Microcontrollers

ApNote AP2900

☐ additional file
APXXXX01.EXE available

Connecting C166 and C500 Microcontrollers to CAN

This application note describes the CAN Protocol, Siemens CAN devices and different options for connecting devices of the Siemens C166 and C500 microcontroller families to the Controller Area Network (CAN).

Author: Axel Wolf / Siemens HL MC AT

AP2900 ApNote - Revision History		
Actual Revision : Rel.02	Previous Revison: Rel. 01	
Page of actual Rel.	Page of prev. Rel.	Subjects changes since last release)
---	---	Minor changes; Register names 81C90/91 adjusted.

Abbreviations and Symbols

The following abbreviations and symbols are used throughout this application note:

ABUS	"Allgemeine Bitserielle Universelle Schnittstelle" (VW protocol)
BL1, BL2	Bit timing registers of the SAE 81C90/91
BRP	Baudrate Prescaler
BSP	Bit Stream Processor
BTL	Bit Timing Logic
CAN	Controller Area Network (Bosch)
CAN_H, CAN_L	Names of the CAN bus lines
CCE	Bit in the Control Register of the CAN module
CiA	CAN in Automation (CAN User Group)
CIL	CPU Interface Logic
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
CTRL	Control Register of the SAE 81C90/91
EML	Error Management Logic
IC	Integrated Circuit
IM	Bit in the MOD Register of the SAE 81C90/91
INIT	Bit in the Control Register of the CAN module
INT	Interrupt Register of the SAE 81C90/91
J1850	Protocol from Chrysler, GM, Ford
MOD	Status- and Control Register of the SAE 81C90/91
NDA	Non-Destructive Arbitration
NRZ	Non-Return-to-Zero
PEC	Peripheral Event Controller
PLCC	Plastic Leaded Chip Carrier
PWM	Pulse Width Modulation
RAM	Random Access Memory
REC	Receive Error Counter
RES	Bit in the MOD Register of the SAE 81C90/91
ROM	Read Only Memory
RTR	Remote Transmission Request
Rx0, Rx1	Receive Inputs of the SAE 81C90/91
SJW	Synchronisation Jump Width
TCEC	Transmit Check Error Counter
TCL	Transceiver Control Logic
TEC	Transmit Error Counter
TSEG1, TSEG2	Segments of the CAN bit cell
Tx0, Tx1	Transmit Outputs of the SAE 81C90/91
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
VAN	Vehicle Area Network (Peugeot, Renault)
XBUS	Chip-internal bus on the C167CR

This application note describes the CAN[1] Protocol, Siemens CAN devices and different options for connecting devices of the Siemens C166 and C500 microcontroller families to the Controller Area Network (CAN). It replaces the German application note "CAN-Anschluß für die Mikrocontrollerfamilien C166 and C500".

After the introduction (Section 1), Section 2 introduces the reader to the Controller Area Network (CAN). Section 3 describes the Siemens CAN devices, the 16-bit microcontroller C167CR, the 8-bit microcontroller C515C, and the Stand-Alone Full-CAN Controller SAE 81C90/91. Section 4 gives hardware examples for connection of Siemens microcontrollers to CAN. Finally, in Section 5, ways of handling the SAE 81C90/91 and the CAN module on the C167CR and the C515C are given.

Whilst every effort has been made to ensure the accuracy of information contained in this application note, the authors cannot be held responsible for any consequences arising from its use.

2 Introduction

Ever increasing quantities of electronic devices are fitted to modern motor vehicles. Examples of such devices include engine management systems, active suspension, ABS, gear control, lighting control, air conditioning, airbags and central locking. All this means more safety and more comfort for the driver and of course a reduction of fuel consumption and exhaust emissions. To improve the behaviour of the vehicle even further, it is necessary for the different control systems (and their sensors) to exchange information. At present, this is usually done by discrete interconnection of the different systems (i.e. point to point wiring). The requirement for information exchange has now grown to such an extent that a cable network with a length of up to several kilometers and many connectors is required. This produces growing problems concerning material cost, production time and reliability.
The solution to this problem is the connection of the control systems via a serial bus system. This bus has to fulfill some special requirements due to its usage in a vehicle:

- **Data Rate**

Much of the data that is exchanged by the control systems (or by sensors) has to be processed in real time which requires very fast transmission. Data items may differ in transmission priority (e.g. Airbag data is likely to be high priority, Air Conditioning data is likely to be low priority). For very high priority data the latency period between the transmission request and the actual start of the transmission is very important and must be minimized. The serial bus therefore must provide very fast transmission, short message length and bus access prioritisation.

- **Data Integrity**

The bus must have a high resistance to Electromagnetic Interference. Erroneous messages must be detected and repeated. To ensure valid data across the sytem, every

[1] Controller Area Network (CAN): License of Robert Bosch GmbH

bus node has to be informed about an error. Bus communication must not be disturbed if one (or more) bus nodes malfunction. Faulty nodes must withdraw from bus communication on their own. The bus must therefore have a linear structure with equal bus nodes (multimaster concept).

- **Data Sharing**

All control systems needing a common data item should be able to simultaneously receive this data item from the bus (e.g. vehicle speed might be used by the engine management system, active suspension ABS and gear control) The bus must therefore have multicast capability.

Several different serial bus systems have been developed for the use in motor vehicles, each of them trying, as far as possible, to fulfill the above requirements. Examples are ABUS from Volkswagen, VAN (Vehicle Area Network) developed by Peugeot and Renault, the J1850 protocol from Chrysler, General Motors and Ford and, of course, the Controller Area Network (CAN) from Robert Bosch GmbH in Germany. These protocols mostly differ in transfer rate, signal coding, message format, error detection, and error handling.

The CAN protocol was defined by Bosch in the mid-eighties. For some time Siemens have also offered CAN devices such as the stand-alone Full-CAN controller SAE 81C90/91 and the C167CR and the C515C microcontrollers (high-end 16-bit or 8-bit microcontrollers respectively with an on-chip CAN module).

CAN has now clearly established a market leading position and indeed a number of vehicle manufacturers have abandoned their proprietry protocols and have chosen the CAN protocol. More than 15 Million CAN nodes are in use worldwide.

2 The Controller Area Network (CAN)

2.1 CAN Basics

CAN is an asynchronous serial bus system with one logical bus line. It has an open, linear bus structure with equal bus nodes. 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. for addition of system function, error recovery or bus monitoring).

The bus logic corresponds to a "wired-AND" mechanism, "recessive" bits (mostly, but not necessarily equivalent to the logic level "1") are overwritten by "dominant" bits (logic level mostly "0"). 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 bus node generates the dominant bus state. Therefore, for the CAN bus line, a medium must be chosen that is able to transmit the two possible bit states (dominant and recessive). One of the most common and cheapest ways is to use a twisted wire pair. The bus lines are then called "CAN_H" and "CAN_L" and may be connected directly to the nodes or via a connector. There's no standard defined by CAN regarding the connector to be used. The twisted wire pair is terminated by terminating resistors at each end of the bus line. The maximum bus speed is 1 MBaud, which can be achieved with a bus length of up to 40 m. For bus lengths longer than 40 m the bus speed must be reduced (a 1000 m bus can be realised with a 40 KBaud bus speed). For a bus length above 1000 m special drivers should be used. At least 30 nodes may be connected without additional equipment. Due to the differential nature of transmission CAN is insensitive to EMI, because both bus lines are affected in the same way which leaves the differential signal unaffected. The bus lines can also be shielded to reduce the electromagnetic emission of the bus itself, especially at high baudrates.

The binary data is coded corresponding to the NRZ code (Non-Return-to-Zero; low level = dominant state; high level = recessive state). To ensure exact synchronization of all bus nodes bit stuffing is used. This means that during the transmission of a message a maximum of five consecutive bits may have the same polarity. Whenever five consecutive bits of the same polarity have been transmitted the transmitter will insert one additional bit of the opposite polarity into the bit stream before transmitting further bits. The receiver also checks the number of bits with the same polarity and removes the stuff bits from the bit stream (= destuffing).

2.2 Addressing and Bus Arbitration

In the CAN protocol it is not bus nodes that are addressed, but the address information is contained in the messages that are transmitted. This is done via an identifier (part of each message) which identifies the message content (e.g. engine speed, oil temperature etc.) The identifier additionally indicates the priority of the message. The lower the binary value of the identifier the higher is the priority of the message.

For bus arbitration, CSMA/CD with NDA is used (Carrier Sense Multiple Access / Collision Detection with Non-Destructive Arbitration). If bus node A wants to transmit a message across the network, it at first checks that the bus is in the idle state ("Carrier Sense") i.e. no node is currently transmitting. If this is the case (and no other node wishes to start a transmission at the same moment) node A becomes the bus master and sends its message. All other nodes switch to receive mode during the first transmitted bit (Start Of Frame bit). After correct reception of the message (which is acknowledged by each node) each bus node checks the message identifier and stores the message, if required. Otherwise, the message is discarded.

If two or more bus nodes start their transmission at the same time ("Multiple Access"), collision of the messages is avoided by bitwise arbitration ("Collision Detection / Non-Destructive Arbitration" together with the "Wired-AND" mechanism, "dominant" bits override "recessive" bits). Each node sends the bits of its message identifier (MSB first) and monitors the bus level. A node that sends a recessive identifier bit but reads back a dominant one loses bus arbitration and switches to receive mode. This condition occurs when the message identifier of a competing node has a lower binary value (dominant state = logic 0) and therefore the competing node is sending a message with a higher priority. In this way, the bus node with the highest priority message wins arbitration without losing time by having to repeat the message. All other nodes automatically try to repeat their transmission once the bus returns to the idle state. It is not permitted for different nodes to send messages with the same identifier as arbitration could fail leading to collisions and errors.

The original CAN specifications (Versions 1.0, 1.2 and 2.0A) defined the message identifier as having a length of 11 bits giving a possible 2048 message identifiers. The specification has since been updated (to version 2.0B) to remove this possible limitation. CAN specification Version 2.0B allows message identifier lengths of 11 and/or 29 bits to be used (an identifier length of 29 bits allows over 536 Million message identifiers). Version 2.0B CAN is often referred to as "Extended CAN", the versions below (1.0, 1.2 and 2.0A) being referred to as "Standard CAN" (see section 2.6.1 for further details).

