APPLICATION NOTE

P8xC591 Microcontroller in CAN Applications

AN00043

Philips
Semiconductors

PHILIPS

Abstract

The P8xC591 is an advanced 8-bit CAN microcontroller for use in general industrial and automotive applications. The CAN controller fully supports the international standard for Controller area network data link layer and medium access control (ISO11898). CAN is a serial bus protocol being primarily intended for transmission of control related data between a number of bus nodes.

This application note provides information how to use the P8xC591 in CAN applications.

APPLICATION NOTE

P8xC591 Microcontroller in CAN Applications

AN00043

Author(s):
Hartmut Habben, Peter Hank

Philips Semiconductors
Systems Laboratory Hamburg
Germany

Keywords

P8xC591, CAN Controller, PeliCAN, CAN 2.0B
Controller Area Network (CAN)

Number of pages : 45

Date: 2000-08-18

Summary

This Application Note covers the CAN related items of the P8xC591. It is assumed the reader is familiar with the P8xC591 data sheet [1] and the use of Controller Area Network (CAN) [2] as specified in ISO11898. Therefore, the discussion will not enter into any great detail on either the CAN specification or the P8xC591 data sheet.

The Application Note describes in more detail complex items of the data sheet and gives the user valuable hints for CAN applications. Several flow diagrams and program examples could be used as a starting point for the development of application software.

Most common tasks like initialisation, reception and transmission of CAN messages as well as more complex functions like automatic bit-rate detection, Higher Layer Protocol support and acceptance filter 'change on the fly' are illustrated. The 'C' code examples represent the minimum requirements needed to accomplish each task.

Contents

1. INTRODUCTION

The P8xC591 is an advanced CAN microcontroller for use in general industrial and automotive applications. In addition to the enhanced functionality of the Philips "Rx+ core" [5] this device provides a number of dedicated hardware functions for these applications.

The CAN controller of the P8xC591 fulfils the complete CAN2.0B specification and provides a direct software migration path from the SJA1000 stand-alone CAN controller [6]. With a superset of CAN features like an enhanced acceptance filter, support for System Maintenance, Diagnostics, System Optimisation and a Receive FIFO characteristics it is intended to be used in versatile application areas.

This Application Note covers the CAN related items of P8xC591 applications. To apply and understand the application examples given in this document, the reader should be familiar with the Philips P8xC591 data sheet [1]. The 'C' code in the examples describe a basic set of software driver routines which could be used as a starting point for development of application software. All used definitions are listed in the Appendix.

2. OVERVIEW

2.1 System Overview

The P8xC591 is designed to work with a minimum number of external components. Figure 2-1 shows the circuitry of a CAN node using the ROM- or OTP EPROM- version of the P8xC591. The only additional components that are required are a crystal plus two capacitors to drive the on-chip oscillator, a capacitor connected to the Reset pin, using the on chip power-on RESET circuitry and a transceiver to connect the P8xC591 to the CAN bus.

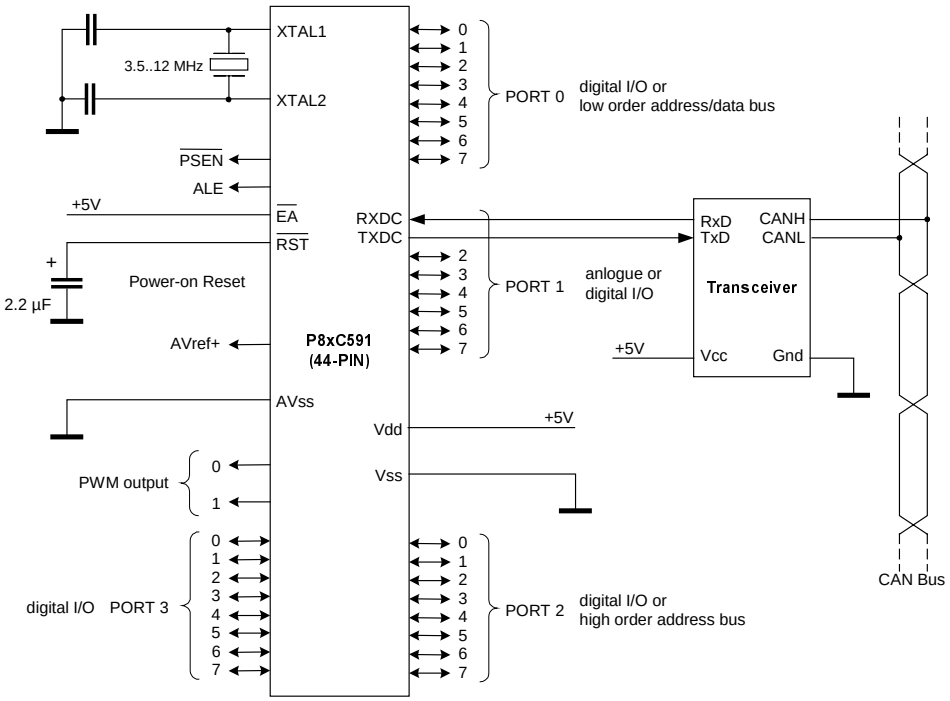

Figure 2-1: Typical P8xC591 CAN Application

2.2 Block Diagram

Figure 2-2 shows the block diagram of the P83C591 (ROM) or the P87C591 (OTP EPROM) versions.

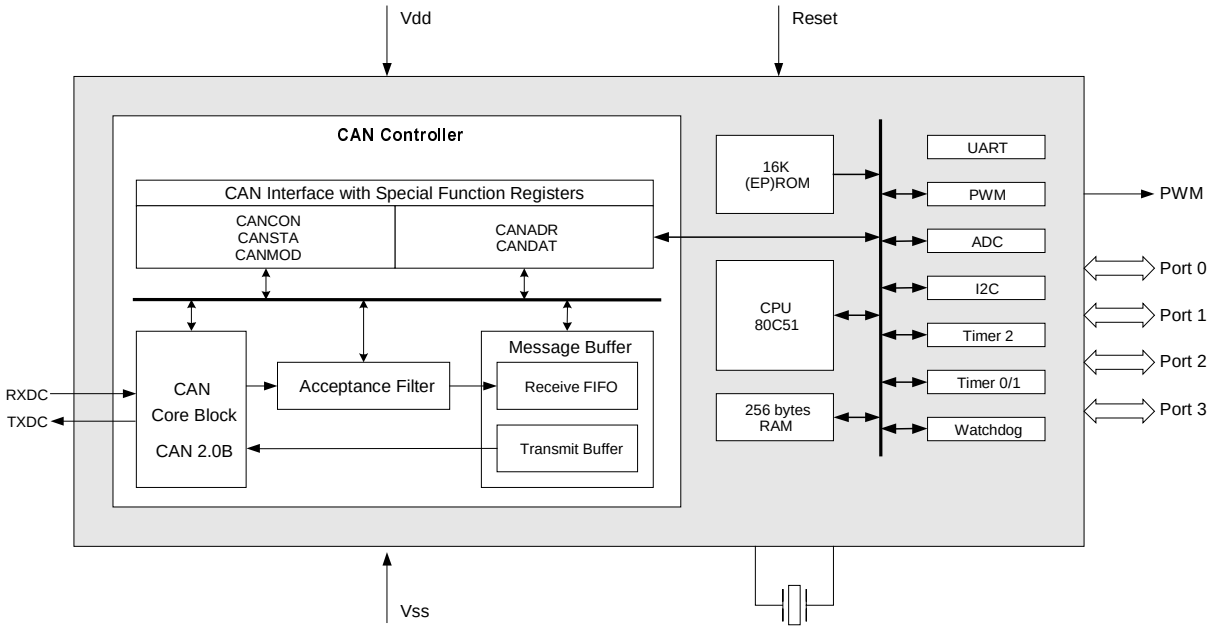

Figure 2-2: Block Diagram of the P8xC591

In addition to a standard set of peripherals the P8xC591 contains a powerful CAN controller block which is also known as PeliCAN. This embedded CAN controller includes the following functional blocks:

The **CAN Core Block** controls the transmission and reception of CAN frames according to the CAN specification CAN2.0B.

The **CAN Interface** consist of 5 Special Function Registers which perform the link between CPU and CAN controller. Access to important CAN registers is accomplished by a fast autoincrement addressing feature and bit-addressable Special Function Registers.

The **Transmit Buffer** of the CAN controller is able to store one complete CAN message (Extended or Standard Frame Format). Whenever a transmission is initiated by the CPU, message bytes are transferred from the Transmit Buffer into the CAN Core Block.

When receiving a message the CAN Core Block converts the serial bit stream into parallel data for the **Acceptance Filter**. With this programmable filter the P8xC591 decides which messages actually are received.

All received messages accepted by the Acceptance Filter are stored within the **Receive FIFO**. Depending on the mode of operation and the data length up to 21 CAN messages can be stored. This enables the user to be more flexible when specifying interrupt services and interrupt priorities for the system because the probability of data overrun conditions is reduced extremely.

2.3 Description of CAN Features

The CAN 2.0B active CAN controller supports 11-bit standard and 29-bit extended identifiers. A maximum CAN bit-rate of 1Mbit/s is already achievable with an 8 MHz clock. An on-chip 64-byte Receive FIFO and a 13-byte transmit buffer is implemented.
In addition to the general CAN features the P8xC591 provides enhanced PeliCAN, System Maintenance, Diagnostics and Optimisation Features known from the SJA1000 stand-alone CAN controller [6] from Philips Semiconductors.

PeliCAN Features:

- Four independently configurable Acceptance Filter Banks
- Four possible Acceptance Filter Configurations in each Bank
- Each filter has two 32-bit specifiers: a 32-bit Code and a 32-bit Mask
- All filters are changeable 'on the fly'
- Acceptance Filter with Higher Layer Protocol Support
- Receive FIFO characteristic
- Listen Only and Self Test Mode
- Receive Interrupt only if FIFO Receive Interrupt Level is reached
- Receive Interrupt immediately at reception of High Priority Data Frames

System Maintenance, Diagnostics and Optimisation Features:

Arbitration Lost Capture	▪ Interrupt on arbitration lost ▪ Detailed CAN bit position of last arbitration lost event is captured
Advanced Error Diagnostics	▪ Error Code Capture with detailed bit position and type of error ▪ Readable Error Counters ▪ Several different Error Interrupts ▪ Programmable error warning limit
Listen Only Mode	▪ Monitor Function ▪ Automatic Bit-rate Detection
CAN Self Test Mode	▪ System Self Tests ▪ Reception of own messages ▪ Global Self Test (acknowledge required) ▪ Local Self Test (no acknowledge required)

2.4 Main differences between P8xC591 and SJA1000 CAN Functionality

SJA1000	P8xC591
One Acceptance Filter Bank with Dual or Single Filter support.	**Four** Acceptance Filter Banks with Dual or Single Filter support in each bank. All filters are changeable 'on the fly' (see also chapter 3.3.3).
One Receive Interrupt configuration: Receive Interrupt is generated on the reception of any message (while the Receive FIFO is not empty).	**Two** Receive Interrupt configurations possible: 1. High Priority Interrupt (see also chapter 3.3.1) 2. Receive Interrupt Level
CAN Bit Timing Calculation t_{scl} = **2** / f_{XTAL} x (32 BRP.5+ ……..+1)	CAN Bit Timing Calculation t_{scl} = **1** / f_{XTAL} x (32 BRP.5+ ……..+1)
BasicCAN Mode [1] PeliCAN Mode	 PeliCAN Mode
Different Transmit output stage configurations with two output pins (TX0, TX1): Normal, Bi-phase, Clock and Test output Mode	Transmit output stage with one output pin (TXDC) in Normal Mode configuration

[1] The SJA1000 is the successor product for the PCA82C200 Stand-alone CAN Controller. The BasicCAN mode keeps the SJA1000 software compatible to the former PCA82C200.

3. CAN FUNCTIONALITY

The CAN protocol 2.0B as specified in [2] is completely handled by the CAN Core Block, see also chapter 2.2. The CPU of the P8xC591 initialises and controls the communication. The following chapters describe all functions necessary to run CAN communication with the P8xC591.

3.1 CPU Interface

3.1.1 Special Function Registers

Access to all PeliCAN registers is performed via five Special Function Registers (SFR). They are mapped into the address range of the P8xC591. For all PeliCAN address locations indirect pointer (CANADR) based addressing is achieved. In addition frequently used PeliCAN registers can be accessed directly via CAN Special Function Registers.

Figure 3-1 shows all CAN Special Function registers including their PeliCAN registers. A direct read/write access to the PeliCAN Mode register is possible via CANMOD. The Command register is accessed while writing to CANCON and the Interrupt register is read while reading CANCON. The CANSTA register allows a write access to the Interrupt Enable register. Reading CANSTA is a direct access to the PeliCAN Status Register. CANSTA is bit-addressable and allows direct addressing of single status flags which is always useful for polling.

