FR FAMILY
32-BIT MICROCONTROLLER
MB91460

C_CAN-CONTROLLER
APPLICATION NOTE
Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-06-27</td>
<td>V1.0, Initial draft, MPI</td>
</tr>
</tbody>
</table>

This document contains 39 pages.
Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for all products delivered free of charge (e.g. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC’s etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment.**

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH’s entire liability and the customer’s exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH’s sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer’s use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH’s and its suppliers’ liability is restricted to intention and gross negligence.

**NO LIABILITY FOR CONSEQUENTIAL DAMAGES**

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.
Contents

REVISION HISTORY ............................................................................................................ 2
WARRANTY AND DISCLAIMER ......................................................................................... 3
CONTENTS .......................................................................................................................... 4
1 INTRODUCTION .............................................................................................................. 6
  1.1 Feature List of C_CAN Controller ............................................................................ 6
2 PREREQUISITE FOR USING THE C_CAN CONTROLLER ......................................... 7
  2.1 Clock Generation and Supply for C_CAN Controller ................................................ 7
  2.2 CAN functionality of GPIO Ports .......................................................................... 7
  2.3 Interface to the CAN Bus ...................................................................................... 7
3 C_CAN CHANNEL BASE ADDRESSES ....................................................................... 9
4 CONFIGURATION OF C_CAN CHANNEL ..................................................................... 10
  4.1 C_CAN Channel Baud Rate .................................................................................. 10
  4.2 Special Modes of C_CAN Channel ........................................................................ 12
  4.2.1 Disable Automatic Re-transmission ................................................................ 12
  4.2.2 Test Modes ...................................................................................................... 12
  4.2.2.1 Silent Mode ............................................................................................... 12
  4.2.2.2 Loop Back Mode ..................................................................................... 13
  4.2.2.3 Loop Back with Silent Mode ..................................................................... 13
  4.2.2.4 Basic Mode ............................................................................................. 14
5 CONFIGURATION OF MESSAGE OBJECTS (MESSAGE BUFFERS) ....................... 15
  5.1 Arbitration (Message ID) ..................................................................................... 16
  5.2 ID Mask ............................................................................................................... 18
6 OPERATION OF C_CAN CHANNEL .............................................................................. 21
  6.1 Basic Mode .......................................................................................................... 21
  6.1.1 Transmission of CAN Data Frames ................................................................ 21
  6.1.1.1 Setting Transmission Requests ................................................................ 21
  6.1.2 Reception of CAN Data Frames ...................................................................... 22
  6.2 Normal Mode ........................................................................................................ 23
  6.2.1 Transmission of CAN Data Frames ................................................................ 23
  6.2.1.1 Setting Transmission Requests ................................................................ 23
  6.2.1.2 Setting Transmission Requests with TXRQST and NEWDAT ................ 25
  6.2.1.3 Clearing Transmission Requests ................................................................ 25
  6.2.2 Reception of CAN Data Frames ...................................................................... 26
<table>
<thead>
<tr>
<th></th>
<th>6.2.2.1</th>
<th>Configuring Message Object to Receive CAN Data Frames</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.2.2.2</td>
<td>Receiving CAN Data Frames</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>6.2.3</td>
<td>Reception of CAN Data Frames using FIFO Buffer</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>6.2.3.1</td>
<td>Configuring Message Objects to Receive CAN Data Frames</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>6.2.3.2</td>
<td>Receiving CAN Data Frames</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>6.2.4</td>
<td>Transmission of CAN Remote Frames</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>6.2.5</td>
<td>Reception of CAN Remote Frames</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Interrupts</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>6.3.1</td>
<td>Transmission Interrupts</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>6.3.1.1</td>
<td>Setting Transmission Requests</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>6.3.1.2</td>
<td>Transmit Interrupt Handler</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>6.3.2</td>
<td>Receive Interrupts</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>6.3.2.1</td>
<td>Configuring Receive Object</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>6.3.2.2</td>
<td>Receive Interrupt Handler</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>6.3.3</td>
<td>Status Change Interrupts</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>6.3.3.1</td>
<td>Configuration</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>6.3.3.2</td>
<td>Status Interrupt Handler</td>
<td>37</td>
</tr>
<tr>
<td>7</td>
<td>APPENDIX</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>7.1</td>
<td>Figures</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>ADDITIONAL INFORMATION</td>
<td></td>
<td>39</td>
</tr>
</tbody>
</table>
1 Introduction

This application note describes the features and the programming of the CAN Controller implemented on the FR Family 32-bit MCUs. This CAN controller is the so-called C_CAN that uses the Bosch IP.

1.1 Feature List of C_CAN Controller

Following are the features of the CAN Controller:
- Supports CAN protocol version 2.0 part A and B
- Bit rates up to 1 MBits/s
- Up to 128 Message Objects
- Each Message Object has its own identifier mask
- Programmable FIFO mode (concatenation of Message Objects)
- Maskable interrupt
- Disabled Automatic Re-transmission Mode for Time Triggered CAN applications
- Programmable Loop Back Mode for self-test operation

For a comparison of the features of the CAN Controller implemented for example on the 16LX Family MCUs called F_CAN to the features of the new CAN Controller (called C_CAN) please refer to Table 1-1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>F_CAN</th>
<th>C_CAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support CAN protocol version 2.0 part A and B</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports Remote Frame</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Maximum Baud Rate</td>
<td>1 MBit/s (when CAN clock is 16 MHz)</td>
<td>1 Mbit/s</td>
</tr>
<tr>
<td>Number of Message Objects</td>
<td>16</td>
<td>32 (up to 128)</td>
</tr>
<tr>
<td>Multi-Level Buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mask Filtering</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Full Bit Comparison</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Full Bit Mask</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Partial Bit Mask</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Masks</td>
<td>2 Masks for all Message Objects</td>
<td>1 Mask for each Message Object</td>
</tr>
<tr>
<td>Disabled Automatic Re-transmission Mode</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Loop Back Mode for self-test</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1-1: Comparison of features supported by F_CAN Controller (MB90340 Series) and C_CAN Controller (FR Family MCUs such as MB91460)
2 Prerequisite for using the C_CAN Controller

The FR Family MCUs can be used for a wide range of applications including CAN applications. To use a CAN controller of the MCU, some configuration needs to be taken care of, depending on the requirements of the application. This configuration is discussed herein.

