Copyright Notice and Proprietary Information

Copyright © 1998, 1999 Robert Bosch GmbH. All rights reserved. This software and manual are owned by Robert Bosch GmbH, and may be used only as authorized in the license agreement controlling such use. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Robert Bosch GmbH, or as expressly provided by the license agreement.

Disclaimer

ROBERT BOSCH GMBH, MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

ROBERT BOSCH GMBH, RESERVES THE RIGHT TO MAKE CHANGES WITHOUT FURTHER NOTICE TO THE PRODUCTS DESCRIBED HEREIN. ROBERT BOSCH GMBH DOES NOT ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR CIRCUIT DESCRIBED HEREIN.
C_CAN

1. About this Document
   1.1. Change Control
       1.1.1. Current Status
       1.1.2. Change History
   1.2. Conventions
   1.3. Scope
   1.4. References
   1.5. Terms and Abbreviations

2. Functional Description
   2.1. Functional Overview
   2.2. Block Diagram
   2.3. Operating Modes
       2.3.1. Software Initialisation
       2.3.2. CAN Message Transfer
       2.3.3. Disabled Automatic Retransmission
       2.3.4. Test Mode
       2.3.5. Silent Mode
       2.3.6. Loop Back Mode
       2.3.7. Loop Back combined with Silent Mode
       2.3.8. Basic Mode
       2.3.9. Software control of Pin CAN_TX

3. Programmer’s Model
   3.1. Hardware Reset Description
   3.2. CAN Protocol Related Registers
       3.2.1. CAN Control Register (addresses 0x01 & 0x00)
       3.2.2. Status Register (addresses 0x03 & 0x02)
           3.2.2.1. Status Interrupts
       3.2.3. Error Counter (addresses 0x05 & 0x04)
       3.2.4. Bit Timing Register (addresses 0x07 & 0x06)
       3.2.5. Test Register (addresses 0x0B & 0x0A)
       3.2.6. BRP Extension Register (addresses 0x0D & 0x0C)
   3.3. Message Interface Register Sets
       3.3.1. IFx Command Request Registers
       3.3.2. IFx Command Mask Registers
           3.3.2.1. Direction = Write
           3.3.2.2. Direction = Read
       3.3.3. IFx Message Buffer Registers
           3.3.3.1. IFx Mask Registers
           3.3.3.2. IFx Arbitration Registers
           3.3.3.3. IFx Message Control Registers
           3.3.3.4. IFx Data A and Data B Registers
       3.3.4. Message Object in the Message Memory
3.4. Message Handler Registers ........................................... 24
  3.4.1. Interrupt Register (addresses 0x09 & 0x08) ..................... 24
  3.4.2. Transmission Request Registers .................................. 25
  3.4.3. New Data Registers ................................................ 25
  3.4.4. Interrupt Pending Registers ....................................... 26
  3.4.5. Message Valid 1 Register ......................................... 26
4. CAN Application ............................................................. 27
  4.1. Management of Message Objects ..................................... 27
  4.2. Message Handler State Machine ................................... 27
    4.2.1. Data Transfer from / to Message RAM .......................... 27
    4.2.2. Transmission of Messages ...................................... 28
    4.2.3. Acceptance Filtering of Received Messages ................... 29
      4.2.3.1. Reception of Data Frame ................................... 29
      4.2.3.2. Reception of Remote Frame ................................. 29
    4.2.4. Receive / Transmit Priority .................................... 30
  4.3. Configuration of a Transmit Object ................................ 30
  4.4. Updating a Transmit Object ......................................... 30
  4.5. Configuration of a Receive Object ................................ 31
  4.6. Handling of Received Messages ..................................... 31
  4.7. Configuration of a FIFO Buffer .................................... 32
  4.8. Reception of Messages with FIFO Buffers .......................... 32
    4.8.1. Reading from a FIFO Buffer .................................... 32
  4.9. Handling of Interrupts ............................................... 34
  4.10. Configuration of the Bit Timing ................................... 34
    4.10.1. Bit Time and Bit Rate ......................................... 35
    4.10.2. Propagation Time Segment ..................................... 36
    4.10.3. Phase Buffer Segments and Synchronisation ................. 37
    4.10.4. Oscillator Tolerance Range .................................... 39
    4.10.5. Configuration of the CAN Protocol Controller .............. 40
    4.10.6. Calculation of the Bit Timing Parameters .................... 41
      4.10.6.1. Example for Bit Timing at high Baudrate ............... 42
      4.10.6.2. Example for Bit Timing at low Baudrate ............... 42
5. CPU Interface ............................................................... 43
  5.1. Customer Interface ................................................... 43
  5.2. Timing of the WAIT output signal ................................ 44
  5.3. Interrupt Timing ..................................................... 44
6. Appendix ............................................................................. 45
  6.1. List of Figures .......................................................... 45
1. About this Document

1.1 Change Control

1.1.1 Current Status
Revision 1.2

1.1.2 Change History

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>By</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft</td>
<td>17.12.98</td>
<td>C. Horst</td>
<td>First Draft</td>
</tr>
<tr>
<td>Draft</td>
<td>07.04.99</td>
<td>C. Horst</td>
<td>DAR Mode added</td>
</tr>
<tr>
<td>Draft</td>
<td>20.04.99</td>
<td>C. Horst</td>
<td>Signal names modified</td>
</tr>
<tr>
<td>Revision 1.0</td>
<td>28.09.99</td>
<td>C. Horst</td>
<td>Revised version</td>
</tr>
<tr>
<td>Revision 1.1</td>
<td>10.12.99</td>
<td>C. Horst</td>
<td>BRP Extension Register added</td>
</tr>
<tr>
<td>Revision 1.2</td>
<td>06.06.00</td>
<td>F. Hartwich</td>
<td>Document restructured</td>
</tr>
</tbody>
</table>

1.2 Conventions

The following conventions are used within this User’s Manual.

**Helvetica bold** Names of bits and signals

*Helvetica italic* States of bits and signals

1.3 Scope

This document describes the C_CAN module and its features from the application programmer’s point of view.

All information necessary to integrate the C_CAN module into an user-defined ASIC is located in the ‘Module Integration Guide’.

1.4 References

This document refers to the following documents.

<table>
<thead>
<tr>
<th>Ref</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FV/SLN1</td>
<td>CAN Specification Revision 2.0</td>
</tr>
<tr>
<td>2</td>
<td>K8/EIS1</td>
<td>Module Integration Guide</td>
</tr>
<tr>
<td>3</td>
<td>K8/EIS1</td>
<td>VHDL Reference CAN User’s Manual</td>
</tr>
<tr>
<td>4</td>
<td>ISO</td>
<td>ISO 11898-1 “Controller Area Network (CAN) - Part 1: Data link layer and physical signalling”</td>
</tr>
</tbody>
</table>
## 1.5 Terms and Abbreviations

This document uses the following terms and abbreviations.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>BSP</td>
<td>Bit Stream Processor</td>
</tr>
<tr>
<td>BTL</td>
<td>Bit Timing Logic</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check Register</td>
</tr>
<tr>
<td>DLC</td>
<td>Data Length Code</td>
</tr>
<tr>
<td>EML</td>
<td>Error Management Logic</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>TTCAN</td>
<td>Time Triggered CAN</td>
</tr>
</tbody>
</table>
2. Functional Description

2.1 Functional Overview

The C_CAN is a CAN module that can be integrated as stand-alone device or as part of an ASIC. It is described in VHDL on RTL level, prepared for synthesis. It consists of the components (see figure 1) CAN Core, Message RAM, Message Handler, Control Registers, and Module Interface.

The CAN_Core performs communication according to the CAN protocol version 2.0 part A and B. The bit rate can be programmed to values up to 1MBit/s depending on the used technology. For the connection to the physical layer additional transceiver hardware is required.

For communication on a CAN network, individual Message Objects are configured. The Message Objects and Identifier Masks for acceptance filtering of received messages are stored in the Message RAM.

All functions concerning the handling of messages are implemented in the Message Handler. Those functions are the acceptance filtering, the transfer of messages between the CAN Core and the Message RAM, and the handling of transmission requests as well as the generation of the module interrupt.

The register set of the C_CAN can be accessed directly by an external CPU via the module interface. These registers are used to control/configure the CAN Core and the Message Handler and to access the Message RAM.

The Module Interfaces delivered with the C_CAN module can easily be replaced by a customized module interface adapted to the needs of the user.

The C_CAN implements the following features:

• Supports CAN protocol version 2.0 part A and B
• Bit rates up to 1 MBit/s
• 32 Message Objects
• Each Message Object has its own identifier mask
• Programmable FIFO mode (concatenation of Message Objects)
• Maskable interrupt
• Disabled Automatic Retransmission mode for Time Triggered CAN applications
• Programmable loop-back mode for self-test operation
• 8-bit non-multiplex Motorola HC08 compatible module interface
• two 16-bit module interfaces to the AMBA APB bus from ARM
2.2 Block Diagram

The design consists of the following functional blocks (see figure 1):

**CAN Core**
CAN Protocol Controller and Rx/Tx Shift Register for serial/parallel conversion of messages.

**Message RAM**
Stores Message Objects and Identifier Masks.

**Registers**
All registers used to control and to configure the C_CAN module.

**Message Handler**
State Machine that controls the data transfer between the Rx/Tx Shift Register of the CAN Core and the Message RAM as well as the generation of interrupts as programmed in the Control and Configuration Registers.

**Module Interface**
Up to now the C_CAN module is delivered with three different interfaces. An 8-bit interface for the Motorola HC08 controller and two 16-bit interfaces to the AMBA APB bus from ARM. They can easily be replaced by a user-defined module interface.

![Block Diagram of the C_CAN](image_url)
2.3 Operating Modes

2.3.1 Software Initialisation
The software initialization is started by setting the bit Init in the CAN Control Register, either by software or by a hardware reset, or by going Bus_Off.

While Init is set, all message transfer from and to the CAN bus is stopped, the status of the CAN bus output CAN_TX is recessive (HIGH). The counters of the EML are unchanged. Setting Init does not change any configuration register.

To initialize the CAN Controller, the CPU has to set up the Bit Timing Register and each Message Object. If a Message Object is not needed, it is sufficient to set its MsgVal bit to not valid. Otherwise, the whole Message Object has to be initialized.

Access to the Bit Timing Register and to the BRP Extension Register for the configuration of the bit timing is enabled when both bits Init and CCE in the CAN Control Register are set.

Resetting Init (by CPU only) finishes the software initialisation. Afterwards the Bit Stream Processor BSP (see section 4.10 on page 34) synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11 consecutive recessive bits (Bus_Idle) before it can take part in bus activities and starts the message transfer.

The initialization of the Message Objects is independent of Init and can be done on the fly, but the Message Objects should all be configured to particular identifiers or set to not valid before the BSP starts the message transfer.

To change the configuration of a Message Object during normal operation, the CPU has to start by setting MsgVal to not valid. When the configuration is completed, MsgVal is set to valid again.

2.3.2 CAN Message Transfer
Once the C_CAN is initialized and Init is reset to zero, the C_CAN's CAN Core synchronizes itself to the CAN bus and starts the message transfer.

Received messages are stored into their appropriate Message Objects if they pass the Message Handler's acceptance filtering. The whole message including all arbitration bits, DLC and eight data bytes is stored into the Message Object. If the Identifier Mask is used, the arbitration bits which are masked to “don’t care” may be overwritten in the Message Object.

The CPU may read or write each message any time via the Interface Registers, the Message Handler guarantees data consistency in case of concurrent accesses.

Messages to be transmitted are updated by the CPU. If a permanent Message Object (arbitration and control bits set up during configuration) exists for the message, only the data bytes are updated and then TxRqst bit with NewDat bit are set to start the transmission. If several transmit messages are assigned to the same Message Object (when the number of Message Objects is not sufficient), the whole Message Object has to be configured before the transmission of this message is requested.

