Application Note, V 1.0, Feb. 2004

AP29002

82C900

Standalone TwinCAN Controller

Microcontrollers

Never stop thinking.

Revision History:	2004-02	**V 1.0**
Previous Version:	-	
Page	Subjects (major changes since last revision)	
All	Updated Layout to Infineon Corporate Design, updated release to 1.0, Content unchanged!	

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Edition 2004-02-01

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents **Page**

1 Product Overview

The TwinCAN module in the 82C900 permits the connection and autonomous handling of two independent CAN buses. It supports the maximum CAN baud rate of 1M baud and complies with CAN protocol version 2.0B (active).

The full-CAN module has 32 message objects, which can be independently assigned to one of the two buses. The scalable FIFO mechanism for reception and transmission greatly improves the real-time behavior of the system. The device provides built-in automatic gateway functionality for data exchange between both CAN buses, minimizing the CPU's message handling load. The gateway feature can also be used for automatic reply to received messages.

The powerful interrupt structure permits application-specific interrupt generation. The enhanced acceptance filtering dedicates one acceptance mask to each message object. In addition, a 16-bit frame count/time-stamp is implemented for each message object. The 82C900's analyzing mode, during which no dominant bits are sent, makes it ideal for diagnostic applications.

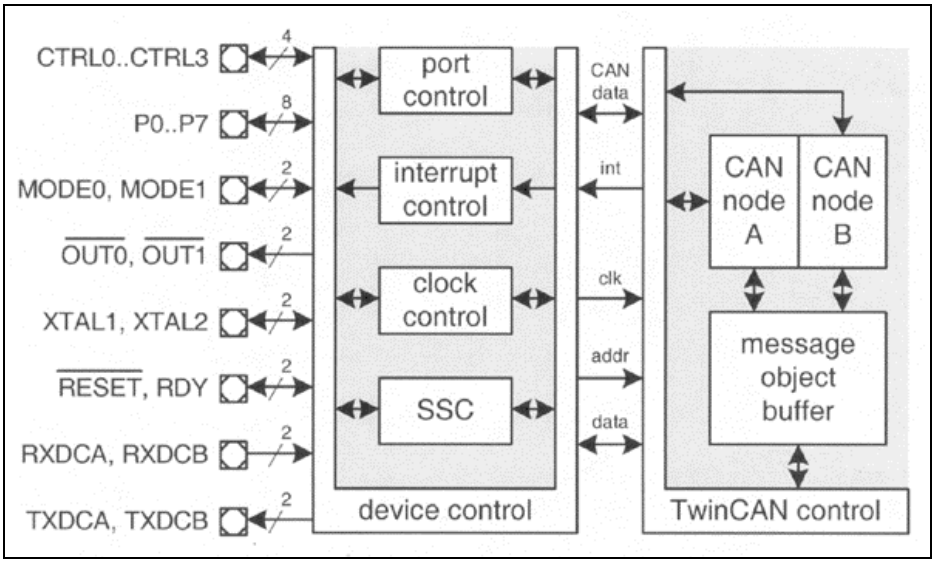

Figure 1 82C900 Architectural Overview

The Infineon 82C900 comes with many useful features, such as:

- Handles two independent CAN buses autonomously
- 32 message objects can be independently assigned to one of the two buses
- Complies with CAN protocol version 2.0B (active)
- Baud rate up to 1Mbaud
- Scalable FIFO mechanism
- Built-in automatic gateway functionality
- Remote frame monitoring
- One acceptance filtering mask per message object
- 16 bit frame count/time-stamp per message object
- Analyzing mode
- Parallel and serial interface
- Can be configured via external serial EEPROM
- Clock output pin
- Power saving features
- 28-pin P-DSO package
- Temperature range -40° to +125° C

The Standalone TwinCAN device provides two interface channels for communication with a host microcontroller: a multiplexed address/data bus and a synchronous serial channel (SSC). The SSC may also be used to read out the initial TwinCAN register configuration from a serial EEPROM without requiring a host microcontroller. A Sleep Mode and a Power-Down Mode can be activated in order to minimize power consumption. The device's clock can be controlled by CAN messages.

2 CAN protocol overview

CAN is an asynchronous serial bus system with one logical bus line. A CAN bus consists of two or more nodes. The number of nodes on the bus may be changed dynamically without disturbing the communication of other nodes. This allows easy connection and disconnection of bus nodes (e.g. node for software upgrade, bus monitoring…).

The bus logic corresponds to a “wired-AND” mechanism, recessive bits are overwritten by dominants bits. As long as no bus node is sending a dominant bit, the bus line is in the recessive state, but a dominant bit from any node generates the dominant Bus State.

The maximum bus speed is 1Mbaud, which can be achieved with a bus length of up to 40m. At least 30 nodes may be connected without additional equipment.

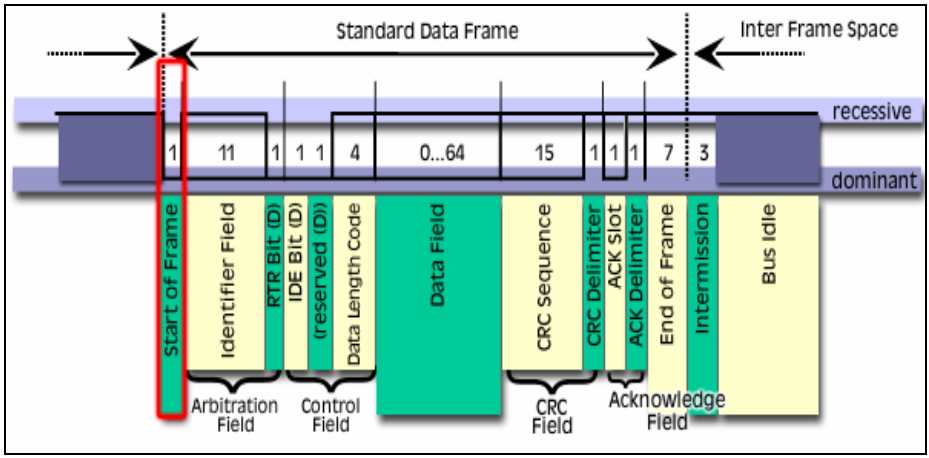

Figure 2 Structure of a Standard data Frame generated by a CAN node

In this kind of frame, the identifier field has a length up to 11 bits giving 2048 message identifiers. But “extended CAN” specifications allow a 29 bits identifier field giving 536 Million messages identifiers. In this application note, only “standard CAN” will be used but C167CR and CS also support “extended CAN” specifications.

For bus arbitration, Carrier Sense Multiple Access / Collision Detection with Non-Destructive Arbitration is used. It means that if a node wants to transmit a message, it checks first that the bus is in the idle state (“Carrier Sense”). If this is the case the node becomes the bus master and sends its message. If many nodes start their transmission at the same time (“Multiple Access”) collision is avoided by bit wise arbitration (Collision Detection with Non-Dest’ructive Arbitration).

Each node sends the bits of its message identifier and monitors the bus level. A node which sends a recessive identifier bit but reads back a dominant one loses bus arbitration and switches to receive mode.

The CAN protocol allows a high data integrity with several means for error checking and can detect:

- Cyclic Redundancy Check (CRC) Errors
- Acknowledge Errors
- Form Errors
- Bit Errors
- Stuff Errors

3 Using special features of the 82C900

The following chapter contains descriptions and example applications to all the important features of the 82C900. The examples are all developed using the 82C900 Starter Kit, Dave 2.1 and the Keil-compiler. The examples that require a host interface use the C167CR Starter Kit. The clock frequency of the crystal on the 82C900 Starter Kit is 24 MHz.

3.1 Using the 82C900 with SSC interface

The SSC interface provides an economical way to connect the 82C900 to a microcontroller. The SSC interface is developed by Infineon and is 100% SPI compatible. Basically there are only 5 port pins necessary for communication via SSC (plus one for each interrupt you want to use). The default baud rate of the SSC interface on the 82C900 is 100 KBaud. But it can work with a speed up to 6.25 MBaud in slave- and 12.5 MBaud in master mode.

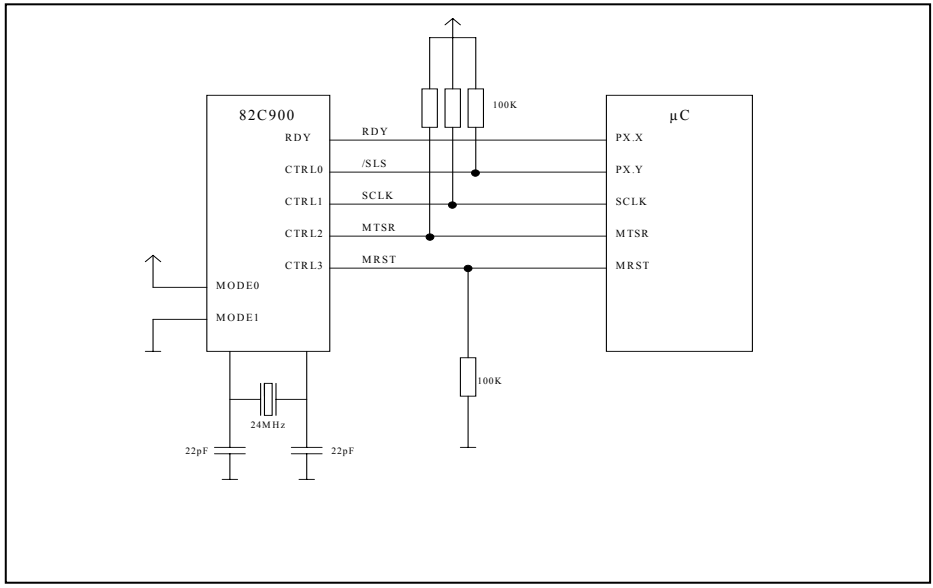

Figure 3 Important connections for the SSC-Data transfer. For RDY and SLS, any available Port of the microcontroller can be used.

An example using the SSC Slave mode of the 82C900 was implemented using the 82C900 evaluation board and an Infineon C167CR as the host microcontroller. The 82C900 evaluation board was connected directly to the Phytec KitCON167 evaluation board for the C167CR.

Note: Please note that this example can also be implemented on many other Infineon Microcontrollers, such as the C161PI. If you want to use the Phytec C161O KitCon board, the board needs to be connected with wires because P3.4 of the KitCon board is used for other functions. So another port pin has to be used for the /RDY signal. But if wires are used noise can be a problem, if the wires are to long or the connectors are not good enough.

The jumper configurations on the 82C900 starterkit are shown in Table 1.

