

XGATE Library: CAN Driver

Providing a Full CAN Mailbox System

by: Stuart Robb and Steve McAslan
MCD Applications
East Kilbride, Scotland

1 Introduction

This driver provides a full mailbox-based CAN solution for the on-chip MSCAN module. Messages are stored in message buffers in RAM. The user may choose up to 16 receive message buffers and 16 transmit message buffers. Each message buffer stores the 11-bit message identifier, up to eight bytes of data and the length of each message. Remote frames, extended identifiers and time-stamping are not supported.

Software running on the XGATE provides the main functionality of the driver and a comprehensive API is provided to allow simple access to the mailbox system from the CPU.

2 CAN Overview

A basic understanding of the CAN protocol is assumed, and a very brief overview is provided here. To help the reader understand the MSCAN specification, it is recommended that the Bosch specification be read first, to familiarize the reader with the terms and concepts contained within this document.

Contents

1	Introduction	1
2	CAN Overview	1
2.1	MSCAN Overview	3
3	Driver Overview	4
4	Driver Configuration	4
5	Application Program Interface	5
5.1	InitCAN	5
5.2	WriteCANMsg	6
5.3	SendCANMsg	6
5.4	ReadCANMsg	6
5.5	ReadCANMsgId	7
5.6	WriteCANMsgId	7
5.7	FindNewCANMsg	7
5.8	CheckCANTxStatus	7
5.9	CheckCANRxStatus	8
5.10	CAN0_RxNotificationISR	8
5.11	CAN1_RxNotificationISR	8
5.12	CAN2_RxNotificationISR	8
5.13	CAN3_RxNotificationISR	8
5.14	CAN4_RxNotificationISR	9
6	Driver Performance	9
7	Driver Implementation	10
7.1	Driver Files	10
7.2	Data Structures	11
7.3	Data Consistency	13
7.4	XGATE Threads	14
7.5	XGATE_CAN_Receive	14
7.6	XGATE_CAN_Transmit	15
8	Building an Application using CodeWarrior Tools	17
9	Building an Application using Cosmic Tools	17

The Controller Area Network (CAN) protocol is a serial communications protocol designed to support distributed real-time control applications at bit rates up to 1 Mbit/s. The CAN protocol was designed to be used primarily, but not exclusively, as a vehicle serial data bus, meeting the specific requirements of this field: real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness and required bandwidth. The low cost of CAN networks is realized by high performance microcontrollers with on-chip CAN modules, such as the M68HC908GZ60 and the MC9S12XDP512.

CAN is a carrier sense multiple access with collision detection (CSMA/CD) protocol. Information is transmitted on the CAN bus in fixed format messages by nodes. The main message formats are called: data frame, remote transmission request frame (remote frame), and error frame. A data frame is used to transmit data and a remote frame is a request for data. An error frame is transmitted automatically by the MSCAN module when an error is detected.

The data frame consists of the following fields: a start bit, an arbitration field, a control field, a data field, a cyclic redundancy check (CRC) field, an acknowledge field and an end of frame field.

A remote frame is similar to a data frame but has no data field. In order to transmit a data frame, the application must specify the arbitration field, part of the control field (Data Length Code, from 0 to 8) and the data field (number of bytes specified by Data Length Code), the other fields are generated automatically by the CAN controller.

The arbitration field contains the message identifier, which has three functions, as follows:

- It defines the priority of the message. CAN is a multi-master protocol, so in the case when more than one node is attempting to start the transmission of data or remote frames simultaneously, the bus access conflict is resolved by bit-wise arbitration using the arbitration field of the message. The message with the highest priority arbitration field wins access to the CAN bus and may continue to transmit the rest of the message. This requires that each message in a system is defined with a unique identifier.
- It is used to label the message. As each message must have a unique identifier, the identifier may be used to label the message contents. For example, the message with identifier 0x123 always contains the latest value from sensor A.
- It is used for message filtering. All nodes test the arbitration field of all received messages with a programmable hardware filter to determine whether to accept the message. Messages that are not relevant to a node can thus be filtered out. An efficient filter implementation will save processor time by eliminating the processing of unwanted messages. Careful selection of identifiers is required to achieve efficient filters on all nodes. Filtering allows any number of nodes to receive and simultaneously act upon the same message providing multicast communication.

CAN specification 2.0A defines an 11-bit identifier; CAN specification 2.0B defines identifiers with 11 bits (standard) and 29 bits (extended).

2.1 MSCAN Overview

The MSCAN modules are specific implementations of CAN controllers for the CAN 2.0B protocol. These are highly efficient CAN controller modules that are optimized for real-time performance and incorporate important features for predictable CAN network traffic.

In order to have predictable, deterministic behavior, a CAN controller must be able to:

- transmit a stream of CAN messages without that stream being interrupted by a CAN message of lower priority from an other node
- manage the internal CAN message queue so that messages queued for transmission are transmitted in order of message priority i.e. the highest priority message in the queue is transmitted first.