The transmission of any number of Message Objects may be requested at the same time, they are transmitted subsequently according to their internal priority. Messages may be updated or set to not valid any time, even when their requested transmission is still pending. The old data will be discarded when a message is updated before its pending transmission has started.

Depending on the configuration of the Message Object, the transmission of a message may be requested autonomously by the reception of a remote frame with a matching identifier.
2.3.3 Disabled Automatic Retransmission

According to the CAN Specification (see ISO11898, 6.3.3 Recovery Management), the C_CAN provides means for automatic retransmission of frames that have lost arbitration or that have been disturbed by errors during transmission. The frame transmission service will not be confirmed to the user before the transmission is successfully completed. By default, this means for automatic retransmission is enabled. It can be disabled to enable the C_CAN to work within a Time Triggered CAN (TTCAN, see ISO11898-1) environment.

The Disabled Automatic Retransmission mode is enabled by programming bit DAR in the CAN Control Register to one. In this operation mode the programmer has to consider the different behaviour of bits TxRqst and NewDat in the Control Registers of the Message Buffers:

- When a transmission starts bit TxRqst of the respective Message Buffer is reset, while bit NewDat remains set.
- When the transmission completed successfully bit NewDat is reset.

When a transmission failed (lost arbitration or error) bit NewDat remains set. To restart the transmission the CPU has to set TxRqst back to one.

2.3.4 Test Mode

The Test Mode is entered by setting bit Test in the CAN Control Register to one. In Test Mode the bits Tx1, Tx0, LBack, Silent and Basic in the Test Register are writable. Bit Rx monitors the state of pin CAN_RX and therefore is only readable. All Test Register functions are disabled when bit Test is reset to zero.

2.3.5 Silent Mode

The CAN Core can be set in Silent Mode by programming the Test Register bit Silent to one.

In Silent Mode, the C_CAN is able to receive valid data frames and valid remote frames, but it sends only recessive bits on the CAN bus and it cannot start a transmission. If the CAN Core is required to send a dominant bit (ACK bit, overload flag, active error flag), the bit is rerouted internally so that the CAN Core monitors this dominant bit, although the CAN bus may remain in recessive state. The Silent Mode can be used to analyse the traffic on a CAN bus without affecting it by the transmission of dominant bits (Acknowledge Bits, Error Frames). Figure 2 shows the connection of signals CAN_TX and CAN_RX to the CAN Core in Silent Mode.

![Figure 2: CAN Core in Silent Mode](image-url)
2.3.6 Loop Back Mode

The CAN Core can be set in Loop Back Mode by programming the Test Register bit LBack to one. In Loop Back Mode, the CAN Core treats its own transmitted messages as received messages and stores them (if they pass acceptance filtering) into a Receive Buffer. Figure 3 shows the connection of signals CAN_TX and CAN_RX to the CAN Core in Loop Back Mode.

This mode is provided for self-test functions. To be independent from external stimulation, the CAN Core ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data/remote frame) in Loop Back Mode. In this mode the CAN Core performs an internal feedback from its Tx output to its Rx input. The actual value of the CAN_RX input pin is disregarded by the CAN Core. The transmitted messages can be monitored at the CAN_TX pin.

2.3.7 Loop Back combined with Silent Mode

It is also possible to combine Loop Back Mode and Silent Mode by programming bits LBack and Silent to one at the same time. This mode can be used for a “Hot Selftest”, meaning the C_CAN can be tested without affecting a running CAN system connected to the pins CAN_TX and CAN_RX. In this mode the CAN_RX pin is disconnected from the CAN Core and the CAN_TX pin is held recessive. Figure 4 shows the connection of signals CAN_TX and CAN_RX to the CAN Core in case of the combination of Loop Back Mode with Silent Mode.
2.3.8 Basic Mode

The CAN Core can be set in Basic Mode by programming the Test Register bit Basic to one. In this mode the C_CAN module runs without the Message RAM.

The IF1 Registers are used as Transmit Buffer. The transmission of the contents of the IF1 Registers is requested by writing the Busy bit of the IF1 Command Request Register to ‘1’. The IF1 Registers are locked while the Busy bit is set. The Busy bit indicates that the transmission is pending.

As soon the CAN bus is idle, the IF1 Registers are loaded into the shift register of the CAN Core and the transmission is started. When the transmission has completed, the Busy bit is reset and the locked IF1 Registers are released.

A pending transmission can be aborted at any time by resetting the Busy bit in the IF1 Command Request Register while the IF1 Registers are locked. If the CPU has reset the Busy bit, a possible retransmission in case of lost arbitration or in case of an error is disabled.

The IF2 Registers are used as Receive Buffer. After the reception of a message the contents of the shift register is stored into the IF2 Registers, without any acceptance filtering.

Additionally, the actual contents of the shift register can be monitored during the message transfer. Each time a read Message Object is initiated by writing the Busy bit of the IF2 Command Request Register to ‘1’, the contents of the shift register is stored into the IF2 Registers.

In Basic Mode the evaluation of all Message Object related control and status bits and of the control bits of the IFx Command Mask Registers is turned off. The message number of the Command request registers is not evaluated. The NewDat and MsgLst bits of the IF2 Message Control Register retain their function, DLC3-0 will show the received DLC, the other control bits will be read as ‘0’.

In Basic Mode the ready output CAN_WAIT_B is disabled (always ‘1’).

2.3.9 Software control of Pin CAN_TX

Four output functions are available for the CAN transmit pin CAN_TX. Additionally to its default function – the serial data output – it can drive the CAN Sample Point signal to monitor CAN_Core’s bit timing and it can drive constant dominant or recessive values. The last two functions, combined with the readable CAN receive pin CAN_RX, can be used to check the CAN bus’ physical layer.

The output mode of pin CAN_TX is selected by programming the Test Register bits Tx1 and Tx0 as described in section 3.2.5 on page 17.

The three test functions for pin CAN_TX interfere with all CAN protocol functions. CAN_TX must be left in its default function when CAN message transfer or any of the test modes Loop Back Mode, Silent Mode, or Basic Mode are selected.
3. Programmer’s Model

The C_CAN module allocates an address space of 256 bytes. The registers are organized as 16-bit registers, with the high byte at the odd address and the low byte at the even address.

The two sets of interface registers (IF1 and IF2) control the CPU access to the Message RAM. They buffer the data to be transferred to and from the RAM, avoiding conflicts between CPU accesses and message reception/transmission.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Reset Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN Base + 0x00</td>
<td>CAN Control Register</td>
<td>0x0001</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x02</td>
<td>Status Register</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x04</td>
<td>Error Counter</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x06</td>
<td>Bit Timing Register</td>
<td>0x2301</td>
<td>write enabled by CCE</td>
</tr>
<tr>
<td>CAN Base + 0x08</td>
<td>Interrupt Register</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x0A</td>
<td>Test Register</td>
<td>0x0000</td>
<td>write enabled by Test</td>
</tr>
<tr>
<td>CAN Base + 0x0C</td>
<td>BRP Extension Register</td>
<td>0x0000</td>
<td>write enabled by CCE</td>
</tr>
<tr>
<td>CAN Base + 0x0E</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0x10</td>
<td>IF1 Command Request</td>
<td>0x0001</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x12</td>
<td>IF1 Command Mask</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x14</td>
<td>IF1 Mask 1</td>
<td>0xFFFF</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x16</td>
<td>IF1 Mask 2</td>
<td>0xFFFF</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x18</td>
<td>IF1 Arbitration 1</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x1A</td>
<td>IF1 Arbitration 2</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x1C</td>
<td>IF1 Message Control</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x1E</td>
<td>IF1 Data A 1</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x20</td>
<td>IF1 Data A 2</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x22</td>
<td>IF1 Data B 1</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x24</td>
<td>IF1 Data B 2</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>CAN Base + 0x28 - 0x3E</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0x40 - 0x54</td>
<td>IF2 Registers</td>
<td>see note 2)</td>
<td>same as IF1 Registers</td>
</tr>
<tr>
<td>CAN Base + 0x56 - 0x7E</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0x80</td>
<td>Transmission Request 1</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x82</td>
<td>Transmission Request 2</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x84 - 0x8E</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0x90</td>
<td>New Data 1</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x92</td>
<td>New Data 2</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0x94 - 0x9E</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0xA0</td>
<td>Interrupt Pending 1</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0xA2</td>
<td>Interrupt Pending 2</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0xA4 - 0xAE</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CAN Base + 0xB0</td>
<td>Message Valid 1</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0xB2</td>
<td>Message Valid 2</td>
<td>0x0000</td>
<td>read only</td>
</tr>
<tr>
<td>CAN Base + 0xB4 - 0xBE</td>
<td>— reserved</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

1) r signifies the actual value of the CAN_RX pin.
2) The two sets of Message Interface Registers - IF1 and IF2 - have identical functions.
3) Reserved bits are read as ‘0’ except for IFx Mask 2 Register where they are read as ‘1’

Figure 5: C_CAN Register Summary
3.1 Hardware Reset Description

After hardware reset, the registers of the C_CAN hold the values described in figure 5.

Additionally the busoff state is reset and the output CAN_TX is set to recessive (HIGH). The value 0x0001 (Init = ‘1’) in the CAN Control Register enables the software initialisation. The C_CAN does not influence the CAN bus until the CPU resets Init to ‘0’.

The data stored in the Message RAM is not affected by a hardware reset. After power-on, the contents of the Message RAM is undefined.

3.2 CAN Protocol Related Registers

These registers are related to the CAN protocol controller in the CAN Core. They control the operating modes and the configuration of the CAN bit timing and provide status information.

3.2.1 CAN Control Register (addresses 0x01 & 0x00)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>Test Mode Enable</td>
</tr>
<tr>
<td></td>
<td>one - Test Mode.</td>
</tr>
<tr>
<td></td>
<td>zero - Normal Operation.</td>
</tr>
<tr>
<td>CCE</td>
<td>Configuration Change Enable</td>
</tr>
<tr>
<td></td>
<td>one - The CPU has write access to the Bit Timing Register (while Init = one).</td>
</tr>
<tr>
<td></td>
<td>zero - The CPU has no write access to the Bit Timing Register.</td>
</tr>
<tr>
<td>DAR</td>
<td>Disable Automatic Retransmission</td>
</tr>
<tr>
<td></td>
<td>one - Automatic Retransmission disabled.</td>
</tr>
<tr>
<td></td>
<td>zero - Automatic Retransmission of disturbed messages enabled.</td>
</tr>
<tr>
<td>EIE</td>
<td>Error Interrupt Enable</td>
</tr>
<tr>
<td></td>
<td>one - Enabled - A change in the bits BOff or EWarn in the Status Register will generate an interrupt.</td>
</tr>
<tr>
<td></td>
<td>zero - Disabled - No Error Status Interrupt will be generated.</td>
</tr>
<tr>
<td>SIE</td>
<td>Status Change Interrupt Enable</td>
</tr>
<tr>
<td></td>
<td>one - Enabled - An interrupt will be generated when a message transfer is successfully completed or a CAN bus error is detected.</td>
</tr>
<tr>
<td></td>
<td>zero - Disabled - No Status Change Interrupt will be generated.</td>
</tr>
<tr>
<td>IE</td>
<td>Module Interrupt Enable</td>
</tr>
<tr>
<td></td>
<td>one - Enabled - Interrupts will set IRQ_B to LOW. IRQ_B remains LOW until all pending interrupts are processed.</td>
</tr>
<tr>
<td></td>
<td>zero - Disabled - Module Interrupt IRQ_B is always HIGH.</td>
</tr>
<tr>
<td>Init</td>
<td>Initialization</td>
</tr>
<tr>
<td></td>
<td>one - Initialization is started.</td>
</tr>
<tr>
<td></td>
<td>zero - Normal Operation.</td>
</tr>
</tbody>
</table>

Note: The busoff recovery sequence (see CAN Specification Rev. 2.0) cannot be shortened by setting or resetting Init. If the device goes busoff, it will set Init of its own accord, stopping all bus activities. Once Init has been cleared by the CPU, the device will then wait for 129
occurrences of Bus Idle (129 * 11 consecutive recessive bits) before resuming normal operations. At the end of the busoff recovery sequence, the Error Management Counters will be reset.

