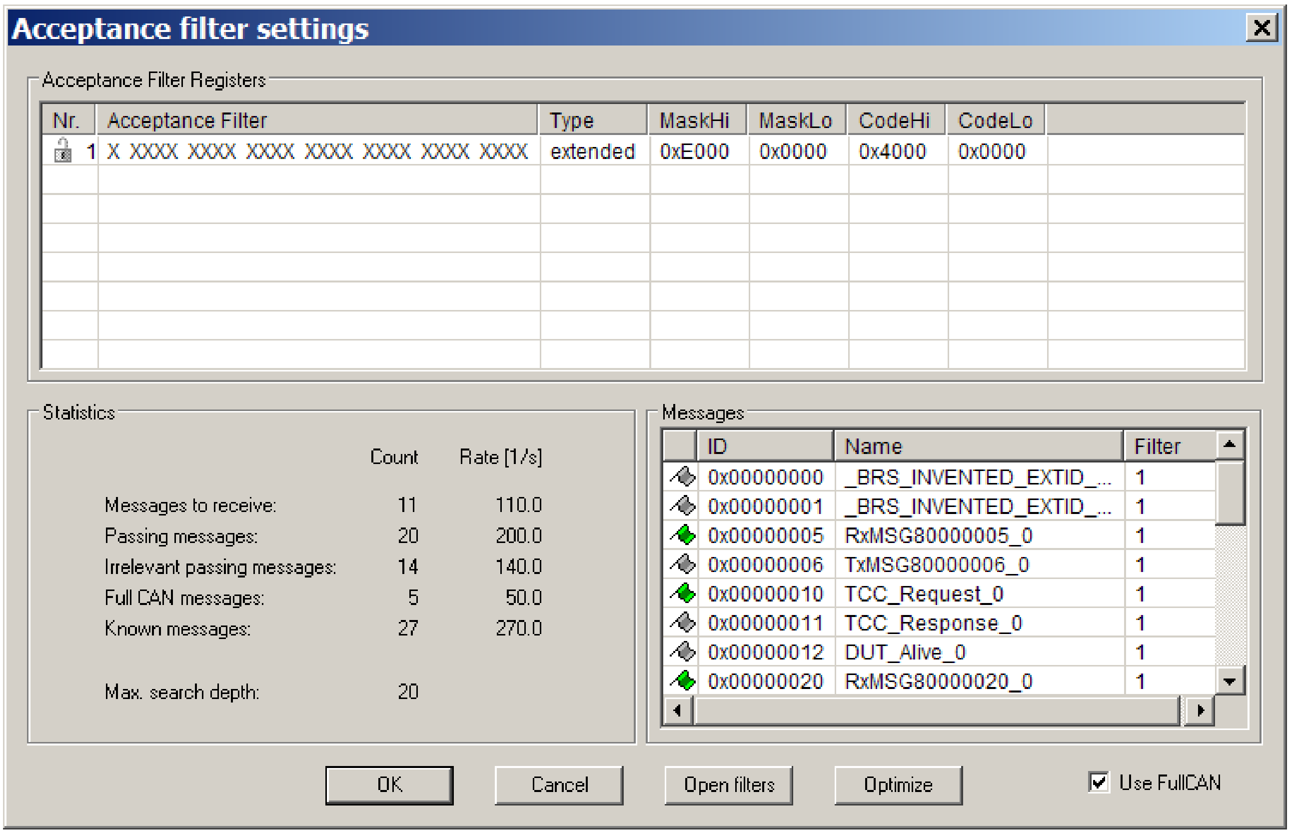


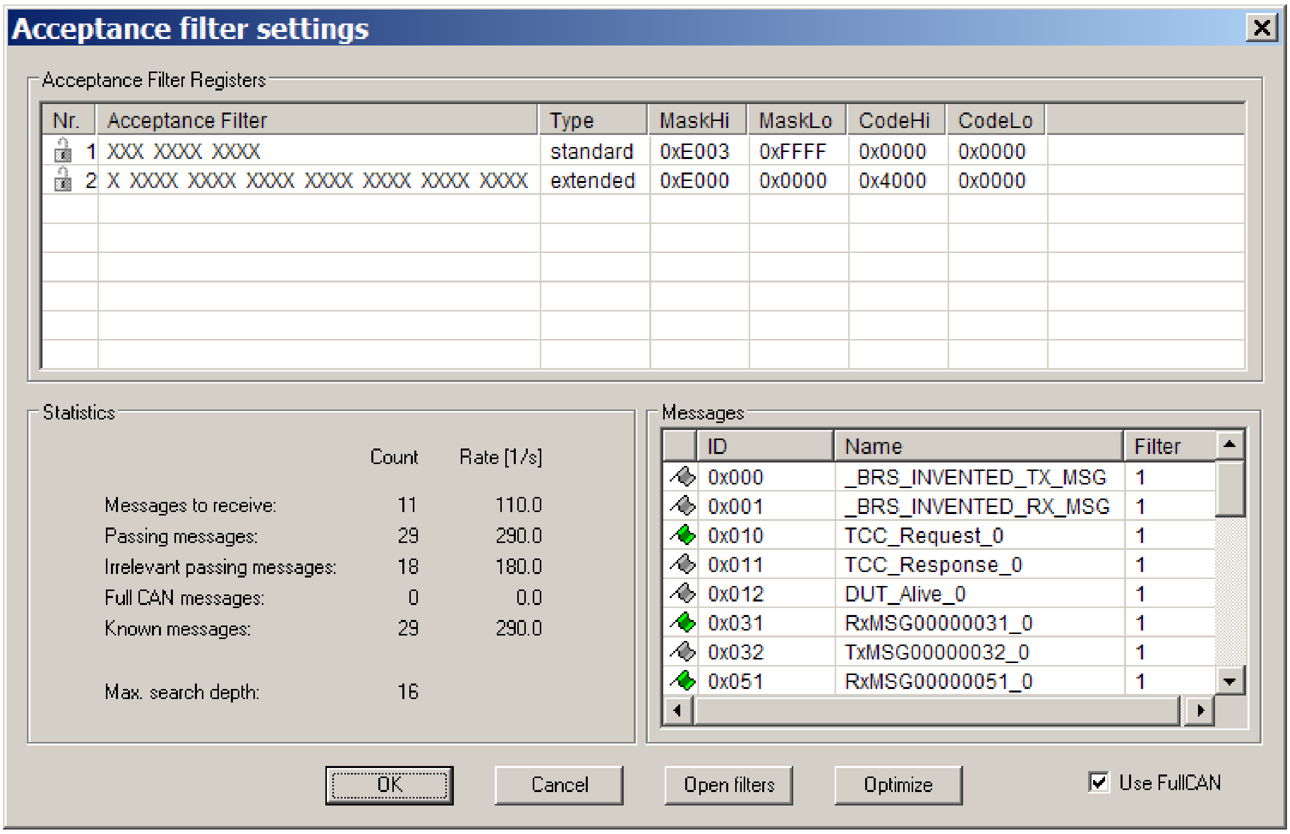
You find detailed information concerning the bus timing settings in the online help of the Generation Tool.

11.1.3 Acceptance Filtering

The dialog of the acceptance filter settings depends from the Id-types in the used database. If there are only standard Id's used, the type of the acceptance filter is standard and if there are only extended Id's used, the type of the acceptance filter is extended (see the two next pictures). Only if there are standard and extended Id's used inside the database, or if the configuration use both types of Id's (for example there is a extended Id-range configured in a standard Id database), there will be two acceptance filter used. The type of the first acceptance filter will be standard and the second will be extended.




Acceptance filter settings
Acceptance Filter Registers
Nr. | Acceptance Filter | Type | MaskHi | MaskLo | CodeHi | CodeLo
1 | X XXXX XXXX XXXX XXXX XXXX XXXX XXXX | extended | 0xE000 | 0x0000 | 0x4000 | 0x0000
Statistics
Count Rate [1/s]
Messages to receive: 11 110.0
Passing messages: 20 200.0
Irrelevant passing messages: 14 140.0
Full CAN messages: 5 50.0
Known messages: 27 270.0
Max. search depth: 20
Messages
ID | Name | Filter
0x00000000 | _BRS_INVENTED_EXTID_... | 1
0x00000001 | _BRS_INVENTED_EXTID_... | 1
0x00000005 | RxMSG80000005_0 | 1
0x00000006 | TxMSG80000006_0 | 1
0x00000010 | TCC_Request_0 | 1
0x00000011 | TCC_Response_0 | 1
0x00000012 | DUT_Alive_0 | 1
0x00000020 | RxMSG80000020_0 | 1
OK
Cancel
Open filters
Optimize
Use FullCAN