Table 1 Jumper settings for SSC-Slave-Mode

JP10	1-2 connected: SLS on CTRL0
JP32	Open (MODE0 = ‘1’)
JP33	Close (MODE1 = ‘0’)
JP38	All open
JP39	3-7 closed, 1-2 depending which interrupts are used 8 open
JP40	All open

The following example sets up the SSC interface of an Infineon 16 bit microcontroller. The code was actually tested on the C167CR and C161PI microcontrollers (on Phytec KitCon evaluation boards). The Code is generic and should work on most other Infineon 16-bit controllers.

```
void SSC_Init(void)
{
        SSCEN = 0;
        // SSC baudrate generator/reload register
        // baudrate = 100 KBaud
        SSCBR = 0x0031;

        // load P3.13 output latch (SCLK) with the
        //  desired clock idle level(high)
        P3 |= 0x2000;

        // set P3.13 direction control (SCLK output)
        DP3  |= 0x2000;

        // ---------- SSC Control Register ----------
        ///  Master Mode
        ///  transfer data width is 8 bit
        ///  transfer/receive MSB first
        ///  shift transmit data on the leading clock edge, latch on
        /// trailing edge
        ///  idle clock line is high, leading clock edge is
        /// high-to-low transition
        ///  ignore receive error
        ///  ignore phase error
        SSCCON = 0xC057;

        DP3  &= 0xFEFF;        // reset P3.8 dir control (MRST input)
        P3   |= 0x0200;        //  set P3.9 output latch (MTSR)
        DP3  |= 0x2240;        // set P3 dir control
}
```

As you can see from the source code, there is nothing special about the SSC initialization.

For transferring data to a register we need to specify its 11-bit address. But since the SSC interface on the 82C900 is only able to handle 8 bits at a time and one of these is used for Read/Write indication, the rest of the address has to be specified in a different way. Therefore, when using the SSC interface, the 82C900's address space is divided into pages. Every page is 128 bytes wide.

So before being able to transfer data to a register, we need to select the page first:

```
void SSC_SetPage(byte PageNumber)
{
        SLS = 0;                // set SLS=0, new communication sequence
        while(RDY==0){};        // wait for RDY==1 (82C900 is ready)

        SSCRIR = 0;             // reset receive interrupt request
        SSCTB = PAGE_WRITEMASK | PAGE;
        // mask for write, load transmit buffer register with
        // PAGE register address

        while(!SSCRIR){};       // wait for end of transmission
        while(RDY==0){};        // wait for RDY==1 (82C900 is ready)

        SSCRIR = 0;             // reset receive interrupt request
        SSCTB = PageNumber;
        // load transmit buffer register with pagenumber

        while(!SSCRIR){};       // wait for end of transmission
        while(RDY==0){};        // wait for RDY==1 (82C900 is ready)
        SLS = 1;                // set SLS=1, communication sequence end
}
```

The 82C900 extends the SSC interface by a ready signal (RDY). This is necessary, because both CAN and SSC interfaces use the same memory area. As long as there is CAN access to the registers, the SSC will be suspended by using the RDY signal to avoid collisions.

The following C function transfers 1 data byte to the address specified as a parameter:

```
void SSC_SendByte(word StartAddress, byte Data)
{
        SSC_SetPage(StartAddress >> 7);
        // extract 4 MSBs (= page number) from StartAddress

        SLS = 0;                // set SLS=0, new communication sequence
        while(RDY==0){};        // wait for RDY==1 -> 82C900 is ready

        SSCRIR = 0;             // reset receive interrupt request
        SSCTB = PAGE_WRITEMASK | StartAddress;
         // mask for write, load transmit buffer register
         // with StartAddress

        while(!SSCRIR){};       // wait for end of transmission
        while(RDY==0){};        // wait for RDY==1 -> 82C900 is ready

        SSCRIR = 0;             // reset receive interrupt request
        SSCTB = Data;  // load transmit buffer register with Data
        while(!SSCRIR){};       // wait for end of transmission

        while(RDY==0){};        // wait for RDY==1 -> 82C900 is ready
        SLS = 1;                // set SLS=1, communication sequence end
}
```

First the four MSB bits are extracted from the address as they specify the page. Then the page is set in the 82C900's `PAGE` register using the `SSC_SetPage` function described before. After that, the write indication bit is added to the remaining 7 address bits and address byte is being transferred via the SSC. Then the data byte is transferred completing the communication sequence.

This procedure is the same for all Infineon microcontrollers containing a SSC peripheral.

Reading a byte from the 82C900 works similar to writing:

First the four MSB bits are extracted from the address as they specify the page. Then the page is set in the 82C900's `PAGE` register using the `SSC_SetPage` function described before.

After that the address and a dummy byte are transferred to the 82C900. The C function shown below the returns the data byte read from the 82C900

```
byte SSC_ReadByte(word StartAddress)
{
        SSC_SetPage(StartAddress >> 7);
          // extract 4 MSBs (= page number) from StartAddress
        StartAddress &= PAGE_READMASK;  // mask for read
        SLS = 0;                // set SLS=0, new communication sequence
        while(RDY==0){};        // wait for RDY==1 -> 82C900 is ready

        SSCRIR = 0;             // reset receive interrupt request
        SSCTB = StartAddress;

          // load transmit buffer register with StartAddress
        while(!SSCRIR){};   // wait for end of transmission

        while(RDY==0){};    // wait for RDY==1 -> 82C900 is ready

        SSCRIR = 0;         // reset receive interrupt request
        SSCTB = StartAddress;
          // load transmit buffer register with StartAddress (dummy)
        while(!SSCRIR){};       // wait for end of transmission

        while(RDY==0){};        // wait for RDY==1 -> 82C900 is ready
        SLS = 1;                // set SLS=1, communication sequence end

        return SSCRB;           // return received byte
}
```

Application Example

With the SSC data transfer routines shown in chapter 3.1 you are able to create a CAN application with the 82C900. The application example sets up two CAN message objects, one for receiving and one for transmitting.

As soon as a CAN message that has the same CAN identifier as this receive object is received, an interrupt service routine is called. This interrupt service routine copies the data received from the CAN message to the transmit object and transmits it on the same CAN node. The source code for this example is in the SSC directory of the code for this ApNote.

```
void main(void)
{
        Project_Init();
        SLS = 1;                        // slave not selected

        // set up CAN node A, 1 Mbaud

        SSC_SendByte(ACR, 0x41);              // set bit CCE in ACR
        SSC_SendByte(ABTR+1, 0x23);
          // set DIV8X=0, TSEG2=3, TSEG1=4
        SSC_SendByte(ABTR, 0x02);             // set SJW=1, BRP=2
        SSC_SendByte(AIMR0, 0x01);
        // take IMC0 into account for INTID generation
        SSC_SendByte(ACR, 0x00);              // clear INIT, CCE in ACR
```

This part of the code initializes the microcontroller and sets up CAN node A in the 82C900 for operation at 1 MBaud. To be able to change the configuration, bit CCE in ACR must be set and then be reset again after the configuration has been changed.

As a next step, the CAN message object for receiving must be set up. This is also done by configuring a number of registers in the 82C900, the ‘CAN Message Object Registers’:

```
//set up CAN message object 0 (RX), node A, interrupt node 1
// /OUT1 (associated with interupt node 1) is connected to P2.11 calling
the ISR when message object is received

        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0x7F);
          // set MSGVAL to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xFD);
          // set INTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0x7F);
          // set RMTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xDF);
          // set TRXQ to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xFD);
          // set NEWDAT to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCFGn,0x10);
          // select Node A, set DIR=0 (receive), 1 data byte (DLC=1)
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCFGn+2,0x01);
          // trigger interrupt node 1 when receiving message
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xF7);
          // set MSGLST to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xFB);
          // set RXIE, receive interrupt is enabled
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGARn+2,0x04);
          // identifier of receive object (0) is 0x0001
        SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xBF);
          // set MSGVAL, message object is valid
```

To be able to configure the message object, it must first be made invalid by resetting `MSGVAL` in register `MSGCTRn`. Then you can configure the message object, set the data direction, data length and associated interrupt node.

In the 82C900 there are two interrupt pins available, that can be associated with up to 4 interrupt nodes each by configuring the `GLOBCTR` register. Then you can associate the single message objects with one interrupt node each. The corresponding interrupt pin will be triggered, when the associated event occurs.

In this application the receive interrupt is enabled and the identifier of the receive message object is set to 0x0001. After that the message is set valid.

Now the transmit message object is set up by setting the corresponding register in the 82C900. It works similar to the receive object, only the data direction and identifier are different and no interrupt node is needed.

```
//set up CAN message object 1 (TX), node A

        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0x7F);
                                        // set MSGVAL to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0xFD);
                                        // set INTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0x7F);
                                        // set RMTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xDF);
                                          // set TRXQ to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCFGn,0x18);
                       // clear NODE in MSGCFGn, set DIR=1 (transmit),
                       // 1 data byte (DLC=1)
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGARn+2,0x08);
                                             // identifier to 0x0002
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xF7);
                                        // set CPUUPD to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xFE);
                                        // set NEWDAT to '10' => set
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0xBF);
                                        // set MSGVAL to '10' => set

        while (1){};
}
```

```
void INT_viIsrEx3(void) interrupt CC11INT
{
        unsigned char databuffer, checkTXRQ;
        databuffer = SSC_ReadByte(MSG_OBJ_BASE(0)+MSGDRn0);
          // get data from received message
        checkTXRQ = SSC_ReadByte(MSG_OBJ_BASE(1)+MSGCTRn+1) | 0x10;
        // checkTXRQ holds LSB of TXRQ

        if(checkTXRQ)
        // TXRQ='01', no transmission requested, ok to send
        {
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xFB);
                                // set CPUUPD
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGDRn0,databuffer);
                          // put data into transmit message object
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xF7);
                                // reset CPUUPD
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xEF);
                                // set TRXQ to '10' => set
        }
        else
        }
                // TXRQ='10', transmission requested, not ok to send
                {
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0x7F);
                          // set MSGVAL to '01' => reset
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGDRn0,databuffer);
                          // put data into transmit message object
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xF7);
                          // reset CPUUPD
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0xBF);
                         // set MSGVAL to '10' => set
                SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xEF);
                         // set TRXQ to '10' => set
        }
}
```

3.2 Using the parallel interface of the 82C900

The 82C900 can also be configured via parallel interface. The host CPU must support an 8-Bit Infineon or Motorola compatible multiplexed bus and 4 control signals: /ALE, /CS, /WR and /RD for an Infineon compatible parallel bus.

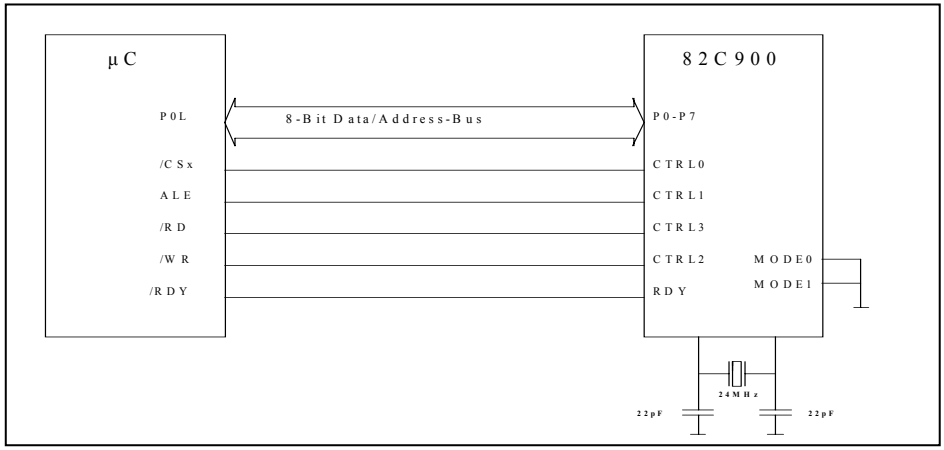

Figure 4 Schematic for the parallel data transfer

In this example, the C167CR is used as the host microcontroller, with the KitCon167 evaluation board. This board can be connected directly to the 82C900 Starter Kit via the KitCon-connector. CS3 from the C167CR is used as the chip select signal to the 82C900. Therefore the ADDRSEL3 and BUSCON3 registers of the C167CR are configured with the appropriate bus settings.