2.3 The different CAN Frames and their Formats

2.3.1 Data Frame

2.3.1.1 Standard CAN Data Frame

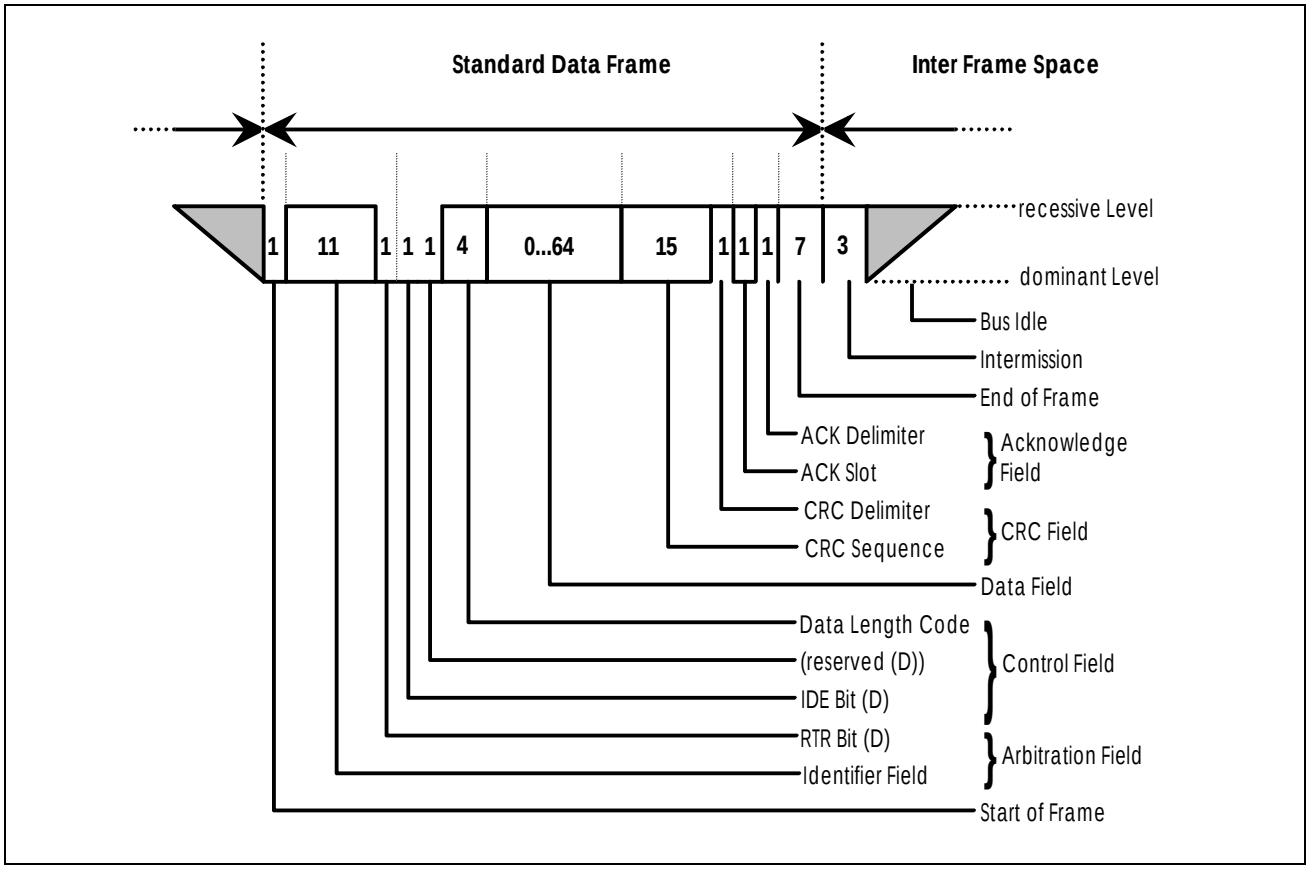

Figure 2.3-1:
Standard CAN Data Frame

A "Data Frame" is generated by a node when the node wishes to transmit data. The Standard CAN Data Frame is shown in figure 2.3-1. In common with all other frames, the frame begins with a Start Of Frame bit (SOF = dominant state) for hard synchronization of all nodes.

The SOF is followed by the Arbitration Field consisting of 12 bits, the 11-bit Identifier (reflecting the contents and priority of the message) and the RTR bit (Remote Transmission Request bit). The RTR bit is used to distinguish a Data Frame (RTR = dominant) from a Remote Frame (see section 2.3.2).

The next field is the Control Field, consisting of 6 bits. The first bit of this field is called the IDE bit (Identifier Extension) and is at dominant state to specify that the frame is a Standard Frame. The following bit is reserved and defined as a dominant bit. The remaining 4 bits of the Control Field are the Data Length Code (DLC) and specify the number of bytes of data contained in the message (0 - 8 bytes).

The data being sent follows in the Data Field which is of the length defined by the DLC above (0, 8, 16, ..., 56 or 64 bits).

The Cyclic Redundancy Field (CRC) follows and is used to detect possible transmission errors. The CRC Field consists of a 15 bit CRC sequence, completed by the recessive CRC Delimiter bit.

The final field is the Acknowledge Field. During the ACK Slot bit the transmitting node sends out a recessive bit. Any node that has received an error free frame acknowledges the correct reception of the frame by sending back a dominant bit (regardless of whether the node is configured to accept that specific message or not). From this it can be seen that CAN belongs to the "in-bit-response" group of protocols. The recessive Acknowledge Delimiter completes the Acknowledge Slot and may not be overwritten by a dominant bit.

Seven recessive bits (End of Frame) end the Data Frame. Between any two frames the bus must remain in the recessive state for at least a further 3 bit times (called Intermission). If, following the frame, no nodes wish to transmit then the bus stays in the recessive state (Bus Idle).

2.3.1.1 Extended CAN Data Frame

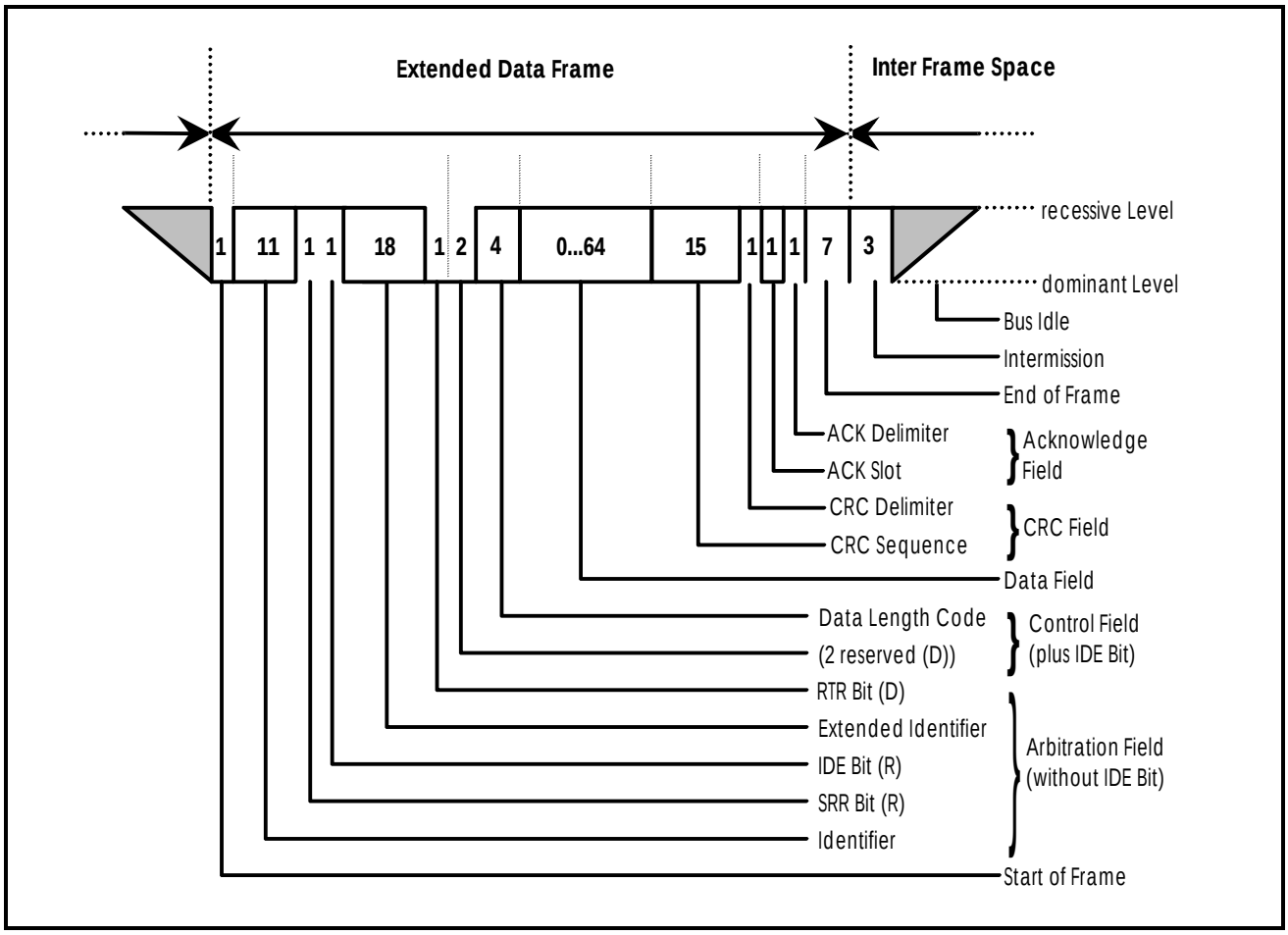

Figure 2.3-2:
Extended CAN Data Frame

The Extended CAN Data Frame is shown in figure 2.3-2.

It should be noted above, that to enable standard and extended frames to be sent across a shared network it is necessary to split the 29 bit extended message identifier into 11bit (most significant) and 18 bit (least significant) sections. This split ensures that the Identifier Extension bit (IDE) can remain at the same bit position in both standard and extended frames (see below).

In the Extended CAN Data Frame the Start Of Frame bit (SOF) is followed by the Arbitration Field consisting of 32 bits. The first 11 bits are the 11 most significant bits of the 29-bit Identifier ("Base-ID"). These 11 bits are followed by the Substitute Remote Request bit (SRR) which is transmitted as recessive. The SRR is followed by the IDE bit which is recessive to denote that the frame is an Extended CAN frame. It should be noted from this, that if arbitration remains unresolved after transmission of the first 11 bits of the identifier, and one of the nodes involved in arbitration is sending a Standard CAN frame (11 bit identifier), then the Standard CAN frame will win arbitration due to the assertion of a dominant IDE bit. Further to this the SRR bit in an Extended CAN frame must be recessive to allow the assertion of a dominant RTR bit by a node that is sending a Standard CAN Remote Frame (see section 2.3.2.1). The SRR and IDE bits are followed by the remaining 18 bits of the identifier ("ID-Extension") and the Remote Transmission Request bit (again RTR = dominant for a Data Frame).