During the waiting time after the resetting of Init, each time a sequence of 11 recessive bits has been monitored, a Bit0Error code is written to the Status Register, enabling the CPU to readily check up whether the CAN bus is stuck at dominant or continuously disturbed and to monitor the proceeding of the busoff recovery sequence.

### 3.2.2 Status Register (addresses 0x03 & 0x02)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOff</td>
<td>Busoff Status</td>
</tr>
<tr>
<td></td>
<td>one The CAN module is in busoff state.</td>
</tr>
<tr>
<td></td>
<td>zero The CAN module is not busoff.</td>
</tr>
<tr>
<td>EWarn</td>
<td>Warning Status</td>
</tr>
<tr>
<td></td>
<td>one At least one of the error counters in the EML has reached the error warning limit of 96.</td>
</tr>
<tr>
<td></td>
<td>zero Both error counters are below the error warning limit of 96.</td>
</tr>
<tr>
<td>EPass</td>
<td>Error Passive</td>
</tr>
<tr>
<td></td>
<td>one The CAN Core is in the error passive state as defined in the CAN Specification.</td>
</tr>
<tr>
<td></td>
<td>zero The CAN Core is error active.</td>
</tr>
<tr>
<td>RxOk</td>
<td>Received a Message Successfully</td>
</tr>
<tr>
<td></td>
<td>one Since this bit was last reset (to zero) by the CPU, a message has been successfully received (independent of the result of acceptance filtering).</td>
</tr>
<tr>
<td></td>
<td>zero Since this bit was last reset by the CPU, no message has been successfully received. This bit is never reset by the CAN Core.</td>
</tr>
<tr>
<td>TxOk</td>
<td>Transmitted a Message Successfully</td>
</tr>
<tr>
<td></td>
<td>one Since this bit was last reset by the CPU, a message has been successfully (error free and acknowledged by at least one other node) transmitted.</td>
</tr>
<tr>
<td></td>
<td>zero Since this bit was reset by the CPU, no message has been successfully transmitted. This bit is never reset by the CAN Core.</td>
</tr>
<tr>
<td>LEC</td>
<td>Last Error Code (Type of the last error to occur on the CAN bus)</td>
</tr>
<tr>
<td>0</td>
<td>No Error</td>
</tr>
<tr>
<td>1</td>
<td>Stuff Error : More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed.</td>
</tr>
<tr>
<td>2</td>
<td>Form Error : A fixed format part of a received frame has the wrong format.</td>
</tr>
<tr>
<td>3</td>
<td>AckError : The message this CAN Core transmitted was not acknowledged by another node.</td>
</tr>
<tr>
<td>4</td>
<td>Bit1Error : During the transmission of a message (with the exception of the arbitration field), the device wanted to send a recessive level (bit of logical value ‘1’), but the monitored bus value was dominant.</td>
</tr>
</tbody>
</table>
5 **Bit0Error**: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), the device wanted to send a *dominant* level (data or identifier bit logical value ‘0’), but the monitored Bus value was *recessive*. During *busoff* recovery this status is set each time a sequence of 11 *recessive* bits has been monitored. This enables the CPU to monitor the proceeding of the busoff recovery sequence (indicating the bus is not stuck at *dominant* or continuously disturbed).

6 **CRCError**: The CRC check sum was incorrect in the message received, the CRC received for an incoming message does not match with the calculated CRC for the received data.

7 **unused**: When the LEC shows the value ‘7’, no CAN bus event was detected since the CPU wrote this value to the LEC.

The LEC field holds a code which indicates the type of the last error to occur on the CAN bus. This field will be cleared to ‘0’ when a message has been transferred (reception or transmission) without error. The unused code ‘7’ may be written by the CPU to check for updates.

3.2.2.1 **Status Interrupts**

A Status Interrupt is generated by bits BOff and EWarn (Error Interrupt) or by RxOk, TxOk, and LEC (Status Change Interrupt) assumed that the corresponding enable bits in the CAN Control Register are set. A change of bit EPass or a write to RxOk, TxOk, or LEC will never generate a Status Interrupt.

Reading the Status Register will clear the Status Interrupt value (8000h) in the Interrupt Register, if it is pending.

3.2.3 **Error Counter** (addresses 0x05 & 0x04)

<table>
<thead>
<tr>
<th>RP</th>
<th>REC6-0</th>
<th>TEC7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>

**RP**
- *Receive Error Passive*
  - **one**: The Receive Error Counter has reached the *error passive* level as defined in the CAN Specification.
  - **zero**: The Receive Error Counter is below the *error passive* level.

**REC6-0**
- *Receive Error Counter*
  - Actual state of the Receive Error Counter. Values between 0 and 127.

**TEC7-0**
- *Transmit Error Counter*
  - Actual state of the Transmit Error Counter. Values between 0 and 255.

3.2.4 **Bit Timing Register** (addresses 0x07 & 0x06)

<table>
<thead>
<tr>
<th>res</th>
<th>TSeg2</th>
<th>TSeg1</th>
<th>SJW</th>
<th>BRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

**TSeg1**
- The time segment before the sample point
  - valid values for **TSeg1** are [1 … 15]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.
The time segment after the sample point

valid values for **TSeg2** are [0 … 7]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.

**SJW**  
(Re)Synchronisation Jump Width

Valid programmed values are 0-3. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.

**BRP**  
Baud Rate Prescaler

The value by which the oscillator frequency is divided for generating the bit time quanta. The bit time is built up from a multiple of this quanta. Valid values for the Baud Rate Prescaler are [0 … 63]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.

**Note:** With a module clock **CAN_CLK** of 8 MHz, the reset value of 0x2301 configures the C_CAN for a bit rate of 500 kBit/s. The registers are only writable if bits **CCE** and **Init** in the CAN Control Register are set.

### 3.2.5 Test Register (addresses 0x0B & 0x0A)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>res</td>
<td>res</td>
<td>res</td>
<td>res</td>
<td>res</td>
<td>res</td>
<td>res</td>
<td>Rx</td>
<td>Tx1</td>
<td>Tx0</td>
<td>LBack</td>
<td>Silent</td>
<td>Basic</td>
<td>res</td>
<td>res</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>

**Rx**  
Monitors the actual value of the **CAN_RX** Pin

- **one** The CAN bus is recessive (**CAN_RX** = ‘1’).
- **zero** The CAN bus is dominant (**CAN_RX** = ‘0’).

**Tx1-0**  
Control of **CAN_TX** pin

- **00** Reset value, **CAN_TX** is controlled by the CAN Core.
- **01** Sample Point can be monitored at **CAN_TX** pin.
- **10** **CAN_TX** pin drives a dominant (‘0’) value.
- **11** **CAN_TX** pin drives a recessive (‘1’) value.

**LBack**  
Loop Back Mode

- **one** Loop Back Mode is enabled.
- **zero** Loop Back Mode is disabled.

**Silent**  
Silent Mode

- **one** The module is in Silent Mode
- **zero** Normal operation.

**Basic**  
Basic Mode

- **one** IF1 Registers used as Tx Buffer, IF2 Registers used as Rx Buffer.
- **zero** Basic Mode disabled.

Write access to the Test Register is enabled by setting bit **Test** in the CAN Control Register. The different test functions may be combined, but **Tx1-0 ≠ “00”** disturbs message transfer.
3.2.6 BRP Extension Register (addresses 0x0D & 0x0C)

<table>
<thead>
<tr>
<th>Address</th>
<th>IF1 Register Set</th>
<th>Address</th>
<th>IF1 Register Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN Base + 0x10</td>
<td>IF1 Command Request</td>
<td>CAN Base + 0x10</td>
<td>IF1 Command Request</td>
</tr>
<tr>
<td>CAN Base + 0x12</td>
<td>IF1 Command Mask</td>
<td>CAN Base + 0x12</td>
<td>IF1 Command Mask</td>
</tr>
<tr>
<td>CAN Base + 0x14</td>
<td>IF1 Mask 1</td>
<td>CAN Base + 0x14</td>
<td>IF1 Mask 1</td>
</tr>
<tr>
<td>CAN Base + 0x16</td>
<td>IF1 Mask 2</td>
<td>CAN Base + 0x16</td>
<td>IF1 Mask 2</td>
</tr>
<tr>
<td>CAN Base + 0x18</td>
<td>IF1 Arbitration 1</td>
<td>CAN Base + 0x18</td>
<td>IF1 Arbitration 1</td>
</tr>
<tr>
<td>CAN Base + 0x1A</td>
<td>IF1 Arbitration 2</td>
<td>CAN Base + 0x1A</td>
<td>IF1 Arbitration 2</td>
</tr>
<tr>
<td>CAN Base + 0x1C</td>
<td>IF1 Message Control</td>
<td>CAN Base + 0x1C</td>
<td>IF1 Message Control</td>
</tr>
<tr>
<td>CAN Base + 0x1E</td>
<td>IF1 Data A 1</td>
<td>CAN Base + 0x1E</td>
<td>IF1 Data A 1</td>
</tr>
<tr>
<td>CAN Base + 0x20</td>
<td>IF1 Data A 2</td>
<td>CAN Base + 0x20</td>
<td>IF1 Data A 2</td>
</tr>
<tr>
<td>CAN Base + 0x22</td>
<td>IF1 Data B 1</td>
<td>CAN Base + 0x22</td>
<td>IF1 Data B 1</td>
</tr>
<tr>
<td>CAN Base + 0x24</td>
<td>IF1 Data B 2</td>
<td>CAN Base + 0x24</td>
<td>IF1 Data B 2</td>
</tr>
</tbody>
</table>

BRPE  Baud Rate Prescaler Extension

0x00-0x0F  By programming BRPE the Baud Rate Prescaler can be extended to values up to 1023. The actual interpretation by the hardware is that one more than the value programmed by BRPE (MSBs) and BRP (LSBs) is used.

3.3 Message Interface Register Sets

There are two sets of Interface Registers which are used to control the CPU access to the Message RAM. The Interface Registers avoid conflicts between CPU access to the Message RAM and CAN message reception and transmission by buffering the data to be transferred. A complete Message Object (see chapter 3.3.4) or parts of the Message Object may be transferred between the Message RAM and the IFx Message Buffer registers (see chapter 3.3.3) in one single transfer.

The function of the two interface register sets is identical (except for test mode Basic). They can be used the way that one set of registers is used for data transfer to the Message RAM while the other set of registers is used for the data transfer from the Message RAM, allowing both processes to be interrupted by each other. Figure 6 gives an overview of the two Interface Register sets.

Each set of Interface Registers consists of Message Buffer Registers controlled by their own Command Registers. The Command Mask Register specifies the direction of the data transfer and which parts of a Message Object will be transferred. The Command Request Register is used to select a Message Object in the Message RAM as target or source for the transfer and to start the action specified in the Command Mask Register.

Figure 6: IF1 and IF2 Message Interface Register Sets
3.3.1 IFx Command Request Registers

A message transfer is started as soon as the CPU has written the message number to the Command Request Register. With this write operation the Busy bit is automatically set to ‘1’ and signal CAN_WAIT_B is pulled LOW to notify the CPU that a transfer is in progress. After a wait time of 3 to 6 CAN_CLK periods, the transfer between the Interface Register and the Message RAM has completed. The Busy bit is set back to zero and CAN_WAIT_B is set back to HIGH (see figure 5.2 on page 44).