The MSCAN modules are unique in having a triple transmit buffer arrangement with internal prioritization, which enables the above requirements to be achieved. The local priority register for each transmit buffer ensures that the transmit buffer that contains the highest priority message is used to arbitrate for CAN bus access. In the case where more than three CAN messages are queued for transmission, the driver software must ensure that the two highest priority queued messages are loaded into the MSCAN transmit buffers at all times.

Thus the highest priority queued message in one transmit buffer is used to arbitrate for bus access. The second highest priority message is already in another transmit buffer, waiting to begin arbitrating for bus access immediately after the message in the first buffer has completed its transmission. The final transmit buffer is available to be loaded with the next highest priority message in the transmit queue.

If the application program schedules a new, higher priority message for transmission, the driver software must overwrite the transmit buffer that contains the lowest priority message with the new, higher priority message and place the lower priority message back in the transmit queue.

As a further performance enhancement, the MSCAN modules have a very flexible programmable acceptance filter that allows an efficient acceptance filter to be achieved. An efficient acceptance filter is one that accepts all the CAN messages that are wanted by a node but that rejects all, or as many as possible, of the CAN messages that are not wanted by that node. An efficient acceptance filter has a significant impact on the time spent by the CPU or XGATE processing received CAN messages. If an unwanted message is accepted by the filter, significant time may be wasted in checking the identifier of the received message with a list of wanted message identifiers.

The MSCAN acceptance filter may be configured as:

- 2 independent 32-bit filters, which test the 29 bits of an extended identifier plus the SRR, IDE and RTR bits, or the 11 bits of a standard identifier plus the RTR and IDE bits.
- 4 independent 16-bit filters, which test the 14 most significant bits of an extended identifier plus the SRR and IDE bits, or the 11 bits of a standard identifier plus the RTR and IDE bits.
- 8 independent 8-bit filters, which test the 8 most significant bits of standard and extended identifiers.

The flexibility of the MSCAN acceptance filter allows for a highly efficient filter to be implemented.

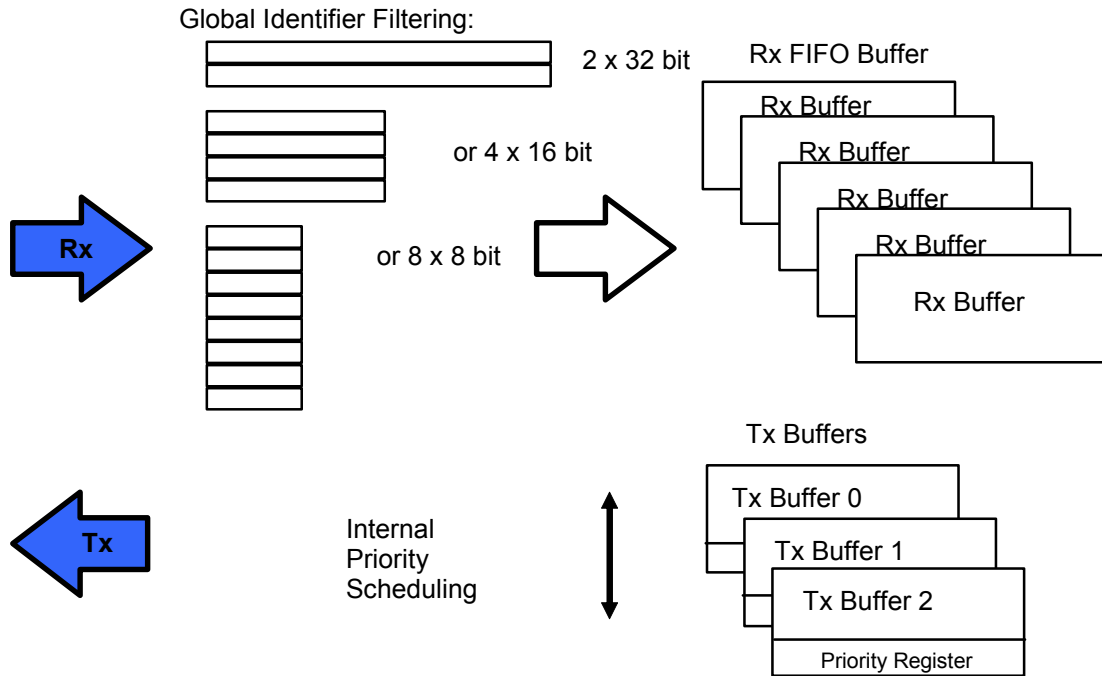


Figure 1. MSCAN Block Diagram

3 Driver Overview

The MSCAN driver software described in this Application Note is designed specifically for use with the MSCAN module in conjunction with the XGATE module on the MC9S12XDP512, and was tested on mask set 0L15Y.

The MSCAN driver software manages the transmission, reception and storage of messages on the CAN bus by the MSCAN modules. CPU routines are provided to initialize the MSCAN module, to read message buffers, to load message buffers with data and to transmit message buffers. Received, accepted messages are sorted into the appropriate message buffers and routines are provided to obtain the status of message buffers and to read the received data.