The next field is the Control Field, consisting of 6 bits. The first 2 bits of this field are reserved and are at dominant state. The remaining 4 bits of the Control Field are the Data Length Code (DLC) and specify the number of data bytes (as for the Standard Data Frame).

The remaining portion of the frame (Data Field, CRC Field, Acknowledge Field, End Of Frame and Intermission) is constructed in the same way as for a Standard Data Frame.

2.3.2 Remote Frame

2.3.2.1 Standard CAN Remote Frame

Normally data transmission is performed on an autonomous basis with the data source node (e.g. a sensor) sending out a Data Frame. It is also possible, however, for a destination node (or nodes) to request the data from the source. For this purpose the destination node sends a "Remote Frame" with an identifier that matches the identifier of the required Data Frame. The appropriate data source node will then send a Data Frame as a response to this remote request.

There are 2 differences between a Remote Frame and a Data Frame. Firstly the RTR-bit is at the recessive state and secondly there is no Data Field. In the very unlikely event of a Data Frame and a Remote Frame with the same identifier being transmitted at the same time, the Data Frame wins arbitration due to the dominant RTR bit following the identifier. In this way, the node that transmitted the Remote Frame receives the desired data immediately.

The format of a Standard Remote Frame is shown in figure 2.3-3 below.

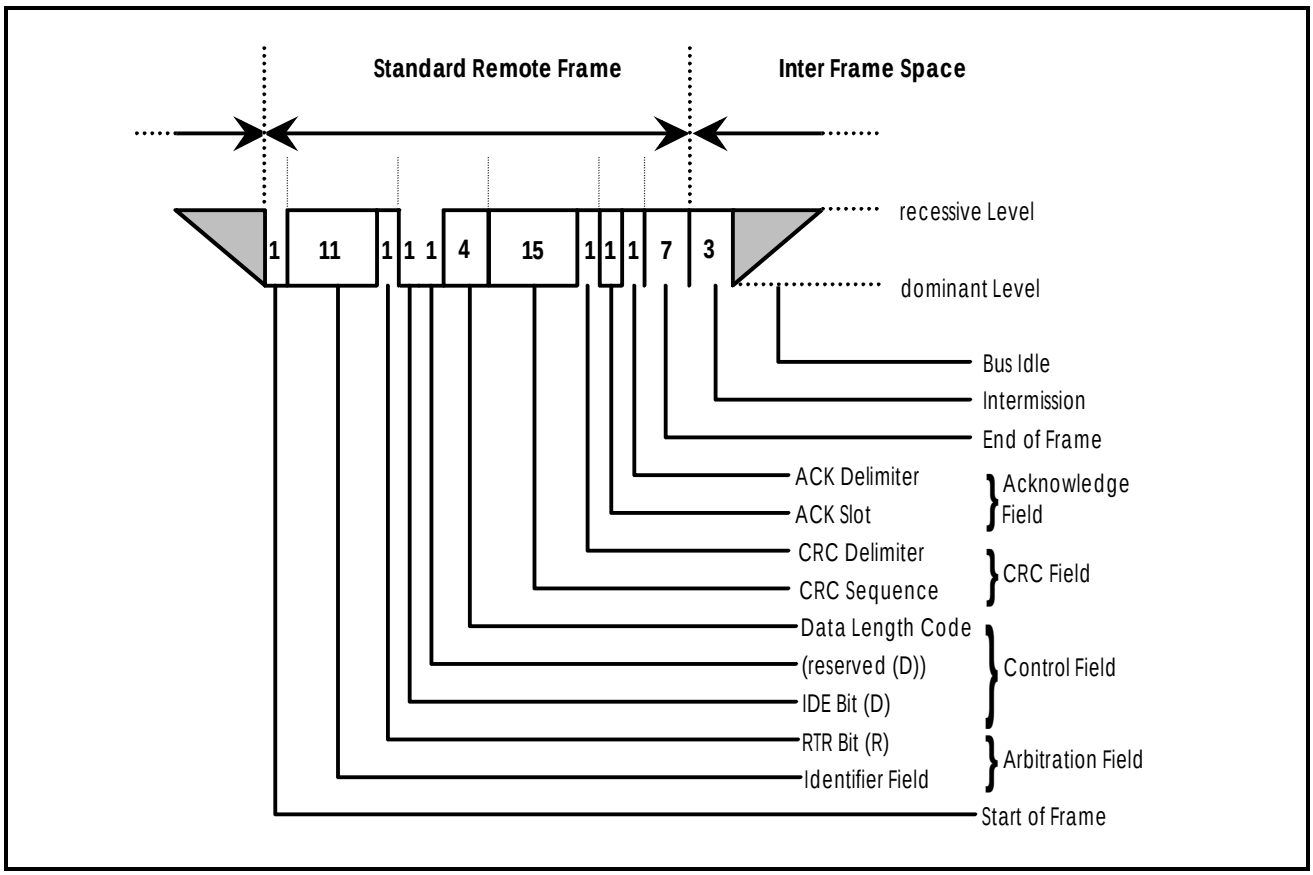

Figure 2.3-3:
Standard CAN Remote Frame

2.3.2.2 Extended CAN Remote Frame

The format of an Extended CAN Remote Frame is shown in figure 2.3-4 below.

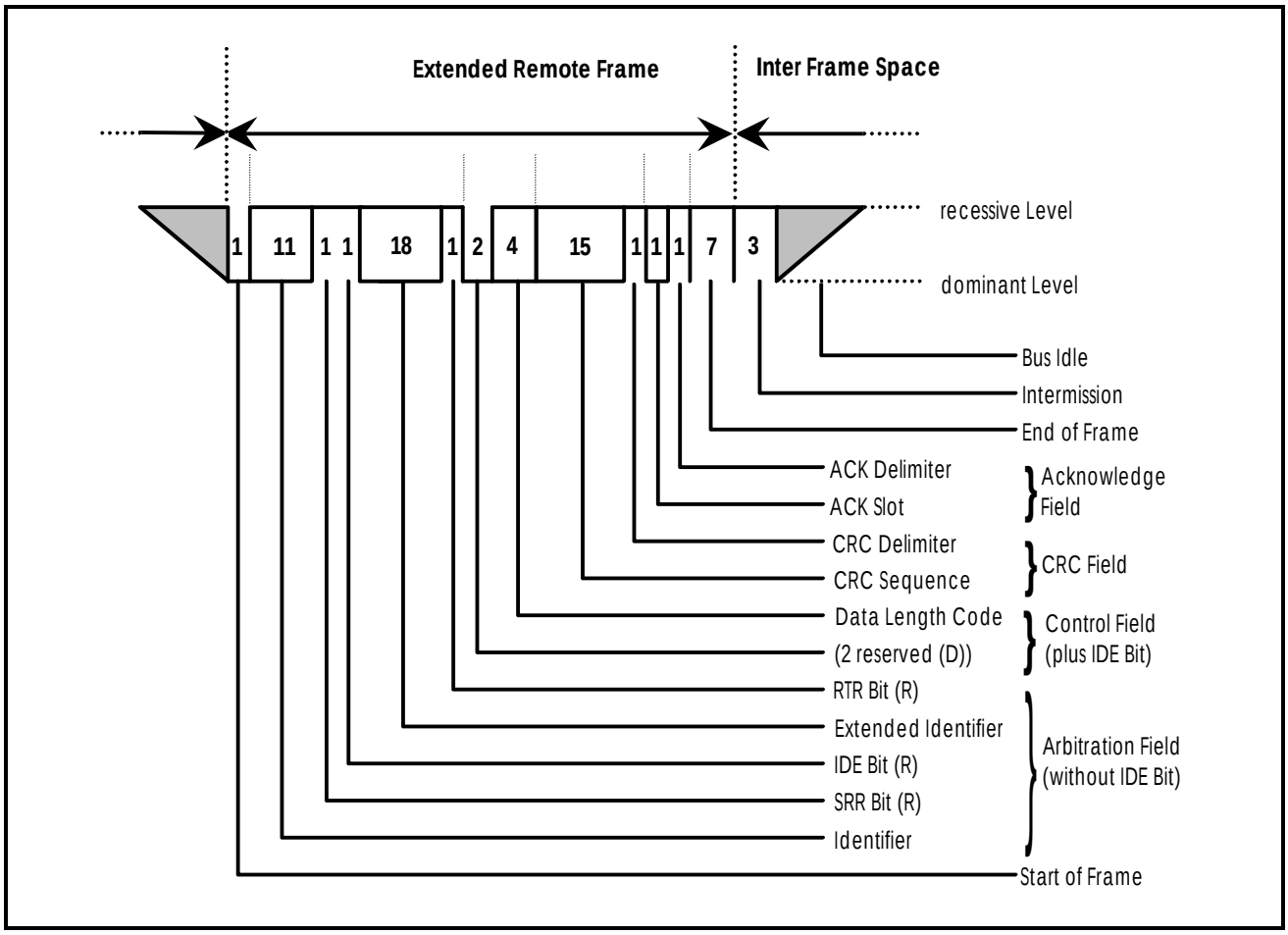

Figure 2.3-4:
Extended CAN Remote Frame

2.3.3 Error Frames, Overload Frame, Interframe Space

2.3.3.1 Error Frames

An *Error Frame* is generated by any node that detects a bus error. An error frame consists of 2 fields, an Error Flag field followed by an Error Delimiter field. The Error Delimiter consists of 8 recessive bits and allows the bus nodes to restart bus communications cleanly after an error. There are, however, two forms of Error Flag fields. The form of the Error Flag field depends on the “error status” of the node that detects the error (see section 2.5 for details of “error status”).

If an “error-active” node detects a bus error then the node interrupts transmission of the current message by generating an “active error flag”. The “active error flag” is composed of six consecutive dominant bits. This bit sequence actively violates the bit stuffing rule. All other stations recognize the resulting bit stuffing error and in turn generate Error Frames themselves. The Error Flag field therefore consists of between six and twelve consecutive dominant bits (generated by one or more nodes). The Error Delimiter field completes the Error Frame. After completion of the Error Frame bus activity returns to normal and the interrupted node attempts to resend the aborted message.

if an “error passive” node detects a bus error then the node transmits an “error passive flag” followed, again, by the Error Delimiter field. The “error passive flag” consists of six consequtive recessive bits, and therefore the Error Frame (for an “error passive” node) consists of 14 recessive bits (i.e. no dominant bits). From this it follows that, unless the bus error is detected by the node that is actually transmitting (i.e. is the bus master), the transmission of an Error Frame by an “error passive” node will not affect any other node on the network. If the bus master node generates an “error passive flag” then this may cause other nodes to generate error frames due to the resulting bit stuffing violation. After transmission of an Error Frame an “error passive” node must wait for 6 consecutive recessive bits on the bus before attempting to rejoin bus communications.