<table>
<thead>
<tr>
<th>IF1 Command Request Register (addresses 0x11 &amp; 0x10)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Message Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy Flag</td>
<td>res res res res res res res res</td>
<td>Message Number</td>
</tr>
</tbody>
</table>

Busy Flag

- **one**: set to one when writing to the IFx Command Request Register
- **zero**: reset to zero when read/write action has finished.

Message Number

- **0x01-0x20**: Valid Message Number, the Message Object in the Message RAM is selected for data transfer.
- **0x00**: Not a valid Message Number, interpreted as 0x20.
- **0x21-0x3F**: Not a valid Message Number, interpreted as 0x01-0x1F.

**Note:** When a Message Number that is not valid is written into the Command Request Register, the Message Number will be transformed into a valid value and that Message Object will be transferred.

3.3.2 IFx Command Mask Registers

The control bits of the IFx Command Mask Register specify the transfer direction and select which of the IFx Message Buffer Registers are source or target of the data transfer.

<table>
<thead>
<tr>
<th>IF2 Command Mask Register (addresses 0x13 &amp; 0x12)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Mask Arb Control ClrIntPnd TxRqst/ NewDat Data A Data B</th>
</tr>
</thead>
<tbody>
<tr>
<td>res WR/RD</td>
<td>Mask Arb Control ClrIntPnd TxRqst/ NewDat Data A Data B</td>
<td></td>
</tr>
</tbody>
</table>

WR/RD

- **one**: Write: Transfer data from the selected Message Buffer Registers to the Message Object addressed by the Command Request Register.
- **zero**: Read: Transfer data from the Message Object addressed by the Command Request Register into the selected Message Buffer Registers.

The other bits of IFx Command Mask Register have different functions depending on the transfer direction:

3.3.2.1 Direction = Write

Mask

- **one**: transfer Identifier Mask + MDir + MXtd to Message Object.
- **zero**: Mask bits unchanged.
Arb  Access Arbitration Bits
one  transfer Identifier + Dir + Xtd + MsgVal to Message Object.
zero  Arbitration bits unchanged.

Control  Access Control Bits
one  transfer Control Bits to Message Object.
zero  Control Bits unchanged.

ClrIntPnd  Clear Interrupt Pending Bit
Note:  When writing to a Message Object, this bit is ignored.

TxRqst/NewDat  Access Transmission Request Bit
one  set TxRqst bit
zero  TxRqst bit unchanged

Note:  If a transmission is requested by programming bit TxRqst/NewDat in the IFx Command Mask Register, bit TxRqst in the IFx Message Control Register will be ignored.

Data A  Access Data Bytes 0-3
one  transfer Data Bytes 0-3 to Message Object.
zero  Data Bytes 0-3 unchanged.

Data B  Access Data Bytes 4-7
one  transfer Data Bytes 4-7 to Message Object.
zero  Data Bytes 4-7 unchanged.

3.3.2.2 Direction = Read

Mask  Access Mask Bits
one  transfer Identifier Mask + MDir + MXtd to IFx Message Buffer Register.
zero  Mask bits unchanged.

Arb  Access Arbitration Bits
one  transfer Identifier + Dir + Xtd + MsgVal to IFx Message Buffer Register.
zero  Arbitration bits unchanged.

Control  Access Control Bits
one  transfer Control Bits to IFx Message Buffer Register.
zero  Control Bits unchanged.

ClrIntPnd  Clear Interrupt Pending Bit
one  clear IntPnd bit in the Message Object.
zero  IntPnd bit remains unchanged.

TxRqst/NewDat  Access New Data Bit
one  clear NewDat bit in the Message Object.
zero  NewDat bit remains unchanged.

Note:  A read access to a Message Object can be combined with the reset of the control bits IntPnd and NewDat. The values of these bits transferred to the IFx Message Control Register always reflect the status before resetting these bits.

Data A  Access Data Bytes 0-3
one  transfer Data Bytes 0-3 to IFx Message Buffer Register.
zero  Data Bytes 0-3 unchanged.

Data B  Access Data Bytes 4-7
one  transfer Data Bytes 4-7 to IFx Message Buffer Register.
zero  Data Bytes 4-7 unchanged.
3.3.3 IFx Message Buffer Registers

The bits of the Message Buffer registers mirror the Message Objects in the Message RAM. The function of the Message Objects bits is described in chapter 3.3.3.

### 3.3.3.1 IFx Mask Registers

<table>
<thead>
<tr>
<th>IF1 Mask 1 Register</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1 Mask 2 Register</td>
<td>MXtd MDir res Msk28-16</td>
</tr>
<tr>
<td>IF2 Mask 1 Register</td>
<td>Msk15-0</td>
</tr>
<tr>
<td>IF2 Mask 2 Register</td>
<td>MXtd MDir res Msk28-16</td>
</tr>
</tbody>
</table>

### 3.3.3.2 IFx Arbitration Registers

<table>
<thead>
<tr>
<th>IF1 Arbitration 1 Register</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1 Arbitration 2 Register</td>
<td>ID5-0</td>
</tr>
<tr>
<td>IF2 Arbitration 1 Register</td>
<td>ID15-0</td>
</tr>
<tr>
<td>IF2 Arbitration 2 Register</td>
<td>ID28-16</td>
</tr>
</tbody>
</table>

### 3.3.3.3 IFx Message Control Registers

<table>
<thead>
<tr>
<th>IF1 Message Control Register</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF2 Message Control Register</td>
<td>NewDat MsgLst IntPnd UMask TxIE RxIE RmtEn TxRqst EoB res res res DLC3-0</td>
</tr>
</tbody>
</table>

### 3.3.3.4 IFx Data A and Data B Registers

The data bytes of CAN messages are stored in the IFx Message Buffer Registers in the following order:

<table>
<thead>
<tr>
<th>IF1 Message Data A1</th>
<th>Data(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF1 Message Data A2</td>
<td>Data(3)</td>
</tr>
<tr>
<td>IF1 Message Data B1</td>
<td>Data(5)</td>
</tr>
<tr>
<td>IF1 Message Data B2</td>
<td>Data(7)</td>
</tr>
<tr>
<td>IF2 Message Data A1</td>
<td>Data(1)</td>
</tr>
<tr>
<td>IF2 Message Data A2</td>
<td>Data(3)</td>
</tr>
<tr>
<td>IF2 Message Data B1</td>
<td>Data(5)</td>
</tr>
<tr>
<td>IF2 Message Data B2</td>
<td>Data(7)</td>
</tr>
</tbody>
</table>

In a CAN Data Frame, Data(0) is the first, Data(7) is the last byte to be transmitted or received. In CAN's serial bit stream, the MSB of each byte will be transmitted first.
3.3.4 Message Object in the Message Memory

There are 32 Message Objects in the Message RAM. To avoid conflicts between CPU access to the Message RAM and CAN message reception and transmission, the CPU cannot directly access the Message Objects, these accesses are handled via the IFx Interface Registers.

Figure 7 gives an overview of the two structure of a Message Object.

<table>
<thead>
<tr>
<th>Message Object</th>
<th>UMask</th>
<th>Msk28-0</th>
<th>MXtd</th>
<th>MDir</th>
<th>EoB</th>
<th>NewDat</th>
<th>MsgSt</th>
<th>RxIE</th>
<th>TxIE</th>
<th>IntPnd</th>
<th>RmtEn</th>
<th>TxRqst</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgVal</td>
<td>ID28-0</td>
<td>Xtd</td>
<td>Dir</td>
<td>DLC3-0</td>
<td>Data 0</td>
<td>Data 1</td>
<td>Data 2</td>
<td>Data 3</td>
<td>Data 4</td>
<td>Data 5</td>
<td>Data 6</td>
<td>Data 7</td>
</tr>
</tbody>
</table>

Table: Structure of a Message Object in the Message Memory

**MsgVal**
- **one**: The Message Object is configured and should be considered by the Message Handler.
- **zero**: The Message Object is ignored by the Message Handler.

**Note**: The CPU must reset the **MsgVal** bit of all unused Message Objects during the initialization before it sets bit **Init** in the CAN Control Register. This bit must also be reset before the identifier **Id28-0**, the control bits **Xtd**, **Dir**, or the Data Length Code **DLC3-0** are modified, or if the Message Object is no longer required.

**UMask**
- **Use Acceptance Mask**
  - **one**: Use Mask (Msk28-0, MXtd, and MDir) for acceptance filtering
  - **zero**: Mask ignored.

**Note**: If the **UMask** bit is set to **one**, the Message Object’s mask bits have to be programmed during initialization of the Message Object before **MsgVal** is set to **one**.

**ID28-0**
- **Message Identifier**
  - ID28 - ID0: 29-bit Identifier (“Extended Frame”).

**Msk28-0**
- **Identifier Mask**
  - **one**: The corresponding identifier bit is used for acceptance filtering.
  - **zero**: The corresponding bit in the identifier of the message object cannot inhibit the match in the acceptance filtering.

**Xtd**
- **Extended Identifier**
  - **one**: The 29-bit (“extended”) Identifier will be used for this Message Object.
  - **zero**: The 11-bit (“standard”) Identifier will be used for this Message Object.

**MXtd**
- **Mask Extended Identifier**
  - **one**: The extended identifier bit (IDE) is used for acceptance filtering.
  - **zero**: The extended identifier bit (IDE) has no effect on the acceptance filtering

**Note**: When 11-bit (“standard”) Identifiers are used for a Message Object, the identifiers of received Data Frames are written into bits **ID28** to **ID18**. For acceptance filtering, only these bits together with mask bits **Msk28** to **Msk18** are considered.
Dir       Message Direction
  one  Direction = transmit: On TxRqst, the respective Message Object is transmitted as a Data Frame. On reception of a Remote Frame with matching identifier, the TxRqst bit of this Message Object is set (if RmtEn = one).
  zero  Direction = receive: On TxRqst, a Remote Frame with the identifier of this Message Object is transmitted. On reception of a Data Frame with matching identifier, that message is stored in this Message Object.

MDir      Mask Message Direction
  one  The message direction bit (Dir) is used for acceptance filtering.
  zero  The message direction bit (Dir) has no effect on the acceptance filtering.

The Arbitration Registers ID28-0, Xtd, and Dir are used to define the identifier and type of outgoing messages and are used (together with the mask registers Msk28-0, MXtd, and MDir) for acceptance filtering of incoming messages. A received message is stored into the valid Message Object with matching identifier and Direction=receive (Data Frame) or Direction=transmit (Remote Frame). Extended frames can be stored only in Message Objects with Xtd = one, standard frames in Message Objects with Xtd = zero. If a received message (Data Frame or Remote Frame) matches with more than one valid Message Object, it is stored into that with the lowest message number. For details see chapter 4.2.3 Acceptance Filtering of Received Messages.

EoB       End of Buffer
  one  Single Message Object or last Message Object of a FIFO Buffer.
  zero  Message Object belongs to a FIFO Buffer and is not the last Message Object of that FIFO Buffer.

Note : This bit is used to concatenate two or more Message Objects (up to 32) to build a FIFO Buffer. For single Message Objects (not belonging to a FIFO Buffer) this bit must always be set to one. For details on the concatenation of Message Objects see chapter 4.7.

NewDat    New Data
  one  The Message Handler or the CPU has written new data into the data portion of this Message Object.
  zero  No new data has been written into the data portion of this Message Object by the Message Handler since last time this flag was cleared by the CPU.

MsgLst    Message Lost (only valid for Message Objects with direction = receive)
  one  The Message Handler stored a new message into this object when NewDat was still set, the CPU has lost a message.
  zero  No message lost since last time this bit was reset by the CPU.

RxIE      Receive Interrupt Enable
  one  IntPnd will be set after a successful reception of a frame.
  zero  IntPnd will be left unchanged after a successful reception of a frame.