4 Driver Configuration

The `xgCAN_init.h` file contains all initialization values, such as which MSCAN modules are used, CAN bit timing, mailbox sizes, and mailbox identifier values. This is the only file that the user should need to modify.

Table 1 shows all the user defined values for MSCAN0. A similar set of values must be defined for each MSCAN module used. Note that the CANx part of each name corresponds to the physical MSCANx module on the MCU. For example, if the MSCAN2 module is to be used, then all the values containing CAN2 must be defined, regardless of whether any other MSCAN modules are used or not.

Table 1. User Defined Values for MSCAN0

CAN_RECEIVE_NOTIFICATION	Define to enable CPU interrupt for receive notification for all enabled MSCAN modules
USE_CAN0	Define to enable MSCAN0. If defined, the following initialization values for MSCAN0 must also be defined.
CANCTL0_CAN0	Define CANCTL0 register value for MSCAN0
CANCTL1_CAN0	Define CANCTL1 register value for MSCAN0
CANBTR0_CAN0	Define CANBTR0 register value for MSCAN0
CANBTR1_CAN0	Define CANBTR1 register value for MSCAN0
CANRIER_CAN0	Define CANRIER register value for MSCAN0. RXFIE must be set to enable receive interrupts.
CANTIER_CAN0	Define CANTIER register value for MSCAN0. This value must be zero.
CANIDAC_CAN0	Define CANIDAC register value for MSCAN0
CANIDAR0_CAN0	Define CANIDAR0–3 register values as a 32-bit number for MSCAN0
CANIDAR1_CAN0	Define CANIDAR4–7 register values as a 32-bit number for MSCAN0
CANIDMR0_CAN0	Define CANIDMR0–3 register values as a 32-bit number for MSCAN0
CANIDMR1_CAN0	Define CANIDMR4–7 register values as a 32-bit number for MSCAN0
RXBOXSIZE_CAN0	Define the number of receive mailboxes required for MSCAN0. Minimum 1, maximum 16
TXBOXSIZE_CAN0	Define the number of transmit mailboxes required for MSCAN0. Minimum 0, maximum 16
CAN0ID1 to CAN0IDx	Define (RXBOXSIZE_CAN0 - 1) receive identifiers, followed by TXBOXSIZE_CAN0 transmit identifiers.

5 Application Program Interface

Once the driver is configured and compiled the user may initialize the driver, and send and receive CAN messages by using the application program interface (API) provided for the CPU. The following functions may be called from any CPU routine.

5.1 InitCAN

Parameters: const XGCANstruct * channel: pointer to ChannelCANx, a structure that contains a pointer to the MSCAN module to be initialized.

 tCAN_Init_Struct * CAN_Init: pointer to CAN_Init_CANx, a structure that contains initialization constants for a particular MSCAN module.

Return: None.

Description: Initializes the MSCAN module with values contained in the structure pointed to by CAN_Init. The initialization constants are statically defined in xgCAN_ini.h. This routine must be called after reset and before any CAN messages are transmitted or received.

5.2 WriteCANMsg

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements and MSCAN module.
 tU08 box: mailbox number to be written to.
 tU08 *len: pointer to length (in bytes) of source data.
 tU08 *data: pointer to source data to be written.

Return: Error = 1 if box < RXBOXSIZE_CANx. Error = 0 if box >= RXBOXSIZE_CANx.

Description: Writes the data and length to the transmit mailbox specified by channel and box.

5.3 SendCANMsg

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements and MSCAN module.
 tU08 box: mailbox number to be queued.

Return: Error = 1 if box < RXBOXSIZE_CANx. Error = 0 if box >= RXBOXSIZE_CANx.

Description: Queues a transmit mailbox for transmission. The mailbox data is subsequently loaded into the next available MSCAN buffer by Xgate_CAN_Transmit. If more than one mailbox is queued for transmission for a particular channel, the highest numbered mailbox is loaded into the next available MSCAN buffer irrespective of the order in which the mailboxes were queued.

5.4 ReadCANMsg

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.
 tU08 box: mailbox number to be read.
 tU08 *len: number of data bytes to be read.
 tU08 *data: destination address for mailbox data bytes.

Return: None.

Description: Reads the data and length from the specified receive or transmit mailbox. If a receive mailbox is read and the corresponding RxStatus bit is set, this bit is cleared by this routine.

5.5 ReadCANMsgId

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.
 tU08 box: mailbox number to be read.
 tU16 *id: pointer to identifier.

Return: None.

Description: Reads the identifier of the specified transmit or receive mailbox.

5.6 WriteCANMsgId

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.
 tU08 box: mailbox number to be read.
 tU16 *id: pointer to identifier.

Return: Error = 1 if box < RXBOXSIZE_CANx. Error = 0 if box >= RXBOXSIZE_CANx.