2.3.3.2 Overload Frame

An Overload Frame has the same format as an “active” Error Frame (i.e. that generated by an “error active” node). An Overload Frame, however can only be generated during Interframe Space. This is the way then an Overload Frame can be differentiated from an Error Frame (an Error Frame is sent during the transmission of a message). The Overload Frame consists of 2 fields, an Overload Flag followed by an Overload Delimiter. The Overload Flag consists of six dominant bits followed by Overload Flags generated by other nodes (as for “active error flag”, again giving a maximum of twelve dominant bits). The Overload Delimiter consists of eight recessive bits. An *Overload Frame* can be generated by a node as a result of 2 conditions, a) if the node detects a dominant bit during Interframe Space (illegal see section 2.5), or b) if due to internal conditions, the node is not yet able to start reception of the next message. A node may generate a maximum of 2 sequential Overload Frames to delay the start of the next message.

2.3.3.3 Interframe Space

Interframe Space separates a preceeding frame (of whatever type) from a following Data or Remote Frame. Interframe space is composed of at least 3 recessive bits, these bits are termed the Intermission. This time is provided to allow nodes time for internal processing before the start of the next message frame. After the Intermission, the bus line remains in the recessive state (Bus Idle) until the next transmission starts.

2.4 The Nominal Bit Time

One bit cell (i.e. one high or low pulse of the NRZ code) is constructed from four segments. Each segment is in turn constructed from an integer multiple of Time Quanta. The Time Quantum is the smallest discrete timing resolution used by a CAN node. A bit time, and therefore by definition also the bit rate, is selected by programming the Bit Timing Logic (BTL) to select the width of the Time Quanta and the number of Time Quanta in the various segments. The nominal bit time with its segments (according to CAN Spec ISO11898, 1993) is shown in figure 2.4-1.

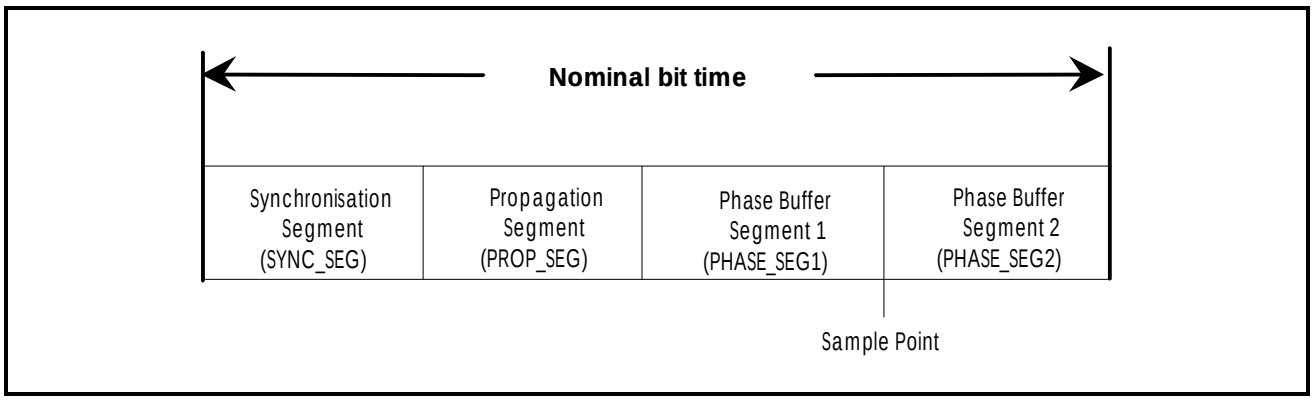

Figure 2.4-1:
Partition of bit time

The SYNC_SEG (Synchronisation Segment) is used to synchronise the various bus nodes. If there is a bit state change between the previous bit and the current bit then the bus state change is expected to occur within this segment. The length of this segment is always 1 Time Quantum (1 BTL cycle).

The PROP_SEG (Propagation Segment) is used to compensate for signal delays across the network. These delays are caused by signal propagation delay on the bus line and through the electronic interface circuits of the bus nodes. PROP_SEG may be 1, 2, 3, ..., 8 or more Time Quanta long. For many CAN module implementations the PROP_SEG and PHASE_SEG1 segments are combined for ease of programming.

The PHASE_SEG1 and PHASE_SEG2 segments are used to compensate for edge phase errors. These segments may be lengthened or shortened by resynchronization. PHASE_SEG1 may be 1, 2, 3, ..., 8 or more Time Quanta long. PHASE_SEG2 is the maximum of PHASE_SEG1 and the information processing time, which is the time segment starting with the sample point reserved for calculation of the subsequent bit level and is less than or equal to two Time Quanta long. The sample point is the point of time at

which the bus level is read and interpreted as the value of that respective bit. Its location is at the end of PHASE_SEG1.

The total number of Time Quanta in a bit time must be between 8 and 25.

As a result of resynchronization, PHASE_SEG1 may be lengthened or PHASE_SEG2 may be shortened. The amount of lengthening or shortening the phase buffer segments has an upper limit given by the resynchronization jump width. The resynchronization jump width may be between 1 and 4 Time Quanta, but it may not be longer than PHASE_SEG1.

2.5 Error Detection and Error Handling

The CAN protocol provides sophisticated error detection mechanisms. The following errors can be detected:

- **Cyclic Redundancy Check (CRC) Error:**

With the Cyclic Redundancy Check the transmitter calculates special check bits for the bit sequence from the start of a frame until the end of the Data Field. This CRC sequence is transmitted in the CRC Field. The receiving node also calculates the CRC sequence using the same formula and performs a comparison to the received sequence. If a mismatch is detected, a CRC error has occured and an Error Frame is generated. The message is repeated.

- **Acknowledge Error:**

In the Acknowledge Field of a message the transmitter checks if the Acknowledge Slot (which it has sent out as a recessive bit) contains a dominant bit. If not, no other node has received the frame correctly, an Acknowledge Error has occured and the message has to be repeated. No Error Frame is generated, though.

- **Form Error:**

If a transmitter detects a dominant bit in one of the four segments End of Frame, Interframe Space, Acknowledge Delimiter or CRC Delimiter then a Form Error has occured and an Error Frame is generated. The message is repeated.

- **Bit Error:**

A Bit Error occurs if a) a transmitter sends a dominant bit and detects a recessive bit or b) if it sends a recessive bit and detects a dominant bit when monitoring the actual bus level and comparing it to the just transmitted bit. In case b) no error occurs during the Arbitration Field and the Acknowledge Slot.

- **Stuff Error:**

If between Start of Frame and CRC Delimiter 6 consecutive bits with the same polarity are detected, the bit stuffing rule has been violated. A stuff error occurs and an Error Frame is generated. The message is repeated.

Detected errors are made public to all other nodes via Error Frames. The transmission of the erroneous message is aborted and the frame is repeated as soon as possible. Furthermore, each CAN node is in one of the three error states "error active", "error passive" oder "bus off" according to the value of the internal error counters. The error-active state is the usual state where the bus node can transmit messages and active Error Frames (made of dominant bits) without any restrictions. In the error-passive state, messages and passive Error Frames (made of recessive bits) may be transmitted. The bus-off state makes it temporarily impossible for the station to participate in the bus communication. During this state, messages can neither be received nor transmitted.

2.6 Different CAN Implementations

2.6.1 Standard CAN, Extended CAN

Those Data Frames and Remote Frames, which only contain the 11-bit identifier, are called Standard Frames according to CAN specification V2.0 part A. With these frames, 2^{11} (=2048) different messages can be identified (identifiers 0-2047). However, the 16 messages with the lowest priority (2032-2047) are reserved. Extended Frames according to CAN specification V2.0 part B own a 29-bit identifier. As already mentioned, this 29-bit identifier is made up of the 11-bit identifier ("Base ID") and the 18-bit Extended Identifier ("ID Extension"). So 2^{29} different identifiers are possible.

CAN modules specified after CAN V2.0 part A are only able to transmit and receive Standard Frames according to the Standard CAN protocol. Messages using the 29-bit identifier cause errors. If a device is specified after CAN V2.0 part B, there’s one more distinction. Modules named "Part B Passive" can transmit and receive Standard Frames but tolerate Extended Frames without generating Error Frames. "Part B Active" devices are able to transmit and receive both Standard and Extended Frames.

2.6.2 Basic CAN, Full CAN

There is one more CAN characteristic concerning the interface between the CAN module and the host CPU, dividing CAN chips into "Basic CAN" and "Full CAN" devices. This has nothing to do with the used protocol though (Standard or Extended CAN), which makes it possible to use both Basic and Full CAN devices in the same network.

In the Basic CAN devices, only basic functions of the protocol are implemented in hardware, e.g. the generation and the check of the bit steam. The decision, if a received message has to be stored or not (acceptance filtering) and the whole message management has to be done by software, i.e. by the host CPU. Mostly the CAN chip also only provides one transmit buffer and one or two receive buffers. So the host CPU load is quite high using Basic CAN modules, therefore these devices should only be used at low baudrates and low bus loads with only a few different messages. The advantages of Basic CAN are the small chip size leading to low costs of these devices.

Full CAN devices do the whole bus protocol in hardware including the acceptance filtering and the message management. They contain several so called message objects which handle the identifier, the data, the direction (receive or transmit) and the information Standard CAN / Extended CAN. During the initialisation of the device, the host CPU

defines which messages are to be sent and which are to be received. The host CPU is informed by interrupt if the identifier of a received message matches with one of the programmed (receive-) message objects. In this way, the CPU load is strongly reduced. Using Full CAN devices, high baudrates and high bus loads with many messages can be handled. These chips are more expensive than the Basic CAN devices, though.

Many Full CAN chips provide a "Basic-CAN-Feature": One of their message objects can be programmed in a way that every message is stored there that does not match with one of the other message objects. This can be very helpful in a number of applications.

3. The Siemens CAN Devices C167CR, C515C and SAE 81C90/91

3.1 The Microcontroller Families C500 and C166 at a Glance

The history of the 8-bit microcontrollers from Siemens is founded on 8051-compatible derivatives like the SAB 80C515 or the SAB 80C517. The improvement of these controllers lead to the C500 family whose C500 core is still fully compatible to the old 8051 core but provides more performance and a more flexible design methodology for further integration. An actual high-performance member of the C500 family is the C515C. It is based on the well-known SAB 80C515A, but has some additional features like an SPI-compatible, synchronous serial interface (SSC) and a CAN module which will be described in detail in section 3.2.