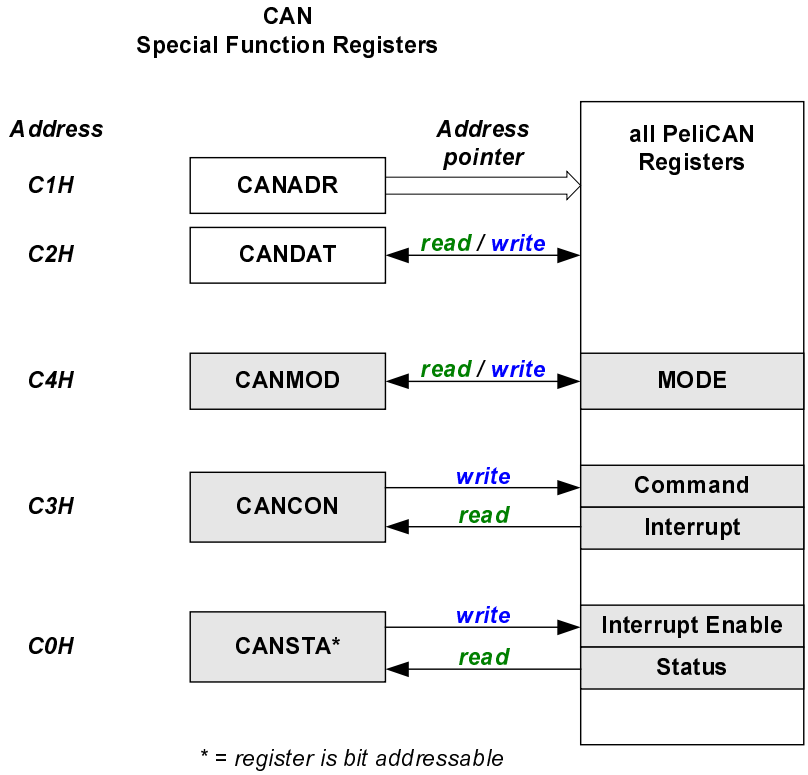

Figure 3-1: CAN Special Function Registers

All other CAN controller registers need to be addressed indirectly. The CANADR register points to the address of the PeliCAN register. During a write access the data to the addressed register has to be written into CANDAT. During a read access data from the addressed register can be read from CANDAT.

The following example illustrates the functionality for direct and indirect addressing of PeliCAN registers:

```
/* Direct addressing of Mode Register                                    */

      CANMOD = 0x01;            /* set Reset Request in Mode Register   */


/* Indirect addressing of Bit Timing Registers 0 and 1                   */

      CANADR = BTR0;            /* set address to BTR0 register         */
      CANDAT = 0x45;            /* write data to BTR0                   */
      CANADR = BTR1;            /* set address to BTR1 register         */
      CANDAT = 0x2B;            /* write data to BTR1                   */
```

3.1.2 Fast CAN Register Access with Auto-Increment

Indirect addressing, as described in the previous chapter, could be time consuming when addressing the CAN message buffers and the acceptance filters. Therefore, the P8xC591 includes an autoincrement mode for fast register addressing. The autoincrement feature is automatically activated when PeliCAN addresses beyond CAN address 31 (decimal) are selected by CANADR. Every read or write access to CANDAT automatically increments the CANADR pointer. This stack-like reading and writing could effectively being used for setting up a new transmit message as described below or for reading the RX Buffer.

```
/* setting up a new CAN message for transmitting                         */

     CANADR = TBF;                      /* point to 591 TX Buffer      */
     CANDAT = TransmitMessage[0];       /* write TX Frame Information  */
     CANDAT = TransmitMessage[1];       /* write TX Identifier 1       */
     CANDAT = TransmitMessage[2];       /* write TX Identifier 2       */
     CANDAT = TransmitMessage[3];       /* write TX Data Byte 1        */
     CANDAT = TransmitMessage[4];       /* write TX Data Byte 2        */
     …..
     …..
     …..

/* copy receive message from RX Buffer into the CPU RAM space           */

     CANADR = RBF;                      /* point to 591 RX Buffer      */
     ReceiveMessage[0] = CANDAT;        /* read RX Frame Information   */
     ReceiveMessage[1] = CANDAT;        /* read RX Identifier 1        */
     ReceiveMessage[2] = CANDAT;        /* read RX Identifier 2        */
     ReceiveMessage[3] = CANDAT;        /* read RX Data Byte 1         */
     ReceiveMessage[4] = CANDAT;        /* read RX Data Byte 2         */
     …..
     …..
```

3.2 Initialisation

3.2.1 Reset Mode and Operating Mode

After a power-up or hardware reset, the CAN controller will be in Reset Mode. In this mode, the RM bit in the Mode register will always be '1'. If the CAN controller is not in Reset Mode, setting the RM bit (either by hardware or software) will force it into Reset Mode. When in Reset Mode, the internal state machine of the CAN controller is frozen.

Typically after a power-up or hardware reset, once the boot-up and initialisation routines are complete, the CPU will put the CAN controller into Operating Mode by software (clearing the RM bit). In Operating Mode, any of the following will cause the RM bit to be set, forcing the CAN controller into Reset Mode:

- Hardware reset
- Software writing '1' to the MOD.0 (RM) bit
- Bus-Off condition

In addition, the 'special modes' of the CAN controller can only be entered from Reset Mode. These modes are the Test Mode, the Receive Polarity Mode, the Self Test Mode and the Listen Only Mode. After leaving the Reset Mode the CAN controller returns to the mode defined within the Mode Register.

3.2.2 CAN Controller set-up

The CAN controller has to be set-up for CAN communication after power-on or after hardware reset.

An initialisation procedure should cover the following items:

- Mode of Operation
- Acceptance Filter
- Bus Timing [3]
- TXDC Output pin configuration
- Interrupts

A Flow Diagram accompanied by a programming example of the initialisation procedure is shown in Figure 3-2.

For the CAN controller initialisation only the following registers and register bits shown in Table 1 are relevant. Most CAN Registers offer a convenient restore feature, where user configurations are kept and not changed after a Hardware Reset or putting the CAN controller into Reset Mode (marked by 'no change' in Table 1).

Register	Bit names	Register values:			
		After Power -up	After HW Reset	By setting MOD.0 (RM) bit[1]	
Mode Register	MOD.7 MOD.6 MOD.5 MOD.4 MOD.3 MOD.2 MOD.1 MOD.0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 0 0 0 0 no change no change 1	Test Mode Receive Polarity Mode Self Test Mode Listen Only Mode Reset Mode
Port 1 Configuration Register	P1M1.1 P1M2.1	0 0	0 0	no change no change	Output driver configuration for the CAN TXDC pin should be set to push-pull[2] means P1M1.1='0', P1M2.1='1'.
Interrupt Enable Register	IER	xxh	no change	no change	CAN related Interrupt Enable Register
Rx Interrupt Level Register	RIL	00h	00h	no change	Receive Interrupt Level
Bus Timing Register 0, see also [3]	BTR0	xxh	no change	no change	Synchronisation Jump Width Baud Rate Pre-scaler
Bus Timing Register 1, see also [3]	BTR1	xxh	no change	no change	Samples Per Bit Time Segment
Acceptance Filter Mode Register	ACFMOD	00h	no change	no change	Message Format for Bank 4, 3, 2, 1 Accept. Filter Mode for Bank 4, 3, 2, 1
Acceptance Filter Enable Register	ACFEN	xxh	no change	no change	Filter 1 & 2 Enable for Bank 4, 3, 2, 1
Acceptance Filter Priority Register	ACFPRIO	xxh	no change	no change	Filter 1 & 2 Priority for Bank 4, 3, 2, 1
Acceptance Code Register	ACR(3 to 0)	xxh	no change	no change	ACR for Bank 4, 3, 2, 1
Acceptance Mask Register	AMR(3 to 0)	xxh	no change	no change	AMR for Bank 4, 3, 2, 1

Table 1: CAN Controller register set-up

[1] CAN controller goes into Reset Mode by setting the MOD.0 (RM) bit in the Mode Register.

[2] After power-up and hardware Reset the default Port1.1 (TXDC) pin configuration is set to "pseudo bi-directional" (P1M1.1='0', P1M2.1='0').

3.2.3 Flow Diagram

Figure 3-2 presents a short and simple CAN controller initialisation routine. The according 'C' code is shown on the next page.

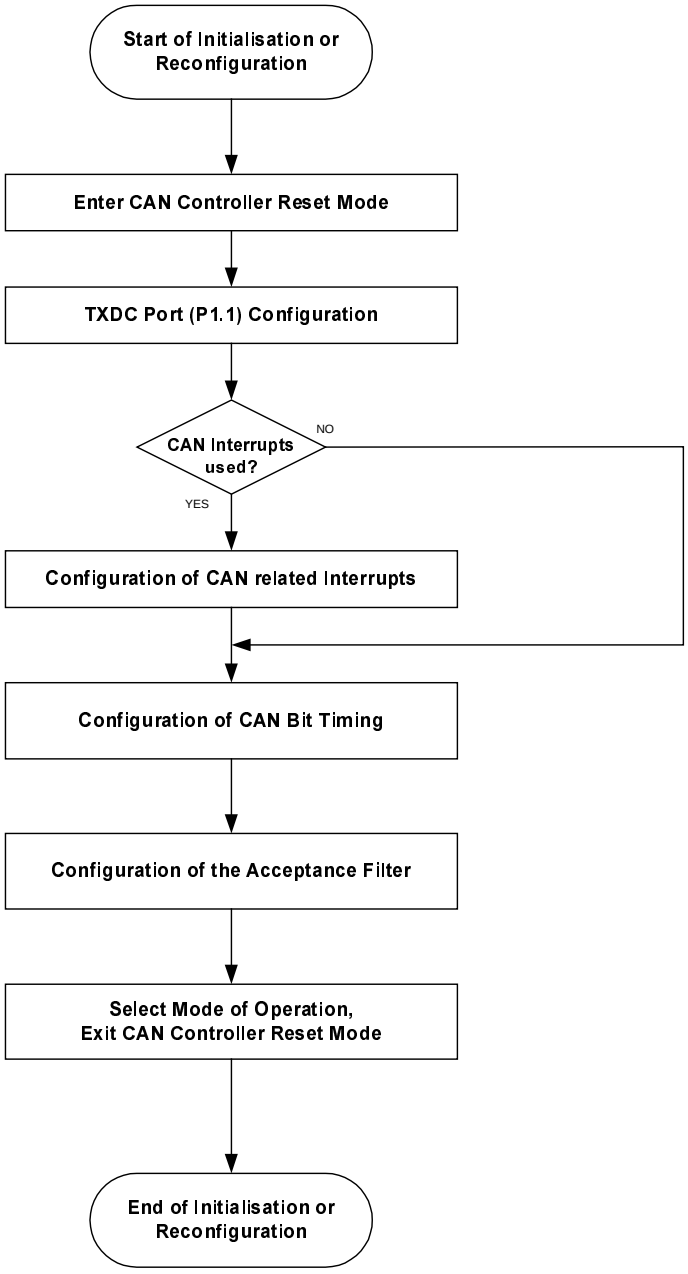

Figure 3-2: Initialisation Flow Diagram

Example 'C' code – CAN Initialisation -

The figure below is the according 'C' code to the Flow Diagram from the previous page.

```
void init_can_controller ( void )
{
/* Enter CAN Controller Reset Mode:
   -------------------------------                                    */
      CANMOD = 0x01;            /* set the CAN controller to reset     */
                                /* mode to start initialization        */

/* TXDC Port (P1.1) Configuration:
   -------------------------------
                                /* Pin TXDC set to push-pull           */
      P1M2 = P1M2 | 0x02;       /* P1M2.1='1', P1M1.1 = '0' (default)  */

/* Configuration of CAN related Interrupts:
   ---------------------------------------
      CANSTA = 0x03;            /* receive and transmit interrupts     */
                                /* are enabled in this example         */

/* Configuration of CAN Bit Timing:
   -------------------------------
      CANADR = BTR0;            /* BTR0 and BTR1 are programmed for    */
      CANDAT = 0x45;            /* 125 kbit/s @12 MHz crystal          */
      CANADR = BTR1;            /* TSEG1 = 12, TSEG2 = 3, SJW = 2      */
      CANDAT = 0x2B;            /* Samples = 1  -> sample point ~81 %  */

/* Configuration of the Acceptance Filter:
   --------------------------------------
      Filter 1 of bank 1 is configured to receive ID = 010.0000.0XXX    */

      CANADR = ACR10;    /* set address to Acc. Code Register 0 (Bank 1)*/
      CANDAT = 0x40;     /* acceptance code 0 used for filtering        */

      CANADR = AMR10;    /* set address to Acc. Mask Register 0 (Bank 1)*/
                                                                       */
      CANDAT = 0x00;     /* bank1: acceptance mask 0                    */
      CANDAT = 0xFF;     /* bank1: acceptance mask 1 don't care         */
      CANDAT = 0xFF;     /* bank1: acceptance mask 2 don't care         */
      CANDAT = 0xFF;     /* bank1: acceptance mask 3 don't care         */

      CANADR = ACFMOD;   /* set address to ACF Mode register            */
      CANDAT = 0x55;     /* single accept. filters for 11-bit ID's (SFF)*/

      CANADR = ACFPRIO;  /* set address to ACF Priority register        */
      CANDAT = 0xFF;     /* high priorities for all filters             */

      CANADR = ACFEN;    /* set address to ACF Enable register          */
      CANDAT = 0x01;     /* enable acceptance filter 1 of bank1         */

/* Select Mode of Operation, Exit CAN Controller Reset Mode:
   --------------------------------------------------------
      CANMOD = 0x00;     /* put CAN controller into operation mode      */
}
```

3.3 Acceptance Filter

With the P8xC591 Acceptance Filter, incoming CAN messages are filtered so that only predefined messages are accepted and stored in the Receive FIFO. If enabled, only these accepted messages generate a Receive Interrupt and thus reduce the CPU processing time for servicing CAN.