TxIE      Transmit Interrupt Enable
  one  IntPnd will be set after a successful transmission of a frame.
  zero  IntPnd will be left unchanged after the successful transmission of a frame.

IntPnd    Interrupt Pending
  one  This message object is the source of an interrupt. The Interrupt Identifier in the Interrupt Register will point to this message object if there is no other interrupt source with higher priority.
  zero  This message object is not the source of an interrupt.
RmtEn  Remote Enable
one   At the reception of a Remote Frame, TxRqst is set.
zero  At the reception of a Remote Frame, TxRqst is left unchanged.

TxRqst  Transmit Request
one   The transmission of this Message Object is requested and is not yet done.
zero  This Message Object is not waiting for transmission.

DLC3-0  Data Length Code
0-8   Data Frame has 0-8 data bytes.
9-15  Data Frame has 8 data bytes

Note : The Data Length Code of a Message Object must be defined the same as in all the corresponding objects with the same identifier at other nodes. When the Message Handler stores a data frame, it will write the DLC to the value given by the received message.

Data 0  1st data byte of a CAN Data Frame
Data 1  2nd data byte of a CAN Data Frame
Data 2  3rd data byte of a CAN Data Frame
Data 3  4th data byte of a CAN Data Frame
Data 4  5th data byte of a CAN Data Frame
Data 5  6th data byte of a CAN Data Frame
Data 6  7th data byte of a CAN Data Frame
Data 7  8th data byte of a CAN Data Frame

Note : Byte Data 0 is the first data byte shifted into the shift register of the CAN Core during a reception, byte Data 7 is the last. When the Message Handler stores a Data Frame, it will write all the eight data bytes into a Message Object. If the Data Length Code is less than 8, the remaining bytes of the Message Object will be overwritten by non specified values.

3.4 Message Handler Registers
All Message Handler registers are read-only. Their contents (TxRqst, NewDat, IntPnd, and MsgVal bits of each Message Object and the Interrupt Identifier) is status information provided by the Message Handler FSM.

3.4.1 Interrupt Register (addresses 0x09 & 0x08)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IntId15-0  Interrupt Identifier (the number here indicates the source of the interrupt)
0x0000  No interrupt is pending.
0x0001-0x0020  Number of Message Object which caused the interrupt.
0x0021-0x7FFF  unused.
0x8000  Status Interrupt.
0x8001-0xFFF  unused

If several interrupts are pending, the CAN Interrupt Register will point to the pending interrupt with the highest priority, disregarding their chronological order. An interrupt remains pending until the CPU has cleared it. If IntId is different from 0x0000 and IE is set, the interrupt line to
the CPU, IRQ_B, is active. The interrupt line remains active until \texttt{IntId} is back to value 0x0000 (the cause of the interrupt is reset) or until \texttt{IE} is reset.

The Status Interrupt has the highest priority. Among the message interrupts, the Message Object’s interrupt priority decreases with increasing message number.

A message interrupt is cleared by clearing the Message Object’s \texttt{IntPnd} bit. The Status Interrupt is cleared by reading the Status Register.

### 3.4.2 Transmission Request Registers

<table>
<thead>
<tr>
<th>Transmission Request 1 Register (addresses 0x81 &amp; 0x80)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{TxRqst16-9}</td>
<td>\texttt{TxRqst8-1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transmission Request 2 Register (addresses 0x83 &amp; 0x82)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{TxRqst32-25}</td>
<td>\texttt{TxRqst24-17}</td>
</tr>
</tbody>
</table>

\textbf{TxRqst32-1} Transmission Request Bits (of all Message Objects)

- \texttt{one}  The transmission of this Message Object is requested and is not yet done.
- \texttt{zero}  This Message Object is not waiting for transmission.

These registers hold the \texttt{TxRqst} bits of the 32 Message Objects. By reading out the \texttt{TxRqst} bits, the CPU can check for which Message Object a Transmission Request is pending. The \texttt{TxRqst} bit of a specific Message Object can be set/reset by the CPU via the IFx Message Interface Registers or by the Message Handler after reception of a Remote Frame or after a successful transmission.

### 3.4.3 New Data Registers

<table>
<thead>
<tr>
<th>New Data 1 Register (addresses 0x91 &amp; 0x90)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NewDat16-9}</td>
<td>\texttt{NewDat8-1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New Data 2 Register (addresses 0x93 &amp; 0x92)</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NewDat32-25}</td>
<td>\texttt{NewDat24-17}</td>
</tr>
</tbody>
</table>

\textbf{NewDat32-1} New Data Bits (of all Message Objects)

- \texttt{one}  The Message Handler or the CPU has written new data into the data portion of this Message Object.
- \texttt{zero}  No new data has been written into the data portion of this Message Object by the Message Handler since last time this flag was cleared by the CPU.

\textbf{MsgLst} These registers hold the \texttt{NewDat} bits of the 32 Message Objects. By reading out the \texttt{NewDat} bits, the CPU can check for which Message Object the data portion was updated. The \texttt{NewDat} bit of a specific Message Object can be set/reset by the CPU via the IFx Message Interface Registers or by the Message Handler after reception of a Data Frame or after a successful transmission.
3.4.4 Interrupt Pending Registers

<table>
<thead>
<tr>
<th>Interrupt Pending 1 Register</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(addresses 0xA1 &amp; 0xA0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntPnd16</td>
<td>IntPnd8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interrupt Pending 2 Register</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(addresses 0xA3 &amp; 0xA2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntPnd32</td>
<td>IntPnd24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IntPnd32-1 Interrupt Pending Bits (of all Message Objects)

- **one**  This message object is the source of an interrupt.
- **zero** This message object is not the source of an interrupt.

These registers hold the IntPnd bits of the 32 Message Objects. By reading out the IntPnd bits, the CPU can check for which Message Object an interrupt is pending. The IntPnd bit of a specific Message Object can be set/reset by the CPU via the IFx Message Interface Registers or by the Message Handler after reception or after a successful transmission of a frame. This will also affect the value of IntId in the Interrupt Register.

3.4.5 Message Valid 1 Register

<table>
<thead>
<tr>
<th>Message Valid 1 Register</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(addresses 0xB1 &amp; 0xB0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MsgVal16</td>
<td>MsgVal8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message Valid 2 Register</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(addresses 0xB3 &amp; 0xB2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MsgVal32</td>
<td>MsgVal24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MsgVal32-1 Message Valid Bits (of all Message Objects)

- **one**  This Message Object is configured and should be considered by the Message Handler.
- **zero** This Message Object is ignored by the Message Handler.

These registers hold the MsgVal bits of the 32 Message Objects. By reading out the MsgVal bits, the CPU can check which Message Object is valid. The MsgVal bit of a specific Message Object can be set/reset by the CPU via the IFx Message Interface Registers.
4. CAN Application

4.1 Management of Message Objects

The configuration of the Message Objects in the Message RAM will (with the exception of the bits MsgVal, NewDat, IntPnd, and TxRqst) not be affected by resetting the chip. All the Message Objects must be initialized by the CPU or they must be not valid (MsgVal = ‘0’) and the bit timing must be configured before the CPU clears the Init bit in the CAN Control Register.

The configuration of a Message Object is done by programming Mask, Arbitration, Control and Data field of one of the two interface register sets to the desired values. By writing to the corresponding IFx Command Request Register, the IFx Message Buffer Registers are loaded into the addressed Message Object in the Message RAM.

When the Init bit in the CAN Control Register is cleared, the CAN Protocol Controller state machine of the CAN_Core and the Message Handler State Machine control the C_CAN’s internal data flow. Received messages that pass the acceptance filtering are stored into the Message RAM, messages with pending transmission request are loaded into the CAN_Core’s Shift Register and are transmitted via the CAN bus.

The CPU reads received messages and updates messages to be transmitted via the IFx Interface Registers. Depending on the configuration, the CPU is interrupted on certain CAN message and CAN error events.

4.2 Message Handler State Machine

The Message Handler controls the data transfer between the Rx/Tx Shift Register of the CAN Core, the Message RAM and the IFx Registers.

The Message Handler FSM controls the following functions:

- Data Transfer from IFx Registers to the Message RAM
- Data Transfer from Message RAM to the IFx Registers
- Data Transfer from Shift Register to the Message RAM
- Data Transfer from Message RAM to Shift Register
- Data Transfer from Shift Register to the Acceptance Filtering unit
- Scanning of Message RAM for a matching Message Object
- Handling of TxRqst flags.
- Handling of interrupts.

4.2.1 Data Transfer from / to Message RAM

When the CPU initiates a data transfer between the IFx Registers and Message RAM, the Message Handler sets the Busy bit in the respective Command Register to ‘1’. After the transfer has completed, the Busy bit is set back to ‘0’ (see figure 8).

The respective Command Mask Register specifies whether a complete Message Object or only parts of it will be transferred. Due to the structure of the Message RAM it is not possible to write single bits/bytes of one Message Object, it is always necessary to write a complete Message Object into the Message RAM. Therefore the data transfer from the IFx Registers to the Message RAM requires of a read-modify-write cycle. First that parts of the Message
Object that are not to be changes are read from the Message RAM and then the complete contents of the Message Buffer Registers are into the Message Object.

Figure 8: Data Transfer between IFx Registers and Message RAM

After the partial write of a Message Object, that Message Buffer Registers that are not selected in the Command Mask Register will set to the actual contents of the selected Message Object.

After the partial read of a Message Object, that Message Buffer Registers that are not selected in the Command Mask Register will be left unchanged.

4.2.2 Transmission of Messages

If the shift register of the CAN Core cell is ready for loading and if there is no data transfer between the IFx Registers and Message RAM, the MsgVal bits in the Message Valid Register TxRqst bits in the Transmission Request Register are evaluated. The valid Message Object with the highest priority pending transmission request is loaded into the shift register by the Message Handler and the transmission is started. The Message Object's NewDat bit is reset.

After a successful transmission and if no new data was written to the Message Object (NewDat = '0') since the start of the transmission, the TxRqst bit will be reset. If TxIE is set, IntPnd will be set after a successful transmission. If the C_CAN has lost the arbitration or if an error occurred during the transmission, the message will be retransmitted as soon as the CAN
bus is free again. If meanwhile the transmission of a message with higher priority has been requested, the messages will be transmitted in the order of their priority.

4.2.3 Acceptance Filtering of Received Messages

When the arbitration and control field (Identifier + IDE + RTR + DLC) of an incoming message is completely shifted into the Rx/Tx Shift Register of the CAN Core, the Message Handler FSM starts the scanning of the Message RAM for a matching valid Message Object.

To scan the Message RAM for a matching Message Object, the Acceptance Filtering unit is loaded with the arbitration bits from the CAN Core shift register. Then the arbitration and mask fields (including \texttt{MsgVal}, \texttt{UMask}, \texttt{NewDat}, and \texttt{EoB}) of Message Object 1 are loaded into the Acceptance Filtering unit and compared with the arbitration field from the shift register. This is repeated with each following Message Object until a matching Message Object is found or until the end of the Message RAM is reached.

If a match occurs, the scanning is stopped and the Message Handler FSM proceeds depending on the type of frame (Data Frame or Remote Frame) received.

4.2.3.1 Reception of Data Frame

The Message Handler FSM stores the message from the CAN Core shift register into the respective Message Object in the Message RAM. Not only the data bytes, but all arbitration bits and the Data Length Code are stored into the corresponding Message Object. This is implemented to keep the data bytes connected with the identifier even if arbitration mask registers are used.

The \texttt{NewDat} bit is set to indicate that new data (not yet seen by the CPU) has been received. The CPU should reset \texttt{NewDat} when it reads the Message Object. If at the time of the reception the \texttt{NewDat} bit was already set, \texttt{MsgLst} is set to indicate that the previous data (supposedly not seen by the CPU) is lost. If the \texttt{RxIE} bit is set, the \texttt{IntPnd} bit is set, causing the Interrupt Register to point to this Message Object.