2.1 Clock Generation and Supply for C_CAN Controller

The CAN module has its own clock prescaler which lets the CAN clock be selected as one among CLKB, CLKMAIN or CLKVCO and also lets the one of these mentioned clock divided by a ratio between 1 to 16. Other than the clock selection the CAN prescaler also provides the register CANCKD using which each of the CAN controller among 0 to 5 can be selectively enabled / disabled

If the CLKCAN source is set core clock CLKB then the clock for the CAN is also modulated (if the clock modulator is enabled).

2.2 CAN functionality of GPIO Ports

Since most of the pins of the MCU are shared to several IO functionalities one has to enable the CAN IO function for the corresponding port pins.

The Receive Pin RX is enabled by setting the corresponding PFR register bit. The Transmit Pin TX is enabled by setting the corresponding PFR register bit.

2.3 Interface to the CAN Bus

The Figure 2-1 shows the example of the Differential High-Speed CAN bus. Such arrangement may provide the speed up to 1 MBits/s. The termination resistor of 120 Ω is located at each end of the bus to minimize reflections and ringing on the waveforms. The logic states of the bits are determined by the differential voltage between the CAN_H and CAN_L signals. In most systems, the recessive state represents logic ‘1’, while logic ‘0’ is provided by the dominant state.

![Figure 2-1: High-Speed CAN Bus](image-url)
C_CAN controller can be interfaced to such CAN bus via a transceiver. Transceiver provides the ability to receive and transmit the messages over the bus. Figure 2-2 shows interfacing of MB91467 microcontroller to Transceiver PCA82C250T. The CAN_H and CAN_L outputs of the transceiver would be connected to corresponding signals of the CAN bus.

![Figure 2-2: MB91467 Interfacing to Transceiver PCA82C250T](image)
3 C_CAN Channel Base Addresses

The base addresses of the C_CAN Channels on FR Family MCUs are:

- **CAN0**: 0x00C000
- **CAN1**: 0x00C100
- **CAN2**: 0x00C200
- **CAN3**: 0x00C300
- **CAN4**: 0x00C400
- **CAN5**: 0x00C500

For details on the available CAN channels on a given FR Family MCU, please refer the datasheet of the MCU.
4 Configuration of C_CAN Channel

This Chapter describes how to configure a C_CAN channel. This includes setting the CAN baud rate corresponding to your CAN network and enabling special function modes (if needed).

4.1 C_CAN Channel Baud Rate

Configuration of baud rate using the Bit Timing Register \textit{BTRn} (also BR Extension Register \textit{BRPERn} in some cases) is an integral part CAN initialization.

C_CAN supports a wide range of bit rate from 1kBPS to 1MBPS. Each CAN node available on a FR series of microcontroller has an independent baud rate generator which can be tuned to get the desired bit or baud rate from CAN clock (i.e. CLKP).

According to the CAN specification, the bit time is divided into the following four segments:
- Synchronization Segment (Sync_Seg)
- Propagation Time Segment (Prop_Seg)
- Phase Buffer Segment 1 (Phase_Seg1)
- Phase Buffer Segment 2 (Phase_Seg2)

The following figure shows CAN Bit Time and the segments:

![CAN Bit Timing](image)

Each segment consists of a specific, programmable number of time quanta. 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 (i.e. CLKCAN) and the Baud Rate Prescaler (BRP):

\[
t_q = \frac{BRP}{f_{sys}}
\]

Baud Rate Prescaler Extension (BRPE) may also need to be considered if the lower baud rate is required at higher CLKCAN clock values.

The following table discusses various parameter related to one CAN Bit and bits of \textit{BTRn} register related to them:
### Table 4-1: Parameters of CAN Bit Time

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync_Seg</td>
<td>1 t₀</td>
<td>This segment is always fixed.</td>
</tr>
<tr>
<td>Prop_Seg</td>
<td>[1..8] tᵣ</td>
<td>These two segments together can be configured using TSEG1 bits (bits 8-11) of BTRn register as below: Prop_Seg + Phase_Seg1 = TSEG1 Value + 1</td>
</tr>
<tr>
<td>Phase_Seg1</td>
<td>[1..8] tᵣ</td>
<td></td>
</tr>
<tr>
<td>Phase_Seg2</td>
<td>[1..8] tᵣ</td>
<td>This segment can be configured using TSEG2 bits (bits 12-14) of BTRn register as below: Phase_Seg2 = TSEG2 Value + 1</td>
</tr>
<tr>
<td>SJW</td>
<td>[1..4] tᵣ</td>
<td>This can be configured using SJW bits (bits 7-6) of BTRn register as below: SJW = SJW Value + 1</td>
</tr>
</tbody>
</table>

**Example 1:** To have a Bit rate of 500Kbps with a frequency of fᵦ = 24 MHz it is possible to set: BRP = 5, TSEG1 = 3, TSEG2 = 2, then the BTRn value would be 0x2305.
4.2 Special Modes of C_CAN Channel

4.2.1 Disable Automatic Re-transmission
If there is any error while the transmission or in case of lost arbitration, the CAN has a provision of automatic re-transmission of such frames. By default this automatic re-transmission of frames is enabled. It can be disabled to enable the CAN to work within a Time Triggered CAN environment. The Disabled Automatic Retransmission mode is enabled by programming bit DAR in the CAN Control RegisterCTRLRn to "1". In this mode the behavior of TXRQST and NEWDAT bits are different than the normal one.

4.2.2 Test Modes

4.2.2.1 Silent Mode
In the silent mode, CAN node is not able to transmit any message frames however it can receive message frames such as Data Frame and Remote Frames. In this mode it can’t send any dominant bits on the bus (such as ACK bit).