Description: Writes the identifier to the specified transmit mailbox.

5.7 FindNewCANMsg

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.

Return: Number of first new message mailbox found, or NONEWCANMSG if none are found.

Description: Searches RxStatus_CANx starting at the highest order bit, corresponding to the highest receive mailbox number, to find the first set bit indicating a new, unread, message.

5.8 CheckCANTxStatus

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.
 tU08 box: mailbox number to be checked.

Return: Status of the specified mailbox.

Description: Checks the status flag of the specified mailbox. Returns status = 1 if the transmit mailbox is queued for transmission and awaiting transfer to MSCAN transmit buffer. Returns status = 0 if the mailbox has not been queued for transmission, or has been queued and transferred to an MSCAN transmit buffer. The status value does not indicate whether a mailbox that has been transferred to an MSCAN transmit buffer has actually been transmitted on the CAN bus.

5.9 CheckCANRxStatus

Parameters: const XGCANstruct *channel: pointer to ChannelCANx, a structure that contains pointers to the relevant mailbox elements.
 tU08 box: mailbox number to be checked.

Return: Status of the specified mailbox.

Description: Checks the status flag of the specified mailbox. Returns status = 1 if the receive mailbox contains a new message at has not been read by ReadCANMsg. Returns status = 0 if the mailbox has not received any CAN message at all, or if the mailbox has not received any new CAN message since the mailbox was last read by ReadCANMsg.

5.10 CAN0_RxNotificationISR

Parameters: None.

Return: None.

Description: Template CPU interrupt service routine for the user to insert their own code. CAN0_RxNotificationISR is triggered by the XGATE when a new CAN message has been received on MSCAN0 and transferred to a mailbox.

5.11 CAN1_RxNotificationISR

Parameters: None.

Return: None.

Description: Template CPU interrupt service routine for the user to insert their own code. CAN1_RxNotificationISR is triggered by the XGATE when a new CAN message has been received on MSCAN1 and transferred to a mailbox.

5.12 CAN2_RxNotificationISR

Parameters: None.

Return: None.

Description: Template CPU interrupt service routine for the user to insert their own code. CAN2_RxNotificationISR is triggered by the XGATE when a new CAN message has been received on MSCAN2 and transferred to a mailbox.

5.13 CAN3_RxNotificationISR

Parameters: None.

Return: None.

Description: Template CPU interrupt service routine for the user to insert their own code. CAN3_RxNotificationISR is triggered by the XGATE when a new CAN message has been received on MSCAN3 and transferred to a mailbox.

5.14 CAN4_RxNotificationISR

Parameters:	None.
Return:	None.
Description:	Template CPU interrupt service routine for the user to insert their own code. CAN4_RxNotificationISR is triggered by the XGATE when a new CAN message has been received on MSCAN4 and transferred to a mailbox.

6 Driver Performance

Table 2 shows the on-chip resources required for operation of this driver.

Table 2. Required Resources

Parameter	Value
Required peripheral use	At least one MSCAN module and up to all MSCAN modules on the MCU

Table 3 shows the performance of the driver when compiled using CodeWarrior for S12X v4.5. In all examples the CAN bus is 100% loaded at 500kbts/s and the S12X is operating at 40MHz.

Since the same XGATE thread services all MSCAN modules the code size will remain the same no matter how many MSCAN modules are in use.

The data size depends on the total number of message buffers. For each message buffer in use the driver will allocate 11 bytes of RAM.

Maximum execution time and load will increase by the number of MSCAN modules in use. For the purposes of measuring driver performance, the worst case loading scenario is used. For a receive buffer this occurs when a CAN message is received whose ID does not match one of the message buffers. Transmit buffers always take the same amount of time to copy to the MSCAN module. For those examples where the driver both receives and transmits the loading is the sum of the receive load and the transmit load.

The maximum latency is based on using all five MSCAN receive buffers. The maximum latency reduces according to the number of CAN busses in use (since each can be simultaneously receiving a message. Transmit latency is approximately one message length in the worst case.

Load and execution time measurements were made using the XGATE load measurement software described in AN3253.

Table 3. Required Resources

Configuration	Code Size	Data Size	Maximum Latency	Maximum Execution Time	Load
One receive buffer (MSCAN0)	130 bytes	11 bytes	~750 μ s	3.5 μ s	~2.3%
One transmit buffer (MSCAN0)	138 bytes	11 bytes	~150 μ s	1.4 μ s	~0.9%
16 receive and 4 transmit buffers (MSCAN0)	268 bytes	220 bytes	~750 μ s	4.9 μ s	~3.2%
8 receive and 8 transmit buffers each on MSCAN0 and MSCAN1	268 bytes	352 bytes	~325 μ s	2.3 μ s	~4.8%

7 Driver Implementation

This section describes the file structure of the project, the data types used, and the software routines provided for the XGATE and the CPU.

7.1 Driver Files

7.1.1 `xgCAN_ini.h`

As described previously, this file contains the configuration parameters for the driver.