Acceptance filter settings
Acceptance Filter Registers
Nr. | Acceptance Filter | Type | MaskHi | MaskLo | CodeHi | CodeLo
1 | XXX XXXX XXXX | standard | 0xE003 | 0xFFFF | 0x0000 | 0x0000
2 | X XXXX XXXX XXXX XXXX XXXX XXXX XXXX | extended | 0xE000 | 0x0000 | 0x4000 | 0x0000
Statistics
Count Rate [1/s]
Messages to receive: 11 110.0
Passing messages: 29 290.0
Irrelevant passing messages: 18 180.0
Full CAN messages: 0 0.0
Known messages: 29 290.0
Max. search depth: 16
Messages
ID | Name | Filter
0x000 | _BRS_INVENTED_TX_MSG | 1
0x001 | _BRS_INVENTED_RX_MSG | 1
0x010 | TCC_Request_0 | 1
0x011 | TCC_Response_0 | 1
0x012 | DUT_Alive_0 | 1
0x031 | RxMSG00000031_0 | 1
0x032 | TxMSG00000032_0 | 1
0x051 | RxMSG00000051_0 | 1
OK
Cancel
Open filters
Optimize
Use FullCAN

The following mask and code values are the raw values written in the CAN cell registers, to set the „Acceptance Filter“.

For multiple Basic CANs there will be one Acceptance Filter for each Basic CAN.

Please use one standard and one extended Filter to handle mixed ID systems. (mixed Filters lead to problems for none fully opened filters because received messages may change the filter during runtime – this is a hardware specific behavior)

MaskHi	“Arbitration Mask Register High” value of the BasicCAN object. The value can be modified by changing “Acceptance Filter” or by using “Open filters” or “Optimize” button.
MaskLo	“Arbitration Mask Register Low” value of the BasicCAN objects. The value can be modified by changing “Acceptance Filter” or by using “Open filters” or “Optimize” button. This box is only available, if extended IDs are used.
CodeHi	”Arbitration Register High” value of the BasicCAN objects. The value can be modified by changing “Acceptance Filter” or by using “Open filters” or “Optimize” button.
CodeLo	“Arbitration Register Low” value of the BasicCAN objects. The value can be modified by changing “Acceptance Filter” or by using “Open filters” or “Optimize” button. This box is only available, if extended IDs are used.

To get further information please refer to the help file of the Generation Tool GENy.

12 Known Issues / Limitations

1. Please refer to the errata sheets of Texas Instruments.

2. Errata DCAN#22:

It could happen that an incorrect payload (data bytes) is stored in mailbox under certain conditions. This is a HW issue present in several revisions (A and earlier). For details please refer to silicon errata. The CAN driver implements the workaround proposal no.1 as it can be applied also to the families without local power down-mode and it does not disturb the other peripherals.

For that purpose the user has to calibrate a "6 NOPs" dummy loop, i.e. the software has to wait for at least 6 CAN clocks cycles (corresponding to the CAN clock input and not CAN bus). The 6 NOPs are not optimized and the group cannot take less than 6 CPU cycles even if the core has a pipeline.
The number of iterations through the "6 NOPs" has the following formula:

ErrataDcan22Iterations = CPU_CLOCK/CAN_CLOCK

	Info If the used version of the silicon is affected by this issue, the calculated value corresponding to ErrataDcan22Iterations has to be entered in the configuration: 1) A user config file has to be created; this will be installed in the CAN driver component of the configuration. 2) This used config file will contain the following line: #define kCanErrata22Iterations <ErrataDcan22Iterations>

Please note that the workaround is by default activated and ErrataDcan22Iterations = 255.

	Info If the used version of the silicon is not affected by this issue, the workaround can be disabled as followings: 1) A user config file has to be created; this will be installed in the CAN driver component of the configuration. 2) This used config file will contain the following line: #define C_DISABLE_DCAN_ISSUE22_WORKAROUND

13 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector-informatik.com