Four identical Acceptance Filter Banks can be configured independently. Each bank has the functionality known from the SJA1000 with the extension, that the filters are 'changeable on the fly' (see also chapter 3.3.3).

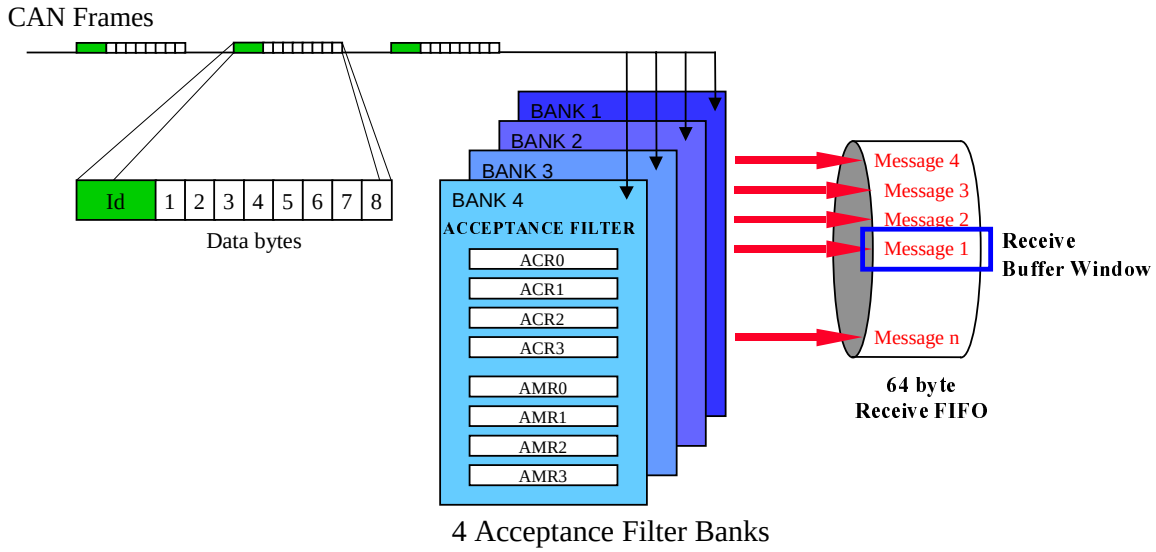

Figure 3-3: Acceptance Filter

The Acceptance Code Registers (ACRn) and the Acceptance Mask Registers (AMRn) define the Acceptance Filter. Within the Acceptance Code Registers the bit patterns of receive messages are defined. The corresponding Acceptance Mask Registers allow defining certain bit positions to be 'don't care' by setting them to '1'. With this feature groups of CAN messages can be defined for reception.

ACR	`101.1010.0101`
AMR	`000.1111.0000`
Accepted messages	`101.xxxx.0101`

Figure 3-4: Example of Acceptance Filtering

As shown in Figure 3-5, each Acceptance Filter Bank can be configured as a Single Filter or as a Dual Filter. Both filter configurations support Standard CAN frames (11-bit ID) as well as Extended CAN frame (29-bit ID). For more details on acceptance filter configurations see the P8xC591 Data Sheet [1].

All four configurations for each filter bank can be defined in the Acceptance Filter Mode Register and are only possible during the Reset Mode of the CAN controller.

ACFMOD Register CAN Address: 1Dh

Symbol	Function
MFORMATBn	Message Format: '0' = SFF (11-bit ID), '1' = EFF (29-bit ID)
AMODEBn	Accept. Filter Mode: '0' = Single Accept. Filter, '1' = Dual Accept. Filter

n = Bank 4, 3, 2 or 1

Table 2: Acceptance Filter Mode Register

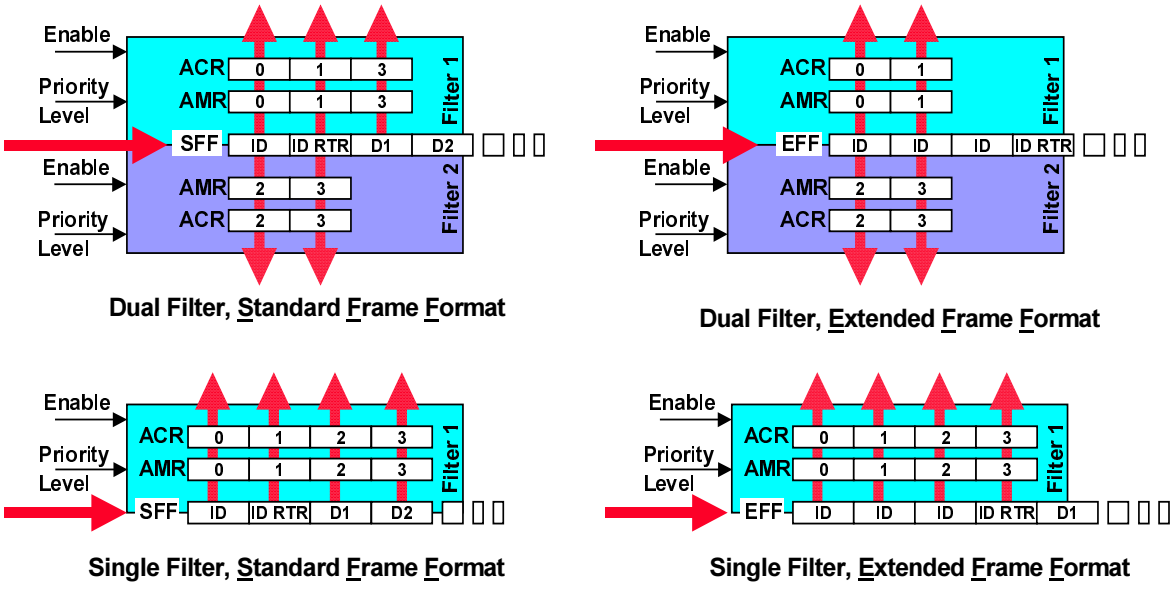

Figure 3-5: Four possible Acceptance Filter Configurations

3.3.1 Acceptance Priority

For each Acceptance Filter two different interrupt configurations are possible.

1. High Priority Interrupt

A Receive Interrupt is generated immediately, if a CAN frame passes an Acceptance Filter which is configured for 'High Priority Interrupt' generation. This allows using, e.g., certain Acceptance Filters for alarm message recognition.

2. Receive Interrupt Level

A Receive Interrupt is generated, if the number of message bytes in the Receive FIFO exceeds the level specified in the Receive Interrupt Level Register, RIL (more details in chapter 3.6.4).

Both interrupt configurations are defined in the Acceptance Filter Priority Register.

ACFPRIO CAN Address: 1Fh

Symbol	Function
BnF2PRIO	Filter 2 Priority, '1' = High Priority Interrupt, '0' = Rx Level Interrupt
BnF1PRIO	Filter 1 Priority, '1' = High Priority Interrupt, '0' = Rx Level Interrupt

n = Bank 4, 3, 2 or 1

Table 3: Acceptance Filter Priority Register

3.3.2 Higher Layer Protocol Support, Acceptance Filtering on Data Bytes Example: DeviceNet Protocol, Explicit Messaging.

The DeviceNet Protocol [7] uses the Standard Frame Format with an 11-bit CAN Identifier. An Explicit Message uses the data field of a CAN frame to carry DeviceNet defined information. The data field (0..8 bytes) of a transmission that contains the complete Explicit Message includes a Message Header and the entire Message Body. The Message Header is specified within the 1st Data Byte of the CAN data field and contains the Destination MAC ID.

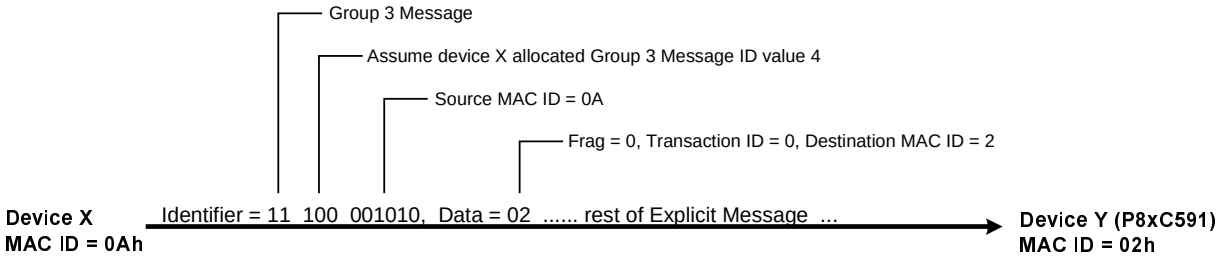

Figure 3-6: Example for an Explicit Messaging Connection

The 11-bit CAN Identifier Field in combination with the MAC ID in the Message defines a Source or Destination MAC ID and has to be examined.
The P8xC591 Acceptance Filter can easily filter both, the Message Header and Source MAC ID.

With the Higher Layer Protocol Support, filtering on the MAC ID in the 1st Data Byte the P8xC591 will get less interrupted and the overall performance of the whole system can be improved significantly.

Additionally the 2nd Data Byte containing the Service Code Information in the Open Explicit Messaging Connection can be used as well for acceptance filtering.

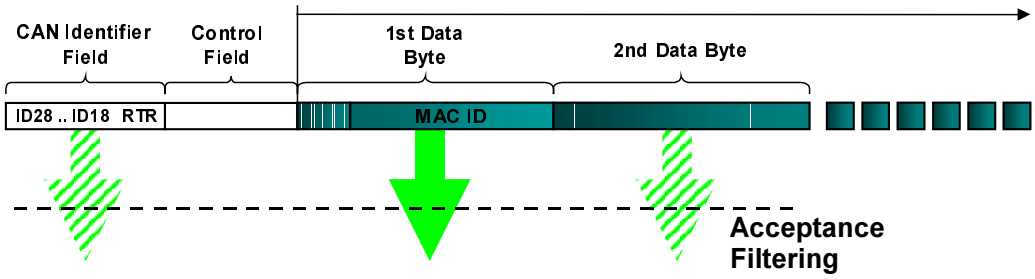

Figure 3-7: DeviceNet Protocol, Explicit Messaging

Example 'C' code – Device Net Initialisation

The following 'C' code is an example for an Acceptance Filter configuration for Explicit Messaging in the DeviceNet protocol, see also Figure 3-6. The P8xC591 is used as the Device Y.
Typically the function would be called during an initialisation of a CAN node when the CAN controller is in Reset Mode. In any other case (CAN controller is **not** in Reset Mode), the call of this function would be a re-configuration of the Acceptance Filter and should be handled as a 'change on the fly' of the filter. In this case the desired filter has to be disabled before calling the function and enabled after exiting the function (see also chapter 3.3.3).

```
/*************************************************************************/
/*                                                                       */
/* TITLE  Example 'C' Code for                                           */
/*        DeviceNet, Explicit Messaging Connection                       */
/*                                                                       */
/*        Functional Description:                                        */
/*        This Function is an example for an Acceptance Filter           */
/*        configuration for Explicit Messaging in the DeviceNet          */
/*        protocol.                                                      */
/*                                                                       */
/*        Message parameters:                                            */
/*        Message Group, Source MAC ID, Destination MAC ID               */
/*        are directly used to set-up a P8xC591 acceptance filter        */
/*                                                                       */
/*        NOTICE: Copyright (C) 2000 PHILIPS Semiconductors              */
/*                                                                       */
/*************************************************************************/

#include "reg591.h"
#include "c591_def.h"

void Explicit_Messaging_Filter_Configuration
                      (BYTE Group, BYTE ID, BYTE SourceMAC, BYTE DestMAC)
{

/* Configuration of the Bank1 Acceptance Code Register 0:
   ------------------------------------------------------              */

      CANADR = ACR10;  /* set address to Acc. Code Register 0 (Bank 1)  */

      CANDAT = (Group << 6)+(ID << 3)+(SourceMAC >> 3);/* ACR0          */
      CANDAT = (SourceMAC << 5);                       /* ACR1          */
      CANDAT =  DestMAC;                               /* ACR2          */
                                                       /* ACR3 not used */

/* Configuration of the Bank1 Acceptance Mask Register 0:
   ------------------------------------------------------
*/
      CANADR = AMR10;  /* set address to Acc. Mask Register 0 (Bank 1)  */

      CANDAT = 0x00;                                   /* AMR0          */
      CANDAT = 0x0F;                                   /* AMR1          */
      CANDAT = 0x00;                                   /* AMR2          */
      CANDAT = 0xFF;                                   /* AMR3          */
}
```

3.3.3 Change Acceptance Filter on the fly

The P8xC591 Acceptance Filter configuration can be changed in two different ways. Besides the most common way, changing the filter during the CAN controller Reset Mode, the P8xC591 offers the new 'change on the fly' feature. With this feature an Acceptance Filter configuration can be changed during a running CAN communication. The Flow Diagram in Figure 3-8 is a general example how to re-configure the Acceptance Filter during the CAN controller Operating Mode. The according 'C' code is shown on the next page.