Note: Connecting the 82C900 directly to the kitCON161PI evaluation board from Phytec will not work due to an address decoder on this board which modifies the bus signals.

Table 2 shows the jumper settings for the 82C900 evaluation board in this example.

Table 2 Jumper Settings for Parallel-Mode

JP10	1-2 connected: /CS on CTRL0
JP32	Close (MODE0 = '0')
JP33	Close (MODE1 = '0')
JP38	All close
JP39	1-2 depending which interrupts are used 3-8 open
JP40	3-6 close(with using /CS3) others open
JP42	2-3 closed

The 82C900 is configured for a long read access by setting bit LAE in the GLOBCTR-Register to '1' (default '0').

Configurations for the /CS3 signal on the C167CR

For this example a 4K memory window with start address 0x200000 is used. In this address range the address/data-bus is an 8-Bit multiplexed bus. To select the 82C900 /CS3 is used. To fulfill the timing for long read access the following important configurations must be made for BUSCON3 (f_{CPU}=20MHz for the C167CR, f_{82C900}=24MHz).

Table 3 Startup file configuration

_MCTC3 = 7	7 Waitstates (/RD at least 360ns)
_RWDC3 = 0	/RD/WR delay at least 10ns
_BTYP3 = 1	8-Bit multiplexed bus
_CSREN3/_CSWEN3 = 1	CS is dependent of RD/WR signal

To reach the 82C900 timing specification for t_{whlh} (/WR high to next ALE) for C167CR f_{CPU} > 5MHz, the functions for the write access of the 82C900 must be copied into the IRAM of the microcontroller. Therefore a user class must be defined with the following entries in the target options window of the Keil-Compiler.

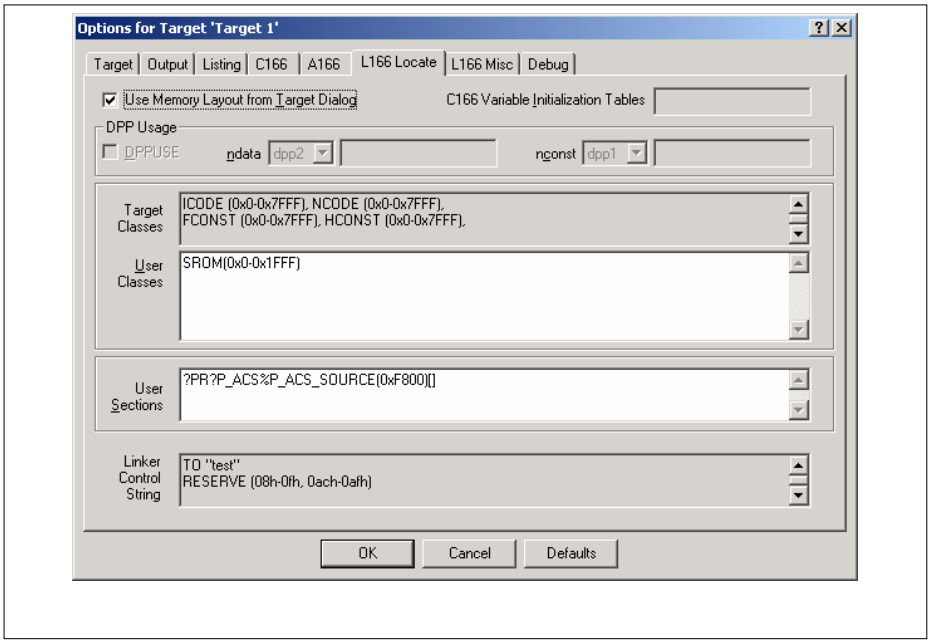

Figure 5 Keil Linker Settings for SROM section

The ?PR?_ACS section must be defined as an SROM section in the User Sections text box. In this example the section ?PR?P_ACS and the class P_ACS_SOURCE are assigned to the execution address 0xF800 and a storage address within the SROM section. Their address range is specified in the User Class text box.

For copying and executing some parts of the code out of the IRAM some macros and the hmemcpy(…) function are needed. They are defined in the SROM.h and string.h header files. A more detailed description of the macros are available in the Keil Application Note 138 on the Keil website.

Functions to access the 82C900 via parallel bus:

The variable _82C900 is defined as:

```
#define  _82C900  ((unsigned char volatile xhuge*) 0x200000)
```

It is the base address of the 82C900 with the described configuration for BUSCON3.

Next is the source code of the functions to access the 82C900.

The following functions will be copied into the IRAM. They are located in the file P_ACS.C (short for “Parallel Access”). It is important that the name of the file is the same as the user section. The next statement must be placed before the code that is copied into the IRAM.

```
#pragma RENAMECLASS (NCODE=P_ACS_SOURCE)

// read data from adr
unsigned char P_Read(byte adr)
{
        unsigned char ret;
        _atomic_(1);// no interrupts during transfer
        IEN = 0;// disable interrupts
        ret = *(_82C900+adr);// read access to the 82C900
        IEN = 1;// enable interrupts again
        return ret;
}

// write data to adr
void P_Write(byte adr, byte data)
{
        _atomic_(1);// no interrupts during transfer
        IEN = 0;// disable interrupts
        *(_82C900+adr) = data; // write access to the 82C900
        _nop_();// to guarantee the timing at fCPU=20MHz
        IEN = 1;// enable interrupts
}
```

These functions fulfill the bus timings for a 20MHz CPU clock. If the microcontroller is used with a higher clock speed, _nop_() can be inserted after the read/write statement to lengthen the time between /RD /WR high and the next ALE signal. The functions are copied to IRAM in the parallel_init() function that is described later.

The following functions send a byte/word or n bytes to the 82C900:

```
// sends one byte to the 82C900 at address 'startaddress'
void P_SendByte(word startaddress, unsigned char data)
{
  // set Pagenumber A10-A7
  P_Write((PAGE&0x00ff), (unsigned char)(startaddress>>7)&0x0E);
  P_Write((startaddress&0x00FF), data)// write data into 82C900 RAM
}

// sends a word to the 82C900
void P_SendWord(word startaddress, word data)
{
  P_Write((PAGE&0x00ff),(unsigned char)(startaddress>>7)&0x0E);
                                          // set Pagenumber A10-A7
  P_Write(((startaddress&0x00FF)+1),(unsigned char)(data>>8)&0x00FF);
                                       // write data into 82C900 RAM
  P_Write((startaddress&0x00FF), (unsigned char) (data&0x00FF));
                                       // write data into 82C900 RAM
}

//sends bytecount bytes to the 82C900 starting with address startaddress
void P_SendnBytes(word startaddress, unsigned char *data, int bytecount)
{
        int i;
        for( i = 0; i < bytecount; i++ )
        {
                P_SendByte(startaddress+i, data[i]);
        }
}
```

These functions read bytes from the 82C900:

```
// read a byte from the 82C900
unsigned char P_ReadByte(word startaddress)
{
  unsigned char ret;
  P_Write((PAGE&0x00FF), (unsigned char)(startaddress >>7)&0x0E);
                                          // set page of 82C900
  ret = P_Read(startaddress&0xFF);    // read Byte
  return ret;
}
```

```
// read bytecount Bytes from the 82C900
void P_ReadnBytes(word startaddress, unsigned char *data, int bytecount)
{
        int i;
        for( i = 0; i < bytecount; i++)
        {
                data[i] = P_ReadByte(startaddress+i);
        }
}
```

Following functions can be used to configure the 82C900 easily:

```
// set the GLOBCTR for long read access
// copy transferfunctions in the IRAM
void Parallel_init()
{
  SROM_PS(P_ACS)// define variables
  // copy functions into IRAM
  hmemcpy(SROM_PS_TRG(P_ACS),SROM_PS_SRC(P_ACS),SROM_PS_LEN(P_ACS));
  P_SendByte(GLOBCTR,0x17);// long read access, OUT0 and  OUT1 INTout
}

// init MSGOBJ obj_nr
void init_MSG(int node, int obj_nr, int dir, int by_nr, int id)
{
        int cfg=0x0000;
        P_SendWord(MSG_OBJ_BASE(obj_nr)+MSGCTRn, 0x5555);
        if( 1 == node) cfg |= 0x0002; // node B(node='1')
        if( 1 == dir)  cfg |= 0x0008; // transmitobject(dir='1')
        cfg |= by_nr<<4;              // number of bytes(by_nr)
        P_SendWord(MSG_OBJ_BASE(obj_nr)+MSGCFGn, cfg);
                                      // send configuration
        P_SendByte(MSG_OBJ_BASE(obj_nr)+MSGARn+2, id<<2);// set ID(id)
        P_SendWord(MSG_OBJ_BASE(obj_nr)+MSGCTRn, 0xFFBF);// MSG valite
}
```

```
// configure the interrupt and the interruptnode for the MSGOBJ msg_obj
void init_INT(char msg_obj, char INT_Node)
{
  unsigned char dir;
  unsigned int INT = 1, node, help = 0;
  // configures the (A/B)IMR0 register for the MSGOBJINT
  node = (P_ReadByte(MSG_OBJ_BASE(msg_obj)+MSGCFGn)&0x02)>>1;// 1 if
node B
  if(msg_obj < 0x10)
  {
    help = P_ReadByte((node?BIMR0:AIMR0)+1)<<8;
    help |= P_ReadByte(node?BIMR0:AIMR0);
    INT = INT << msg_obj; // take MSGOBJ INT into account
    P_SendWord(node?BIMR0:AIMR0,INT | help);
  }
  else
  {
    help = P_ReadByte((node?BIMR0:AIMR0)+3)<<8;
    help |= P_ReadByte((node?BIMR0:AIMR0)+2);
    INT = INT << (msg_obj-0x10);
    P_SendWord((node?BIMR0:AIMR0)+2,INT | help);
  }
  // reads the direction of the MSGOBJ
  dir = (P_ReadByte(MSG_OBJ_BASE(msg_obj)+MSGCFGn)&0x08);
  if(dir) // enable transmitt interrupt
  {
    P_SendByte(MSG_OBJ_BASE(msg_obj)+MSGCFGn+2, (INT_Node<<4)&0xF0);
                              // set Interrupt Node for the MSGOBJ
    P_SendByte(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xFFED);
                              // enable Interrupt and clear INTPND
  }
  else   // enable receive interrupt
  {
    P_SendByte(MSG_OBJ_BASE(msg_obj)+MSGCFGn+2, (INT_Node)&0x0F);
                              // set Interrupt Node for the MSGOBJ
    P_SendByte(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xF9);
                              // enable Interrupt and clear INTPND
  }
}
```

Functions to transmit data via CAN-Bus or to read received data from the 82C900:

```
// load wd_nr bytes into the MSGOBJ and transmit them via CAN-BUS
void SendData(int msg_obj, unsigned char *data, int bt_nr)
{
        int  msg_dat=MSG_OBJ_BASE(msg_obj)+MSGDRn0;
                                        // address of the MSGOBJ-data
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xFB7F);
                                        // inhibit transmission.
        P_SendnBytes(msg_dat,data,bt_nr);
                                        // transmit new data to the MSGOBJ
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xE6BD);

// Reads bt_nr Bytes from the MSGOBJ into the datafield
void ReadData(int msg_obj, unsigned char *data, int bt_nr)
{
        int msg_dat=MSG_OBJ_BASE(msg_obj)+MSGDRn0;
                                        // address of the MSGOBJ-data
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xFD7F);
                                        // reset MSGVAL,NEWDAT
        P_ReadnBytes(msg_dat,data,bt_nr);     // read bt_nr bytes
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xFDBF);
                                        // set MSGVAL, no data lost
}
```

Application example for the parallel interface

This example configures with the described functions both CAN-Nodes of the 82C900 to a baud rate of 1 Mbaud. Message object 0 is setup as a receiving object for node A with the receive interrupt triggering INTnode 0. Message object 1 is set up as a transmit object for node B and message object 2 is a transmitting object of node A. All transmit objects have two data bytes.