Hence if there are two nodes on the CAN bus: one is in normal mode (Node 1) and the other being in Silent mode (Node 2) and when the Node 1 sends a message, Node 2 would receive the message but wont be able to acknowledge it. In such situation, the transmit error counter at the Node 1 would get increased to reflect this error condition. Node 2 wouldn’t encounter any error, since the ACK bit is rerouted internally so that it monitors this dominant bit, although the CAN bus may remain in recessive state.

Figure 4-2: CAN Core in Silent Mode
4.2.2.2 Loop Back Mode

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. In this mode, the message that CAN node transmits, can be monitored on TX pin and hence would be received by the nodes available on the bus. The same message would be received by the same node since CAN Core performs an internal feedback from its TX output to its RX input. The CAN Core ignores acknowledge errors in this mode.

![Figure 4-3: CAN Core in Loop Back Mode](image)

4.2.2.3 Loop Back 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 CAN can be tested without affecting a running CAN system connected to the pins TX and RX. In this mode the RX pin is disconnected from the CAN Core and the TX pin is held recessive.

![Figure 4-4: CAN Core in Loop Back combined with Mode](image)
4.2.2.4 Basic Mode

In the Basic Mode, the C_CAN node runs without using the Message Objects in the RAM. It uses the IF1 registers as Transmit Buffer and uses the IF2 registers as Receive Buffer. Using this mode, one can avoid initializing all the message objects (32 up to 128) in the Message RAM.

The BUSY bit of IF1CREQn registers needs to be used in this mode only and has no meaning in the normal mode (i.e. while using Message RAM).

**Transmission:** After setting all the IF1 registers according to the transmit message requirements, the transmission (of contents of IF1 registers) is requested by setting the BUSY bit in IF1CREQn to 1. After setting the bit to 1, transmission data (contents of IF1 registers) is locked. Once the message gets transmitted (depending upon the condition on the bus) the CAN core clears the BUSY bit to indicate that the transmission is successfully completed. The pending transmission (BUSY = 1) can also aborted by the CPU by clearing the BUSY bit to 0.

**Reception:** In the Basic Mode, the evaluation of all Message Object related control and status bits and also the evaluation of the control bits of the IFx Command Mask Registers IF1CMSKn is turned off. Hence, after the reception of a message the contents of the shift register is stored into the IF2 Registers, without any acceptance filtering.
5 Configuration of Message Objects (Message Buffers)

For efficient handling of CAN messages sent or received by C_CAN channel the message objects (message buffers) provided by C_CAN channel needs to be prepared before starting the C_CAN channel. Nonetheless, it is also possible to change the configuration of the message objects while the C_CAN channel is participating in the CAN communication.

Handling of the message objects from the CPU is significantly different from the way this is done on the former F-CAN controller. In C_CAN controller, to ensure data consistency at all times the CPU has no direct access to the message buffers.

Each C_CAN channel contains two sets of interface registers. These interface registers can be modified/read directly by the CPU. The information which has to be read / written from / to one of the message objects is transferred by these interface registers.

In Figure 5-1 the register structure of a C_CAN channel containing the two sets of interface registers is shown:

In the following sections and sample codes interface registers IF1 or IF2 are used in random occurrence. However, both sets of interface registers behave exactly in the same way and hence can be used interchangeably. A set of interface registers consists of the following parts described next.

![Figure 5-1: Register Structure of C_CAN Channel](image-url)
5.1 Arbitration (Message ID)

The Figure 5-2 shows the arbitration bits those are parts of IFxARB1n and IFxARB2n registers. In the case of transmission, these bits indicates ID of the message to transmitted and in the case of reception, these bits indicates the acceptance filter.

If the Standard Identifier (11-bit) is used, then it is programmed to bits ID28 to ID18. The Standard Identifier is also transmitted in the same order (i.e. from ID28 to ID18 – 11 bit identifier as shown in the Figure 5-3). The 7 most significant bits ID28 to ID22 must not be all 'recessive'. The bits ID17 to ID0 are all ignored.

If the Extended Identifier (29-bit) is used, then it is programmed to bits ID28 to ID0. In the figure Figure 5-4, the 11 bit identifier corresponds to bit ID28 to ID18 and the 18 bit identifier corresponds to ID17 to ID0.

The XTD bit is set to 1, if Extended Identifier is used. For Standard Identifier it is cleared to 0. If the message object is required to be set for transmission, then the DIR bit needs to be set, else for reception it is required to be cleared. For the unused message object MSGVAL bit needs to be cleared and the same can be set if a particular message object is configured completely and should be considered by the message handler.
The following code snippet shows how to configure a given message object to transmit a message with Extended Identifier \texttt{0x1237FFFF}.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*----------------------------------------------------------------------*/
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xF237;  // =>
  // MSGVAL = 1;  // msg buffer valid
  // XTD = 1;    // id = extended (29-bit)
  // DIR = 1;    // direction = transmit
  // ID = 0x1237FFFF;  // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to transmit a message with Standard Identifier \texttt{0x60F}.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*----------------------------------------------------------------------*/
IF1ARB20 = 0xB83C;  // =>
  // MSGVAL = 1;  // msg buffer valid
  // XTD = 0;    // id = standard (11-bit)
  // DIR = 1;    // direction = transmit
  // ID = 0x60F;  // ID = 0x60F
```

The following code snippet shows how to configure a given message object to receive a message with Extended Identifier \texttt{0x1237FFFF}.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*----------------------------------------------------------------------*/
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0x9237;  // =>
  // MSGVAL = 1;  // msg buffer valid
  // XTD = 1;    // id = extended (29-bit)
  // DIR = 0;    // direction = receive
  // ID = 0x1237FFFF;  // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive a message with Standard Identifier \texttt{0x60F}.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*----------------------------------------------------------------------*/
IF1ARB20 = 0x983C;  // =>
  // MSGVAL = 1;  // msg buffer valid
  // XTD = 0;    // id = standard (11-bit)
  // DIR = 0;    // direction = receive
  // ID = 0x60F;  // ID = 0x60F
```
5.2 ID Mask

The mask bits are only relevant to the received messages (data and remote frames).