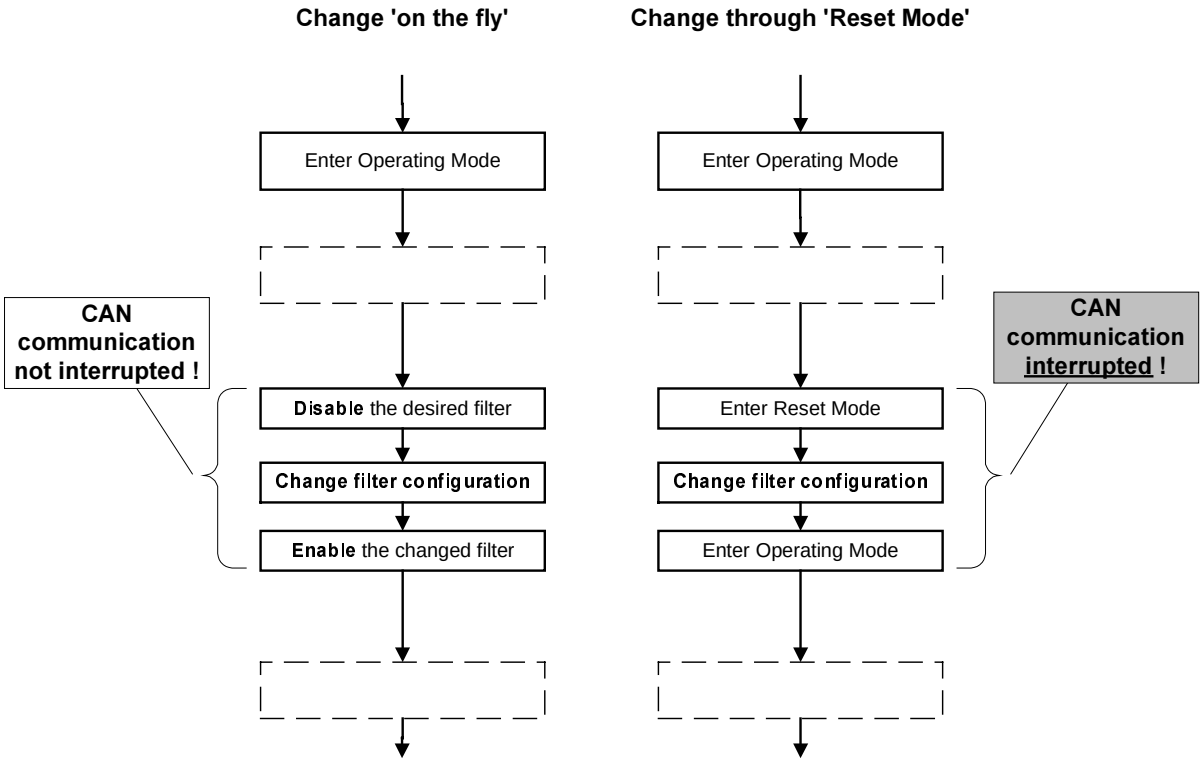

Figure 3-8: Change of Acceptance Filter Flows

Change 'on the fly'

Each defined Acceptance Filter can be disabled and enabled with certain bit-locations in the Acceptance Filter Enable Register. If the corresponding filter is disabled, it is possible to change the Acceptance Filter configuration during normal operation. This very convenient way of changing a filter where the CAN bus communication is not interrupted is called 'change on the fly' feature.

Change through 'Reset Mode'

During Reset Mode of the CAN controller, all Acceptance Filters can be defined or re-defined. During normal operation this method has some disadvantages:

- CAN communication is interrupted and some messages could be lost during this time.
- Change process takes more time.

Example 'C' code – Change on the fly -

Below is the according 'C' code to the Flow Diagram from the previous page. In this case a Standard CAN Frame Format (SFF) with 11-bit identifier is used and one of the filters is changed on the fly.

```
/*****************************************************************************/
/*                                                                           */
/* TITLE  Example 'C' Code for                                               */
/*        the 'change on the fly' feature of the P8xC591 CAN Controller      */
/*                                                                           */
/*        Functional Description:                                            */
/*        This Function is a typical example for the 'change on the fly'     */
/*        feature for changing a configuration of the P8xC591 Acceptance     */
/*        Filter during normal Operating Mode. A Standard CAN Frame          */
/*        Format (SFF) with 11-bit identifier was used in this example.      */
/*                                                                           */
/*        NOTICE: Copyright (C) 2000 PHILIPS Semiconductors                  */
/*                                                                           */
/*****************************************************************************/

void change_on_the_fly ( void )
{
/* Disable the desired Filter:
   ---------------------------                                               */
      CANADR = 0x1E;            /* set address to ACF Enable register   */
      CANDAT = CANDAT & 0xFE;   /* disable acceptance filter 1 of bank1 */

/* Change filter configuration:
   ----------------------------
   In this re-configuration example all eleven bits of the CAN messages
   are used for message filtering. Only messages with the CAN-ID: 301hex
   equals 01100000001b are supposed to be received.
   The first two data bytes are not used for filtering and are set to
   don't care.                                                               */

      CANADR = ACR10;  /* set address to Acc. Code Register 0 (Bank 1)  */
      CANDAT = 0x60;   /* bank1: ACR0 used for filtering                */
      CANDAT = 0x20;   /* bank1: ACR1 only upper 4 bits used filtering  */
                       /* bank1: acceptance code 2 don't care, not used */
                       /* bank1: acceptance code 3 don't care, not used */

      CANADR = AMR10;  /* set address to Acc. Mask Register 0 (Bank 1)  */
      CANDAT = 0x00;   /* bank1: AMR0                                   */
      CANDAT = 0x0F;   /* bank1: AMR1 only upper 4 bits used            */
      CANDAT = 0xFF;   /* bank1: AMR2 accept. mask 2 don't care         */
      CANDAT = 0xFF;   /* bank1: AMR3 accept. mask 3 don't care         */

/* Enable the changed Filter:
   --------------------------                                                */
      CANADR = ACFEN;           /* set address to ACF Enable register  */
      CANDAT = CANDAT | 0x01;   /* enable acceptance filter 1 of bank1 */
}
```

3.4 CAN Interrupts

The CAN controller of the P8xC591 has 8 different CAN interrupts, which may be used to initiate immediate actions by the CPU on certain states of the CAN controller. The general interrupt flow in Figure 3-9 gives an overview of all possible CAN interrupts. A CAN interrupt is generated when one or more bits of the CAN Interrupt Register are set. It depends very much on the system and the requested behaviour, in which order the interrupts are served.
Note that other routines might use the CANADR as well. If this register is modified within the ISR, it is recommended to store the current CANADR value at the beginning of the interrupt service routine and restore it at the end.

Examples for the reaction on transmission and reception interrupts are discussed in chapters 3.5.2 and 3.6.2. Detailed flows and examples for wake-up, data overrun, CAN error handling and arbitration lost interrupt services are described in [4].

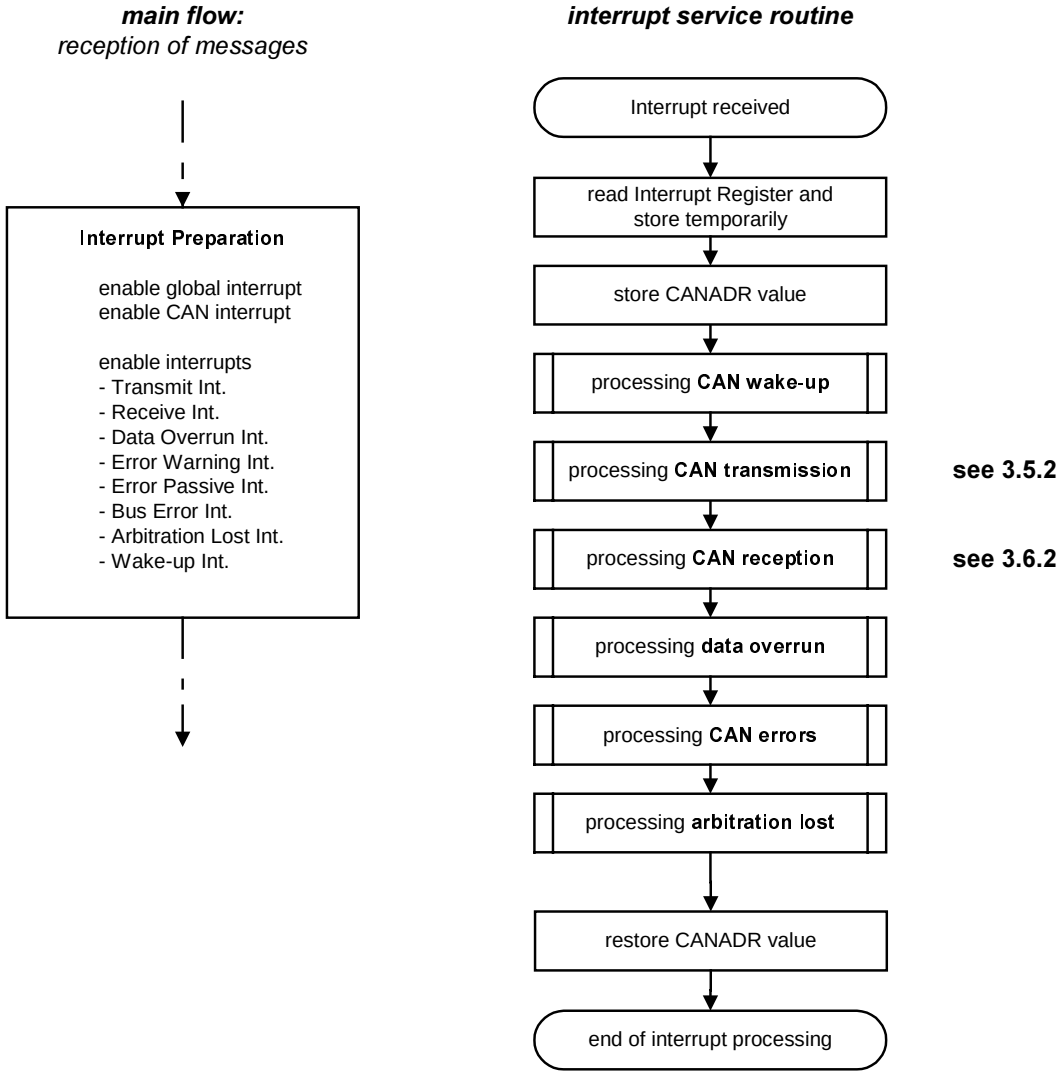

Figure 3-9: General interrupt flow

Example 'C' code - CAN Interrupt Service -

```
/* ***************************************************************** */
/* CAN Interrupt Service                                             */
/* This function can be used as a general interrupt service. All CAN    */
/* interrupt sources of the P8xC591 are considered. Before a call to a  */
/* certain sub process is performed the contents of the interrupt reg.  */
/* is saved. Additionally CANADR is temporarily stored and restored.    */
/* ***************************************************************** */
void ECAN_int_service(void) interrupt 13 using 3  /* high priority      */
{
  InterruptRegCopy = CANCON;            /* read interrupt register      */
  CANADR_save = CANADR;                 /* save CANADR                  */

  /* CAN Transmit Interrupt ? */
  if ((InterruptRegCopy & 0x02) == 0x02)
   {
     /* Transmit Interrupt Handling */
   }
  /* CAN Receive Interrupt ? */
  if ((InterruptRegCopy & 0x01) == 0x01)
   {
     do {
         RX_Service();
        } while (RBS);                  /* Receive Buffer empty ?       */
   }
  /* Data Overrun Interrupt ? */
  if ((InterruptRegCopy & 0x08) == 0x08)
   {
    /* Data Overrun Handling  */
   }
  /* Error Interrupt ? */
  if ((InterruptRegCopy & 0x04) == 0x04)
   {
    /* Error Handling */
   }
  /* Error Passive Interrupt ? */
  if ((InterruptRegCopy & 0x20) == 0x20)
   {
    /* Error Passive Handling */
   }
  /* Bus Error Interrupt ? */
  if ((InterruptRegCopy & 0x80) == 0x80)
   {
    /* Bus Error Handling */
   }
  /* Arbitration Lost Interrupt ? */
  if ((InterruptRegCopy & 0x40) == 0x40)
   {
    /* Arbitration Lost Handling */
   }
    CANADR = CANADR_save;                   /* restore CANADR           */

  /* ----- end of EX0_int_service -----                                 */
 }
```

3.5 Transmission

A transmission of a message is done autonomously by the CAN controller according to the CAN protocol specification. First, the CPU has to transfer the Transmit message into the Transmit Buffer and set the 'Transmission Request' flag in the Command Register. The transmission process can be controlled either by an interrupt request or by polling status flags.

3.5.1 Polling Controlled Transmission

As shown in Figure 3-10 the transmit interrupt of the CAN controller is disabled for this type of transmission control. As long as the P8xC591 is transmitting a message, the Transmit Buffer is locked for writing. Thus the CPU has to check the 'Transmit Buffer Status' flag (TBS) of the Status Register, if a new message can be placed into the Transmit Buffer.

- The Transmit Buffer is locked:

  Polling the Status Register periodically, the CPU waits, until the Transmit Buffer is released.