After the configuration message object 1 transmits the data {0x55, 0xAA} with the ID 0x0001 on node B. The message is received by message object 0 at node A. Interrupt CC10INT (pin 2.10 of C167CR) is triggered from INT0 (OUT0 of 82C900). The interrupt function reads the data from the 82C900 and sends it back via message object 2 with the ID 0x0002. The transfers can be watched with a CAN-Analyzer or an Oscilloscope. The source code for this example is in the “parallel” directory.

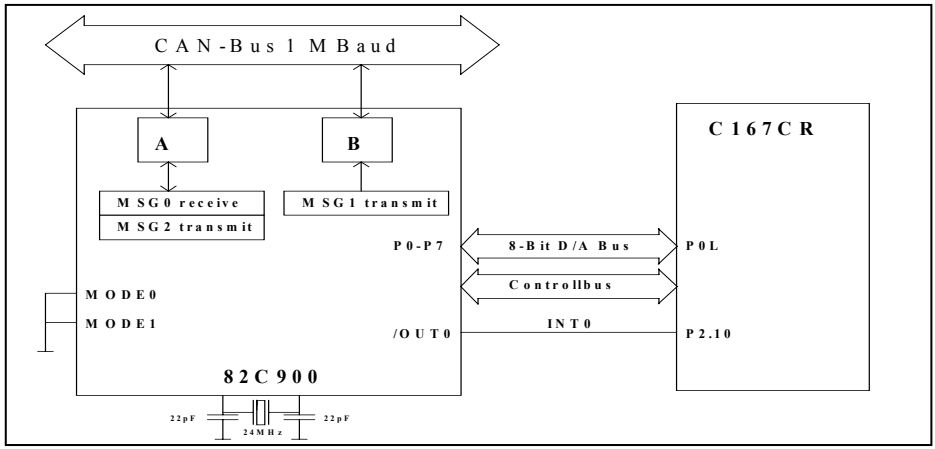

Figure 6 Schematic of the example application

The interrupt function reads only the received data from MSGOBJ0 and sends it back via MSGOBJ2.

```
void INT_viIsrEx2(void) interrupt CC10INT
{
  // USER CODE BEGIN (IR_IsrEx2,1)
        unsigned char data[2]={0x00,0x00};
        ReadData(0,data,2);
        SendData(2,data,2);
  // USER CODE END
}
```

The initialization functions for the CAN-Nodes:

```
// Node A 1 MBaut, IMC ist taken into account*/
void init_NodeA(void)
{
  P_SendByte(ACR,0x41);  // set bit CCE in ACR
  P_SendByte(ABTR,0x02); // set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1, BRP=2
  P_SendByte(ABTR+1,0x23);// set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1, BRP=2
  P_SendByte(AIMR0,0x01);// take IMC0 into acount for INTID generation
  P_SendByte(ACR,0x00);              // clear INIT, CCE in ACR
}

void init_NodeB(void)
{
  P_SendByte(BCR,0x41); // set bit CCE in ACR
  P_SendByte(BBTR,0x02);// set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1, BRP=2
  P_SendByte(BBTR+1,0x23);// set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1, BRP=2
  P_SendByte(BIMR0,0x01);// take IMC0 into acount for INTID generation
  P_SendByte(BCR,0x00); // clear INIT, CCE in ACR
}
```

Main-function of the example application:

```
void main(void)
{
  // USER CODE BEGIN (Main,1)
        unsigned char data[2]={0x55,0xAA};

  // USER CODE END

 // USER CODE BEGIN (Main,2)

  Parallel_init(); // set GLOBCTR of the 82C900
  Project_Init();
  init_NodeA(); // init node A
  init_NodeB();        // init node B
  init_MSG(0,0,0,2,0x0001);//MSG0 is a receive OBJ, Node A, 2 Databytes
  init_INT(0,0); //configure receive interrupt for MSGOBJ0 trigger Node0
  init_MSG(1,1,1,2,0x0001);//MSG1 is a transmitting obj,Node B, 2 Datab.
  init_MSG(0,2,1,2,0x0002);//MSG0 is a transmitting OBJ,Node A,2 Datab.
  SendData(1,data,2);


  while(1){}
  // USER CODE END
}
```

3.3 Hardware Gateway functionality

The 82C900 also provides the possibility to act as a gateway between two CAN-Busses. It doesn't matter if the CAN-Busses work at the same speed. So it can be used to connect a High- and a Low-speed CAN Bus with or without any control of a microcontroller. The Gateway function has two modes. The first mode is the Normal Gateway Mode. This mode provides the possibility to send messages from e.g. Node A to Node B and at the same time it can forward remote frames from Node B to Node A. But it needs two message objects for each direction.

The second mode is the Shared-Gateway-Mode. This mode only needs one message object per direction.

But it can only receive messages/remote frames from one Node at a time. So it can happen, that a message on CAN-Bus A gets lost without any detection, during the time the gateway waits for a remote frame on CAN-Bus B.

3.4 Normal Gateway Mode

The following section describes how to connect two CAN-Busses operating with different speeds. CAN-Bus 1 with a speed of 1 Mbaud will be connected to Node A and CAN-Bus 2 with a speed of 125 Kbaud will be connected to Node B. The SSC-Interface will be used for the configuration.

In this application two messages objects will be configured. MSGOB0 will be a receiving object on node A and MSGOBJ1 will be a transmitting object on node B. Both are additionally configured for the normal gateway mode. So all messages that are received on node A with the ID 0x001 will be automatically transmitted on node B with the same identifier. To configure a bi-directional gateway two more message objects need to be configured with just switched nodes. The source files and the project are in the "Normal_Gateway" directory.

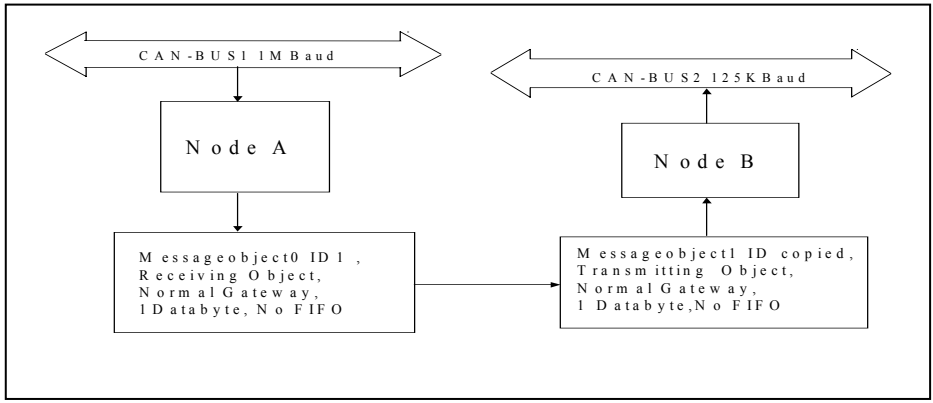

Figure 7 Two Message objects in Normal Gateway Mode connecting two CAN-Busses.

First the nodes need to be configured.

```
// set up CAN node A, 1 MBaud

        SSC_SendByte(ACR, 0x41);       // set bit CCE in ACR
        SSC_SendByte(ABTR+1, 0x23);    // set DIV8X=0, TSEG2=3, TSEG1=4
        SSC_SendByte(ABTR, 0x02);      // set SJW=1, BRP=2
        SSC_SendByte(AIMR0, 0x01);     //take IMC0 into account for
                                       //INTID generation
        SSC_SendByte(ACR, 0x00);       // clear INIT, CCE in ACR


// setup CAN node B, 125 KBaud

        SSC_SendByte(BCR, 0x41);       // set bit CCE in BCR
        SSC_SendByte(BBTR+1, 0x23);    // set DIV8X=0, TSEG2=3, TSEG1=4
        SSC_SendByte(BBTR, 0x17);      // set SJW=0, BRP=22
        SSC_SendByte(BIMR0, 0x01);     // take IMC0 into account for
                                       // INTID generation
        SSC_SendByte(BCR, 0x00);       // clear INIT, CCE in ACR
```

Now message object 0 is configured as receive object for the normal gateway mode without FIFO-Buffering. The gateway capability is configured in the Message Object n FIFO/Gateway Control Register MSGFGCR.

```
// message object 0
// receiving object from CAN node A, normal gateway mode,
// received messages are directly transmitted on
// destinaton side(CAN node B)
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0x7F);
                                  // set MSGVAL to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xFD);
                                  // set INTPND to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0x7F);
                                  // set RMTPND to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xDF);
                                  // set TRXQ to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xFD);
                                  // set NEWDAT to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCFGn,0x10);
       // select Node A, set DIR=0(receive), 1 data byte (DLC=1)
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn+1,0xF7);
                                  // set MSGLST to '01' => reset
SSC_SendByte(MSG_OBJ_BASE(0)+MSGARn+2,0x04);
               // identifier of receive object (0) is 0x0001
SSC_SendByte(MSG_OBJ_BASE(0)+MSGFGCRn,0x00);
                                       // FSIZE = 0 => No FIFO
SSC_SendByte(MSG_OBJ_BASE(0)+MSGFGCRn+1,0x0D);
       // GDFS='1',SRREN='0',IDC="1", DLCC='1', FD='0',SDT='0'
SSC_SendByte(MSG_OBJ_BASE(0)+MSGFGCRn+2,0x01);
                        // CANPTR = 1 => MSOBJ 1 is destination
SSC_SendByte(MSG_OBJ_BASE(0)+MSGFGCRn+3,0x04);
                                  // MMC ='100' => normal gateway
SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xBF);
                        // set MSGVAL, message object is valid
```

Table 4 Settings for the MSGFGCR-Register for the receiving object

FSIZE = “0x00”	Turns the FIFO-Buffer off
GDFS = ‘1’	Received message will be automatically transmitted from the transmitting object.
IDC = ‘1’	The identifier of the received message object will be copied to the transmitting object.
DLCC=’1’	The Data length code of the receiving object is automatically copied to the transmitting object
SDT=’0’	No single data transfer. The message object stays valid after a successful datatransfer.
CANPTR=”0x01”	Message object 1 is the destination object.