The \texttt{TxRqst} bit of this Message Object is reset to prevent the transmission of a Remote Frame, while the requested Data Frame has just been received.

4.2.3.2 Reception of Remote Frame

When a Remote Frame is received, three different configurations of the matching Message Object have to be considered:

1) \texttt{Dir = '1' (direction = transmit), RmtEn = '1', UMask = '1' or '0'}

   At the reception of a matching Remote Frame, the \texttt{TxRqst} bit of this Message Object is set. The rest of the Message Object remains unchanged.

2) \texttt{Dir = '1' (direction = transmit), RmtEn = '0', UMask = '0'}

   At the reception of a matching Remote Frame, the \texttt{TxRqst} bit of this Message Object remains unchanged; the Remote Frame is ignored.

3) \texttt{Dir = '1' (direction = transmit), RmtEn = '0', UMask = '1'}

   At the reception of a matching Remote Frame, the \texttt{TxRqst} bit of this Message Object is reset. The arbitration and control field (Identifier + IDE + RTR + DLC) from the shift register is stored into the Message Object in the Message RAM and the \texttt{NewDat} bit of this Message Object is set. The data field of the Message Object remains unchanged; the Remote Frame is treated similar to a received Data Frame.
4.2.4 Receive / Transmit Priority

The receive/transmit priority for the Message Objects is attached to the message number. Message Object 1 has the highest priority, while Message Object 32 has the lowest priority. If more than one transmission request is pending, they are serviced due to the priority of the corresponding Message Object.

4.3 Configuration of a Transmit Object

Figure 9 shows how a Transmit Object should be initialised.

<table>
<thead>
<tr>
<th>MsgVal</th>
<th>Arb</th>
<th>Data</th>
<th>Mask</th>
<th>EoB</th>
<th>Dir</th>
<th>NewDat</th>
<th>MsgLst</th>
<th>RxEI</th>
<th>TxEI</th>
<th>IntPnd</th>
<th>RmtEn</th>
<th>TxRqst</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>appl.</td>
<td>appl.</td>
<td>appl.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>appl.</td>
<td>0</td>
<td>appl.</td>
<td>0</td>
<td>appl.</td>
</tr>
</tbody>
</table>

Figure 9: Initialisation of a Transmit Object

The Arbitration Registers (ID28-0 and Xtd bit) are given by the application. They define the identifier and type of the outgoing message. If an 11-bit Identifier (“Standard Frame”) is used, it is programmed to ID28 - ID18, ID17 - ID0 can then be disregarded.

If the TxIE bit is set, the IntPnd bit will be set after a successful transmission of the Message Object.

If the RmtEn bit is set, a matching received Remote Frame will cause the TxRqst bit to be set; the Remote Frame will autonomously be answered by a Data Frame.

The Data Registers (DLC3-0, Data0-7) are given by the application, TxRqst and RmtEn may not be set before the data is valid.

The Mask Registers (Msk28-0, UMask, MXtd, and MDir bits) may be used (UMask='1') to allow groups of Remote Frames with similar identifiers to set the TxRqst bit. For details see section 4.2.3.2, handle with care. The Dir bit should not be masked.

4.4 Updating a Transmit Object

The CPU may update the data bytes of a Transmit Object any time via the IFx Interface registers, neither MsgVal nor TxRqst have to be reset before the update.

Even if only a part of the data bytes are to be updated, all four bytes of the corresponding IFx Data A Register or IFx Data B Register have to be valid before the content of that register is transferred to the Message Object. Either the CPU has to write all four bytes into the IFx Data Register or the Message Object is transferred to the IFx Data Register before the CPU writes the new data bytes.

When only the (eight) data bytes are updated, first 0x0087 is written to the Command Mask Register and then the number of the Message Object is written to the Command Request Register, concurrently updating the data bytes and setting TxRqst.

To prevent the reset of TxRqst at the end of a transmission that may already be in progress while the data is updated, NewDat has to be set together with TxRqst. For details see section 4.2.2.

When NewDat is set together with TxRqst, NewDat will be reset as soon as the new transmission has started.
4.5 Configuration of a Receive Object

Figure 9 shows how a Receive Object should be initialised.

<table>
<thead>
<tr>
<th>MsgVal</th>
<th>Arb</th>
<th>Data</th>
<th>Mask</th>
<th>EoB</th>
<th>Dir</th>
<th>NewDat</th>
<th>MsgLst</th>
<th>RxIE</th>
<th>TxIE</th>
<th>IntPnd</th>
<th>RmtEn</th>
<th>TxRqst</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>appl.</td>
<td>appl.</td>
<td>appl.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>appl.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 10: Initialisation of a Receive Object

The Arbitration Registers (ID28-0 and Xtd bit) are given by the application. They define the identifier and type of accepted received messages. If an 11-bit Identifier (“Standard Frame”) is used, it is programmed to ID28 - ID18, ID17 - ID0 can then be disregarded. When a Data Frame with an 11-bit Identifier is received, ID17 - ID0 will be set to ‘0’.

If the RxIE bit is set, the IntPnd bit will be set when a received Data Frame is accepted and stored in the Message Object.

The Data Length Code (DLC3-0) is given by the application. When the Message Handler stores a Data Frame in the Message Object, it will store the received Data Length Code and eight data bytes. If the Data Length Code is less than 8, the remaining bytes of the Message Object will be overwritten by non specified values.

The Mask Registers (Msk28-0, UMask, MXtd, and MDir bits) may be used (UMask=’1’) to allow groups of Data Frames with similar identifiers to be accepted. For details see section 4.2.3.1. The Dir bit should not be masked in typical applications.

4.6 Handling of Received Messages

The CPU may read a received message any time via the IFx Interface registers, the data consistency is guaranteed by the Message Handler state machine.

Typically the CPU will write first 0x007F to the Command Mask Register and then the number of the Message Object to the Command Request Register. That combination will transfer the whole received message from the Message RAM into the Message Buffer Register. Additionally, the bits NewDat and IntPnd are cleared in the Message RAM (not in the Message Buffer).

If the Message Object uses masks for acceptance filtering, the arbitration bits show which of the matching messages has been received.

The actual value of NewDat shows whether a new message has been received since last time this Message Object was read. The actual value of MsgLst shows whether more than one message has been received since last time this Message Object was read. MsgLst will not be automatically reset.

By means of a Remote Frame, the CPU may request another CAN node to provide new data for a receive object. Setting the TxRqst bit of a receive object will cause the transmission of a Remote Frame with the receive object’s identifier. This Remote Frame triggers the other CAN node to start the transmission of the matching Data Frame. If the matching Data Frame is received before the Remote Frame could be transmitted, the TxRqst bit is automatically reset.
4.7 Configuration of a FIFO Buffer

With the exception of the **EoB** bit, the configuration of Receive Objects belonging to a FIFO Buffer is the same as the configuration of a (single) Receive Object, see section 4.5.

To concatenate two or more Message Objects into a FIFO Buffer, the identifiers and masks (if used) of these Message Objects have to be programmed to matching values. Due to the implicit priority of the Message Objects, the Message Object with the lowest number will be the first Message Object of the FIFO Buffer. The **EoB** bit of all Message Objects of a FIFO Buffer except the last have to be programmed to **zero**. The **EoB** bits of the last Message Object of a FIFO Buffer is set to **one**, configuring it as the **End of the Block**.

4.8 Reception of Messages with FIFO Buffers

Received messages with identifiers matching to a FIFO Buffer are stored into a Message Object of this FIFO Buffer starting with the Message Object with the lowest message number.

When a message is stored into a Message Object of a FIFO Buffer the **NewDat** bit of this Message Object is set. By setting **NewDat** while **EoB** is zero the Message Object is locked for further write accesses by the Message Handler until the CPU has written the **NewDat** bit back to zero.

Messages are stored into a FIFO Buffer until the last Message Object of this FIFO Buffer is reached. If none of the preceding Message Objects is released by writing **NewDat** to zero, all further messages for this FIFO Buffer will be written into the last Message Object of the FIFO Buffer and therefore overwrite previous messages.

4.8.1 Reading from a FIFO Buffer

When the CPU transfers the contents of Message Object to the IFx Message Bugger registers by writing its number to the IFx Command Request Register, the corresponding Command Mask Register should be programmed the way that bits **NewDat** and **IntPnd** are reset to **zero** (**TxRqst/NewDat** = ‘1’ and **ClrIntPnd** = ‘1’). The values of these bits in the Message Control Register always reflect the status before resetting the bits.

To assure the correct function of a FIFO Buffer, the CPU should read out the Message Objects starting at the FIFO Object with the lowest message number.

Figure 11 shows how a set of Message Objects which are concatenated to a FIFO Buffer can be handled by the CPU.
Figure 11: CPU Handling of a FIFO Buffer

START Message Interrupt

Read Interrupt Pointer

case Interrupt Pointer
  0x8000h
  else 0x0000h

Status Change Interrupt Handling

MessageNum = Interrupt Pointer

Write MessageNum to IFx Command Request
(Read Message to IFx Registers,
  Reset NewDat = 0,
  Reset IntPnd = 0)

Read IFx Message Control

NewDat = 1

Yes

Read Data from IFx Data A,B

EoB = 1

Yes

MessageNum = MessageNum + 1

No

No

END
4.9 Handling of Interrupts

If several interrupts are pending, the CAN Interrupt Register will point to the pending interrupt with the highest priority, disregarding their chronological order. An interrupt remains pending until the CPU has cleared it.

The Status Interrupt has the highest priority. Among the message interrupts, the Message Object’s interrupt priority decreases with increasing message number.

A message interrupt is cleared by clearing the Message Object’s IntPnd bit. The Status Interrupt is cleared by reading the Status Register.

The interrupt identifier IntId in the Interrupt Register indicates the cause of the interrupt. When no interrupt is pending, the register will hold the value zero. If the value of the Interrupt Register is different from zero, then there is an interrupt pending and, if IE is set, the interrupt line to the CPU, IRQ_B, is active. The interrupt line remains active until the Interrupt Register is back to value zero (the cause of the interrupt is reset) or until IE is reset.

The value 0x8000 indicates that an interrupt is pending because the CAN Core has updated (not necessarily changed) the Status Register (Error Interrupt or Status Interrupt). This interrupt has the highest priority. The CPU can update (reset) the status bits RxOk, TxOk and LEC, but a write access of the CPU to the Status Register can never generate or reset an interrupt.

All other values indicate that the source of the interrupt is one of the Message Objects, IntId points to the pending message interrupt with the highest interrupt priority.

The CPU controls whether a change of the Status Register may cause an interrupt (bits EIE and SIE in the CAN Control Register) and whether the interrupt line becomes active when the Interrupt Register is different from zero (bit IE in the CAN Control Register). The Interrupt Register will be updated even when IE is reset.

The CPU has two possibilities to follow the source of a message interrupt. First it can follow the IntId in the Interrupt Register and second it can poll the Interrupt Pending Register (see section 3.4.4).

An interrupt service routine reading the message that is the source of the interrupt may read the message and reset the Message Object’s IntPnd at the same time (bit ClrIntPnd in the Command Mask Register). When IntPnd is cleared, the Interrupt Register will point to the next Message Object with a pending interrupt.

4.10 Configuration of the Bit Timing

Even if minor errors in the configuration of the CAN bit timing do not result in immediate failure, the performance of a CAN network can be reduced significantly.

In many cases, the CAN bit synchronisation will amend a faulty configuration of the CAN bit timing to such a degree that only occasionally an error frame is generated. In the case of arbitration however, when two or more CAN nodes simultaneously try to transmit a frame, a misplaced sample point may cause one of the transmitters to become error passive.