- The Transmit Buffer is released:

  The CPU writes the new message into the Transmit Buffer and sets the flag 'Transmission Request' (TR) of the Command Register, which will cause the start of the transmission.

The CAN message is transmitted successfully when the Transmit Complete Status bit is set.

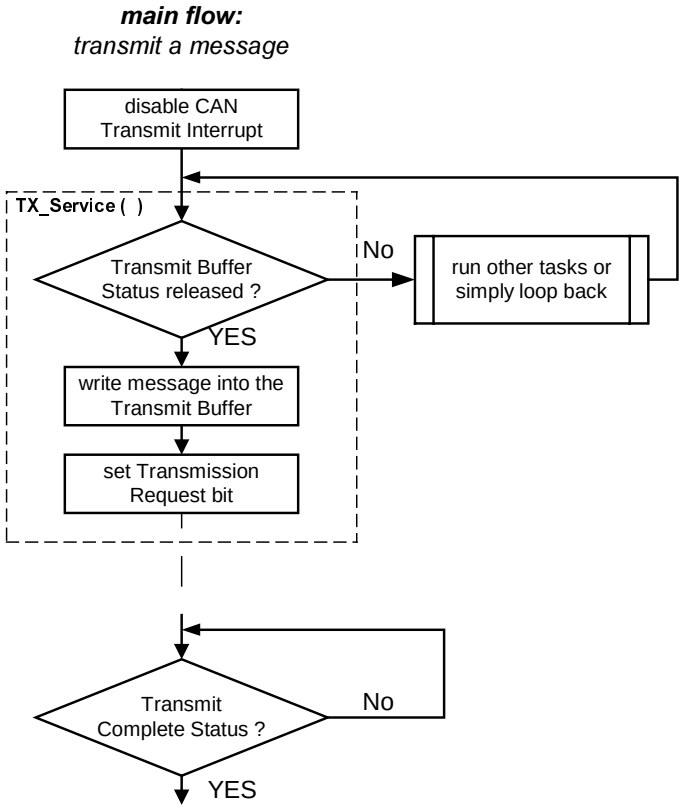

Figure 3-10: Flow Diagram 'Transmission of a message' (polling controlled)

3.5.2 Interrupt Controlled Transmission

According to the main processing of the controller, the transmit interrupt of the CAN controller and the global interrupt(s) of the P8xC591 must be enabled prior to the start of an interrupt controlled transmission. The interrupt enable flags are located in the Interrupt Enable Register of the CAN controller. As long as the P8xC591 is transmitting a message, the Transmit Buffer is locked for writing. Thus the CPU has to check the 'Transmit Buffer Status' flag (TBS) of the Status Register, if a new message can be placed into the Transmit Buffer. The procedure of writing a certain CAN message into the Transmit Buffer and setting the Transmission Request is similar to the process described in the previous chapter.

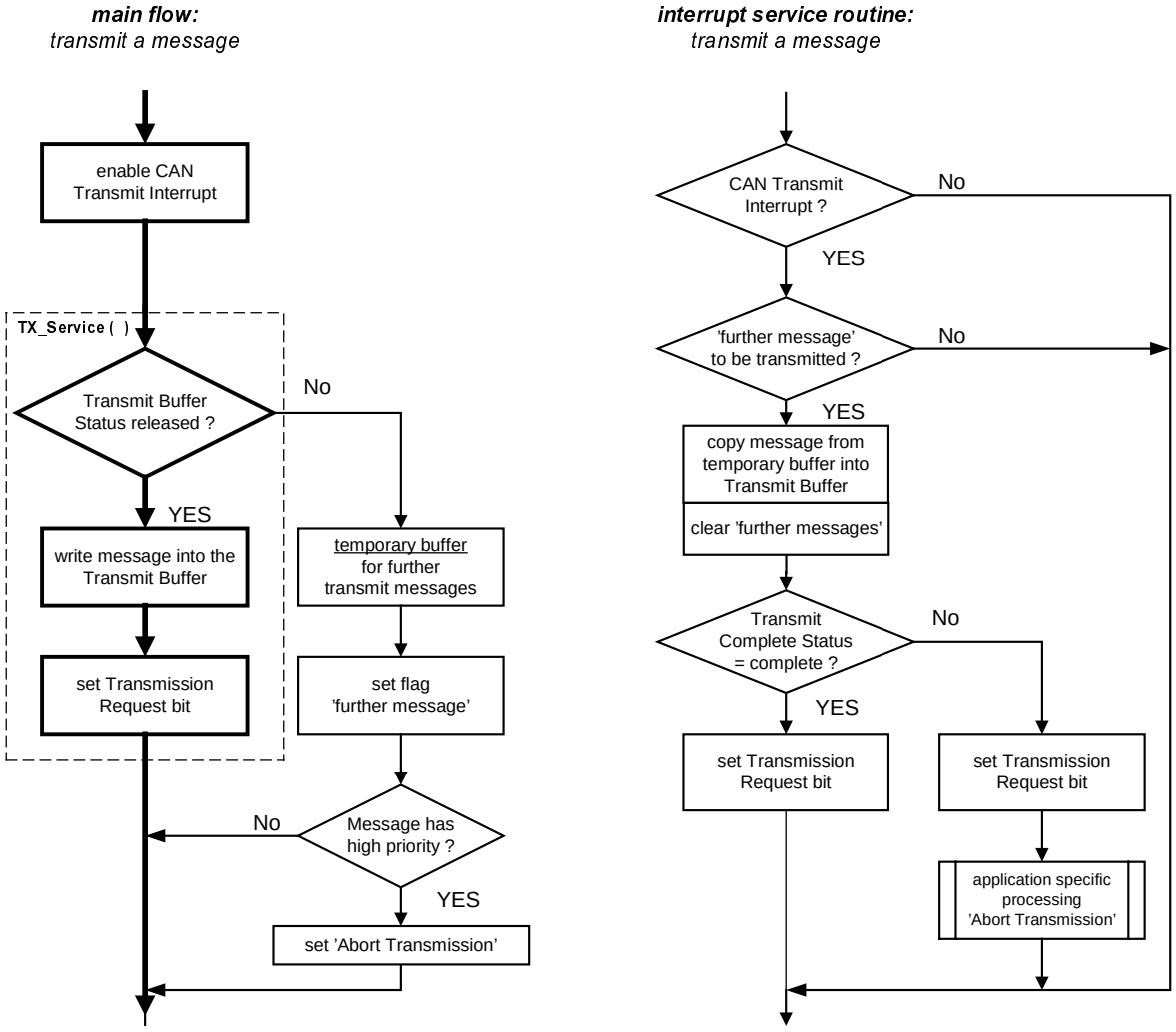

Figure 3-11: Example Flow Diagram 'Transmission of a message' (interrupt controlled)

The diagram in Figure 3-11 shows the standard flow of interrupt controlled message transmission.

Figure 3-11 also includes a solution for scheduling transmit messages that cannot be transmitted because the Transmit Buffer is not released. In this case the 'abort transmit' function of the CAN controller is used.

- The Transmit Buffer is locked:

  In case of the Transmit Buffer is locked, the CPU has to store the new message temporarily in the data memory and sets a software flag 'further message', indicating that a further message is waiting for transmission.

  The start of a next transmit message will in this case be handled during the interrupt service routine, which is initiated at the end of the current running transmission. Upon reception of an interrupt from the CAN controller, the CPU checks the type of interrupt, see also Figure 3-9. In case of a Transmit Interrupt and the 'further message' flag is set, the waiting message has to be copied from the data memory into the Transmit Buffer and the 'further message' flag is cleared. The 'Transmission Request' flag of the Command Register is set, which will cause the CAN controller to start the transmission.

- The Transmit Buffer is released:

  The CPU writes the new message into the Transmit Buffer and sets the flag 'Transmission Request', which will cause the P8xC591 to start the transmission. At the end of a successful transmission, the CAN controller generates a Transmit Interrupt.

3.5.2.1 Abort Transmission

The transmit request of a message, may be aborted using the 'Abort Transmission' command by setting the corresponding bit in the Command Register. This feature of the P8xC591 CAN controller may be used, e.g., for transmitting an urgent message prior to the message, which has been written into the transmit buffer previously and was not transmitted successfully until now.
Figure 3-11 shows a flow using the transmit interrupt. It illustrates the situation, where a message has to be aborted in order to transmit a message with a high priority. Other reasons for aborting a message may require different interrupt flows.

In case a transmit message is still waiting for being served due to different reasons, the Transmit Buffer is locked (see also Figure 3-11). If a transmission of an urgent message is requested, the Abort Transmission bit is set in the Command Register.

When the message waiting to be served has either been transmitted successfully or aborted, the Transmit Buffer is released and a Transmit Interrupt is generated. During the interrupt processing the Transmission Complete flag of the Status Register has to be checked, whether a previous transmission was successful or not. The status 'incomplete' indicates, that the transmission was aborted. In this case the CPU can run through a special routine dealing with a strategy for abort transmission, e.g., repeat the transmission of the aborted message.

A 'C' code example for the TX_Service routine which can be used for both polling and interrupt controlled transmission is given on the next page.

Example 'C' code - TX_Service –

```
/* ******************************************************************** */
/* Transmit Service Function                                            */
/* This function assumes that a CAN transmit message is already         */
/* available in the RAM space of the Microcontroller beginning at       */
/* TransmitMessage[0]. The order of the transmit bytes are organized as */
/* described in the Transmit Buffer chapter of the data sheet.          */
/* ******************************************************************** */

void TX_Service (BYTE *TransmitMessage)
{
  BYTE  Length;              /* CAN Data Length Code                    */
  BYTE  i;                   /* index                                   */
  bit   FF;                  /* FF = 0 (Standard CAN Frame)             */
                             /* FF = 1 (Extended CAN Frame)             */

  if (TBS == 1)              /* Transmit Buffer Status = released ?     */
    {
     /* write message into the Transmit Buffer                          */

     FF     = TransmitMessage[0] & 0x80; /* get Frame Format            */
     Length = TransmitMessage[0] & 0x0F; /* get DLC                     */
     if (Length > 0x08)
       Length = 0x08;

     CANADR = TBF;                       /* point to 591 TX Buffer      */
     CANDAT = TransmitMessage[0];        /* write TX Frame Information  */
     CANDAT = TransmitMessage[1];        /* write TX Identifier 1       */
     CANDAT = TransmitMessage[2];        /* write TX Identifier 2       */

     if (FF)
       {                                 /* Extended Frame Message      */
        CANDAT = TransmitMessage[3];     /* write TX Identifier 3       */
        CANDAT = TransmitMessage[4];     /* write TX Identifier 4       */
       }

     for (i=0; i<Length; i++)            /* write data bytes            */
       {
        if (FF)
          CANDAT = TransmitMessage[i+5];
        else
          CANDAT = TransmitMessage[i+3];
       }

     /* set Transmission Request bit */
     CANCON  = 0x01;
    }
  else
    {
     /* P8xC591 transmit buffer is not released */
     /* run other tasks or call function again  */
    }
}
```

3.6 Reception

3.6.1 Polling Controlled Reception

A typical flow for polling controlled reception is shown in Figure 3-12. The Receive Interrupt of the CAN controller is disabled for this type of reception control. The CPU reads the Status Register of the CAN controller on a regular basis, checking if the Receive Buffer Status flag (RBS) indicates, that at least one message has been received.

- The Receive Buffer Status flag indicates 'empty', i.e., no message has been received:

  The CPU continues with the current task until a new request for checking the Receive Buffer Status is generated.

- The Receive Buffer Status flag indicates 'full', i.e., one or more messages have been received:

  The CPU gets the first message from the CAN controller and sets the Release Receive Buffer flag in the Command Register. The CPU can process each received message before checking for further messages, as indicated in Figure 3-12.
  But it is also possible to download all messages into the data memory by polling the Receive Buffer Status bit again and process all received messages together later. In this case the data memory has to be large enough to store more than one message before they are processed.
  After transferring and processing one or all messages, the CPU can continue with other tasks.

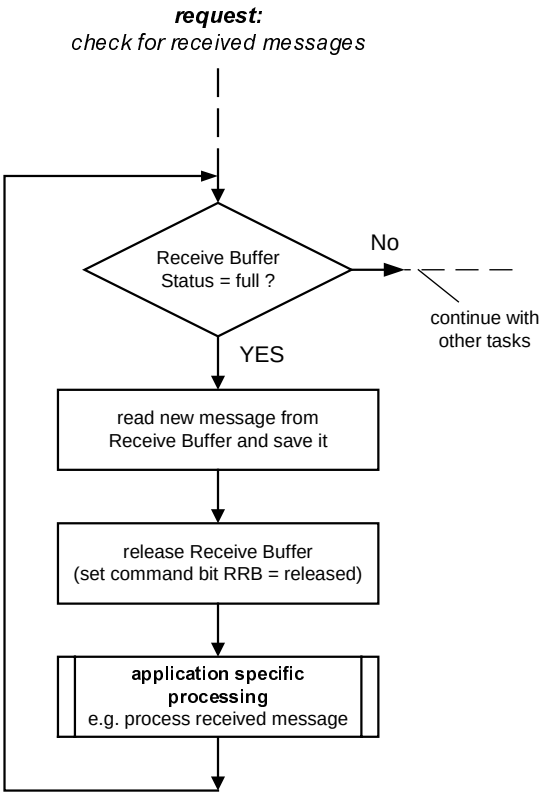

Figure 3-12: Flow Diagram 'Reception of a message' (polling controlled)

3.6.2 Interrupted Controlled Reception

According to the main processing of the controller as given in Figure 3-13, the receive interrupt of the CAN controller and the global interrupt(s) of the P8xC591 must be enabled prior to an interrupt controlled reception of messages. The receive interrupt enable flag is located in the Interrupt Enable Register.