7.1.2 `xgCAN_drv.h`

This file contains function prototypes and external variable declarations for `xgCAN_drv.c` and some constant definitions that generally do not need to be changed.

7.1.3 `xgCAN_drv.c`

This file contains declarations of all variables accessed by the CPU and the driver functions that can be called by the CPU, described in the Application Program Interface. This file must be compiled by the S12X CPU compiler.

7.1.4 `xgate_CAN.c`

This file contains the MSCAN transmit and receive interrupt handler routines for the Cosmic compiler. These routines are executed on the XGATE module and so this file must be compiled by the Cosmic S12X XGATE compiler. Use `xgate_CAN.cxgate` instead if you are using the CodeWarrior tool chain. The code generated by this file is copied to RAM by the initialization startup routine.

7.1.5 `xgate_CAN.cxgate`

This file contains the MSCAN transmit and receive threads for the CodeWarrior compiler. The “.cxgate” file name extension is required for the CodeWarrior XGATE compiler to recognize it. Use `xgate_CAN.c` instead if you are using the Cosmic tool chain. The code generated by this file is copied to RAM by the application startup routine.

7.1.6 `xgate_vectors.h`

This file contains external declarations for `xgate_vectors.c` and `xgate_vectors.cxgate`.

7.1.7 `xgate_vectors.c`

This file contains the XGATE vector table. This file must be compiled by the Cosmic S12X XGATE compiler. Use `xgate_vectors.cxgate` instead if you are using the CodeWarrior tool chain. The data in this file is copied to RAM by the application startup code.

7.1.8 xgate_vectors.cxgate

This file is identical to xgate_vectors.c. The “.cxgate” file name extension is required for the CodeWarrior XGATE compiler to recognize it.

7.1.9 Additional Files

Main.c	As well as containing some example code to enable a very simple application to be compiled and linked, this file contains a few functions required to initialize the PLL, XGATE and interrupt modules.
per_XDx512_L15Y.h	This file contains external declarations for per_XDx512_L15Y.c
per_XDx512_L15Y.c	This file contains declarations for all peripheral registers on the MC9S12XDP512. The constant S12XDP512 must be defined, as a compiler command line argument for example, to ensure that the correct set of peripheral registers are included. Typedefs for all peripheral register structures are contained in the header files located in the “Includes” project sub-directory.
s12x_vectors.c	This file contains the CPU vector table and templates for the MSCAN Receive notification interrupt service routines. Any unexpected interrupts will cause the MCU to enter Background Debug Mode if the BDM module is active, such as in special single chip mode. This is useful for debugging purposes.

7.2 Data Structures

In the following description, the 'CANx' in all identifiers represents the MSCAN channel being addressed, e.g. ID_Table_CAN0 is the identifier table for MSCAN0. There is one copy of each of these data variables for each used MSCAN module.

7.2.1 Receive Mailboxes

There are RXBOXSIZE_CANx receive mailboxes. In the current implementation, there may be a maximum of 16 receive mailboxes for each MSCAN channel.

Receive mailboxes are mailboxes 0 to (RXBOXSIZE_CANx - 1). Mailboxes 1 to (RXBOXSIZE_CANx - 1) receive only messages with a matching identifier.

Mailbox 0 is a special mailbox used to catch all messages that are accepted by the MSCAN filter and that do not have a matching identifier in mailboxes 1 to (RXBOXSIZE_CANx - 1).

7.2.2 Transmit Mailboxes

There are TXBOXSIZE_CANx transmit mailboxes. In the current implementation, there may be a maximum of 16 transmit mailboxes for each MSCAN channel. The transmit mailboxes are in addition to the receive mailboxes, so there may be a maximum of 32 mailboxes in total. Transmit mailboxes are mailboxes RXBOXSIZE_CANx to (RXBOXSIZE_CANx + TXBOXSIZE_CANx - 1).

7.2.3 Mailbox Components

Each mailbox is composed of one element from each of the arrays described below (mailboxes are not structures for efficiency reasons). Each array has (RXBOXSIZE_CANx + TXBOXSIZE_CANx) elements.

7.2.4 ID_Table_CANx:

Array of words containing the 11-bit identifier shifted left by 5 bits (to match the MSCAN identifier registers).

7.2.5 MsgData_CANx:

Array of arrays, each of MSGLENGTH bytes. MSGLENGTH is normally 8, corresponding to the maximum CAN message data length

7.2.6 MsgLen_CANx:

Array of bytes containing the actual message length.

Each mailbox also has one status bit. If the mailbox is a receive mailbox, its status bit is located in RxStatus_CANx. In this case, the status bit is set when a new message is received into the mailbox and cleared when the mailbox is read. If the mailbox is a transmit mailbox, its status bit is located in TxStatus_CANx. In this case, the status bit is set when the mailbox is queued for sending and cleared when it is transferred to an MSCAN hardware buffer for transmission.