To meet the requirements of today's and future embedded control applications Siemens developed the SAB 80C166 in 1990 being the first member of the 16-bit microcontroller family C166. The controllers own a registerbank-oriented CPU with a four-stage pipeline being able to process most of the instructions in just one machine cycle of 80 ns @ 25 MHz CPU clock. The freely programmable interrupt system has response times of typically 400 ns and can handle a large number of independent internal and external interrupt sources at 16 priority levels. The family members SAB 80C166, C167, C165 and C163 are equipped with a well-balanced mix of modular, autonomous peripherals like a 10-bit ADC with up to 16 channels, Capture/Compare units, serial interfaces, a PWM unit, complex timer units and also a CAN module. The implemented XBUS makes it easy to extend the standard derivatives by further application specific peripheral functions. The controllers contain up to 4 Kbytes internal RAM and up to 128 Kbytes ROM or Flash-EPROM, respectively. The whole C166 architecture is made for fast instruction execution and minimum response time to external events. This combination, however, provides highest real-time performance.

3.2 The CAN Module on the C167CR / C515C

3.2.1 The Functional Blocks of the CAN Module

The CAN modules on the C167CR and the C515C are fully compatible concerning the CAN functionality. There are only differences in the interface to the CPU, the module clock generation, the internal registers (8 bit wide on the C515C, 16-bit wide on the C167CR), and the interrupt functionality. A block diagram of the CAN module is shown in figure 3.2-1 (connected to the C167CR here).

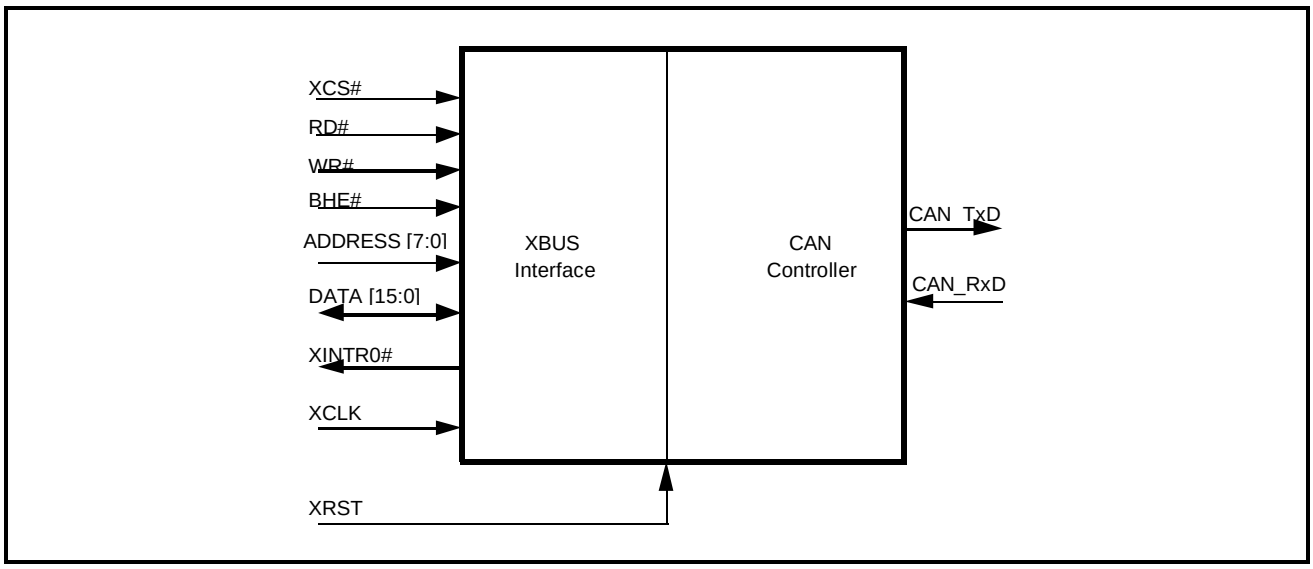

Figure 3.2-1:
Block Diagram of the CAN module (connected to the C167CR here)

The CAN module is made of two major blocks. One of them represents the interface to the CPU.

In the C167CR, the CAN module is connected to the bus controller (and therefore to the CPU) via the XBUS. From a user's point of view, this XBUS can be regarded as an internal representation of the external bus. Connecting the CAN module to the XBUS in 16-bit demultiplexed bus mode offers the fastest possible accesses. All registers of the CAN module are organized as 16-bit registers, located on word addresses. However, all registers may be accessed bytewise in order to select special actions without affecting other mechanisms. These registers reside in a special CAN address area of 256 bytes, which is mapped into segment 0 and uses addresses EF00h through EFFFh.

In the C515C, the CAN module is connected to the internal bus. Again, the registers are mapped into a special address area of 256 bytes using addresses F700h through F7FFh. The registers can be accessed with MOVX-instructions. An access to this special memory space requires the modification of the XPAGE register to the value F7h. Furthermore, the bits XMAP0 and XMAP1 in the SYSCON register must be set to "0".

The second block of the CAN module on the C167CR and the C515C is the CAN controller itself, which is derived from the stand-alone component AN 82527. The CAN controller provides all resources that are required to run the Standard CAN protocol (11-bit identifiers) as well as the Extended CAN protocol (29-bit identifiers). It provides a sophisticated object layer to relieve the CPU of as much overhead as possible when controlling many different message objects (up to 15). This includes bus arbitration, resending of garbled messages, error handling, interrupt generation, etc. In order to implement the physical layer, external CAN transceiver components have to be connected to the C167CR / C515C.

The CAN controller combines several functional blocks (see figure 3.2-2) that work in parallel and contribute to the controller's performance. These units and the functions they provide are described below.

- The *Transmit / Receive Shift Register* holds the destuffed bit stream from the bus line to allow the parallel access to the whole data or Remote Frame for the acceptance match test and the parallel transfer of the frame to and from the Intelligent Memory.

- The *Bit Stream Processor (BSP)* is a sequencer controlling the sequential data stream between the Tx/Rx Shift Register, the CRC Register, and the bus line. The BSP also controls the Error Management Logic (EML) and the parallel data stream between the Tx/Rx Shift Register and the Intelligent Memory such that the processes of reception, arbitration, transmission, and error signalling are performed according to the CAN protocol. Note that the automatic retransmission of messages which have been corrupted by noise or other external error conditions on the bus line is handled by the BSP.

- The *Cyclic Redundancy Check Register* generates the Cyclic Redundancy Check (CRC) code to be transmitted after the data bytes and checks the CRC code of incoming messages. This is done by dividing the data stream by the code generator polynomial.

- The *Error Management Logic (EML)* is responsible for the fault confinement of the CAN device. Its counters, the Receive Error Counter and the Transmit Error Counter, are incremented and decremented by commands from the Bit Stream Processor. According to the values of the error counters, the CAN controller is set into the states *error active*, *error passive* or *busoff*.

- The CAN controller is *error active*, if both error counters are below the *error passive* limit of 128. It is *error passive*, if at least one of the error counters equals or exceeds 128. It goes *busoff*, if the Transmit Error Counter equals or exceeds the *busoff* limit of 256. The device remains in this state, until the *busoff* recovery sequence is finished. Additionally, there is the bit EWRN in the Status Register, which is set, if at least one of the error counters equals or exceeds the error warning limit of 96. EWRN is reset if both error counters are less than the error warning limit.

- The *Bit Timing Logic (BTL)* monitors the busline input CAN_RxD and handles the busline related bit timing according to the CAN protocol. The BTL synchronises on a *recessive* to *dominant* busline transition at *Start of Frame* (hard synchronisation) and on any further *recessive* to *dominant* busline transition, if the CAN controller itself does not transmit a *dominant* bit (resynchronisation). The BTL also provides programmable time segments to compensate for the propagation delay time and for phase shifts and to define the position of the *Sample Point* in the bit time. The programming of the BTL depends on the baudrate and on external physical delay times.

- The *Intelligent Memory* (CAM/RAM Array) provides storage for up to 15 message objects of maximum 8 data bytes length. Each of these objects has a unique identifier and its own set of control and status bits. After the initial configuration, the Intelligent Memory can handle the reception and transmission of data without further CPU actions.

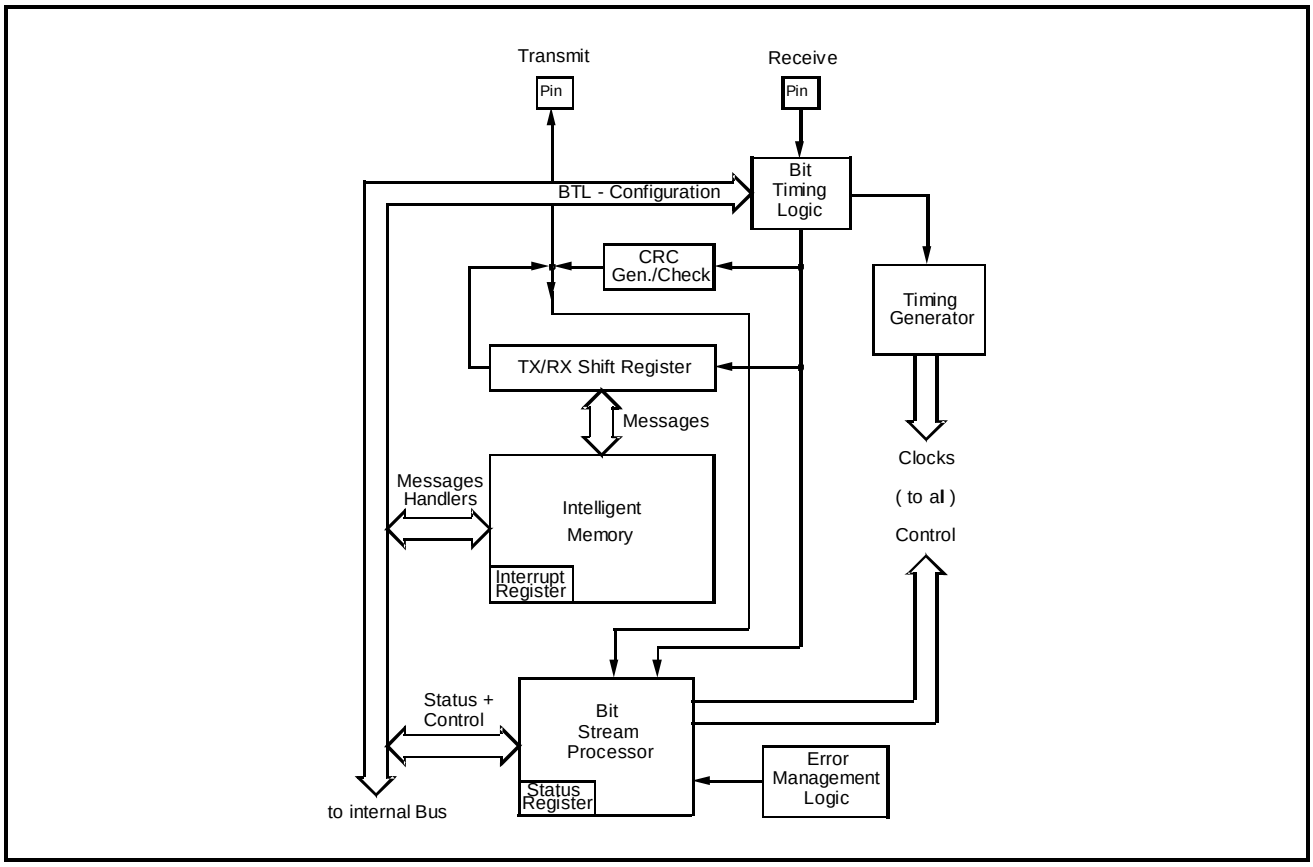

Figure 3.2-2:
Block Diagram of the CAN Controller on the C167CR / C515C

3.2.2 Control Registers of the CAN Controller

All registers reside in the already mentioned 256-byte wide CAN address areas (shown in figure 3.2-3) together with the message objects. The most important registers are now described.