The figure shows the Mask bits those are parts of IFxMSK1n and IFxMSK2n registers. These bits specify which parts arbitration registers are considered and evaluated. If any of the highlighted bits in the mask registers is 1 then only the corresponding bit of arbitration register is evaluated for the message acceptance filtering, else it would be ignored.

Example: While reception of message/frame with Extended Identifier, if the bit MSK19 of mask register is set to 1 and if the bit ID19 bit of arbitration register is 0, the ID19 of the incoming message should be 0 to pass the acceptance filtering.

![Figure 5-5: Mask Bits of Message Object](image)

The following code snippet shows how to configure a given message object to receive a message with Extended Identifier 0x1237FFFF.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* /*-----------------------------------------------------------(C) Fujitsu Microelectronics Europe GmbH */
/*-------------------------------------------------------------------*/
IF1MSK10 = 0xFFFF;
IF1MSK20 = 0xFFFF;  // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
// use ID28-ID0 for acceptance filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 1;   // id = extended (29-bit)
// DIR = 0;   // direction = receive
// ID = 0x1237FFFF;  // ID = 0x1237FFFF
```

The same can also be done by simply clearing the UMSK bit of IF1MCTR0 register to 0. This way there is no need to configure mask registers at all.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* /*-----------------------------------------------------------(C) Fujitsu Microelectronics Europe GmbH */
/*-------------------------------------------------------------------*/
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 1;   // id = extended (29-bit)
// DIR = 0;   // direction = receive
// ID = 0x1237FFFF;  // ID = 0x1237FFFF
IF1MCTR0 UMSK = 0;  // ignore mask
```
The following code snippet shows how to configure a given message object to receive messages with Extended Identifier ranging from 0x1237FFF0 to 0x1237FFFF.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*--------------------------------------------------------------------------------*/
IF1MSK10 = 0xFFF0;
IF1MSK20 = 0xFFFF;  // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 1;   // id = extended (29-bit)
// DIR = 0;   // direction = receive
// ID = 0x1237FFFF;  // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive messages with Extended Identifier 0x1237FFFF7 and 0x1237FFFF.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*--------------------------------------------------------------------------------*/
IF1MSK10 = 0xFFF7;
IF1MSK20 = 0xFFFF;  // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
// use ID28-ID0 (except ID3) for acceptance filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 1;   // id = extended (29-bit)
// DIR = 0;   // direction = receive
// ID = 0x1237FFFF;  // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive a message with Standard Identifier 0x60F.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*--------------------------------------------------------------------------------*/
IF1MSK20 = 0xFFFC;  // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
// use ID28-ID18 for acceptance filtering
IF1ARB20 = 0x983C;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 0;   // id = standard (11-bit)
// DIR = 0;   // direction = receive
// ID = 0x60F;  // ID = 0x60F
```
The same can also be done by simply clearing the UMSK bit of IF1MCTR0 register to 0. This way there is no need to configure mask registers at all.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*****************************************************************************/
IF1ARB20 = 0x983C; // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
// use ID28-ID12 for acceptance filtering
// MSGVAL = 1; // msg buffer valid
// ID = 0x60F; // ID = 0x60F
IF1MCTR0 UMSK = 0; // ignore mask
```

The following code snippet shows how to configure a given message object to receive a messages with Standard Identifier ranging from 0x600 to 0x60F.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*****************************************************************************/
IF1MSK20 = 0xFFC0; // =>
// MXTD = 1;   // use XTD for acceptance filtering
// MDIR = 1;   // use DIR for acceptance filtering
// use ID28-ID12 for acceptance filtering
IF1ARB20 = 0x983C; // =>
// MSGVAL = 1; // msg buffer valid
// XTD = 0;   // id = standard (11-bit)
// DIR = 0;   // direction = receive
// ID = 0x60F; // ID = 0x60F
```
6 Operation of C_CAN Channel

6.1 Basic Mode

As already discussed, in the Basic Mode the C_CAN node runs without using the Message Objects in the RAM. It uses the IF1 registers as Transmit Buffer and uses the IF2 registers as Receive Buffer. The following example demonstrated the entry to the basic mode.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                             */
/* (C) Fujitsu Microelectronics Europe GmbH                                    */
/*-----------------------------------------------------------------------------*/
COER0_OE = 1;               // enable CAN0 output
PIER10_IE0 = 1;              // enable CAN0 input
CTRLR0_INIT = 1;            // stop CAN Operation
CTRLR0_CCE = 1;             // enable BTR Configuration Change
BTR0 = value;               // set bit timing to desired value
CTRLR0_CCE = 0;             // disable BTR Configuration Change
CTRLR0_TEST = 1;            // enable test mode
TESTRO_BASIC = 1;           // enable basic mode
CTRLR0_INIT = 0;            // start CAN Operation

/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                             */
/* (C) Fujitsu Microelectronics Europe GmbH                                    */
 /*-----------------------------------------------------------------------------*/
IF1CMSK0 = 0x00B2;          // =>
    // WRRD = 1;           // write
    // ARB = 1;            // access arbitration
    // CONTROL = 1;        // access control
    // DATAA = 1           // access data bytes 0-3

IF1ARB20 = 0xA00C;          // =>
    // MSGVAL = 1;        // msg buffer valid
    // DIR = 1;           // direction = transmit
    // ID = 3             // ID = 3

IF1MCTR0 = 0x0182;          // =>
    // TXRQST = 1;        // set transmit request
    // EOB = 1;           // end of buffer = 1
    // DLC = 2;           // data length = 2 bytes
IF1DTA10 = 0xFFAA;          // set data
IF1CREQ0 = 0x8000;         // transmit contents of IF Register 1
```
6.1.2 Reception of CAN Data Frames