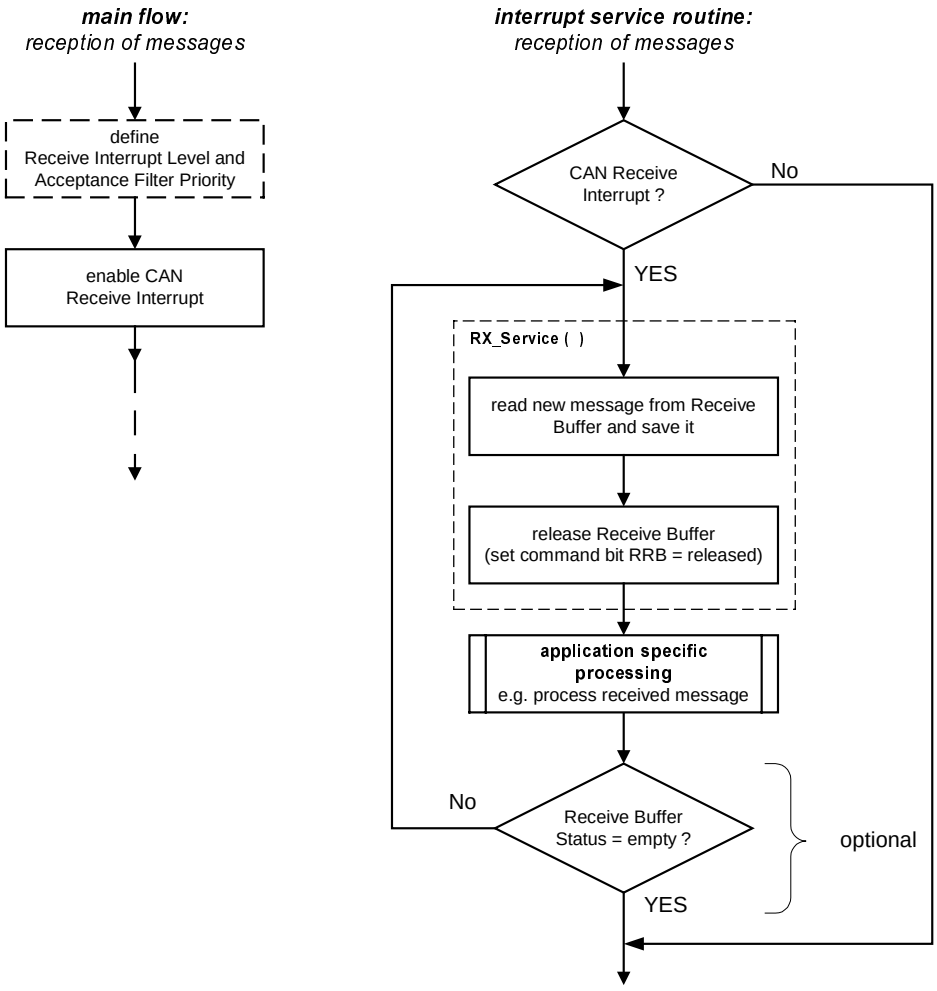

Figure 3-13: Flow Diagram 'Reception of a message' (interrupt controlled)

If the P8xC591 has received a message, which has passed the acceptance filter and has been placed into the Receive FIFO, a receive interrupt is generated. Thus the CPU can react immediately, transferring the received message into its message memory and sets the Release Receive Buffer flag 'RRB' in the Command Register. Further messages in the Receive FIFO will generate a new receive interrupt, so it is not necessary to read all messages available in the Receive FIFO during one interrupt. Nevertheless, at the end of processing the receive interrupt, the CPU can check for further messages by reading the Receive Buffer Status flag (RBS). This option is always useful when the Receive Level Interrupt is used, see also chapter 3.6.4. As given in Figure 3-13, the whole reception process may be done during the interrupt service routine without interaction of the main program.

Example 'C' code - RX_Service –

```
/* ***************************************************************** */
/* Receive Service Function                                          */
/* This function is used to copy a CAN message from the Receive FIFO */
/* into the microcontroller memory space starting at ReceiveMessage[0]. */
/* The function allows automatic message length handling by using DLC */
/* and automatic detection of 11 or 29-bit CAN messages.             */
/* ***************************************************************** */

void RX_Service ()
{
  BYTE  Length;                   /* CAN Data Length Code            */
  BYTE  i;                        /* index                           */
  bit   FF;                       /* FF = 0 (Standard CAN Frame)     */
                                  /* FF = 1 (Extended CAN Frame)     */
  /* read new message from Receive Buffer and store it               */
  CANADR = RBF;                   /* point to 591 RX Buffer          */
  ReceiveMessage[0] = CANDAT;     /* read and store Frame Info Byte  */

  FF     = ReceiveMessage[0] & 0x80; /* get Frame Format             */
  Length = ReceiveMessage[0] & 0x0F; /* get DLC                      */
  if (Length > 0x08)
      Length = 0x08;

  ReceiveMessage[1] = CANDAT;     /* read and store RX Identifier 1  */
  ReceiveMessage[2] = CANDAT;     /* read and store RX Identifier 2  */

  if (FF)
    {
     ReceiveMessage[3] = CANDAT;  /* read and store RX Identifier 3  */
     ReceiveMessage[4] = CANDAT;  /* read and store RX Identifier 4  */
    }

  for (i=0; i<Length; i++)        /* read and store data bytes       */
    {
     if (FF)
       ReceiveMessage[i+5] = CANDAT;
     else
       ReceiveMessage[i+3] = CANDAT;
    }

  /* release Receive Buffer */
  CANCON = 0x04;
```

3.6.3 Data Overrun Handling

In case the Receive FIFO is full but another message is being received, a Data Overrun is signalled to the CPU by setting the Data Overrun Status in the Status Register. If enabled, a Data Overrun Interrupt is generated. A reason for running into a Data Overrun situation could be, that the CPU is extremely overloaded and has no time to fetch received messages from the Receive Buffer in time. Normally a system should avoid Data Overrun conditions. If Data Overrun situations cannot be avoided an application specific 'Data Overrun strategy' should be implemented, e.g., requesting important messages. However, upon Data Overrun, the Receive FIFO must be read out and released for new messages. Finally the 'Clear Data Overrun' command has to be given to terminate this status.

3.6.4 Receive Interrupt - Level or High Priority

The user can define a Receive Interrupt Level for reception. The default value for RIL is '00'. In this case, every valid CAN message that has passed the acceptance filter generates an interrupt for the CPU, if the receive interrupt is enabled.

The interrupt behaviour is different if the user specifies a certain level within the Receive Interrupt Level Register (RIL). As soon as the specific amount of bytes is stored in the RXFIFO and the last message has a valid status a receive interrupt is generated. As described in chapter 3.3.1 the acceptance filter priority register must be configured for **Rx Level Interrupt** generation.

Example: RIL = 0x14 (20d)

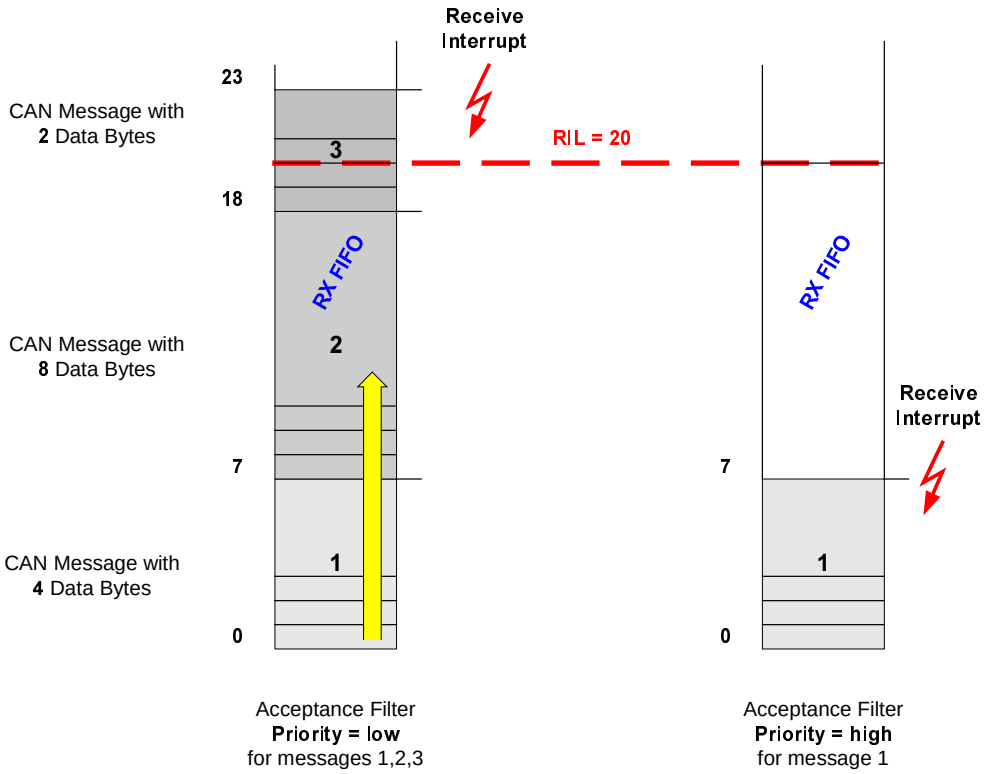

Figure 3-14: Receive Interrupt Level Example

To support message reception of high priority messages the user can define the priority for a certain message within the acceptance filter priority register ACFPRIO. As soon as a CAN message passes a 'high-priority' acceptance filter (ACFPRIO.x = 1), a receive **High Priority Interrupt** is generated immediately upon complete reception, regardless of the value of the RIL register, see Figure 3-14.

Example 'C' code - Receive Interrupt Level -

```
/* ******************************************************************** */
/* Receive Interrupt Level                                              */
/* This part of the initialisation shows how a Receive Interrupt Level  */
/* can be defined. This includes also the definition of the acceptance  */
/* filter priority.                                                     */
/* ******************************************************************** */

void init_can_controller ( void )
{

/* Enter CAN Controller Reset Mode
   ------------------------------                                       */
      CANMOD = 0x01;          /* set the CAN controller to reset mode   */
      ..
      ..
      ..
      CANADR = RIL;
      CANDAT = 0x14;          /* define receive interrupt level at 20d  */
      ..
      ..
      ..

      /* Bank 1 */
      CANADR = ACR10;         /* Acceptance Code Register 0             */
      CANDAT = 0xA0;
      ..
      /* Bank 2 */
      CANADR = ACR20;         /* Acceptance Code Register 0             */
      CANDAT = 0xB0;
      ..
      ..
      CANADR = ACFMOD;        /* point to ACF Mode register             */
      CANDAT = 0x05;          /* single filter for standard CAN frames  */
                              /* for bank1 and bank 2                   */

      CANADR = ACFPRIO;       /* point to ACF Priority register         */
      CANDAT = 0x04;          /* low  priority for filter 1 of bank1    */
                              /* high priority for filter 1 of bank2    */

      CANADR = ACFEN;         /* point to ACF Enable register           */
      CANDAT = 0x05;          /* enable acceptance filter 1 of bank1    */
                              /* enable acceptance filter 1 of bank2    */
      ..
      ..
      ..

/* Exit CAN controller Reset Mode
   ---------------------------------                                    */
      CANMOD = 0x00;          /* set the CAN controller into            */
                              /* operating mode                         */
}
```

3.7 Automatic Bit-rate Detection

The major drawback of existing trial and error concepts for automatic bit-rate detection is the generation of CAN error frames, which is not acceptable.
The CAN controller of the P8xC591 supports the requirements for automatic bit-rate detection with a unique 'Listen Only Mode' feature. In this mode the CAN controller does not generate error frames, only message reception is possible (true monitor function). This section briefly describes an application example without influencing running operations on the network. It requires that at least one node is actually sending CAN messages on the network. The number of messages needed to successfully detect the bit-rate varies depending on several factors, for example the initial test bit-rate and CAN bus load.
For automatic bit-rate detection a pre-defined table within the software contains all possible bit-rates including their bit-timing parameters [3]. Before starting message reception with the highest possible bit-rate, both the receive interrupt and the bus error interrupt are enabled. As soon as bus errors are detected the next lower bit-rate is selected. Whenever a receive interrupt occurs, the correct bit-rate is found. Now the device can be switched into Operating Mode. For verification purposes a 2nd CAN message should be received. From now on the CAN controller can participate on network communication. The example 'C' code listed on the next pages uses the initialisation from chapter 3.2.