The analysis of such sporadic errors requires a detailed knowledge of the CAN bit synchronisation inside a CAN node and of the CAN nodes’ interaction on the CAN bus.
4.10.1 Bit Time and Bit Rate

CAN supports bit rates in the range of lower than 1 kBit/s up to 1000 kBit/s. Each member of the CAN network has its own clock generator, usually a quartz oscillator. The timing parameter of the bit time (i.e. the reciprocal of the bit rate) can be configured individually for each CAN node, creating a common bit rate even though the CAN nodes’ oscillator periods \( f_{osc} \) may be different.

The frequencies of these oscillators are not absolutely stable, small variations are caused by changes in temperature or voltage and by deteriorating components. As long as the variations remain inside a specific oscillator tolerance range \( df \), the CAN nodes are able to compensate for the different bit rates by resynchronising to the bit stream.

According to the CAN specification, the bit time is divided into four segments (see figure 12). The Synchronisation Segment, the Propagation Time Segment, the Phase Buffer Segment 1, and the Phase Buffer Segment 2. Each segment consists of a specific, programmable number of time quanta (see Table 1). The length of the time quantum \( t_q \), which is the basic time unit of the bit time, is defined by the CAN controller’s system clock \( f_{sys} \) and the Baud Rate Prescaler (BRP): \( t_q = \frac{BRP}{f_{sys}} \). The C_CAN’s system clock \( f_{sys} \) is the frequency of its CAN_CLK input.

The Synchronisation Segment Sync_Seg is that part of the bit time where edges of the CAN bus level are expected to occur; the distance between an edge that occurs outside of Sync_Seg and the Sync_Seg is called the phase error of that edge. The Propagation Time Segment Prop_Seg is intended to compensate for the physical delay times within the CAN network. The Phase Buffer Segments Phase_Seg1 and Phase_Seg2 surround the Sample Point. The (Re-)Synchronisation Jump Width (SJW) defines how far a resynchronisation may move the Sample Point inside the limits defined by the Phase Buffer Segments to compensate for edge phase errors.

![Figure 12: Bit Timing](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRP</td>
<td>([1 .. 32])</td>
<td>defines the length of the time quantum ( t_q )</td>
</tr>
<tr>
<td>Sync_Seg</td>
<td>(1 t_q)</td>
<td>fixed length, synchronisation of bus input to system clock</td>
</tr>
<tr>
<td>Prop_Seg</td>
<td>([1 .. 8] t_q)</td>
<td>compensates for the physical delay times</td>
</tr>
<tr>
<td>Phase_Seg1</td>
<td>([1 .. 8] t_q)</td>
<td>may be lengthened temporarily by synchronisation</td>
</tr>
<tr>
<td>Phase_Seg2</td>
<td>([1 .. 8] t_q)</td>
<td>may be shortened temporarily by synchronisation</td>
</tr>
<tr>
<td>SJW</td>
<td>([1 .. 4] t_q)</td>
<td>may not be longer than either Phase Buffer Segment</td>
</tr>
</tbody>
</table>

This table describes the minimum programmable ranges required by the CAN protocol

Table 1 : Parameters of the CAN Bit Time
A given bit rate may be met by different bit time configurations, but for the proper function of the CAN network the physical delay times and the oscillator’s tolerance range have to be considered.

### 4.10.2 Propagation Time Segment

This part of the bit time is used to compensate physical delay times within the network. These delay times consist of the signal propagation time on the bus and the internal delay time of the CAN nodes.

Any CAN node synchronised to the bit stream on the CAN bus will be out of phase with the transmitter of that bit stream, caused by the signal propagation time between the two nodes. The CAN protocol’s non-destructive bitwise arbitration and the dominant acknowledge bit provided by receivers of CAN messages require that a CAN node transmitting a bit stream must also be able to receive dominant bits transmitted by other CAN nodes that are synchronised to that bit stream. The example in figure 13 shows the phase shift and propagation times between two CAN nodes.

*Figure 13: The Propagation Time Segment*

In this example, both nodes A and B are transmitters performing an arbitration for the CAN bus. The node A has sent its Start of Frame bit less than one bit time earlier than node B, therefore node B has synchronised itself to the received edge from recessive to dominant. Since node B has received this edge delay(A_to_B) after it has been transmitted, B’s bit timing segments are shifted with regard to A. Node B sends an identifier with higher priority and so it will win the arbitration at a specific identifier bit when it transmits a dominant bit while node A transmits a recessive bit. The dominant bit transmitted by node B will arrive at node A after the delay(B_to_A).

Due to oscillator tolerances, the actual position of node A’s Sample Point can be anywhere inside the nominal range of node A’s Phase Buffer Segments, so the bit transmitted by node B must arrive at node A before the start of Phase_Seg1. This condition defines the length of Prop_Seg.

If the edge from recessive to dominant transmitted by node B would arrive at node A after the start of Phase_Seg1, it could happen that node A samples a recessive bit instead of a dominant bit, resulting in a bit error and the destruction of the current frame by an error flag.
The error occurs only when two nodes arbitrate for the CAN bus that have oscillators of opposite ends of the tolerance range and that are separated by a long bus line; this is an example of a minor error in the bit timing configuration (Prop_Seg to short) that causes sporadic bus errors.

Some CAN implementations provide an optional 3 Sample Mode. The C_CAN does not. In this mode, the CAN bus input signal passes a digital low-pass filter, using three samples and a majority logic to determine the valid bit value. This results in an additional input delay of \(1\ \text{t}_q\), requiring a longer Prop_Seg.

### 4.10.3 Phase Buffer Segments and Synchronisation

The Phase Buffer Segments (Phase_Seg1 and Phase_Seg2) and the Synchronisation Jump Width (SJW) are used to compensate for the oscillator tolerance. The Phase Buffer Segments may be lengthened or shortened by synchronisation.

Synchronisations occur on edges from recessive to dominant, their purpose is to control the distance between edges and Sample Points.

Edges are detected by sampling the actual bus level in each time quantum and comparing it with the bus level at the previous Sample Point. A synchronisation may be done only if a recessive bit was sampled at the previous Sample Point and if the actual time quantum’s bus level is dominant.

An edge is synchronous if it occurs inside of Sync_Seg, otherwise the distance between edge and the end of Sync_Seg is the edge phase error, measured in time quanta. If the edge occurs before Sync_Seg, the phase error is negative, else it is positive.

Two types of synchronisation exist: Hard Synchronisation and Resynchronisation. A Hard Synchronisation is done once at the start of a frame; inside a frame only Resynchronisations occur.

- **Hard Synchronisation**

  After a hard synchronisation, the bit time is restarted with the end of Sync_Seg, regardless of the edge phase error. Thus hard synchronisation forces the edge which has caused the hard synchronisation to lie within the synchronisation segment of the restarted bit time.

- **Bit Resynchronisation**

  Resynchronisation leads to a shortening or lengthening of the bit time such that the position of the sample point is shifted with regard to the edge.

  When the phase error of the edge which causes Resynchronisation is positive, Phase_Seg1 is lengthened. If the magnitude of the phase error is less than SJW, Phase_Seg1 is lengthened by the magnitude of the phase error, else it is lengthened by SJW.

  When the phase error of the edge which causes Resynchronisation is negative, Phase_Seg2 is shortened. If the magnitude of the phase error is less than SJW, Phase_Seg2 is shortened by the magnitude of the phase error, else it is shortened by SJW.

  When the magnitude of the phase error of the edge is less than or equal to the programmed value of SJW, the results of Hard Synchronisation and Resynchronisation are the same. If the magnitude of the phase error is larger than SJW, the Resynchronisation cannot compensate the phase error completely, an error of \((\text{phase error} - \text{SJW})\) remains.

  Only one synchronisation may be done between two Sample Points. The Synchronisations maintain a minimum distance between edges and Sample Points, giving the bus level time to stabilise and filtering out spikes that are shorter than \((\text{Prop_Seg} + \text{Phase_Seg1})\).
Apart from noise spikes, most synchronisations are caused by arbitration. All nodes synchronise “hard” on the edge transmitted by the “leading” transceiver that started transmitting first, but due to propagation delay times, they cannot become ideally synchronised. The “leading” transmitter does not necessarily win the arbitration, therefore the receivers have to synchronise themselves to different transmitters that subsequently “take the lead” and that are differently synchronised to the previously “leading” transmitter. The same happens at the acknowledge field, where the transmitter and some of the receivers will have to synchronise to that receiver that “takes the lead” in the transmission of the dominant acknowledge bit.

Synchronisations after the end of the arbitration will be caused by oscillator tolerance, when the differences in the oscillator’s clock periods of transmitter and receivers sum up during the time between synchronisations (at most ten bits). These summarized differences may not be longer than the SJW, limiting the oscillator’s tolerance range.

The examples in figure 14 show how the Phase Buffer Segments are used to compensate for phase errors. There are three drawings of each two consecutive bit timings. The upper drawing shows the synchronisation on a “late” edge, the lower drawing shows the synchronisation on an “early” edge, and the middle drawing is the reference without synchronisation.

In the first example an edge from recessive to dominant occurs at the end of Prop_Seg. The edge is “late” since it occurs after the Sync_Seg. Reacting to the “late” edge, Phase_Seg1 is lengthened so that the distance from the edge to the Sample Point is the same as it would have been from the Sync_Seg to the Sample Point if no edge had occurred. The phase error of this “late” edge is less than SJW, so it is fully compensated and the edge from dominant to recessive at the end of the bit, which is one nominal bit time long, occurs in the Sync_Seg.

In the second example an edge from recessive to dominant occurs during Phase_Seg2. The edge is “early” since it occurs before a Sync_Seg. Reacting to the “early” edge, Phase_Seg2 is shortened and Sync_Seg is omitted, so that the distance from the edge to the Sample Point is the same as it would have been from an Sync_Seg to the Sample Point if no edge had
occurred. As in the previous example, the magnitude of this “early” edge’s phase error is less than SJW, so it is fully compensated.

The Phase Buffer Segments are lengthened or shortened temporarily only; at the next bit time, the segments return to their nominal programmed values.

In these examples, the bit timing is seen from the point of view of the CAN implementation’s state machine, where the bit time starts and ends at the Sample Points. The state machine omits Sync_Seg when synchronising on an “early” edge because it cannot subsequently redefine that time quantum of Phase_Seg2 where the edge occurs to be the Sync_Seg.

The examples in figure 15 show how short dominant noise spikes are filtered by synchronisations. In both examples the spike starts at the end of Prop_Seg and has the length of (Prop_Seg + Phase_Seg1).

In the first example, the Synchronisation Jump Width is greater than or equal to the phase error of the spike’s edge from recessive to dominant. Therefore the Sample Point is shifted after the end of the spike; a recessive bus level is sampled.

In the second example, SJW is shorter than the phase error, so the Sample Point cannot be shifted far enough; the dominant spike is sampled as actual bus level.

\[
\text{Sync}_\text{Seg} \quad \text{Prop}_\text{Seg} \quad \text{Phase}_\text{Seg1} \quad \text{Phase}_\text{Seg2}
\]

![Figure 15: Filtering of Short Dominant Spikes](image)

### 4.10.4 Oscillator Tolerance Range

The oscillator tolerance range was increased when the CAN protocol was developed from version 1.1 to version 1.2 (version 1.0 was never implemented in silicon). The option to synchronise on edges from dominant to recessive became obsolete, only edges from recessive to dominant are considered for synchronisation. The only CAN controllers to implement protocol version 1.1 have been Intel 82526 and Philips 82C200, both are superseded by successor products. The protocol update to version 2.0 (A and B) had no influence on the oscillator tolerance.