And message object 1 is the transmitting object connected to Node B.

```
// message objekt 1
// transmitting to CAN node B,
// normal gateway(remoteframes copied to node A)

        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0x7F);
                                        // set MSGVAL to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0xFD);
                                        // set INTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0x7F);
                                        // set RMTPND to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xDF);
                                        // set TRXQ to '01' => set
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCFGn,0x1A);
// NODE=1 => CAN nod B , set DIR=1 (transmit), 1 data byte (DLC=1)
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xF7);
                                        // set CPUUPD to '01' => reset
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn+1,0xFD);
                                        // set NEWDAT to '01' => set
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGFGCRn,0x00);
                                        // FSIZE = 0 => No FIFO
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGFGCRn+1,0x02);
                        // GDFS='1',SRREN='1',IDC="1", FD='0',SDT='0'
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGFGCRn+2,0x00);
                                        // CANPTR = 0 => MSOBJ 0 is source
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGFGCRn+3,0x04);
                                        // MMC ='100' => normal gateway
        SSC_SendByte(MSG_OBJ_BASE(1)+MSGCTRn,0xBF);
                                        // set MSGVAL to '10' => set
```

Table 5 Settings for the MSGFGCR-Register for the transmitting object

FSIZE = “0x00”	Turns the FIFO-Buffer off
SRREN = ‘1’	A remote frame, received from the destination object, will be send from the source object.
MMC=’100’	Object is set to normal gateway mode
SDT=’0’	No single data transfer. The message object stays valid after a successful data transfer.
CANPTR=”0x00”	Message object 0 is the source object.

Note: The automatic transmission only works, if NEWDAT of the destination object is set to ‘01’

The bits that are not described are not important for this configuration. To configure a bi-directional-gateway only two more objects must be configured in the same way but with swapped Nodes.

3.5 Shared Gateway

The following application creates an bi-directional gateway between CAN-Bus1 (1M Baud) and CAN-BUS2 (125K Baud) using message objects one and two in shared gateway mode. The objects transmit the received data automatically on the destination side and switch than back to the source side to wait for the next message. So all remote frames on the destination side will get lost. The source code files for this application are in the “shared_gateway” directory.

Table 6 Settings for the MSGFGCR-Register for Shared-Gateway

FSIZE = “0x00”	No FIFO possible in shared-gateway-mode
GDFS = ‘1’	Received message will be automatically transmitted from the transmitting object.
IDC = ‘0’	Must be reset in shared-gateway-mode
DLCC=’0’	Must be reset in shared-gateway-mode
SDT=’0’	No single data transfer. The message object stays valid after a successful data transfer.
CANPTR=OBJ-Num	CANPTR must be the OBJ-Number of the shared-gateway Object

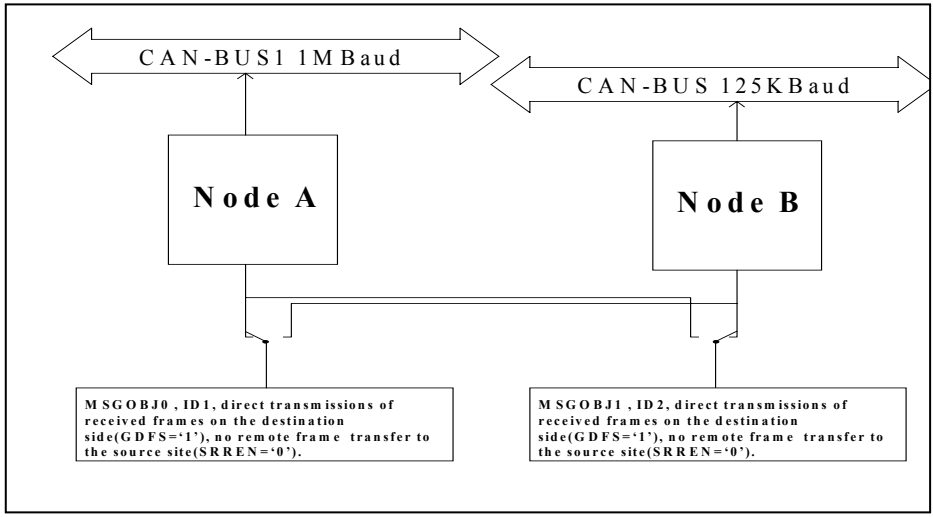

Figure 8 Configuration of the 82C900 for the Shared-Gateway-Mode

The parallel interface is used to configure the 82C900. See the description of the used functions in the description of the parallel interface. The following functions configure Node A for 1M Baud and Node B for 125K Baud:

```
// Node A 1 MBaud, IMC ist taken into account*/
void init_NodeA(void)
{
        P_SendByte(ACR,0x41);   // set bit CCE in ACR
        P_SendByte(ABTR,0x02);  //set BRP=2
        P_SendByte(ABTR+1,0x23)// set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1
        P_SendByte(AIMR0,0x01) //take IMC0 into acount for INTID gen.
        P_SendByte(ACR,0x00);  // clear INIT, CCE in ACR
}
```

```
void init_NodeB(void)
{
        P_SendByte(BCR,0x41);    // set bit CCE in BCR
        P_SendByte(BBTR,0x17);   // set BRP=0x17
        P_SendByte(BBTR+1,0x23);// set DIV8X=0, TSEG2=3, TSEG1=4, SJW=1
        P_SendByte(BIMR0,0x01); // take IMC0 into acount for INTID gen.
        P_SendByte(BCR,0x00);   // clear INIT, CCE in BCR
}
```

The next function can be used to configure a message object for the Shared-Gateway-Mode:

```
// init the MSGOBJ obj_nr as shared gateway as descriped in the
// usersmanual
void init_shared_gateway( char obj_nr, char GDFC, char SRREN)
{
        word cfg = 0x0500 |(obj_nr&0x1F);
     // set MMC='101' and CANPTR to obj_nr
        P_SendWord(MSG_OBJ_BASE(obj_nr)+MSGFGCRn+2,cfg);
        cfg = 0x0000;               // reset FSIZE,IDC, DLCC
        if(GDFC)  cfg |= 0x0100;
     // automatic transmission on destinationsite
        if(SRREN) cfg |= 0x0200;              // waiting on remoteframes
        P_SendWord(MSG_OBJ_BASE(obj_nr)+MSGFGCRn,cfg);
}
```

Main function of the example application, that configures the 82C900 as described above:

```
void main(void)
{
  Project_Init();

  // USER CODE BEGIN (Main,2)
        Parallel_init();// configure 82C900 for long readaccess.
        init_NodeA();// init node A for 1 MBaute
        init_NodeB();// init node B for 125KBaute
        init_MSG(0,0,0,1,0x0001);//MSGOBJ0, node A, 1 Databyte, ID 1
        init_shared_gateway( 0, 1,0);// enable shared gateway for OBJ0
        init_MSG(1,1,0,1,0x0002);// MSGOBJ1, node B, 1 Databyte, ID 2
        init_shared_gateway( 1, 1, 0);// enable gateway for OBJ1

        while(1);
// USER CODE END
}
```

3.6 Stand Alone mode

The 82C900 can also be used in a standalone mode without a direct connection to a microcontroller. The whole configuration can be made with an SPI-compatible EEPROM. If the 82C900 should be setup via CAN-Messages only one node and one message object for the initialization need to be configured via SPI-EEPROM.

3.7 SPI Master Mode

If the mode pins are both high during reset, the 82C900 works as master for a SPI-data transfer. So it can be easily connected to an SPI-compatible EEPROM. The communication protocol is described in the 82C900 users manual in chapter 2.2.1. This stand alone mode is unique to Infineon devices and it is patented by Infineon.

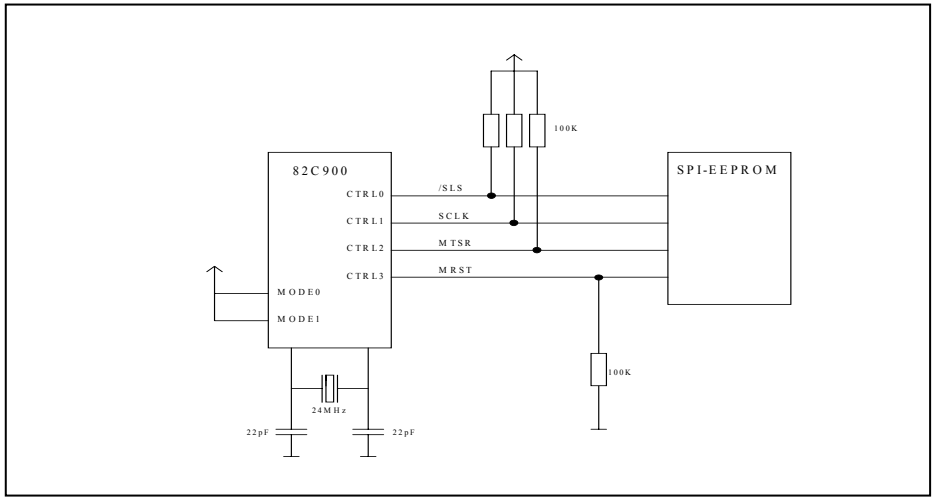

Figure 9 Schematic for the 82C900 in Standalone mode(SPI-Master)

Table 7 Jumper settings for SSC-Master-Mode

JP10	3-4 connected: SLS on EEPROM
JP32	Open
JP33	Open
JP38	All open
JP39	3-7 closed, 1-2 open 8 open
JP40	All open

The Table 8 shows the EEPROM-data for configuring CAN-Node A (1Mbaud) and Message object 0 as the configuration object for the initialization via CAN-Bus. For the communication there are transmitted two bytes for address and data size and one to four bytes of data. No checksum byte is used. The SPI-transfer has a speed of 100 Kbaud.