7.2.7 Channels

Each MSCAN module and corresponding set of mailboxes is identified by a 'channel' identifier. Each channel identifier is a structure consisting of the following elements:

pCAN:	pointer to the MSCAN register structure
pID:	pointer to the ID_Table_CANx array
pBuffer:	pointer to the MsgData_CANx array
pLength:	pointer to the MsgLen_CANx array
pRxStatus:	pointer to the RxStatus_CANx variable
pTxStatus:	pointer to the TxStatus_CANx variable
RxBoxSize:	RXBOXSIZE_CANx

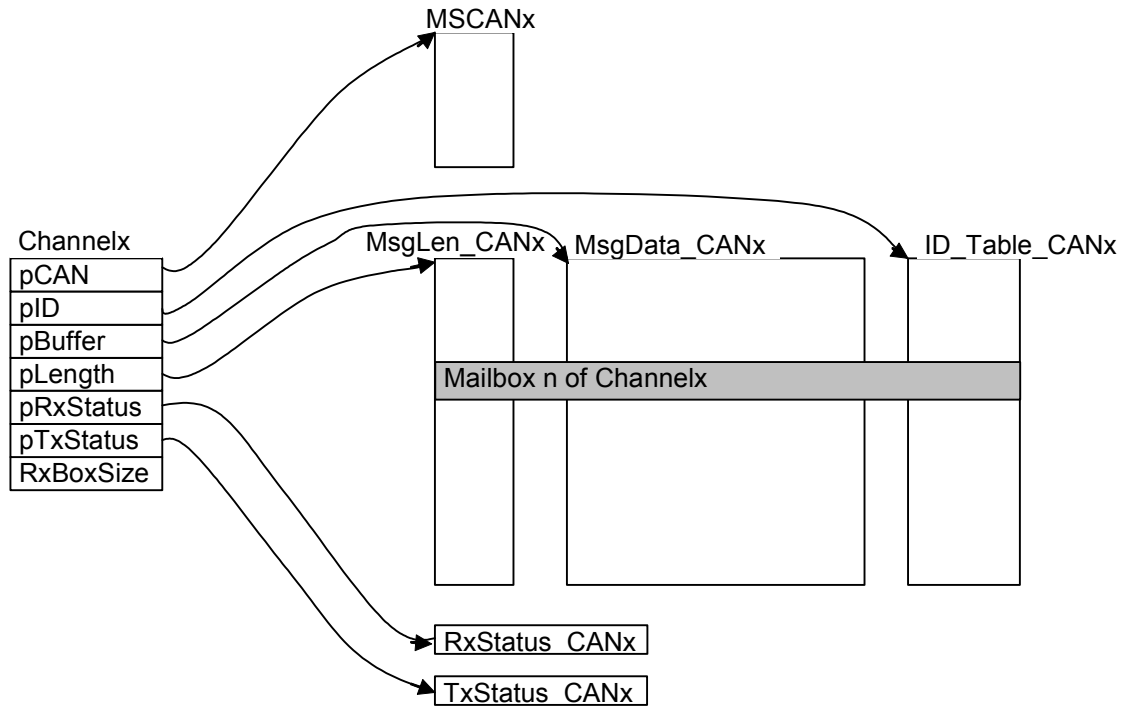


Figure 2. MSCAN Driver Data Structure

There are two versions of each channel structure, one for the CPU and one for the XGATE. This is because the variables have different addresses in the CPU memory space and the XGATE memory space, and therefore a CPU pointer to a variable has a different value to an XGATE pointer to the same variable.

7.3 Data Consistency

Whenever two processes can access the same data or resource at the same time, issues with data consistency can arise. In this application, for example, both the CPU and the XGATE access the message mailboxes. If, for example, the CPU were allowed to read a mailbox at the same time as the XGATE was writing new data into the same mailbox, the CPU could potentially read a mixture of old and new data. This situation is avoided by using semaphores. The use of semaphores is simplified by the provision of 8 semaphore flags within the XGATE module. These semaphore flags are not linked to any hardware modules and do not prevent concurrent access, but do enable very compact code to set and clear the flags and also resolve any potential issues with concurrent access to the semaphore flags themselves.

This CAN driver uses one of the 8 available semaphores. All of the CPU routines that access the mailboxes or status values lock the designated semaphore flag before proceeding to access the mailbox. Similarly, the XGATE routines also lock the same semaphore flag before accessing a mailbox. If the CPU has already locked the semaphore, the XGATE will not be able to lock it, and vice versa. If the CPU cannot lock the semaphore because the XGATE has already locked it, then the CPU routine does not attempt to access the mailbox until the XGATE has released its lock, and vice versa. The CPU and the XGATE must both release their lock on the semaphore as soon as they are finished accessing the mailbox.