In basic mode the evaluation of all message object related control and status bits and evaluation of the control bits of the command mask registers is turned off. The message number (Identifier) of the command request register is not evaluated either. Hence no configuration is required as it would have been required in the normal operation mode. The NEWDAT bit indicates whether a new message frame is arrived. The message related information such as DLC, data bytes and identifier can then be copied to the software buffer. At the end, the NEWDAT bit should be cleared.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
char length;
long int data[2];
long int id;

if (1 == IF2MCTR0_NEWDAT)     // if new message arrived?
{
    length = IF2MCTR0_DLC; // get received data length
data[0] = IF2DTA0;    // get the data bytes 0 - 3
data[1] = IF2DTB0;    // get the data bytes 4 - 7

    if (1 == IF2ARB0_XTD)
    {
        id = (IF2ARB0 & 0x1FFFFFFF);   //Extended ID
    }
    else
    {
        id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL)); // Standard ID
    }

    IF2MCTR0_NEWDAT = 0;  // clear NEWDAT flag
}
```
6.2 Normal Mode

The following example demonstrated the entry to the normal mode.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                */
/* (C) Fujitsu Microelectronics Europe GmbH                        */
/*----------------------------------------------------------------------------*/
COER0_OE = 1;       // enable CAN0 output
PIER0_IE0 = 1;      // enable CAN0 input
CTRLR0_INIT = 1;    // stop CAN Operation

CTRLR0_CCE = 1;     // enable BTR Configuration Change
BTR0 = value;       // set bit timing to desired value
CTRLR0_CCE = 0;     // disable BTR Configuration Change
CTRLR0_INIT = 0;    // start CAN Operation
```

6.2.1 Transmission of CAN Data Frames

Before transmitting a CAN message, the message is constructed in one of the two Interface Register Sets. Then, the message must be transferred to a message object.

6.2.1.1 Setting Transmission Requests

There are two different methods to request the transmission of a message. First, the Interface Register Set for the message can set the transmit request in the Message Control Register as bit IFxMCTRn:TXRQST. If this bit is set and the message is transferred from the Interface Register Set to the Message Buffer, the message will be transferred.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                */
/* (C) Fujitsu Microelectronics Europe GmbH                        */
/*----------------------------------------------------------------------------*/
IF1CMSK0 = 0x00B2;        // =>
// WRRD = 1;   // write
// ARB = 1;   // access arbitration
// CONTROL = 1;   // access control
// DATAA = 1   // access data bytes 0-3

IF1ARB20 = 0xA00C;       // =>
// MSGVAL = 1;         // msg buffer valid
// DIR = 1;            // direction = transmit
// ID = 3              // ID = 3

IF1MCTR0 = 0x0182;       // =>
// TXRQST = 1;       // set transmit request
// EOB = 1;           // end of buffer = 1
// DLC = 2;           // data length = 2 bytes
IF1DTA10 = 0xFFAA;       // set data
IF1CREQ0 = 1;            // transfer IF Register 1 to message object 1
```
Second, a message can be configured without the TXRQST bit set. The command to transfer the message to the Message Buffer is given using the IFxCMSKn register. In this register, the TXREQ bit can be set. If the TXREQ bit is set, a transmission is requested for that message, even if the IFxMCTRn:TXRQST bit is not set.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                           */
/* (C) Fujitsu Microelectronics Europe GmbH                                  */
/*----------------------------------------------------------------------------*/
IF1CMSK0 = 0x00B6;      // =>
  // WRWD = 1;  // write
  // ARB = 1;  // access arbitration
  // CONTROL = 1;  // access control
  // DATAA = 1  // access data bytes 0-3
  // TXREQ = 1  // set TXRQST
IF1ARB20 = 0xA00C;       // =>
  // MSGVAL = 1;         // msg buffer valid
  // DIR = 1;            // direction = transmit
  // ID = 3              // ID = 3
IF1MCTR0 = 0x0082;       // =>
  // EOB = 1;            // end of buffer = 1
  // DLC = 2;            // data length = 2 bytes
IF1DTA10 = 0xFFAA;       // set data
IF1CREQ0 = 1;            // transfer IF Register 1 to message object 1
```
6.2.1.2 Setting Transmission Requests with TXRQST and NEWDAT

When a message needs to be transmitted with just the updated data bytes (and everything else including the arbitration bits and control bits are same), then along with the new data bytes, the TXRQST and NEWDAT flag needs to be set together. This prevents the reset of TXRQST at the end of a transmission that may already be in progress while the data is updated. When NEWDAT is set together with the TXRQST, NEWDAT will be reset as soon as new transmission has started.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                                 */
/* (C) Fujitsu Microelectronics Europe GmbH                                        */
/*--------------------------------------------------------------------------------*/
IF1CMSK0 = 0x00B2; // =>
    // WRRD = 1; // write
    // ARB = 1; // access arbitration
    // CONTROL = 1; // access control
    // DATAA = 1 // access data bytes 0~3
IF1ARB20 = 0xA00C; // =>
    // MSGVAL = 1; // msg buffer valid
    // DIR = 1; // direction = transmit
    // ID = 3 // ID = 3
IF1MCTR0 = 0x0182; // =>
    // TXRQST = 1; // set transmit request
    // EOB = 1; // end of buffer = 1
    // DLC = 2; // data length = 2 bytes
IF1DTA0 = 0xFFAA; // set data
IF1CREQ0 = 1; // transfer IF Register 1 to message buffer 1
...
IF1CMSK0 = 0x0093; // set no transmit request, wr
IF1MCTR0 = 0x8188; // =>
    // NEWDAT = 1; // set new data
    // TXRQST = 1; // set transmit request
    // DLC = 8; // data length = 8 byte
IF1DTA0 = 0xFFFF0000; // set data
IF1DTB0 = 0xAAAA5555; // set data
IF1CREQ0 = 1; // transfer IF Register 1 to message object 1
```

6.2.1.3 Clearing Transmission Requests

The transmission request of a message, that is stored in a message buffer but is not yet transmitted, can be cleared. As with all operations on a message, the message or at least the corresponding bits must be loaded into an Interface Register Set. Then the existing request can be cleared by clearing the IFxMCTRn:TXRQST and storing the message back into the message buffer.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                                 */
/* (C) Fujitsu Microelectronics Europe GmbH                                        */
/*--------------------------------------------------------------------------------*/
IF1CMSK0 = 0x0010; // read control bits
IF1CREQ0 = 0x01; // message object 1
IF1MCTR0_TXRQST = 0; // ... then cancel Transmit
IF1CMSK0 = 0x0090; // write control
IF1CREQ0 = 0x01; // to message object 1
```
6.2.2 Reception of CAN Data Frames

6.2.2.1 Configuring Message Object to Receive CAN Data Frames

Before receiving a message, the Message Object has to be configured appropriately with the command mask, ID mask and arbitration fields.