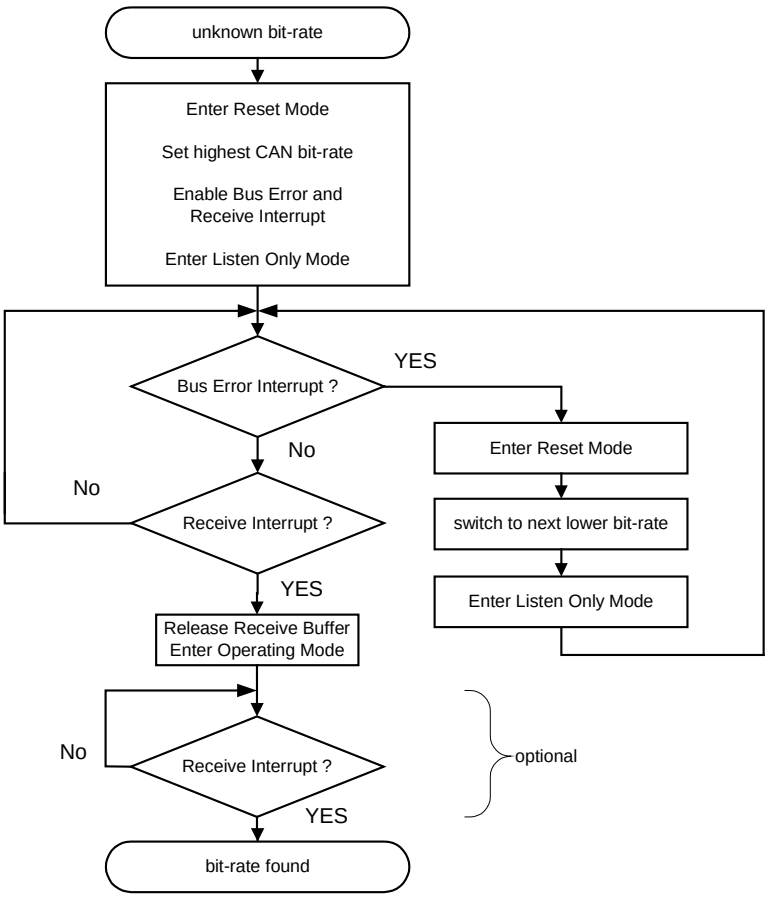

Figure 3-15: Algorithm of CAN Bit-rate Detection

Example 'C' code - Automatic Bit-rate Detection -

```
void auto_bit_rate_detection ( void )
{
  /* Global configurations:
   ----------------------                                              */
  EA = 1;                    /* enable global interrupt                */
  ECAN = 1;                  /* enable CAN interrupt                   */
  CANSTA = 0x81;             /* enable Receive & Bus Error Interrupts  */
  InterruptRegCopy = 0x00;   /* Buffer to store values of the
                                CAN Interrupt Register during the ISR  */
  try = 1;                   /* Counter for bit-rate configurations    */
  bitratefound = 0;          /* bit-rate found flag                    */
  errcnt = 0;                /* bus error counter                      */

  initialise_can_controller (); /* CAN Controller initialisation with
                                   highest bit-rate & Listen Only Mode  */

  /* CAN Controller starts with highest bit-rate
  ------------------------------------------------------------------- */
  do {
       if ((InterruptRegCopy & 0x80) == 0x80)  /* bus error interrupt   */
          {
            if (errcnt == error_max)   /* if error_max bus errors have  */
              {                        /* been detected switch to next  */
                try++;                 /* lower bit-rate                */
                change_bit_rate (try); /* change to lower bit-rate      */
                errcnt = 0;            /* Clear bus error counter       */
                if (try == 8)          /* if end of bit-rate table is   */
                    try = 1;           /* reached, restart trial        */
              }
            CANADR = ECC;              /* read error code capture and   */
            temp_reg1 = CANDAT;        /* enable for next bus error     */
            InterruptRegCopy = 0x00;   /* Clear InterruptRegBuffer      */
          }

       if ((InterruptRegCopy & 0x01) == 0x01 )  /* receive interrupt    */
          {
            bitratefound = 1;          /* Set bitratefound flag         */
            InterruptRegCopy = 0x00;   /* Clear InterruptRegBuffer      */
          }                            /* clear Rx interrupt            */
     } while (bitratefound == 0);      /* wait until bit-rate is found  */

  /* CAN Controller has detected the CAN system bit-rate
  ------------------------------------------------------------------- */

  CANMOD = 0x01;                     /* CAN contr. = reset mode         */
  CANMOD = 0x00;                     /* CAN contr. = operating mode     */

  /* CAN Controller is waiting for 2nd CAN frame (optional)
  ------------------------------------------------------------------- */

  do { } while ((InterruptRegCopy & 0x01) != 0x01);

} /* End of bit-rate detection */
```

Subroutine 'change_bit_rate'

```
void change_bit_rate (unsigned int try1)
{ /* CAN Bit Rate Table:
     -------------------                                                */
   CANMOD = 0x01; /* CAN controller = reset mode                        */
   switch (try1)
   {          /* Note: Maximum oscillator tolerance for each CAN node
                       is +/- 0.1%.
                       The following bus timing parameters for the
                       P8xC591 are based on XTAL=12MHz.                 */

      case 1:  CANADR = BTR0;  /* CAN Bit Rate 1000 kbit/s,             */
               CANDAT = 0x40;  /* SJW=2, BRP=1                          */
               CANADR = BTR1;  /* SP=83.3%, NBT=12                      */
               CANDAT = 0x18;  /* SAM=0, TSEG1=9, TSEG2=2               */
               break;
      case 2:  CANADR = BTR0;  /* CAN Bit Rate  800 kbit/s,             */
               CANDAT = 0x40;  /* SJW=2, BRP=1                          */
               CANADR = BTR1;  /* SP=86.6%, NBT= 15                     */
               CANDAT = 0x1B;  /* SAM=0, TSEG1=12, TSEG2=2              */
               break;
      case 3:  CANADR = BTR0;  /* CAN Bit Rate  500 kbit/s,             */
               CANDAT = 0x41;  /* SJW=2, BRP=2                          */
               CANADR = BTR1;  /* SP=83.3%, NBT= 12                     */
               CANDAT = 0x18;  /* SAM=0, TSEG1=9, TSEG2=2               */
               break;
      case 4:  CANADR = BTR0;  /* CAN Bit Rate  250 kbit/s,             */
               CANDAT = 0x42;  /* SJW=2, BRP=3                          */
               CANADR = BTR1;  /* SP=81.3%, NBT= 16                     */
               CANDAT = 0x2B;  /* SAM=0, TSEG1=12, TSEG2=3              */
               break;
      case 5:  CANADR = BTR0;  /* CAN Bit Rate  125 kbit/s              */
               CANDAT = 0x45;  /* SJW=2, BRP=6                          */
               CANADR = BTR1;  /* SP=81.3%, NBT= 16                     */
               CANDAT = 0x2B;  /* SAM=0, TSEG1=12, TSEG2=3              */
               break;
      case 6:  CANADR = BTR0;  /* CAN Bit Rate   50 kbit/s,             */
               CANDAT = 0x4B;  /* SJW=2, BRP=12                         */
               CANADR = BTR1;  /* SP=85%, NBT= 20                       */
               CANDAT = 0x2F;  /* SAM=0, TSEG1=16, TSEG2=3              */
               break;
       case 7: CANADR = BTR0;  /* CAN Bit Rate   20 kbit/s,             */
               CANDAT = 0x5D;  /* SJW=2, BRP=30                         */
               CANADR = BTR1;  /* SP=85%, NBT= 20                       */
               CANDAT = 0x2F;  /* SAM=0, TSEG1=16, TSEG2=3              */
               break;
       case 8: CANADR = BTR0;  /* CAN Bit Rate   10 kbit/s,             */
               CANDAT = 0x7B;  /* SJW=2, BRP=60                         */
               CANADR = BTR1;  /* SP=85%, NBT= 20                       */
               CANDAT = 0x2F;  /* SAM=0, TSEG1=16, TSEG2=3              */
               break;
      default: break;
   }
}
```

Interrupt Service Routine

```
/* ********************************************************************** */
/*      PROC : ECAN_int_service                                           */
/* ********************************************************************** */

void ECAN_int_service(void) interrupt 13 using 3 /* CAN ISR             */
{
   InterruptRegCopy = CANCON;   /* Store values from interrupt register */

   if ((InterruptRegCopy & 0x80) == 0x80)
     {
       errcnt++;                /* error counter is incremented         */
       if (errcnt == error_max)
         {
          CANMOD = 0x01;        /* Reset Mode  */
         }
     }
   if ((InterruptRegCopy & 0x01) == 0x01)
     {
      CANCON = 0x04;            /* Release Receive Buffer               */
     }

}  /* ----- End of CAN ISR -----                                          */
```

3.8 CAN Controller Self Tests

The CAN controller of the P8xC591 supports two different options for self tests:

- Global Self Test (setting the self reception request bit in **normal Operating Mode**)
- Local Self Test (setting the self reception request bit in **Self Test Mode**)

3.8.1 Global Self Test

A global self test is always performed in a running system. Figure 3-16 shows that at least one other CAN node must be connected onto the bus for giving an acknowledge.

This feature of the P8xC591 CAN controller is intended to be used, e.g., to verify proper operation of a certain CAN node within a system.

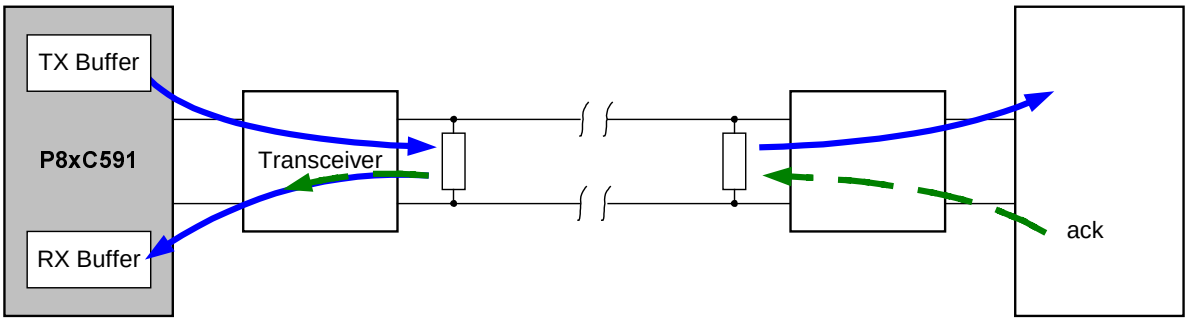

Figure 3-16: Global Self Test

As shown in the following 'C' code example the initiation of a global self test is similar to starting a normal CAN transmission. However, in this case the transmit message is also received and stored in the receive buffer of theP8xC591. Note that the acceptance filters have to be configured to receive this CAN message.

```
/* Global Self Test                                              */

     CANADR = TBF;                        /* point to 591 TX Buffer      */
     CANDAT = TransmitMessage[0];         /* write TX Frame Information  */
     CANDAT = TransmitMessage[1];         /* write TX Identifier 1       */
     CANDAT = TransmitMessage[2];         /* write TX Identifier 2       */
     ...
     ...
     CANCON = 0x10;                       /* Self Reception Request      */
     ...
```

Similar to the TX_Service routine presented in chapter 3.5, first the transmit buffer is loaded followed by the **Self Reception Request** command.

As soon as the CAN frame is transmitted completely, both a transmit interrupt and a receive interrupt are generated, if enabled.

3.8.2 Local Self Test

A local self test, e.g., can be used perfectly for single node tests because an acknowledge from other nodes is not needed. In this case the CAN controller has to be in '**Self Test Mode**'. Note that a physical layer interface must be available including a CAN bus line and a termination network as shown in Figure 3-17. Entering the Self Test Mode is possible only if the Reset Mode is entered previously.