7.4 XGATE Threads

Message reception and transmission events trigger service requests from the MSCAN module. These are serviced by the XGATE module and so the CPU is not interrupted at all. These threads are never called by the application and so do not form part of the Application Program Interface. Threads for the XGATE differ from ISRs for the CPU in that they can have unique data associated with them. This is achieved by having a program vector and a data vector for each XGATE thread. This means that all MSCAN receive interrupt requests can be serviced by a single XGATE routine. The necessary information about the MSCAN to be serviced and the location of the message buffers is contained in the “channel” identifier, the address of which is contained in each data vector. After processing a received CAN message, the XGATE can optionally set a channel interrupt flag, causing an interrupt to the CPU.

7.4.1 XGATE Compiler Differences

The initial versions of the CodeWarrior and Cosmic compilers treat the XGATE data vector slightly differently, which has an impact on the “C” code of XGATE routines that make use of the data vector. The CodeWarrior compiler treats the data vector as the 'argument' to the XGATE thread, and so the 'argument' has to be either a 16-bit constant or a 16-bit pointer. On the other hand, Cosmic treats the data vector like a frame pointer, and so the XGATE thread arguments are the data elements pointer to by the vector, rather than the vector itself. In practice, this means that the C code written for CodeWarrior has one extra level of de-reference compared to Cosmic. The resulting differences can be seen by comparing the file `xgate_CAN.cxgate` (CodeWarrior) with `xgate_CAN.c` (Cosmic). At the assembly level however, there is no difference as XGATE register R1 is initialized with the data vector value by hardware.

7.5 XGATE_CAN_Receive

7.5.1 Parameters (CodeWarrior)

XGCANstruct *channel: ChannelCANx, a pointer to a structure that contains pointers to the relevant mailbox elements and MSCAN module.

7.5.2 Parameters (Cosmic)

XGCANstruct channel: ChannelCANx, a structure that contains pointers to the relevant mailbox elements and MSCAN module.

7.5.3 Description

This single function handles receive interrupts from all MSCAN modules, and is executed by the XGATE module. This routine searches ID_Table_CANx elements (RXBOXSIZE_CANx -1) down to 1 for a matching identifier to the received identifier. If a match is found, the data and length are copied into the appropriate mailbox. If no match is found, the identifier, data and length are copied into mailbox 0. A status bit, corresponding to the updated mailbox number, is set in RxStatus_CANx. If CAN_RECEIVE_NOTIFICATION is defined a channel interrupt flag is set, causing the corresponding

MSCAN receive interrupt to occur and enabling the CPU to be immediately notified of newly received data.