The *Control Register* and the *Status Register* contain important status- and control bits e.g. used for initialisation and interrupt control. The cause for a pending interrupt can be read from the *Interrupt-Register.* In the *Bit-Timing-Register*, the desired baudrate on the CAN bus and the length of the time segments described in section 2.4 are defined.

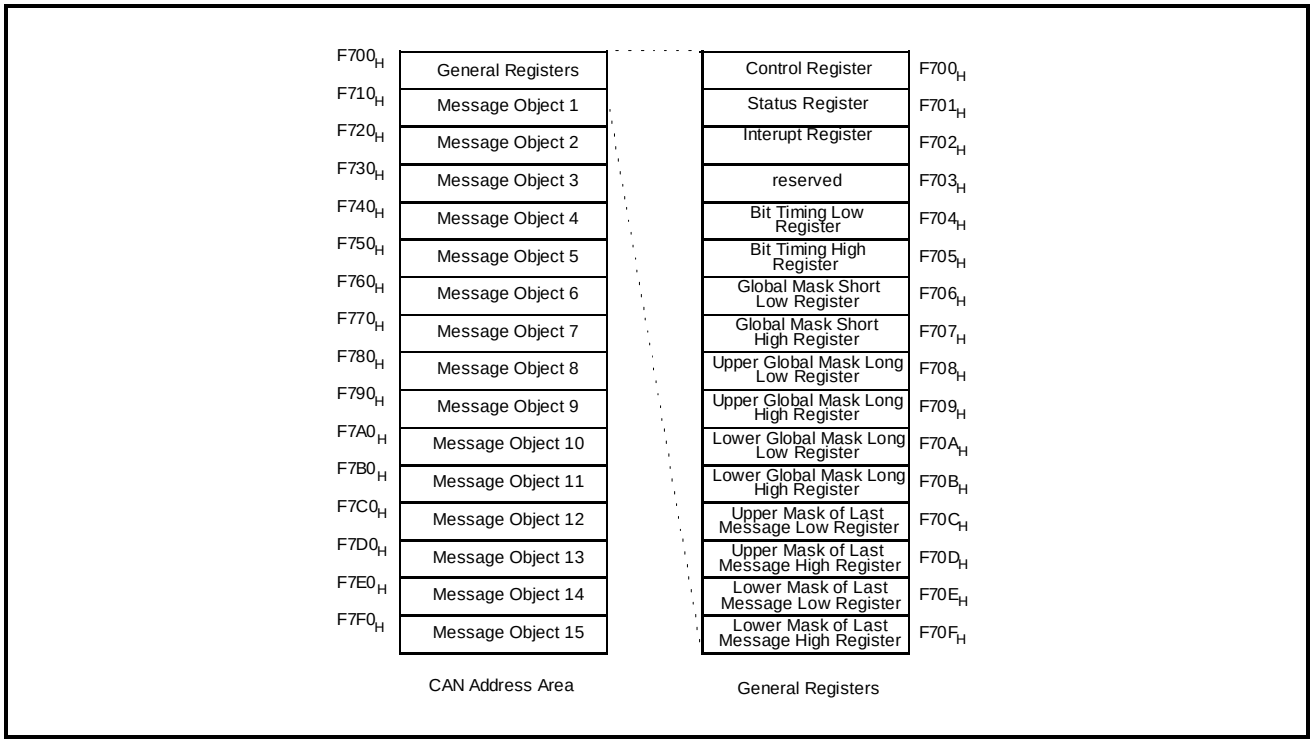

Figure 3.2-3a:
CAN addressing area on the C515C

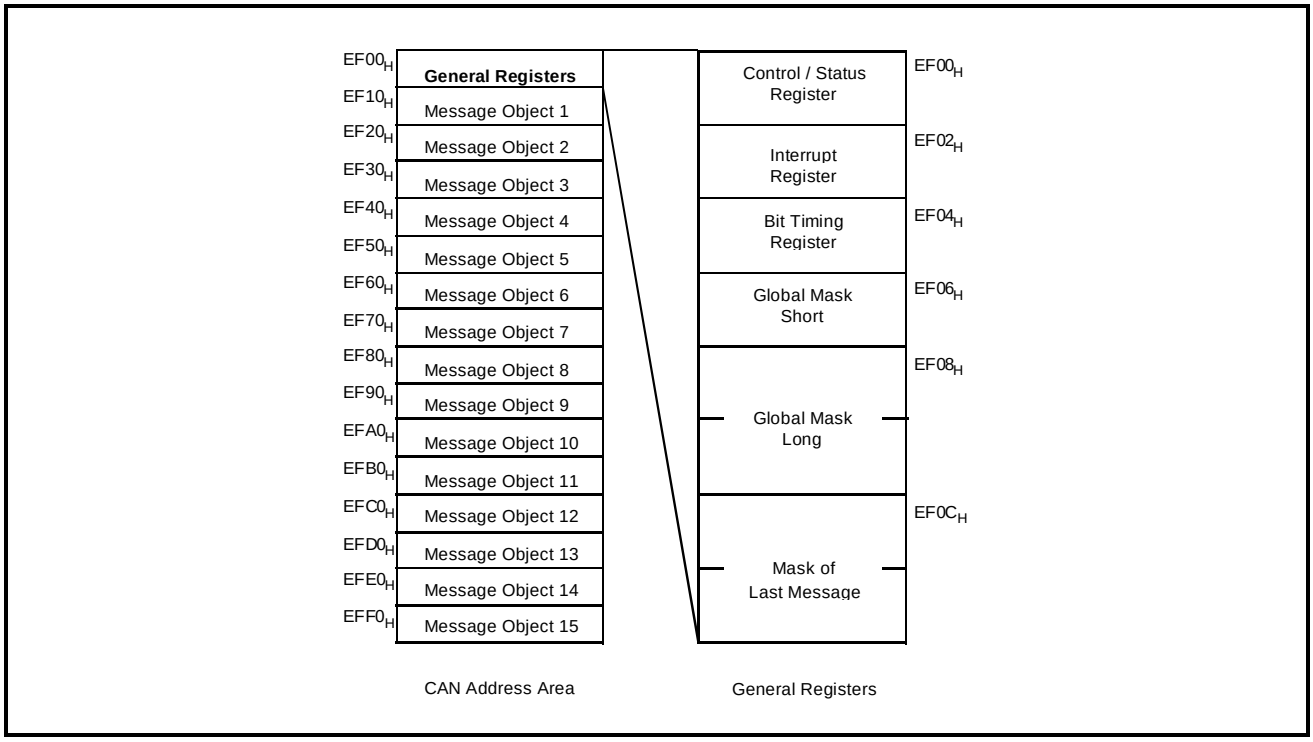

Figure 3.2-3b:
CAN addressing area on the C167CR

There are three mask registers in the CAN controller, called *"Global Mask Short"*, *"Global Mask Long"* and *"Mask of Last Message"*. Incoming frames are masked with their appropriate global mask. Therefore incoming Standard Frames are masked with the standard 11-bit mask in Global Mask Short, while incoming Extended Frames are masked with the extended 29-bit identifier in Global Mask Long. A bit in the mask registers holding a "0" means "don't care", i.e. the respective bit position of the message's identifier is not relevant during the acceptance filtering. In this way a message object accepts not only one specific message but all messages only differing in the previously masked bits. The last message object (MO 15) is used for the "Basic CAN" feature. It has an additional individually programmable acceptance mask for the complete Arbitration Field called Mask of Last Message which is ANDed with the global mask that corresponds to the incoming message. This allows classes of messages to be especially received in this message object by masking some (or all) bits of the identifier.

3.2.3 The Message Objects

The CAN controller provides storage for up to 15 message objects of maximum 8 bytes data length. Each of these objects has a unique identifier and its own set of control and status bits (see figure 3.2-4a and 3.2-4b). Each object can be configured with its direction as either transmit or receive, except the last message which is only a receive buffer with a special mask register.

An object with its direction set as transmit can be configured to be automatically sent whenever a Remote Frame with a matching identifier (taking into account the respective global mask register) is received over the CAN bus. By requesting the transmission of a message with the direction set as receive, a Remote Frame can be sent to request that the appropriate object be sent by some other node. Each object has separate transmit and receive interrupts and status bits, allowing the CPU full flexibility in detecting when a Remote/Data Frame has been sent or received.

For general purpose two masks for acceptance filtering can be programmed, one for identifiers of 11 bits and one for identifiers of 29 bits. However the CPU must configure bit XTD (Normal or Extended Frame Identifier) for each valid message to determine whether a standard or extended frame will be accepted.

The last message object has its own programmable mask for acceptance filtering, allowing a large number of infrequent objects to be handled by the system.