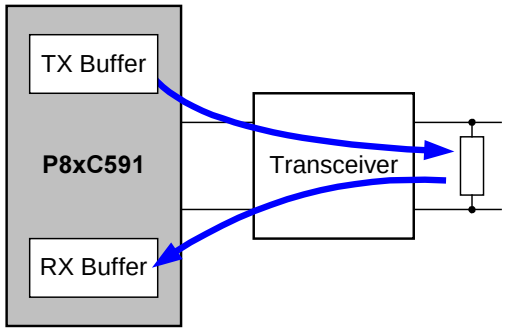

Figure 3-17: Local Self Test

As already described for the global self test, the transmission is also started with the **Self Reception Request** command (see 'C' code example).

```
/* Local Self Test                                                    */

     ..

     CANMOD = 0x01;                    /* Enter Reset Mode            */
     CANMOD = 0x04;                    /* Enter Self Test Mode        */

     CANADR = TBF;                     /* point to 591 TX Buffer      */
     CANDAT = TransmitMessage[0];      /* write TX Frame Information  */
     CANDAT = TransmitMessage[1];      /* write TX Identifier 1       */
     CANDAT = TransmitMessage[2];      /* write TX Identifier 2       */
     ...
     ...
     CANCON = 0x10;                    /* Self Reception Request      */
     ...
```

4. REFERENCES

[1] Data Sheet P8xC591, Philips Semiconductors, 2000

[2] CAN Specification Version 2.0, Parts A and B, Philips Semiconductors, 1992

[3] Jöhnk, E. and Dietmayer, K.: Application Note: Determination of Bit Timing Parameters for the CAN Controller SJA1000, Philips Semiconductors, 1997

[4] Hank, P. and Jöhnk, E.: Application Note: SJA 1000 Stand-alone CAN Controller, Philips Semiconductors, 1997

[5] Data Handbook IC28: 80C51 and XA Microcontrollers, 2000

[6] Data Handbook IC18: Semiconductors for In-Car Electronics, SJA1000 - Stand-alone CAN controller, 2000

[7] DeviceNet Protocol Specification Version 2.0, Open DeviceNet Vendors Association,1997

5. APPENDIX

```
/*****************************************************************************/
/*                                                                           */
/* Project  : P8xC591 Application Note AN00043                               */
/* Author: Hartmut Habben PHILIPS Semiconductors -SLHamburg                  */
/*                                                                           */
/* TITLE: Definition for the CAN controller of the P8xC591                   */
/*                                                                           */
/* 1.   Name: C591_def.h                                                     */
/*                                                                           */
/* 2.   Modification History:                                                */
/*      H.Habben, June 27, 2000    #1.000,      initial version              */
/*                                                                           */
/* 3.   NOTICE: Copyright (C) 2000 PHILIPS Semiconductors                    */
/*                                                                           */
/* 4    Build Environment:                                                   */
/*      Keil Software Development Tools for 8051, C51, version 5.20          */
/*                                                                           */
/*****************************************************************************/

#define RIL                     0x05  // Receive Interrupt Level
#define BTR0                    0x06  // Bus Timing 0
#define BTR1                    0x07  // Bus Timing 1
#define RMC                     0x09  // Receive Message Counter
#define RBSA                    0x0A  // Receive Buffer Start Address
#define ALC                     0x0B  // Arbitr. Lost Capture
#define ECC                     0x0C  // Error Code Capture
#define EWLR                    0x0D  // Error Warning Limit
#define RXERR                   0x0E  // Rx Error Counter
#define TXERR                   0x0F  // Tx Error Counter
#define ACFMOD                  0x1D  // ACF Mode
#define ACFEN                   0x1E  // ACF Enable
#define ACFPRIO                 0x1F  // ACF Priority
#define ACR10                   0x20  // Acceptance Code 0 (Bank 1)
#define AMR10                   0x24  // Acceptance Mask 0 (Bank 1)
#define ACR20                   0x28  // Acceptance Code 0 (Bank 2)
#define AMR20                   0x2C  // Acceptance Mask 0 (Bank 2)
#define ACR30                   0x30  // Acceptance Code 0 (Bank 3)
#define AMR30                   0x34  // Acceptance Mask 0 (Bank 3)
#define ACR40                   0x38  // Acceptance Code 0 (Bank 4)
#define AMR40                   0x3C  // Acceptance Mask 0 (Bank 4)
#define RBF                     0x60  // Receive Buffer
#define TBF                     0x70  // Transmit Buffer

/* local type definition                                                     */

#define BYTE      unsigned char
#define WORD      unsigned int
#define DWORD     unsigned long
```

```
/*****************************************************************************/
/*                                                                           */
/* Project  : P8xC591 Application Note AN00043                               */
/* Author: Hartmut Habben PHILIPS Semiconductors -SLHamburg                  */
/*                                                                           */
/* TITLE: Header file with Special Function Register Declarations            */
/*        for the Philips P8xC591 CAN Microcontroller                        */
/*                                                                           */
/* 1.   Name: REG591.h                                                       */
/*                                                                           */
/* 2.   Modification History:                                                */
/*      H.Habben, June 27, 2000    #1.000,      initial version              */
/*                                                                           */
/* 3    NOTICE: Copyright (C) 2000 PHILIPS Semiconductors                    */
/*                                                                           */
/* 4    Build Environment:                                                   */
/*      Keil Software Development Tools for 8051, C51, version 5.20          */
/*                                                                           */
/*****************************************************************************/

/*  BYTE Register  */
sfr P0     = 0x80; // Port 0
sfr P1     = 0x90; // Port 1
sfr P2     = 0xA0; // Port 2
sfr P3     = 0xB0; // Port 3


sfr PSW    = 0xD0; // Program Status Word
sfr ACC    = 0xE0; // Accumulator
sfr AUXR   = 0x8E; // Auxiliary
sfr AUXR1  = 0xA2; // Auxiliary
sfr B      = 0xF0; // B register
sfr SP     = 0xB9; // Stack Pointer
sfr DPL    = 0x82; // Data Pointer High
sfr DPH    = 0x83; // Data Pointer Low
sfr PCON   = 0x87; // Power Control
sfr TCON   = 0x88; // Timer Control
sfr TMOD   = 0x89; // Timer Mode
sfr TL0    = 0x8A; // Timer Low 0
sfr TL1    = 0x8B; // Timer Low 1
sfr TH0    = 0x8C; // Timer High 0
sfr TH1    = 0x8D; // Timer High 1
sfr IEN0   = 0xA8; // Interrupt Enable 0
sfr IEN1   = 0xE8; // Interrupt Enable 1
sfr IP0    = 0xB8; // Interrupt Priority 0
sfr IP0H   = 0xB7; // Interrupt Priority 0 high
sfr IP1    = 0xF8; // Interrupt Priority 1
sfr IP1H   = 0xF7; // Interrupt Priority 1 high
sfr S0ADDR = 0xCB; // Serial 0 Slave Address
sfr S0ADEN = 0xF9; // Serial 0 slave Address Mask
sfr S0CON  = 0x98; // Serial 0 Control
sfr S0BUF  = 0x99; // Serial 0 Data Buffer
sfr S1ADR  = 0xDB; // Serial 1 Address
sfr S1STA  = 0xD9; // Serial 1 Status
sfr S1CON  = 0xD8; // Serial 1 Control
sfr S1DAT  = 0xDA; // Serial 1 Data
```

```
sfr ADCON  = 0xC5; // A/D Control
sfr ADCH   = 0xC6; // A/D Converter high
sfr CTCON  = 0xEB; // Capture Control
sfr CTH3   = 0xCF; // Capture high 3
sfr CTH2   = 0xCE; // Capture high 2
sfr CTH1   = 0xCD; // Capture high 1
sfr CTH0   = 0xCC; // Capture high 0
sfr CMH2   = 0xCB; // Compare high 2
sfr CMH1   = 0xCA; // Compare high 1
sfr CMH0   = 0xC9; // Compare high 0
sfr CTL3   = 0xAF; // Capture low 3
sfr CTL2   = 0xAE; // Capture low 2
sfr CTL1   = 0xAD; // Capture low 1
sfr CTL0   = 0xAC; // Capture low 0
sfr CML2   = 0xAB; // Compare low 2
sfr CML1   = 0xAA; // Compare low 1
sfr CML0   = 0xA9; // Compare low 0

/* ------ CAN Special Function Registers ------- */

sfr CANADR = 0xC1; // CAN Address
sfr CANDAT = 0xC2; // CAN Data
sfr CANMOD = 0xC4; // CAN Mode Register
sfr CANSTA = 0xC0; // CAN Status  / Interrupt Enable
sfr CANCON = 0xC3; // CAN Command / Interrupt Register
/* ------------------------------------------- */

sfr P1M1   = 0x92; // Port 1 output mode 1
sfr P1M2   = 0x93; // Port 1 output mode 2
sfr P2M1   = 0x94; // Port 2 output mode 1
sfr P2M2   = 0x95; // Port 2 output mode 2
sfr P3M1   = 0x94; // Port 3 output mode 1
sfr P3M2   = 0x95; // Port 3 output mode 2
sfr PWMP   = 0xFE; // PWM Prescaler
sfr PWMP1  = 0xFD; // PWM Register 1
sfr PWMP0  = 0xFC; // PWM Register 0
sfr RTE    = 0xEF; // Reset Enable
sfr STE    = 0xEE; // Set Enable
sfr TMH2   = 0xED; // Timer high 2
sfr TML2   = 0xEC; // Timer low 2
sfr TM2IR  = 0xC8; // Timer 2 Int Flag Reg
sfr T3     = 0xFF; // Timer 3
```

```
/*  ------------------ BIT addressable registers -------------------------- *
/
/*  PSW         */                      /*  IP1         */
sbit CY    = 0xD7;                      sbit PT2   = 0xFF;
sbit AC    = 0xD6;                      sbit PCAN  = 0xFE;
sbit F0    = 0xD5;                      sbit PCM1  = 0xFD;
sbit RS1   = 0xD4;                      sbit PCM0  = 0xFC;
sbit RS0   = 0xD3;                      sbit PCT3  = 0xFB;
sbit OV    = 0xD2;                      sbit PCT2  = 0xFA;
sbit F1    = 0xD1;                      sbit PCT1  = 0xF9;
sbit P     = 0xD0;                      sbit PCT0  = 0xF8;

/*  TCON        */                      /*  P1          */
sbit TF1   = 0x8F;                      sbit P1_7  = 0x97;
sbit TR1   = 0x8E;                      sbit P1_6  = 0x96;
sbit TF0   = 0x8D;                      sbit P1_5  = 0x95;
sbit TR0   = 0x8C;                      sbit P1_4  = 0x94;
sbit IE1   = 0x8B;                      sbit P1_3  = 0x93;
sbit IT1   = 0x8A;                      sbit P1_2  = 0x92;
sbit IE0   = 0x89;                      sbit P1_1  = 0x91;
sbit IT0   = 0x88;                      sbit P1_0  = 0x90;

/*  IEN0        */                      /*  P2          */
sbit EA    = 0xAF;                      sbit P2_7  = 0xA7;
sbit EAD   = 0xAE;                      sbit P2_6  = 0xA6;
sbit ES1   = 0xAD;                      sbit P2_5  = 0xA5;
sbit ES0   = 0xAC;                      sbit P2_4  = 0xA4;
sbit ET1   = 0xAB;                      sbit P2_3  = 0xA3;
sbit EX1   = 0xAA;                      sbit P2_2  = 0xA2;
sbit ET0   = 0xA9;                      sbit P2_1  = 0xA1;
sbit EX0   = 0xA8;                      sbit P2_0  = 0xA0;

/*  IEN1        */                      /*  P3          */
sbit ET2   = 0xEF;                      sbit P3_7  = 0xB7;
sbit ECAN  = 0xEE;                      sbit P3_6  = 0xB6;
sbit ECM1  = 0xED;                      sbit P3_5  = 0xB5;
sbit ECM0  = 0xEC;                      sbit P3_4  = 0xB4;
sbit ECT3  = 0xEB;                      sbit P3_3  = 0xB3;
sbit ECT2  = 0xEA;                      sbit P3_2  = 0xB2;
sbit ECT1  = 0xE9;                      sbit P3_1  = 0xB1;
sbit ECT0  = 0xE8;                      sbit P3_0  = 0xB0;

/*  IP0         */                      /*  P3 alternate functions */
                                        sbit rd    = 0xB7;
sbit PAD   = 0xBE;                      sbit wr    = 0xB6;
sbit PS1   = 0xBD;                      sbit T1    = 0xB5;
sbit PS0   = 0xBC;                      sbit T0    = 0xB4;
sbit PT1   = 0xBB;                      sbit int1  = 0xB3;
sbit PX1   = 0xBA;                      sbit int0  = 0xB2;
sbit PT0   = 0xB9;                      sbit TXD   = 0xB1;
sbit PX0   = 0xB8;                      sbit RXD   = 0xB0;
```

```
/*  S0CON       */
sbit SM0   = 0x9F;
sbit SM1   = 0x9E;
sbit SM2   = 0x9D;
sbit REN   = 0x9C;
sbit TB8   = 0x9B;
sbit RB8   = 0x9A;
sbit TI    = 0x99;
sbit RI    = 0x98;

/*  S1CON       */

sbit CR0   = 0xD8;
sbit CR1   = 0xD9;
sbit AA    = 0xDA;
sbit SI    = 0xDB;
sbit STO   = 0xDC;
sbit STA   = 0xDD;
sbit ENS1  = 0xDE;
sbit CR2   = 0xDF;

/*  Accumulator  */
sbit ACC_7  = 0xE7;
sbit ACC_6  = 0xE6;
sbit ACC_5  = 0xE5;
sbit ACC_4  = 0xE4;
sbit ACC_3  = 0xE3;
sbit ACC_2  = 0xE2;
sbit ACC_1  = 0xE1;
sbit ACC_0  = 0xE0;
```

```
/*  TM2IR       */
sbit T20V   = 0xCF;
sbit CMI2   = 0xCE;
sbit CMI1   = 0xCD;
sbit CMI0   = 0xCC;
sbit CTI3   = 0xCB;
sbit CTI2   = 0xCA;
sbit CTI1   = 0xC9;
sbit CTI0   = 0xC8;

/*  CANSTA      */
/*  (READ)      */
sbit BS     = 0xC7;
sbit ES     = 0xC6;
sbit TS     = 0xC5;
sbit RS     = 0xC4;
sbit TCS    = 0xC3;
sbit TBS    = 0xC2;
sbit DOS    = 0xC1;
sbit RBS    = 0xC0;

/*  (WRITE)     */
sbit BEIE   = 0xC7;
sbit ALIE   = 0xC6;
sbit EPIE   = 0xC5;
sbit WUIE   = 0xC4;
sbit DOIE   = 0xC3;
sbit EIE    = 0xC2;
sbit TIE    = 0xC1;
sbit RIE    = 0xC0;
```