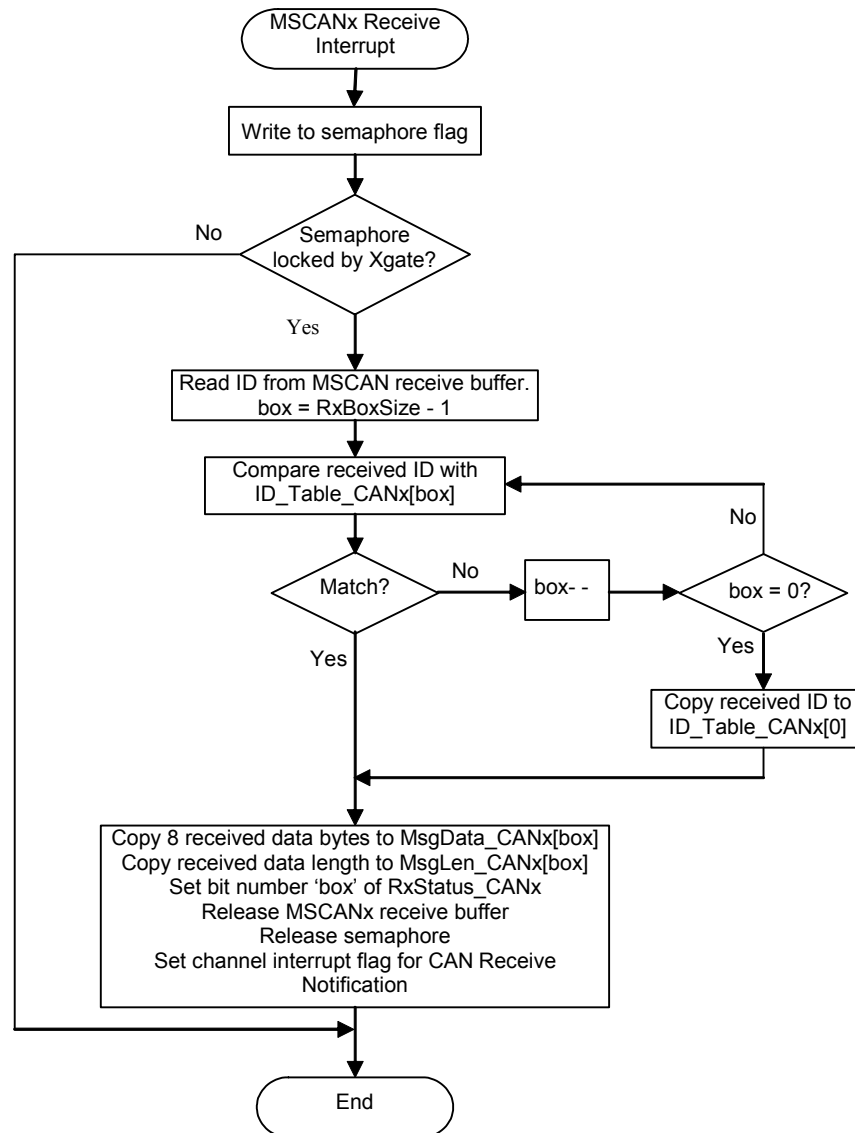


Figure 3. XGATE_CAN_Receive Flowchart

7.6 XGATE_CAN_Transmit

7.6.1 Parameters (CodeWarrior)

XGCANstruct *channel: ChannelCANx, a pointer to a structure that contains pointers to the relevant mailbox elements and MSCAN module.

7.6.2 Parameters (Cosmic)

XGCANstruct channel: ChannelCANx, a structure that contains pointers to the relevant mailbox elements and MSCAN module.

7.6.3 Description

This single function handles transmit interrupts from all MSCAN modules, and is executed by the XGATE module. This routine searches TxStatus_CANx for a queued message, starting at the highest order bit, corresponding to the highest mailbox number. If no bit is set, the routine exits. Otherwise the status bit is cleared and the message Id, data and length are copied into the available MSCAN buffer and marked for transmission.

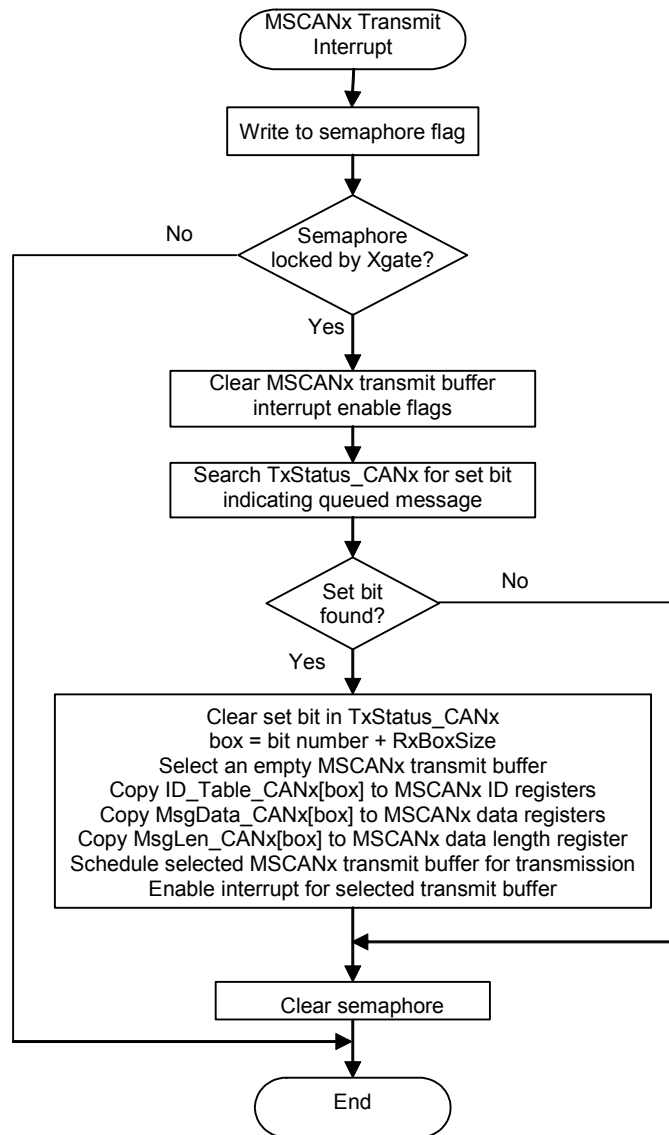


Figure 4. XGATE_CAN_Transmit Flowchart

8 Building an Application using CodeWarrior Tools

The example CodeWarrior project file, CodeWarrior.mcp, divides the source files into two groups, Core Sources and XGATE Sources. However, only files with the '.cxgate' file name extension are compiled with the XGATE compiler, regardless of which group they are in. The linker command file, linker.prm, then specifies the addresses to which the various object files are linked. Note that the CodeWarrior linker file used 'paged' addressing, where bits 23–16 of an address signify a page number.

9 Building an Application using Cosmic Tools

The example Cosmic project file, Cosmic.prj, divides the source files into two groups, Core Sources and XGATE Sources. The options for each group then define the appropriate executables to compile the sources. The linker command file, x512_a.lkf, then specifies the addresses to which the various object files are linked. The CPU and XGATE initial stack pointer values are also specified in this file. Note that the Cosmic linker file uses linear addressing.

How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:

Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:

Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:

Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:

Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2005, 2006. All rights reserved.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see <http://www.freescale.com> or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to <http://www.freescale.com/epp>.

Document Number: AN2726

Rev. 1

04/2006