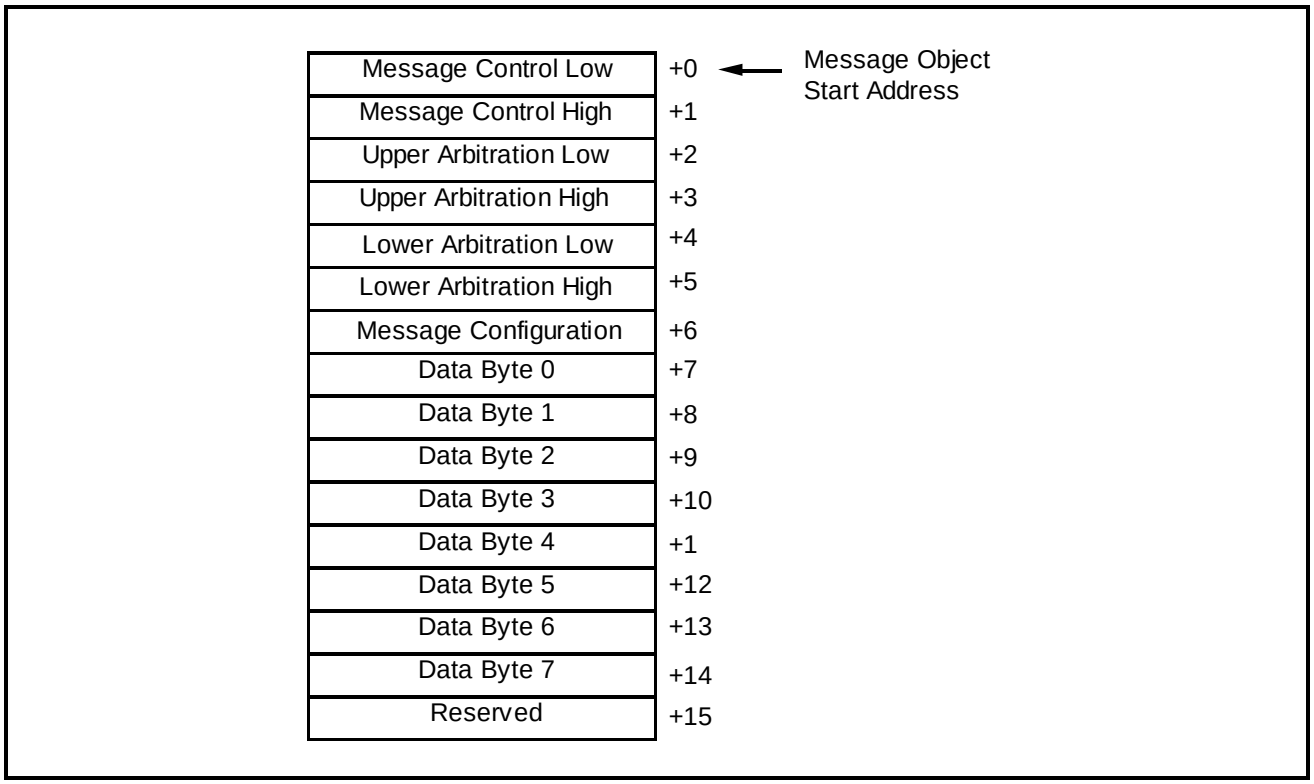

Figure 3.2-4a:
Message Objects in the C515C

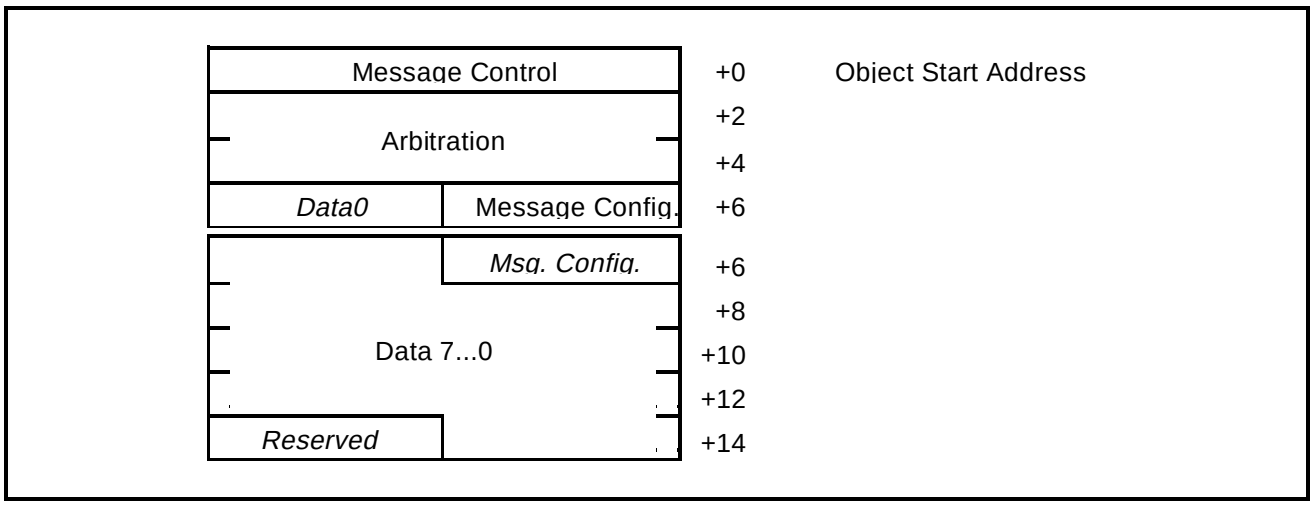

Figure 3.2-4b:
Message Objects in the C167CR

3.2.4 Initialization of the CAN Controller

To initialize the CAN controller, two bits in the control register (INIT and CCE) have to be set first. Then the register concerning the bit timing and the global mask registers must be configured. Afterwards each message object is to be initialized or is to be declared as not valid if it is not needed. The procedure is finished when bits INIT and CCE in the control register are reset. Now the CAN controller synchronizes to the data transfer on the CAN bus by waiting for 11 consecutive recessive bits. Finally it can participate in the CAN bus communication. Please also read section 5.4.1 for more information about the initialization of this device.

Further information about the CAN module can be found in section 5 or in the Siemens literature "Description of the On-Chip CAN-Module" and the manuals of the C167CR and the C515C.

3.3 The Stand-Alone Full-CAN Controller SAE 81C90/91

3.3.1 Important Features of the SAE 81C90/91

The CAN controllers SAE 81C90 (P-LCC-44-package, with two 8-bit I/O-Ports, and SAE 81C91 (P-LCC-28-package; without I/O-Ports) can be operated with up to 20 MHz. They support the CAN spezification V2.0A (B passive), i.e. the controllers can handle messages with the 11-bit standard identifier. Frames with the 29-bit extended identifier are not stored, but tolerated. Via a flexible, programmable interface the connection to different implementations of the physical layer (ISO/OSI layer 1) is possible. The connection to the host controller can be set up either in parallel via a multiplexed 8-bit address-/ data bus compatible to Siemens 8- and 16-bit microcontrollers or via a fast synchronous serial interface (up to 5 MBit/s). With the 8 bit wide registers all functions of the device are controlled from the host CPU. There are software drivers available for the SAE 81C90/91 from different CAN tool suppliers.

The SAE 81C90/91 can handle 16 CAN message objects. Each of these comprises eight data bytes and two descriptor bytes containing important information about the configuration of the relevant object. Message objects 0-7 are additionally equipped with two time-stamp bytes, from which the currency of the data in the data bytes can be inferred. For message object 0, a so called "monitor mode" is available with which this message object can be configured to receive all messages that are not covered by other memory locations. This device, too, thus offers basic CAN functionality.

3.3.2 Functional blocks of the SAE 81C90/91

The functional blocks of the SAE 81C90/91 are shown in figure 3.3-1.

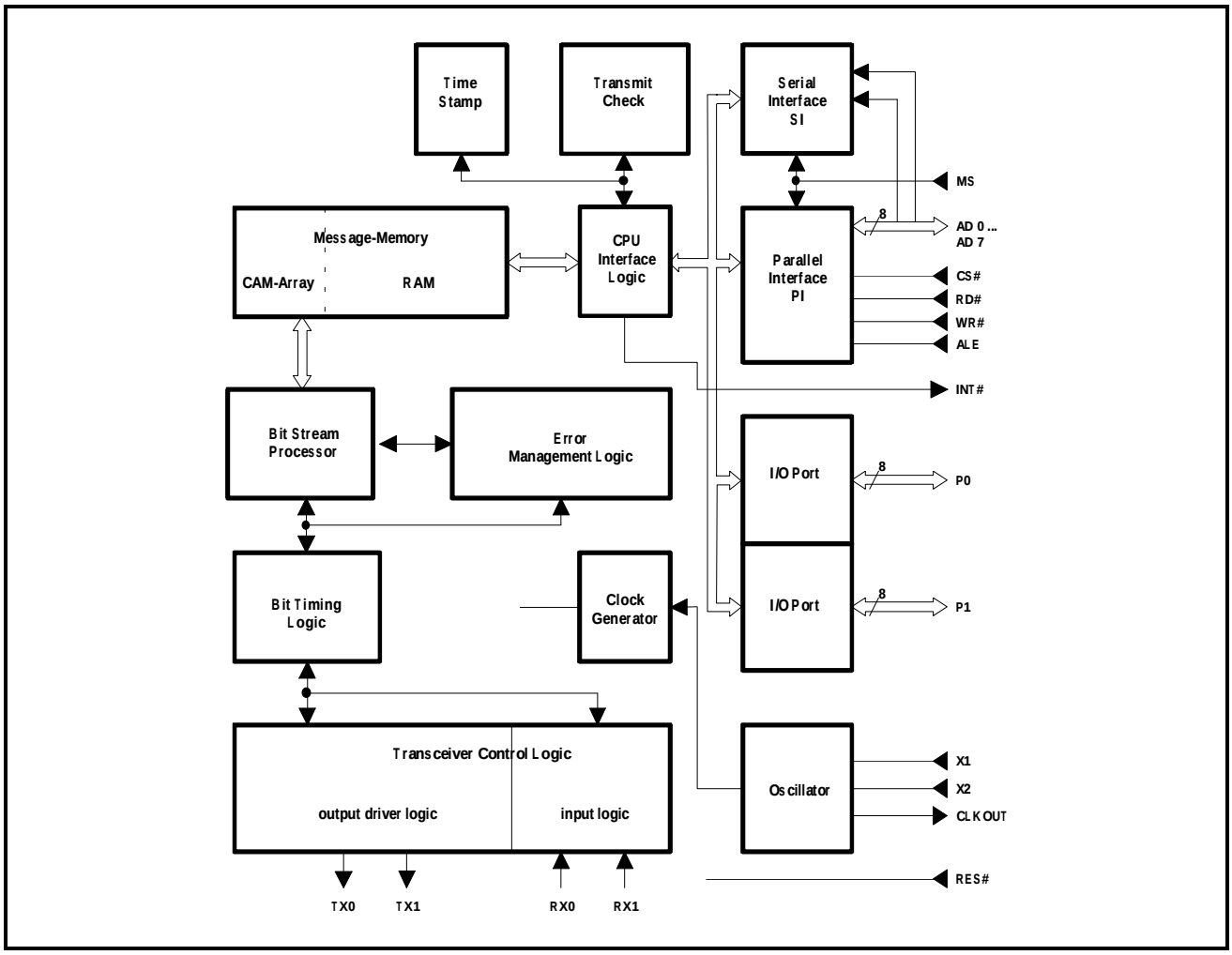

Figure 3.3-1:
Block Diagram of the SAE 81C90/91 (I/O Ports only available for SAE 81C90)

The *Message Memory* contains the 16 message objects. The SAE 81C90/91 as well owns a *Bit-Stream-Prozessor (BSP)*, which controls the complete CAN protocol, handles the different CAN frames (Data Frame, Remote Frame), and performs the frame check. The BSP reports errors to the *Error Management Logic* which returns information about the error rate back to the BSP and to the *CPU Interface Logic* (CIL). The CIL controls the accesses of the host CPU via the parallel or serial interface and interprets the included instructions.