The command mask register should be configured to write the mask, arbitration and control bits. The mask register configuration decides whether a single message would be received by the Message Object or it would allow a group of messages thru its acceptance filter. Here the acceptance filtering is done in such a way that it would just allow just a single message with. The arbitration register is configured to receive message with Standard Identifier (11-bit) 0x004.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                             */
/* (C) Fujitsu Microelectronics Europe GmbH                                  */
/*---------------------------------------------------------------------------*/
IF2CMSK0 = 0x00F0;         // =>
// WRRD = 1;               // write
// MSK = 1;                // access mask
// ARB = 1;                // access arbitration
// CONTROL = 1;            // access control

IF2MSK20 = 0xFFFC;         // =>
// MXTD = 1;               // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
// use ID28-ID18 for acceptance filtering

IF2ARB20 = 0x8010;         // =>
// MSGVAL = 1;             // msg buffer valid
// XTD = 0;                // id = standard (11-bit)
// DIR = 0;                // direction = receive
// ID = 4;                 // ID = 0x004

IF2MCTR0 = 0x1080;         // =>
// UMASK = 1;              // use mask for acceptance filtering
// EOB = 1;                // end of buffer = 1

IF2CREQ0 = 2;              // transfer IF Register 2 to message object 2
```
The same can also be done by simply clearing the UMSK bit of IF1MCTR0 register to 0. This way there is no need to configure mask registers at all.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
char length;
long int data[2];
long int id;

if (NEWDT10 & 2) {
    // read message-buffer 2
    IF2CMSK0 = 0x00B0; // =>
    // WRRD = 1; // write
    // MSK = 0; // don’t access mask
    // ARB = 1; // access arbitration
    // CONTROL = 1; // access control
    IF2ARB20 = 0x8010; // =>
    // MSGVAL = 1; // msg buffer valid
    // XTD = 0; // id = standard (11-bit)
    // DIR = 0; // direction = receive
    // ID = 4; // ID = 0x004
    IF2MCTR0 = 0x0080; // =>
    // UMASK = 0; // ignore mask
    // EOB = 1; // end of buffer = 1
    IF2CREQ0 = 2; // transfer IF Register 2 to message object 2
    length = IF2MCTR0_DLC; // get received data length
    data[0] = IF2DTA0; // get the data bytes 0-3
    data[1] = IF2DTB0; // get the data bytes 4-7
    id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL)); // Standard ID
}
```
6.2.3 Reception of CAN Data Frames using FIFO Buffer

C_CAN provides the possibility to concatenate two or more message objects to form a FIFO buffer. The identifiers and the masks of these objects have to be programmed to the matching values and the EOB bit of all such message objects (to be concatenated) except the last message object of a FIFO buffer is required to be set to 0. The EOB bit of the last message object of a FIFO buffer is set to 1.

6.2.3.1 Configuring Message Objects to Receive CAN Data Frames

The following snippet of code demonstrates how a FIFO buffer described above is realized. Here 4 message objects from 2 to 5 are concatenated to form a FIFO buffer with the same message id 0x004.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
int cnt;

IF2CMSK0 = 0x00F0;   // =>
// WRRD = 1;         // write
// MSK = 1;          // access mask
// ARB = 1;          // access arbitration
// CONTROL = 1;      // access control

IF2MSK20 = 0xFFFC;   // =>
// MXTD = 1;         // use XTD for acceptance filtering
// MDIR = 1;         // use DIR for acceptance filtering
// use ID28-ID18 for acceptance filtering

IF2ARB20 = 0x8010;   // =>
// MSGVAL = 1;       // msg buffer valid
// XTD = 0;          // id = standard (11-bit)
// DIR = 0;          // direction = receive
// ID = 4;           // ID = 0x004

for (cnt = 2; cnt <= 5; cnt++)
{
    if (cnt < 5)
    {
        IF2MCTR0 = 0x1000;  // =>
        // UMASK = 1;   // use mask for acceptance filtering
        // EOB = 0;     // end of buffer = 0
    }
    else
    {
        IF2MCTR0 = 0x1080;  // =>
        // UMASK = 1;   // use mask for acceptance filtering
        // EOB = 1;     // end of buffer = 1
    }
}

IF2CREQ0 = cnt;   // transfer IF Register 2 to message object 2 to 5
```
6.2.3.2 Receiving CAN Data Frames

The following snippet of code demonstrates how a FIFO buffer data can be read and transferred. Here it is considered that message objects 2 to 5 are concatenated.

```c
typedef struct
{
    char length;
    long int data[2];
    long int id;
} received_data;

int cnt;
received_data RX_data[4];

for (cnt = 2; cnt <= 5; cnt++)
{
    if ((NEWDT10 & (0x01 << (cnt-1)))) // new data arrived?
    {
        // read message-buffer
        IF2CMSK0 = 0x003F;      // =>
        // WRRD = 1      // read
        // MASK = 0     // don’t transfer mask
        // ARB = 1     // transfer arbitration
        // CONTROL = 1    // transfer control
        // CIP = 1     // clear INTPND bit (if RXIE is set)
        // TXREQ = 1     // clear NEWDAT bit
        // DATAA = 1     // transfer data bytes 0-3
        // DATAB = 1     // transfer data bytes 4-7
        IF2CREQ0 = cnt;           // transfer Message Register 2 to IF 2