The tolerance range df for an oscillator’s frequency \(f_{\text{osc}}\) around the nominal frequency \(f_{\text{nom}}\) with \((1 - df) \cdot f_{\text{nom}} \leq f_{\text{osc}} \leq (1 + df) \cdot f_{\text{nom}}\) depends on the proportions of Phase_Seg1, Phase_Seg2,
SJW, and the bit time. The maximum tolerance df is the defined by two conditions (both shall be met):

\[ \text{I: } df \leq \frac{\min(\text{Phase}_1, \text{Phase}_2)}{2} \cdot (13 \cdot \text{bit time} - \text{Phase}_2) \]

\[ \text{II: } df \leq \frac{\text{SJW}}{20 \cdot \text{bit time}} \]

It has to be considered that SJW may not be larger than the smaller of the Phase Buffer Segments and that the Propagation Time Segment limits that part of the bit time that may be used for the Phase Buffer Segments.

The combination Prop_Seg = 1 and Phase_Seg1 = Phase_Seg2 = SJW = 4 allows the largest possible oscillator tolerance of 1.58%. This combination with a Propagation Time Segment of only 10% of the bit time is not suitable for short bit times; it can be used for bit rates of up to 125 kBit/s (bit time = 8 μs) with a bus length of 40 m.

4.10.5 Configuration of the CAN Protocol Controller

In most CAN implementations and also in the C_CAN, the bit timing configuration is programmed in two register bytes. The sum of Prop_Seg and Phase_Seg1 (as TSEG1) is combined with Phase_Seg2 (as TSEG2) in one byte, SJW and BRP are combined in the other byte (see figure 16).

In these bit timing registers, the four components TSEG1, TSEG2, SJW, and BRP have to be programmed to a numerical value that is one less than its functional value; so instead of values in the range of \([1..n]\), values in the range of \([0..n-1]\) are programmed. That way, e.g. SJW (functional range of \([1..4]\)) is represented by only two bits.

Therefore the length of the bit time is (programmed values) \([TSEG1 + TSEG2 + 3] \cdot t_q\) or (functional values) \([\text{Sync}_1 + \text{Prop}_1 + \text{Phase}_1 + \text{Phase}_2] \cdot t_q\).

---

Figure 16: Structure of the CAN Core's CAN Protocol Controller

The data in the bit timing registers are the configuration input of the CAN protocol controller. The Baud Rate Prescaler (configured by BRP) defines the length of the time quantum, the
basic time unit of the bit time; the Bit Timing Logic (configured by TSEG1, TSEG2, and SJW) defines the number of time quanta in the bit time.

The processing of the bit time, the calculation of the position of the Sample Point, and occasional synchronisations are controlled by the BTL state machine, which is evaluated once each time quantum. The rest of the CAN protocol controller, the Bit Stream Processor (BSP) state machine is evaluated once each bit time, at the Sample Point.

The Shift Register serializes the messages to be sent and parallelizes received messages. Its loading and shifting is controlled by the BSP.

The BSP translates messages into frames and vice versa. It generates and discards the enclosing fixed format bits, inserts and extracts stuff bits, calculates and checks the CRC code, performs the error management, and decides which type of synchronisation is to be used. It is evaluated at the Sample Point and processes the sampled bus input bit. The time after the Sample point that is needed to calculate the next bit to be sent (e.g. data bit, CRC bit, stuff bit, error flag, or idle) is called the Information Processing Time (IPT).

The IPT is application specific but may not be longer than 2 \( t_q \); the C_CAN's IPT is 0 \( t_q \). Its length is the lower limit of the programmed length of Phase_Seg2. In case of a synchronisation, Phase_Seg2 may be shortened to a value less than IPT, which does not affect bus timing.

### 4.10.6 Calculation of the Bit Timing Parameters

Usually, the calculation of the bit timing configuration starts with a desired bit rate or bit time. The resulting bit time (1/bit rate) must be an integer multiple of the system clock period.

The bit time may consist of 4 to 25 time quanta, the length of the time quantum \( t_q \) is defined by the Baud Rate Prescaler with \( t_q = (\text{Baud Rate Prescaler})/f_{\text{sys}} \). Several combinations may lead to the desired bit time, allowing iterations of the following steps.

First part of the bit time to be defined is the Prop_Seg. Its length depends on the delay times measured in the system. A maximum bus length as well as a maximum node delay has to be defined for expandible CAN bus systems. The resulting time for Prop_Seg is converted into time quanta (rounded up to the nearest integer multiple of \( t_q \)).

The Sync_Seg is 1 \( t_q \) long (fixed), leaving (bit time – Prop_Seg – 1) \( t_q \) for the two Phase Buffer Segments. If the number of remaining \( t_q \) is even, the Phase Buffer Segments have the same length, \( \text{Phase}_\text{Seg2} = \text{Phase}_\text{Seg1} \), else \( \text{Phase}_\text{Seg2} = \text{Phase}_\text{Seg1} + 1 \).

The minimum nominal length of Phase_Seg2 has to be regarded as well. Phase_Seg2 may not be shorter than the CAN controller's Information Processing Time, which is, depending on the actual implementation, in the range of \([0..2] t_q \).

The length of the Synchronisation Jump Width is set to its maximum value, which is the minimum of 4 and Phase_Seg1.

The oscillator tolerance range necessary for the resulting configuration is calculated by the formulas given in section 4.10.4

If more than one configuration is possible, that configuration allowing the highest oscillator tolerance range should be chosen.

CAN nodes with different system clocks require different configurations to come to the same bit rate. The calculation of the propagation time in the CAN network, based on the nodes with the longest delay times, is done once for the whole network.

The CAN system's oscillator tolerance range is limited by that node with the lowest tolerance range.
The calculation may show that bus length or bit rate have to be decreased or that the oscillator frequencies’ stability has to be increased in order to find a protocol compliant configuration of the CAN bit timing.

The resulting configuration is written into the Bit Timing Register:

\[(\text{Phase\_Seg2-1})&(\text{Phase\_Seg1+Prop\_Seg-1})&(\text{SynchronisationJumpWidth-1})&(\text{Prescaler-1})\]

### 4.10.6.1 Example for Bit Timing at high Baudrate

In this example, the frequency of CAN\_CLK is 10 MHz, BRP is 0, the bit rate is 1 MBit/s.

- \[t_q\] = 100 ns = \[t_{\text{CAN\_CLK}}\]
- delay of bus driver = 50 ns
- delay of receiver circuit = 30 ns
- delay of bus line (40m) = 220 ns
- \[t_{\text{Prop}}\] = 600 ns = 6 \cdot \[t_q\]
- \[t_{\text{SJW}}\] = 100 ns = 1 \cdot \[t_q\]
- \[t_{\text{TSeg1}}\] = 700 ns = \[t_{\text{Prop}}\] + \[t_{\text{SJW}}\]
- \[t_{\text{TSeg2}}\] = 200 ns = Information Processing Time + 1 \cdot \[t_q\]
- \[t_{\text{Sync-Seg}}\] = 100 ns = 1 \cdot \[t_q\]
- bit time = 1000 ns = \[t_{\text{Sync-Seg}}\] + \[t_{\text{TSeg1}}\] + \[t_{\text{TSeg2}}\]

\[\text{tolerance for CAN\_CLK} = \frac{0.39\%}{2 \times (13 \times \text{bit time} - \text{PB2})} = \frac{0.1\mu s}{2 \times (13 \times 1\mu s - 0.2\mu s)}\]

In this example, the concatenated bit time parameters are (2-1)\_3&(7-1)\_4&(1-1)\_2&(1-1)\_6, the Bit Timing Register is programmed to = 0x1600.

### 4.10.6.2 Example for Bit Timing at low Baudrate

In this example, the frequency of CAN\_CLK is 2 MHz, BRP is 1, the bit rate is 100 KBit/s.

- \[t_q\] = 1 \mu s = 2 \cdot \[t_{\text{CAN\_CLK}}\]
- delay of bus driver = 200 ns
- delay of receiver circuit = 80 ns
- delay of bus line (40m) = 220 ns
- \[t_{\text{Prop}}\] = 1 \mu s = 1 \cdot \[t_q\]
- \[t_{\text{SJW}}\] = 4 \mu s = 4 \cdot \[t_q\]
- \[t_{\text{TSeg1}}\] = 5 \mu s = \[t_{\text{Prop}}\] + \[t_{\text{SJW}}\]
- \[t_{\text{TSeg2}}\] = 4 \mu s = Information Processing Time + 3 \cdot \[t_q\]
- \[t_{\text{Sync-Seg}}\] = 1 \mu s = 1 \cdot \[t_q\]
- bit time = 10 \mu s = \[t_{\text{Sync-Seg}}\] + \[t_{\text{TSeg1}}\] + \[t_{\text{TSeg2}}\]

\[\text{tolerance for CAN\_CLK} = \frac{1.58\%}{2 \times (13 \times \text{bit time} - \text{PB2})} = \frac{4\mu s}{2 \times (13 \times 10\mu s - 4\mu s)}\]

In this example, the concatenated bit time parameters are (4-1)\_3&(5-1)\_4&(4-1)\_2&(2-1)\_6, the Bit Timing Register is programmed to = 0x34C1.
5. CPU Interface

The interface of the C_CAN module consist of two parts (see figure 17). The Generic Interface which is a fix part of the C_CAN module and the Customer Interface which can be adapted to the customers requirements.

![Diagram of the module interface](image)

Figure 17: Structure of the module interface

5.1 Customer Interface

The purpose of the Customer Interface is to adapt the timings of the module-external signals to the timing requirements of the module and to buffer and drive the external signals. Number and names of the module pins depend on the Customer Interface used with the actual implementation.

The Customer Interface also supplies the clock and reset signals for the module.

The minimum clock frequency required to operate the C_CAN module with a bit rate of 1 MBit/s is 8 MHz. The maximum clock frequency is dependent on synthesis constraints and on the technology which is used for synthesis. The read / write timing of the C_CAN module depends on the Customer Interface used with the actual implementation.

Up to now three different Customer Interfaces are available for the C_CAN module. An 8-bit interface for the Motorola HC08 controller and two 16-bit interfaces to the AMBA APB bus from
5.2 Timing of the WAIT output signal

Figure 18 shows the timing at the modules WAIT output pin CAN_WAIT_B with respect to the modules internal clock CAN_CLK. The number of clock cycles needed for a transfer between the IFx Registers and the Message RAM can vary between 3 and 6 clock cycles depending on the state of the Message Handler (idle, scan Message RAM, load/store shift register, ...).

![Figure 18: Timing of WAIT output signal CAN_WAIT_B](image)

5.3 Interrupt Timing

Figure 19 shows the timing at the modules interrupt pin CAN_INT (active low) with respect to the modules internal clock CAN_CLK.

![Figure 19: Timing of interrupt signal CAN_INT](image)
6. Appendix

6.1 List of Figures

Figure 1: Block Diagram of the C_CAN .......................................................... 8
Figure 2: CAN Core in Silent Mode ................................................................. 10
Figure 3: CAN Core in Loop Back Mode ......................................................... 11
Figure 4: CAN Core in Loop Back combined with Silent Mode ....................... 11
Figure 5: C_CAN Register Summary ............................................................... 13
Figure 6: IF1 and IF2 Message Interface Register Sets ................................. 18
Figure 7: Structure of a Message Object in the Message Memory .................. 22
Figure 8: Data Transfer between IFx Registers and Message RAM ............... 28
Figure 9: Initialisation of a Transmit Object ............................................... 30
Figure 10: Initialisation of a Receive Object ............................................... 31
Figure 11: CPU Handling of a FIFO Buffer ............................................... 33
Figure 12: Bit Timing ...................................................................................... 35
Figure 13: The Propagation Time Segment .................................................. 36
Figure 14: Synchronisation on “late” and “early” Edges .................................. 38
Figure 15: Filtering of Short Dominant Spikes ............................................. 39
Figure 16: Structure of the CAN Core’s CAN Protocol Controller .................. 40
Figure 17: Structure of the module interface .............................................. 43
Figure 18: Timing of WAIT output signal CAN_WAIT_B .............................. 44
Figure 19: Timing of interrupt signal CAN_INT ............................................ 44

EOF