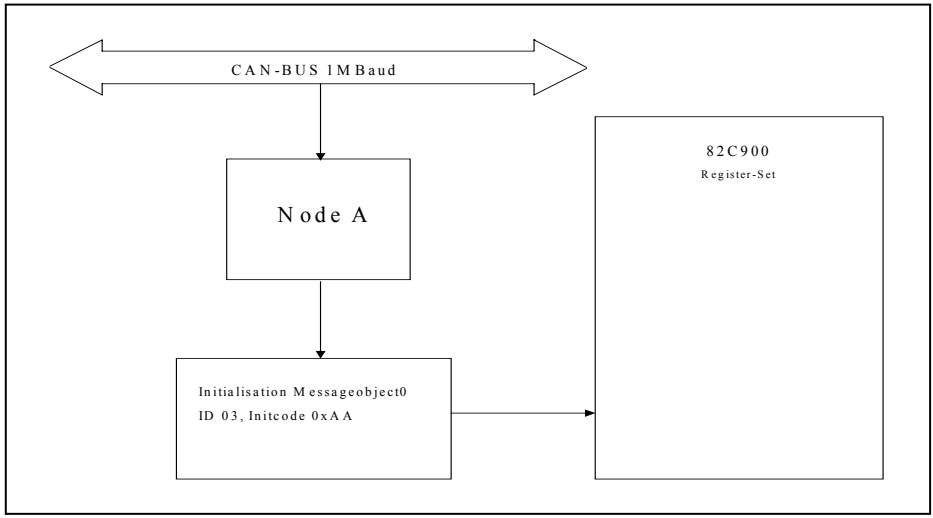

Figure 10 82C900 after loading the EEPROM-Data

Table 8 Contents of the EEPROM

EEPROM-ADR	ADH	ADL	D0	D1	D2	D3	Description
0x0000	0x22	0x00	0x41	-	-	-	1 Data byte(0x41) => 0x0200(ACR) set Init and CCE => Node ready for configurations
0x0003	0x42	0x0C	0x02	0x23	-	-	2Databyte(0x2302)=>0x020C (ABTR) set Node A to 1MBaud
0x0007	0x22	0x18	0x01	-	-	-	1 Data byte => 0x0218(AIMR0) take IMC0 into account for INTID
0x000A	0x22	0x00	0x00	-	-	-	1Databyte => 0x0200(ACR) clear INIT and CCE. Node is conf.

0x000D	0x 40	0x 44	0x 80	0x AA	-	-	2 Data bytes => 0x0044(CANINIT) Message object 0 is init object. Init via CAN enabled. Initcode is 0xAA
0x0011	0x 23	0x 14	0x 72	-	-	-	1 Data byte => 0x0314(MSGCFG0) Messageobject0 is receiving object of Node A with 7 Data bytes
0x0014	0x 23	0x 0A	0x 0C	-	-	-	1 Data byte => 0x030A(MSGAR0+2) Object identifier = 3
0x0017	0x 23	0x 10	0x BF	-	-	-	1 Data byte => 0x0310(MSGCTR0) set the messageobject0 to valid
0x001A	0x 82	0x F0	0x 00	0x 00	0x 00	0x 00	4 Data bytes => 0x02F0(ICR) Stops the Data transfer via SPI and enable the configuration via CAN-Bus

3.8 Configuration via CAN-Bus

After the configuration of one Node and one message object, the 82C900 can be configured via CAN-Bus. These initmessages contain as a first byte the Initcode (here 0xAA), then two bytes of address and one to four bytes of data. The following sequence will configure the second Node for 125 KBaud and 2 more message objects so that 82C900 will work as unidirectional gateway between the two CAN-Busses.

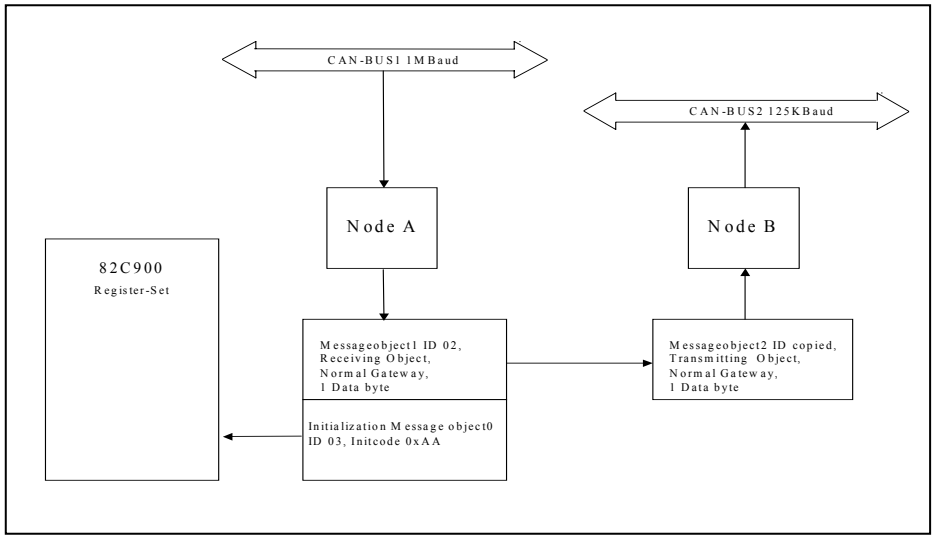

Figure 11 82C900 after the initialization via CAN-Messages

Table 9 Data Bytes of CAN Messages used to configure the 82C900

Init-code	AD H	AD L	D0	D1	D2	D3	Description
0xAA	0x 02	0x 40	0x 41	-	-	-	Set INIT and CCE in BCR(0x0240)
0xAA	0x 02	0x 4C	0x 17	0x 23	-	-	BBTR(0x024C) set Node to 125KBaute
0xAA	0x 02	0x 58	0x 01	-	-	-	BMIR0(0x0258) INT pending stat is taken into account
0xAA	0x 02	0x 40	0x 00	-	-	-	Clear INIT and CCE in BCR => Node B is configured.
0xAA	0x 03	0x 30	0x 55	0x 55	-	-	Reset MSGCTR1(0x0330)
0xAA	0x 03	0x 34	0x 10	-	-	-	MSGCFG1(0x0334) MSGOBJ belongs to Node A, receiving object, 1 Data byte
0xAA	0x 03	0x 2A	0x 08	-	-	-	(MSGAR1+2) set MSGOBJ ID =”2”
0xAA	0x 03	0x 38	0x 00	0x 0D	0x 02	0x 04	MSGFGCR1(0x0318) object in normal gateway mode, identifier and data length are copied to the destination site. No FIFO. MSGOBJ2 is destination object.
0xAA	0x 03	0x 30	0x BF	-	-	-	MSGCTR1(0x0330) set MSGOBJ valid
0xAA	0x 03	0x 50	0x 55	0x 55	-	-	Reset MSGCTR2(0x0350)
0xAA	0x 03	0x 54	0x 1A	-	-	-	MSGCFG2(0x0354) MSGOBJ belongs to Node B, transmitting object, 1 Data byte
0xAA	0x 03	0x 58	0x 00	0x 02	0x 01	0x 04	MSGFGCR2(0x0358) object in normal gateway mode. No FIFO. MSGOBJ1 is source. Remote frames are transferred to the source side.
0xAA	0x 03	0x 50	0x BF	-	-	-	MSGCTR1(0x0350) set MSGOBJ valid

3.9 Using the parallel port as general purpose I/O

If the 82C900 is used in the SSC/SPI mode (Slave or Master), it is possible to configure the port pin P1-P8 as general purpose I/O port. The function of each pin can be configured in the registers IOMODE0/2. If the pins are used as input the value can be read from the register INREG and if the pins are used as output, the value must be written in the register OUTREG. This can be done by writing directly into the registers via SPI or CAN-INIT messages or by configuring MSGOBJs as INMSG and OUTMSG. These MSGOBJ's are configured in the register CANIO. The OUTMSG must be a receiving object and the first data byte of each message is written into OUTREG and INMSG must be an transmitting object. It transmits the value of INREG as the first byte of the message, if TXRQ is set to '10' e.g. via SPI or with a matching remote frame. So it is possible to communicate easily with connected peripherals via CAN messages especially in stand-alone mode.

Application example

For the following example the 82C900 is used in the SSC-slave mode. See the related chapter for detailed descriptions. The SSC-master is a C167CR on a KitCon167 board. The 82C900 StarterKit and the KitCon167 board are connected via KitCon-connector. The port pins P0-P7 of the 82C900 are connected to P2.0-P2.7 of the C167CR. Therefore wires are used connecting P0-P7 on JP41 and the StarterKit with the pins for P2.0-P2.7 on the KitCon-connector (pins 85-92).

Note: In this configuration all jumpers of JP38 must be open!!

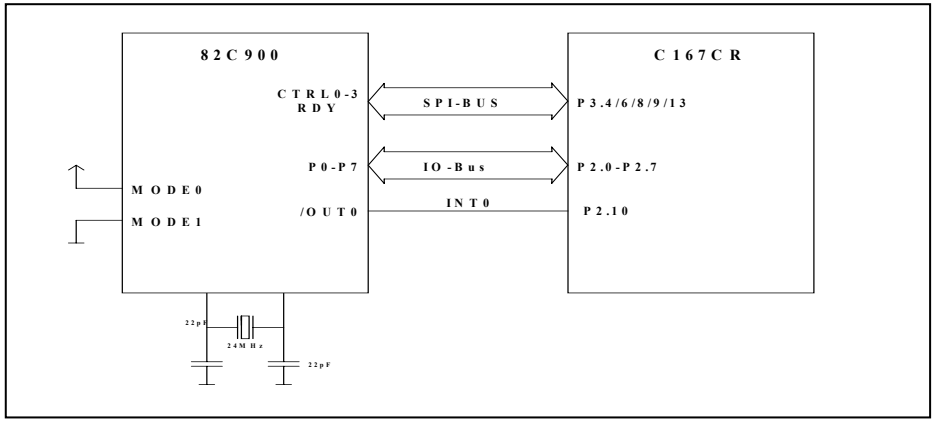

Figure 12 Hardware connections for the application example

The application configures via SSC P0-P3 of the 82C900 as output and P4-P7 as inputs and creates MSG0 as a transmitting object with one data byte for OUTMSG and MSG1 as receiving object with one data byte for INMSG. The configuration for the parallel port of the 82C900 is made in the registers IOMODE0/2 and for IN/OUTMSG in the register CANIO.

If the 82C900 receives a message in OUTMSG with an ID of 1, it puts the first data byte automatically in OUTREG. The contents of the register can be seen on all pins that are configured as output (P4-P7). The receive interrupt of MSG0 triggers EX2IN of the C167CR. The interrupt routine reads out P2.4-P2.7 and writes the data to P2.0-P2.3. Now the contents of INREG can be read out with a remote frame with the ID 2 (INMSG).

If the first data byte of the send message is 0xC0, the first data byte of the message that contains the data of INREG will be 0xCC. The values that are written to as input configured pins have no influence on the value of the pins. But the INREG contains the actual values of all pins even if they are configured as output. So all not used bits of the INMSG-data should be masked out.