        RX_data[cnt-2].length = IF2MCTR0_DLC; // get received data length
        RX_data[cnt-2].data[0] = IF2DTA0;       // get the data bytes 0 - 3
        RX_data[cnt-2].data[1] = IF2DTB0;       // get the data bytes 4 - 7

        RX_data[cnt-2].id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL)); // Standard ID
    }
    if (1 == IF2MCTR0_EOB)  // end of buffer reached?
    {
        break;
    }
}
```
6.2.4 Transmission of CAN Remote Frames

CAN Remote frame is different than the data frame from the fact that it doesn’t have data bytes and the RTR bit is set. The following figures shows a typical remote frame (remote transmission request).

![Figure 6-1: CAN Remote Frame](image)

To transmit the remote frame with a given identifier the message object needs to be configured as Receive Object and the TXRQST bit of the message control register needs to be set to 1 in order to transmit the same. Once the bus is found free the CAN controller will transmit the remote transmission request on the bus. If the matching data frame is received by Receive Object before the remote frame could be transmitted, the TXRQST bit is automatically reset.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
IF2CMSK0 = 0x00B0;  // =>
// WRRD = 1;   // write
// MSK = 0;   // don't access mask
// ARB = 1;   // access arbitration
// CONTROL = 1;  // access control
IF2ARB20 = 0x8010;  // =>
// MSGVAL = 1;  // msg buffer valid
// XTD = 0;   // id = standard (11-bit)
// DIR = 0;   // direction = receive
// ID = 4;   // ID = 0x004
IF2MCTR0 = 0x0182;  // =>
// TXRQST = 1;  // set transmit request for remote frame
// UMASK = 0;   // ignore mask
// EOB = 1;   // end of buffer = 1
// DLC = 2;   // requested data length = 2 bytes
IF2CREQ0 = 2;   // transfer IF Register 2 to message object 2
```
6.2.5 Reception of CAN Remote Frames

To receive the remote transmission request with a given identifier the message object needs to be configured as **Transmit Object** and the **RMTEN** bit of the message control register needs to be set to 1.

Once the remote transmission request is transmitted by the **Receive Object** (described in the above section), it would be received by **Transmit Object** since both has the matching identifiers. Since the **RMTEN** bit of message control register is set, the **TXRQST** of this message object is set automatically. That's how the remote frame would be autonomously be answered by the data frame. That means the data frame with the identifier 0x004 with 2 data bytes 0xAA and 0xFF is transmitted on the bus by **Transmit Object**.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                           */
/*                                                                         */
/* (C) Fujitsu Microelectronics Europe GmbH                                  */
/*---------------------------------------------------------------------------*/
IF1CMSK0 = 0x00B2; // =>
    // WRRD = 1;  // write
    // ARB = 1;   // access arbitration
    // CONTROL = 1;  // access control
    // DATAA = 1  // access data bytes 0-3
IF1ARB20 = 0xA010; // =>
    // MSGVAL = 1;  // msg buffer valid
    // DIR = 1;    // direction = transmit
    // ID = 4     // ID = 0x004
IF1MCTR0 = 0x0282; // =>
    // RMTEN = 1;  // remote enable
    // EOB = 1;    // end of buffer = 1
    // DLC = 2;   // data length = 2 bytes
IF1DTA10 = 0xFFAA; // set data
IF1CREQ0 = 1; // transfer IF Register 1 to message object 1
```
6.3 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.

6.3.1 Transmission Interrupts

6.3.1.1 Setting Transmission Requests

For the transmission handling using interrupts, the IE bit of CAN control register needs to be set and also the TXIE bit of message control register needs to be set. The rest of the transmit object initialization remains same.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/*                 (C) Fujitsu Microelectronics Europe GmbH */
/*------------------------------------------------------------*/
CTRLR0_IE = 1;  // module interrupt enable
IF1CMSK0 = 0x00B2;  // =>
    // WRRD = 1;  // write
    // ARB = 1;  // access arbitration
    // CONTROL = 1;  // access control
    // DATAA = 1  // access data bytes 0-3
IF1ARB20 = 0xA00C;  // =>
    // MSGVAL = 1; // msg buffer valid
    // DIR = 1;  // direction = transmit
    // ID = 3   // ID = 3
IF1MCTR0 = 0x0982;  // =>
    // TXIE = 1;  // transmit interrupt enable
    // TXRQST = 1; // set transmit request
    // EOB = 1;  // end of buffer = 1
    // DLC = 2;  // data length = 2 bytes
IF1DTA10 = 0xFFAA;  // set data
IF1CREQ0 = 1;  // transfer IF Register 1 to message object 1
```
6.3.1.2 Transmit Interrupt Handler

Transmission handler part of the C_CAN ISR need to be determine the cause of the interrupt first. This is done with the help of interrupt, interrupt pending, message valid, new data and transmission request register as shown below. It should also clear the interrupt pending bit using the command mask register.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*--------------------------------------------------------------------------------*/
unsigned int IntPointer = 0x0000;
unsigned long IntBuffer;

IntPointer = INT0;
IntPointer = IntPointer & 0x00FF; // use only the lower six bits
if( (IntPointer >= 1) && (IntPointer <= 0x20) ) // valid buffer number
{
    IntBuffer = 0x01 << (IntPointer-1);
    if( (MSGVAL0 & IntBuffer) != 0 && (INTPND0 & IntBuffer) != 0) // message buffer is valid
        if( (NEWDT0 & IntBuffer) != 0 ) // is a receive interrupt
            IF1CMSK0 = 0x0018; // =>
            // WRRD = 0 // read
            // CONTROL = 1 // transfer control
            // CIP = 1 // clear INTPND bit
            IF1CREQ0 = IntPointer; // start transfer
        else if ( (TREQR0 & IntBuffer) == 0 ) // is transmission done
            IF1CREQ0 = IntPointer; // start transfer
}
```
6.3.2 Receive Interrupts

6.3.2.1 Configuring Receive Object

For the reception handling using interrupts, the IE bit of CAN control register needs to be set and also the RXIE bit of message control register needs to be set. The rest of the receive object initialization remains same.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
CTRLR0_IE = 1;  // module interrupt enable
IF2CMSK0 = 0x00B0;  // =>
  // WRRD = 1;  // write
  // MSK = 0;   // don’t access mask
  // ARB = 1;   // access arbitration
  // CONTROL = 1;  // access control
IF2ARB20 = 0x8010;  // =>
  // MSGVAL = 1;  // msg buffer valid
  // XTD = 0;   // id = standard (11-bit)
  // DIR = 0;   // direction = receive
  // ID = 4;   // ID = 0x004
IF2MCTR0 = 0x0480;  // =>
  // RXIE = 1;   // receive interrupt enable
  // UMASK = 0;   // ignore mask
  // EOB = 1;   // end of buffer = 1
IF2CREQ0 = 2;   // transfer IF Register 2 to message object 2
```
6.3.2.2 Receive Interrupt Handler