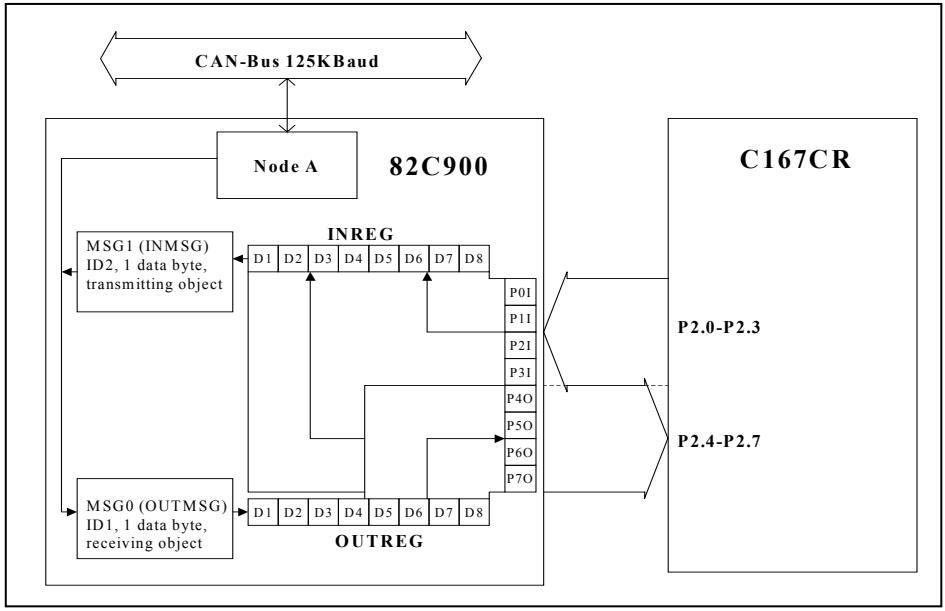

Figure 13 Configuration of the 82C900

The complete source files for this application are located in the “P_Port_GPIO” directory.

Function to configure node A and the message objects via SSC:

```
// configures node A for a CAN-Bus speed of 125KBaud
void NodeA(void)
{
  SSC_SendByte(ACR,0x41);     // set bit CCE in ACR
  SSC_SendByte(ABTR,0x17);    // set SJW=1, BRP=17
  SSC_SendByte(ABTR+1,0x23); //set DIV8X=0,TSEG2=3,TSEG1=4
  SSC_SendByte(AIMR0,0x01);// take IMC0 into acount for INTID generation
  SSC_SendByte(ACR,0x00);     // clear INIT, CCE in ACR
}


// configures MSGOBJ(obj_nr) for node with the direction dir, id and
// by_nr data bytes
void SSC_init_MSG(int node, int obj_nr, int dir, int by_nr, int id)
{
  unsigned char cfg=0x00;
  SSC_SendByte(MSG_OBJ_BASE(obj_nr)+MSGCTRn, 0x55);
  SSC_SendByte(MSG_OBJ_BASE(obj_nr)+MSGCTRn+1, 0x55);
  if( 1 == node) cfg |= 0x02;    // node B(node='1')
  if( 1 == dir)  cfg |= 0x08;    // transmitobject(dir='1')
  cfg |= by_nr<<4;               // number of bytes(by_nr)
  SSC_SendByte(MSG_OBJ_BASE(obj_nr)+MSGCFGn, cfg);// send configuration
  SSC_SendByte(MSG_OBJ_BASE(obj_nr)+MSGARn+2, id<<2);// set ID(id)
  SSC_SendByte(MSG_OBJ_BASE(obj_nr)+MSGCTRn, 0xBF);  // MSG valite
}
```

Main function of the example application:

```
void main(void)
{
  // USER CODE BEGIN (Main,1)
  // USER CODE END

  Project_Init();

  // USER CODE BEGIN (Main,2)
       NodeA();
       SSC_init_MSG(0,0,0,1,1);// MSG0, receive, Node A, id = 1
       SSC_init_MSG(0,1,1,1,2);// MSG1, transmit, Node A, id = 2

       // enable receive interrupt for MSG0
       SSC_SendByte(MSG_OBJ_BASE(0)+MSGCFGn+2,0x00);
       SSC_SendByte(MSG_OBJ_BASE(0)+MSGCTRn,0xF9);

       // configuring the IO pins. 1-4 Input, 5-8 Output
       SSC_SendByte(IOMODE0,0x00);   // 0/1 standard input
       SSC_SendByte(IOMODE0+1,0x00); // 2/3 standard input
       SSC_SendByte(IOMODE2,0x88);   // 4/5 standard output
       SSC_SendByte(IOMODE2+1,0x88); // 6/7 standard output

       // configure and enable the IN/OUTMSG
       SSC_SendByte(CANIO+1,0x81);// MSG1 INMSG
       SSC_SendByte(CANIO,0x80);// MSG0 OUTMSG

       while(1);
  // USER CODE END
}
```

Interrupt routine:

```
// copies bits from input to output side
void INT_viIsrEx2(void) interrupt CC10INT
{
  // USER CODE BEGIN (IR_IsrEx2,1)
               P2_0 = P2_4;
               P2_1 = P2_5;
               P2_2 = P2_6;
               P2_3 = P2_7;
  // USER CODE END
}
```

3.10 The FIFO-capability of the 82C900

The 82C900 provides the possibility to connect some message objects to one FIFO-buffer. This can be useful to minimize the time where the microcontroller communicates with the 82C900 or to minimize the risk of data loss during times of high CPU-loads or especially when the 82C900 acts as normal gateway between two CAN-Busses with different speeds. If a FIFO-buffer is used, the 82C900 manages the whole communication between the CAN-Bus and the message objects of the buffer. The communication between the microcontroller and the FIFO must be handled by software. The configuration of the FIFO is made in the MSGFGCR-register of the 82C900. See the further description in the user's manual of the 82C900.

Application example for the FIFO-buffer without gateway function

The following application uses the parallel interface to configure the 82C900 with the C167CR. The C167CR, KC167 and the 82C900 starter kit must be configured as described in the chapter for the parallel interface. The connections between the 82C900 and the C167CR are similar to the connections of the shared-gateway example.

The example configures two FIFO-buffers with 8 messages objects each. One buffer is configured for receiving messages (base object 8) and the other transmitting them (base object 16) without waiting for remote frames. All message objects of the receiving buffer have the same configuration with enabled receiving interrupts. So the 82C900 triggers an interrupt at the moment it receives messages with the ID 0x001. During the execution of the interrupt routine the C167CR reads the received data, and downloads it directly to the transmitting buffer and transmits the data on node B with the ID 0x002. The transmitting object with the ID 0x003 is used to transmit error messages, and the receiving message can be used to see how the INTPND processing works.

This function is very similar to the SendData(xx) function except that the data is not automatically transmitted. So the TXRQ-bit in MSGCTR must be set or a remote frame must be received before the 82C900 transmits the data. The source files are located in the "FIFO" subdirectory.

```
void LoadRemoteData(int msg_obj, unsigned char *data, int bt_nr)
{
        int  msg_dat=MSG_OBJ_BASE(msg_obj)+MSGDRn0;
                                              // address of the MSGOBJ-data
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xFB7F);
                                                //inhibit transmission.
        P_SendnBytes(msg_dat,data,bt_nr);// send new data to the MSGOBJ
        P_SendWord(MSG_OBJ_BASE(msg_obj)+MSGCTRn, 0xF6BD);// enable data
}
```

This function can be used to initialize a FIFO-buffer and to send or read data:

```
// initialize FSIZE(see users manual) MSGOBJ for on FIFO-Buffer using
// MSGOBJ(base) as baseobject. the FIFO-Buffer is related to node and
// has the direction dir and id. FD(MSGFGCRn) is independent from dir
// and all objects has an 11-bit identifier and by_nr databytes.
void init_FIFO(int base, char FSIZE, char node, char dir, char FD,
                                                    char id, char by_nr)
{
  word cfg = 0x0000;
  int i,res;
  for( i=0; i<(FSIZE);i++)
  {
     res = base+i;
     init_MSG(node,res,dir,by_nr,id);
                                  //init the OBJ's for the FIFO-Buffer
     if(!dir) init_INT(res,0); // enable receive interrupt on node 0
     if(dir&&!FD)P_SendByte(MSG_OBJ_BASE(res)+MSGCTRn, 0x7F);
                                                     // set OBJ invalid
     if(!i) cfg = 0x0200 | (base & 0x1F); // base OBJ
     else cfg = 0x0300 | (base & 0x1F);   // slave OBJ's
     P_SendWord(MSG_OBJ_BASE(res)+MSGFGCRn+2,cfg);
     if(FD) cfg = 0x2000;     // update of CANPTR(base) on trans.event
     else cfg = 0x0000 ;// update of CANPTR(base) on receive event
     cfg |= ((FSIZE-1) & 0x1F); // set FIFO-Size
     P_SendWord(MSG_OBJ_BASE(res)+MSGFGCRn,cfg);
  }
}
```

The following functions can be used to up/download data from/to the FIFO-buffer. The load-function automatically searches for the next free object. If there is no free message object in the buffer the function returns with zero. The read-function searches for and reads the data from the first message object, where the NEWDAT-flag is set. If the MSGLST-flag is set, the function returns zero.

```
// load the data in the first free object of the FIFO-Buffer with the
// base object base. the data can be transmitted automatically, if
// remote is '0'. else it waits for remote frames addressing the FIFO
// returns zero if their is no free object in the buffer
int load_FifoObj(int base, unsigned char* data, char remote)
{
  int by_nr, next, i=0,FSIZE,obj;
  unsigned char help;
  by_nr = (P_ReadByte(MSG_OBJ_BASE(base)+MSGCFGn)&0xF0)>>4;
    // read datasize(base)
  FSIZE = P_ReadByte(MSG_OBJ_BASE(base)+MSGFGCRn)& 0x1F;
// read FSIZE(base)
  next = P_ReadByte(MSG_OBJ_BASE(base)+MSGFGCRn+2);// read CANPTR(base)
  for( i=0; obj = base+(next+i)%(FSIZE+1),
   ((help=P_ReadByte(MSG_OBJ_BASE(obj)+1+MSGCTRn))&0x22)>0; i++)
  {
    if((FSIZE)==i ) return 0;
  }
  // send data directly
  if(!remote) SendData(obj, data, by_nr);
  // data is send on remoteframe
  else LoadRemoteData(obj, data, by_nr);
  return 1;}
```

```
// reads the data from the next OBJ with the NEWDAT flag set.
// return:
// 0 data read OK
// 1 data can be invalid MSGLST was set
// -1 no OBJ in the FIFO with set NEWDAT flag
int read_FifoObj(int base, unsigned char* data)
{
  int by_nr, next, i=0,FSIZE, obj, ret;
  by_nr = (P_ReadByte(MSG_OBJ_BASE(base)+MSGCFGn)&0xF0)>>4;
                                                  // read datasize(base)
  FSIZE = P_ReadByte(MSG_OBJ_BASE(base)+MSGFGCRn)& 0x1F;
                                                    // read FSIZE(base)
  next = P_ReadByte(MSG_OBJ_BASE(base)+MSGFGCRn+2);// read CANPTR(base)
  // search for next new message
  for( i=1;obj = base+(next-i)%(FSIZE+1),
               (P_ReadByte(MSG_OBJ_BASE(obj)+1+MSGCTRn)&03)==2 ; i++);
  // calculate next OBJ
  obj = base+(next-i+1)%(FSIZE+1); // take last OBJ with new data
  if(!((P_ReadByte(MSG_OBJ_BASE(obj)+1+MSGCTRn)&03)==2)) return -1;
                                                    // no new data
  // start reading data
  P_SendByte(MSG_OBJ_BASE(obj)+MSGCTRn, 0xFD);// reset INTPND
  P_ReadnBytes(MSG_OBJ_BASE(obj)+MSGDRn0,data,by_nr);// read bt_nr bytes
  ret = P_ReadByte(MSG_OBJ_BASE(obj)+MSGCTRn+1)&0x08 == 1;
                                                  // messagelost occured
  P_SendByte (MSG_OBJ_BASE(obj)+MSGCTRn+1, 0xFD); // reset NEWDAT
  return ret;
}
```

Interrupt and main-function:

```
void INT_viIsrEx2(void) interrupt CC10INT
{
  // USER CODE BEGIN (IR_IsrEx2,1)
  unsigned char data[4];
  int obj;
  IEN =0;
  while(obj = P_ReadByte(AIR))// read out INTPND
  {
    obj -= 2; // calculate MSGOBJ number
    if(obj > 0x07 && obj < 0x10) // interrupt from rec. fifo
    {
      int read;
      while(!(read = read_FifoObj(8,data)))
```

```
      {
        load_FifoObj(16,data,0); // copy data to transmitting fifo
      }
      if(1 == read) // messages got lost
      {
        data[0] = 0xFF; // mark as invaltite
        SendData(0,data,4); // send alert
      }
    }
     else// readout other MSGOBJ
    {
      ReadData(obj,data,4);
      data[3] = obj; // which MSGOBJ
      SendData(0,data,4);
      P_SendByte(MSG_OBJ_BASE(obj)+MSGCTRn,0xFD);// reset INTPND
    }
    P_SendByte(AIR,0x00); // force INTPND to update
  }
  IEN = 1;
  // USER CODE END
}

void main(void)
{
  // USER CODE BEGIN (Main,1)
  unsigned char data[4];
  Parallel_init();
  // USER CODE END
  Project_Init();
  init_NodeA();// configure node A 1MBaute
  init_NodeB();// configure node B 125 KBaute
  init_FIFO(8,8,0,0,0,0x001,4);// Init receiving FIFO for node a
  init_FIFO(16,8,1,1,1,0x002,4);// Init transmitting FIFO for node b
  init_MSG(0,0,1,4,0x003);
  init_MSG(0,1,0,4,0x004);
  init_INT(1,0);
  while(1);
}
```

Additional information

If the FIFO is configured for automatic transmission (TXRQ set) but the base object is invalid the FIFO won't start transmitting the Data. If the base object is valid and the TXRQ-flag is set the 82C900 will transmit messages and increment the CANPTR until the TXRQ-flag is reset in one MSGOBJ or the OBJ is invalid and continue when both flags are set. Setting the base object invalid won't stop the transmission until CANPTR=base object.

If the FIFO is configured for answering remote frames (TXRQ is not set) the 82C900 starts transmitting messages when the base object is valid and a remote frame is received. If one MSGOBJ is invalid no data will be sent after receiving a remote frame but the CANPTR will be incremented. It can happen that some data is sent twice, if the number of received remote frames is bigger than the number of MSGOBJs in the FIFO. If the base object is invalid no message will be sent on a matching remote frame even if the CANPTR addressed MSGOBJ is valid. The FIFO will continue to transmit messages on matching remote frames after setting the base object valid again.

The same is valid for the receiving FIFO. If the FIFO (FD = '1') transmits remote frames it increases CANPTR, even if the base object is invalid. And the FIFO (FD = '0') doesn't increase CANPTR if the base object is invalid. For transmitting actions the increasing of the CANPTR are independent of MSGVAL of the base object and for receiving actions the increasing of CANPTR are dependent of MSGVAL of the base object.

The INTPND register is not updated after read access to it as usual in the Infineon microcontrollers. It is only updated after a new interrupt request or after a write access to the A/BIR register.

3.11 Using the FIFO-buffering in a gateway application

If two CAN-Busses with different speeds are connected via the gateway function a data loss can easily occur when the source-object of the gateway receives the messages faster than the destination-object can transmit them on the slower CAN-Bus. The 82C900 provides a solution for this problem, using a FIFO-Buffer as a destination-object. The FIFO-Buffer must be configured in the same way as the transmitting Buffer in the last example. The source-object is pretty similar to the source-object of a Normal-Gateway except that FISIZE is not zero but equal FSIZE of the destination-buffer and CANPTR must be initialized with the object-number of the base object of the destination-buffer.

The following example configures message object zero as the source object connected to node A at 1 MBaud, and a FIFO of 8 message objects as destination object connected to node B at 125 KBaud. The FIFO-Buffer is configured as a transmitting buffer with message object 16 as base object. The ID and the data length code is copied from the source side to the destination side.

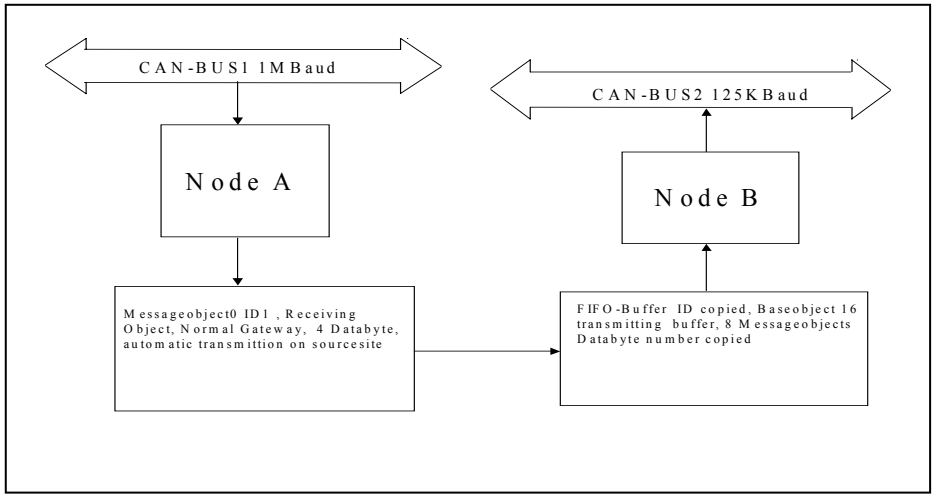

Figure 14 Normal Gateway with FIFO-Buffer

The source code of this application example is located in the “FIFO_GATEWAY” directory.

This function configures a MSGOBJ for Normal Gateway Mode with a FIFO-Buffer as destination object:

```
// initialize an existing object as source-object(obj), for an existing
// FIFO-Bufferwith the baseobj(fifo_buffer) and configure it with IDC
// and DLCC
void FifoGatewaySource(char obj, char IDC, char DLCC, char GDFS,
                                                        char fifo_base)
{
  unsigned char FSIZE;
  int cfg = 0x0000;
  FSIZE = P_ReadByte(MSG_OBJ_BASE(fifo_base)+MSGFGCRn);
                                                  // read FSIZE of Buffer
  cfg =  FSIZE & 0x001F;
  if(IDC) cfg |= 0x0400; // ID is copied
  if(DLCC) cfg |= 0x0800;// DLC is copied
  if(GDFS) cfg |= 0x0100;// automatic transmition
  P_SendWord(MSG_OBJ_BASE(obj)+MSGCTRn, 0xFF7F);// OBJ invalid
  P_SendWord(MSG_OBJ_BASE(obj)+MSGFGCRn,cfg);//set register with cfg
  cfg = 0x0400 | (fifo_base & 0x1F);// set CANPTR and Normal Gatew. mode
  P_SendWord(MSG_OBJ_BASE(obj)+MSGFGCRn+2,cfg);
  P_SendWord(MSG_OBJ_BASE(obj)+MSGCTRn, 0xFFBF);// MSG valite
}
```

Main-function of the application:

```
void main(void)
{
  // USER CODE BEGIN (Main,1)
   Parallel_init();
  // USER CODE END

   Project_Init();

  // USER CODE BEGIN (Main,2)
   init_NodeA();// configure node A 1MBaute
   init_NodeB();// configure node B 125 KBaute
   init_FIFO(16,8,1,1,1,0x002,4);// Init transmitting FIFO
   init_MSG(0,0,0,4,0x003);// init MSGOBJ 0
   FifoGatewaySource(0,1,1,1,16);// init MSGOBJ 0 as Source-object
   while(1);
  // USER CODE END
}
```

The 82C900 does not provide the possibility to use a FIFO-Buffer as source object. So you need to use it only as destination-object. And it is also not possible to forward remote frames from the destination side to the source side. If the function is needed it must be done by software on a microcontroller connected to the 82C900 (description in the 82C900 users manual).

3.12 Power Saving Modes

The 82C900 provides two different types of power-saving modes.

If the 82C900 is in Sleep Mode, the TwinCAN, the SSC and the 8-Bit port logic are gated off. But the clock is still running. So it is possible to wake it up with a falling edge on /CS, /SLS or /RESET or with CAN messages on node A or B, if the bits CANAWU or CANBWU in CLKCTR are set. The messages that wakes up the 82C900 won't get lost.

If the 82C900 is in Power Down Mode the clock stops working and it can only be woken up by a /RESET.

The Modes can be entered in different ways. One way is the direct configuration in the CLKCTR-register via Parallel Interface. There are problems using the SSC to configure the sleep mode directly in the CLKCTR-register (see errata sheet for details). The other way is to use one receiving message object with 2 data bytes to configure the CLKCTR-register. This object is configured in register CANPWD. The first byte of the received message from SLPMSG must be the SLPCODE and the second byte is written into the CLKCTR-register.

Table 10 CLKCTR configurations for the Power Saving Modes

PWD='0' & CANCLK ='1'	Normal Operation Mode
PWD='0' & CANCLK='1'	Sleep Mode
PWD='1' & CANCLK='X'	Power Down Mode

Related Functions

The following function shows how to setup the two Power Down Modes via parallel-interface and how to configure the SleepMSGOBJ. The source code files are in the “power_save” subdirectory.

```
// set the 82C900 in the power down mode.
void PowerDown(void)
{
        P_SendByte(CLKCTR,0x80); // set PWD in CLKCTR
}

// this function configures the 82C900 for the sleep mode, and set
// the wake up modes CANAWU(actA) and CANBWU(actB)
void sleep(char actA, char actB)
{
  char cfg;
  cfg = P_ReadByte(CLKCTR)&0x30;
  if(actA) cfg |= 0x01; // wakeup on MSG at A
  if(actB) cfg |= 0x02; // wakeup on MSG at B
  P_SendByte(CLKCTR,cfg); // set CAN(A/B)WU an reset CANCLK
}

// configures MSGOBJ obj_nr with the ID id as SLPMSG. The object has
// two Data bytes. First must be sleep_code and the second is written to
// register CLKCTR. The MSGOBJ is related to node.
void init_SleepObj(char obj_nr, char id, char node, char sleep_code)
{
  int cfg = 0x0080; // set SLPEN: enable power down via CANMSG
  init_MSG(node, obj_nr, 0, 2, id);// receiving object, 2 data bytes
  cfg |= obj_nr & 0x1F; // set sleepmsg to obj_nr
  cfg |= sleep_code << 8; // configure sleep code
  P_SendWord(CANPWD, cfg);
}
```

http://www.infineon.com

Published by Infineon Technologies AG