Reception handler part of the C_CAN ISR need to be determine the cause of the interrupt first. This is done with the help of interrupt, interrupt pending, message valid and new data register as shown below. It should also clear the interrupt pending bit and new data bit using the command mask register.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*---------------------------------------------------------------------------*/
unsigned int IntPointer = 0x0000;
unsigned long IntBuffer;

IntPointer = INTR0;
IntPointer = IntPointer & 0x00FF; // use only the lower six bits
if((IntPointer >= 1) && (IntPointer <= 0x20) ) // valid buffer number
{
    IntBuffer = 0x01 << (IntPointer-1);
    // Check whether the interrupt source is a valid buffer
    if(((MSGVAL0 & IntBuffer) != 0) && ((INTPND0 & IntBuffer) != 0)))
    // message buffer is valid
    {
        // Check whether the interrupt cause is receive or transmit
        if( (NEWDT0 & IntBuffer) != 0 ) // is a receive interrupt
            {
                IF2CMSK0 = 0x003F; // =>
                // WRRD = 0 // read
                // MASK = 0 // don’t transfer mask
                // ARB = 1 // transfer arbitration
                // CONTROL = 1 // transfer control
                // CIP = 1 // clear INTPND bit
                // TXREQ = 1 // clear NEWDAT bit
                // DATAA = 1 // transfer data bytes 0-3
                // DATAB = 1 // transfer data bytes 4-7
                IF2CREQ0 = IntPointer; // start transfer
            }
    }
}
```
6.3.3 Status Change Interrupts

6.3.3.1 Configuration

For the status change handling using interrupts, the IE, SIE and EIE bit of CAN control register needs to be set.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES. */
/* (C) Fujitsu Microelectronics Europe GmbH */
/*--------------------------------------------------------------------------------*/
CTRLR0_IE = 1;       // module interrupt enable
CTRLR0_SIE = 1;      // status change interrupt enable
CTRLR0_EIE = 1;      // error interrupt enable
```
6.3.3.2 Status Interrupt Handler

A Status Interrupt is generated by bits BOFF and EWARN (Error Interrupt) or by RXOK, TXOK, and LEC (Status Change Interrupt) assuming that the corresponding enable bits in the CAN Control Register are set. In case of Status Interrupt the Interrupt register INTRn reflects a value of 0x8000. Reading the Status Register STATRn will clear the Status Interrupt value (0x8000) in the Interrupt Register. Then the value of the status register determines the cause of status interrupt and then the corresponding action can be taken. The following is an example of Status handler part of the C_CAN ISR.

```c
/* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
/* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
/* ELIGIBILITY FOR ANY PURPOSES.                                          */
/*                                                                         */
/*---------------------------------------------------------------------------*/
unsigned short IntPointer = 0x0000;
unsigned char canstatus;

IntPointer = INTR0;
if( (IntPointer & 0x8000) == 0x8000 )   // Status Interrupt??
{
    // Read the Status Register
    // This will also clear pending Status/Error Interrupt
    canstatus = STATR0;
    if( (canstatus & 0x80) == 0x80 ) // BusOff??
        { // Busoff handling
        ...
    }
    if( (canstatus & 0x40) == 0x40 ) // Error Warning??
        { // Error Warning handling
        ...
    }
    if( (canstatus & 0x20) == 0x20 ) // Error Passive??
        { // Error Passive handling
        ...
    }
    if( (canstatus & 0x10) == 0x10 ) // RXOK??
        { // Clear RXOK Flag in Status register
        STATR0_RXOK = 0;
        ...
    }
    if( (canstatus & 0x08) == 0x08 ) // TXOK??
        { // Clear TXOK Flag in Status register
        STATR0_TXOK = 0;
        ...
    }
    if( (canstatus & 0x07) != 0x00 ) // LEC??
        { // Clear LEC in Status register
        STATR0_LEC = 0;
        ...
    }
}
```
7 Appendix

7.1 Figures

Figure 2-1: High-Speed CAN Bus ................................................................. 7
Figure 2-2: MB91467 Interfacing to Transceiver PCA82C250T ...................... 8
Figure 4-1: CAN Bit Timing ............................................................................. 10
Figure 4-2: CAN Core in Silent Mode .............................................................. 12
Figure 4-3: CAN Core in Loop Back Mode ...................................................... 13
Figure 4-4: CAN Core in Loop Back combined with Mode ......................... 13
Figure 5-1: Register Structure of C_CAN Channel ......................................... 15
Figure 5-2: Arbitration Bits of Message Object ............................................. 16
Figure 5-3: CAN Data Frame with Standard Identifier ............................... 16
Figure 5-4: CAN Data Frame with Extended Identifier ............................... 16
Figure 5-5: Mask Bits of Message Object ................................................... 18
Figure 6-1: CAN Remote Frame ................................................................. 30
8 Additional Information

Information about FUJITSU Microcontrollers can be found on the following Internet page:
http://mcu.emea.fujitsu.com/

The software example related to this application note is:
91460_can
91460_can_isrs
91460_can_uart_terminal

It can be found on the following Internet page:
http://mcu.emea.fujitsu.com/mcu_product/mcu_all_software.htm