The task of the *Bit Timing Logic* is again the synchronization to the bit stream and the bus timing. The *Transceiver Control Logic (TCL)* contains the output driver and the input comparator. The *Transmit Check Unit* is a special feature of the SAE 81C90/91. When transmitting a message, each bit is read back via the normal receive path and compared with the bit just sent. If a mismatch occurs, the Transmit Check Error Counter (TCEC) is

incremented by one and the actual message is invalidated by an Error Frame. In this way also the chip-internal conversion of the data stored in parallel to the serial bit stream (which is not covered by the CAN protocol itself) is monitored. If the TCEC reaches 4, the device enters the bus-off state.

3.3.3 The most important Control Registers of the SAE 81C90/91

Via the host CPU and the 8-bit control registers all functions of the SAE 81C90/91 are controlled. The most important registers are described in the following section.

The register MOD contains the two bits IM and RES, which are necessary for the initialization of the device, and some status bits. In the control register CTRL the Monitor Mode of message object 1 and the Transmit Check Unit can be enabled. The interrupt register INT shows occured interrupts which can be individually enabled / disabled in the interrupt mask register IMSK. In the bit length registers BL1 and BL2 the bit timing segments TSEG1, TSEG2 and SJW are configured. The desired baudrate is controlled via the baudrate prescaler register BRPR. Different physical layers can be connected to the SAE 81C90/91 by programming the output control register OC. The descriptor registers are filled with the 11 bits of the respective identifier, the RTR bit (which distinguishes between Data- and Remote Frames), and the data length code.

3.3.4 Initialization of the SAE 81C90/91 and Bus Access

Similar to the C167CR / C515C two special bits have to be set to be able to initialize the device. These bits are IM and RES in the register MOD. Afterwards the registers concerning the bit timing can be written and the message objects must be configured. Resetting bits RES and IM ends the initialization phase and the controller can participate in the CAN bus activities. Please also read section 5.5.1 for more information about the initialization of this device.

The SAE 81C90/91 contains no implementation of the physical layer, which again requires a CAN transceiver.

Further information about the SAE 81C90/91 Stand Alone Full CAN Controller can be found in section 5 or in the corresponding Data Sheet.

4. Examples for the Connection of Siemens Microcontrollers to CAN

4.1 Connecting the C167CR / C515C to CAN

Figure 4.1-1 shows the connection between the C167CR and the CAN bus. The CAN module is connected to the outside world via two pins, CAN_RxD (P4.5) and CAN_TxD, of port 4. CAN_RxD receives data from the physical layer of the CAN bus, while CAN_TxD transfers data to the physical layer. The physical layer is not implemented in the CAN module, so the C167CR is likewise connected via an external transceiver to the CAN bus lines CAN_H and CAN_L. In this example, the CAN bus is a shielded, twisted wire pair with termination resistors at both ends of the bus lines. If the transceiver has an input pin for slope control or "stand-by", it can be driven via a port pin of the C167CR (pin P4.7 in this case).

Notes:

- If the CAN module is used, port 4 may not be programmed to output all eight segment address lines. A maximum of four segment address lines is possible in this case. The address range that can be accessed via the address lines is then reduced to 1 MByte.
- P4.5 (CAN_RxD) has to be configured as input by the user. (If the CAN module is not used, P4.5 can be used as general purpose I/O).
- In some versions / production steps of the C167CR, CAN_TxD is configured as output by hardware and therefore cannot be used as general purpose I/O pin P4.6. Please contact your distributor or your local Siemens office if you need further information about this subject.

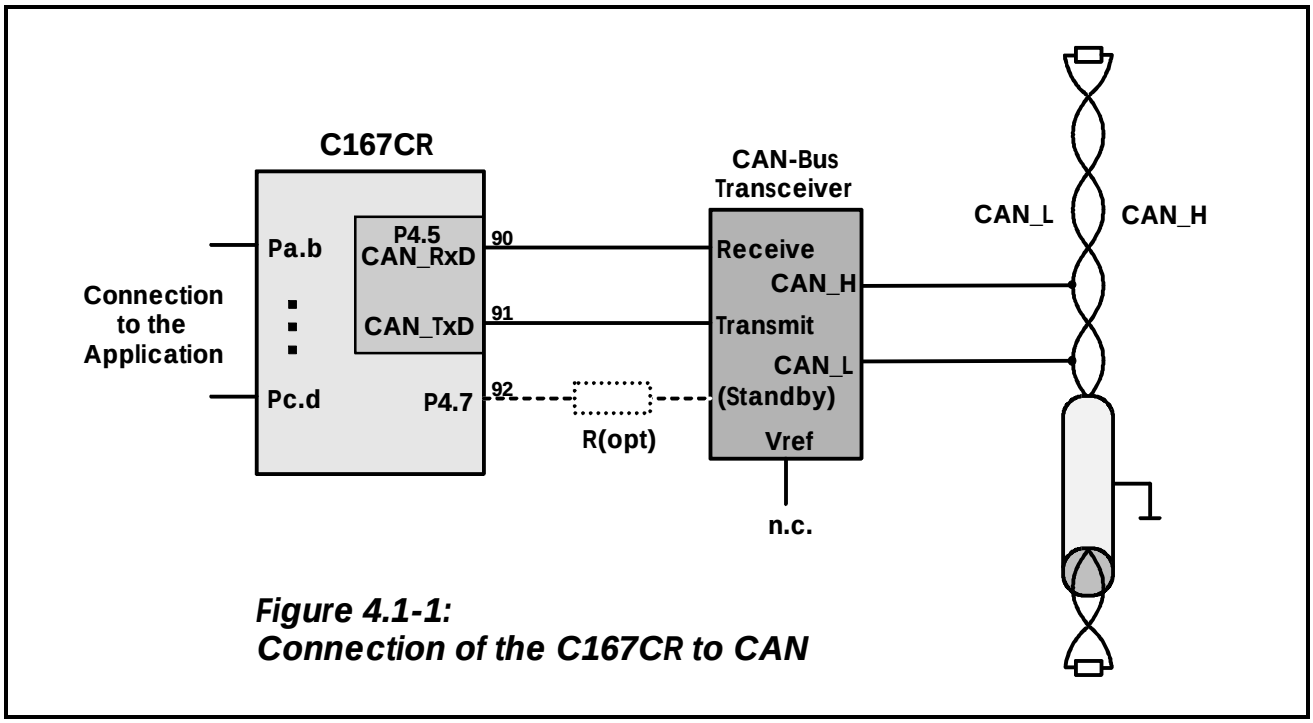

Figure 4.1-1:
Connection of the C167CR to CAN

Figure 4.1-2 shows the connection between the C515C and the CAN bus. This time the CAN module is connected to the CAN transceiver via the port pins "RXDC" and "TXDC" of port 4. For the control of a slope control or "stand-by" pin, portpin P4.5 was chosen in this example.

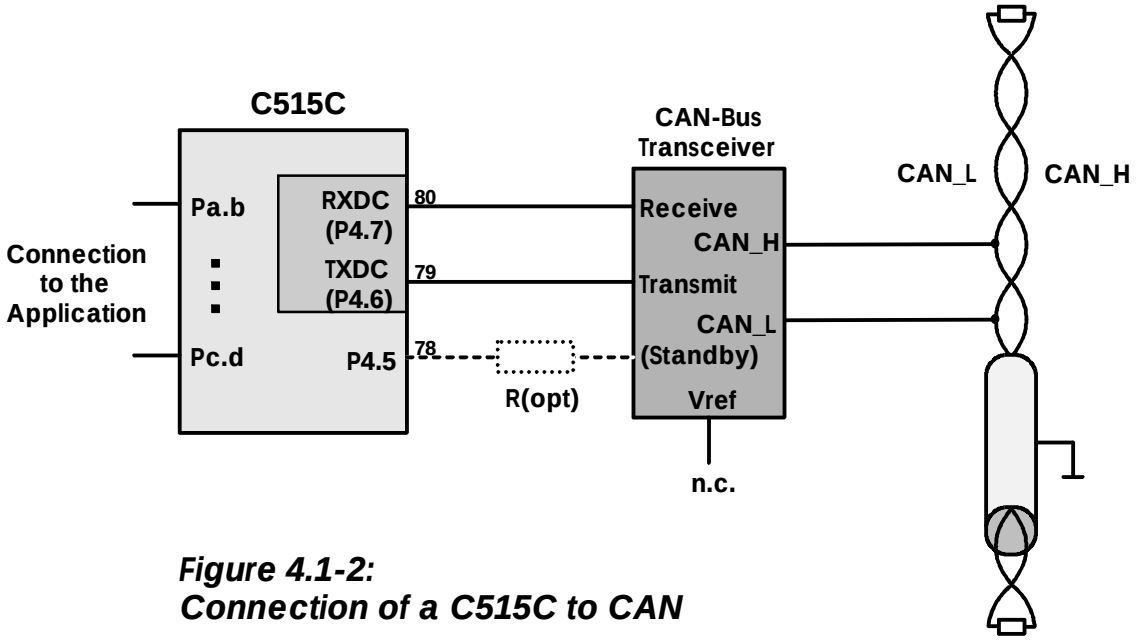

Figure 4.1-2:
Connection of a C515C to CAN

4.2 Connecting the SAB 80C166 to CAN using the parallel interface of the SAE 81C*90*

Of course not all members of the C500- and C166 family own an integrated CAN module. But also these controllers can be used in a Controller Area Network. In figure 4.2-1, the connection between a SAB 80C166 and a SAE 81C90 (P-LCC 44, with I/O ports) is shown. Both controllers are connected via the multiplexed 8-bit bus (address/data lines AD0 to AD7, signals WR#, RD#, ALE). This parallel connection is selected by applying a low logic level to pin MS (a high level would activate the serial interface). With the aid of the remaining address lines AD8 to AD17, the chip-select signal for the CAN controller (active low) can be generated via decoding logic, depending on the external memory configuration. The SAB 80C166 can use the multiplexed 8-bit bus for the corresponding memory range, and one of the other bus modes, for example the faster non-multiplexed 16-bit bus (with individual bus timing in each case), for the accesses to the remaining address space.

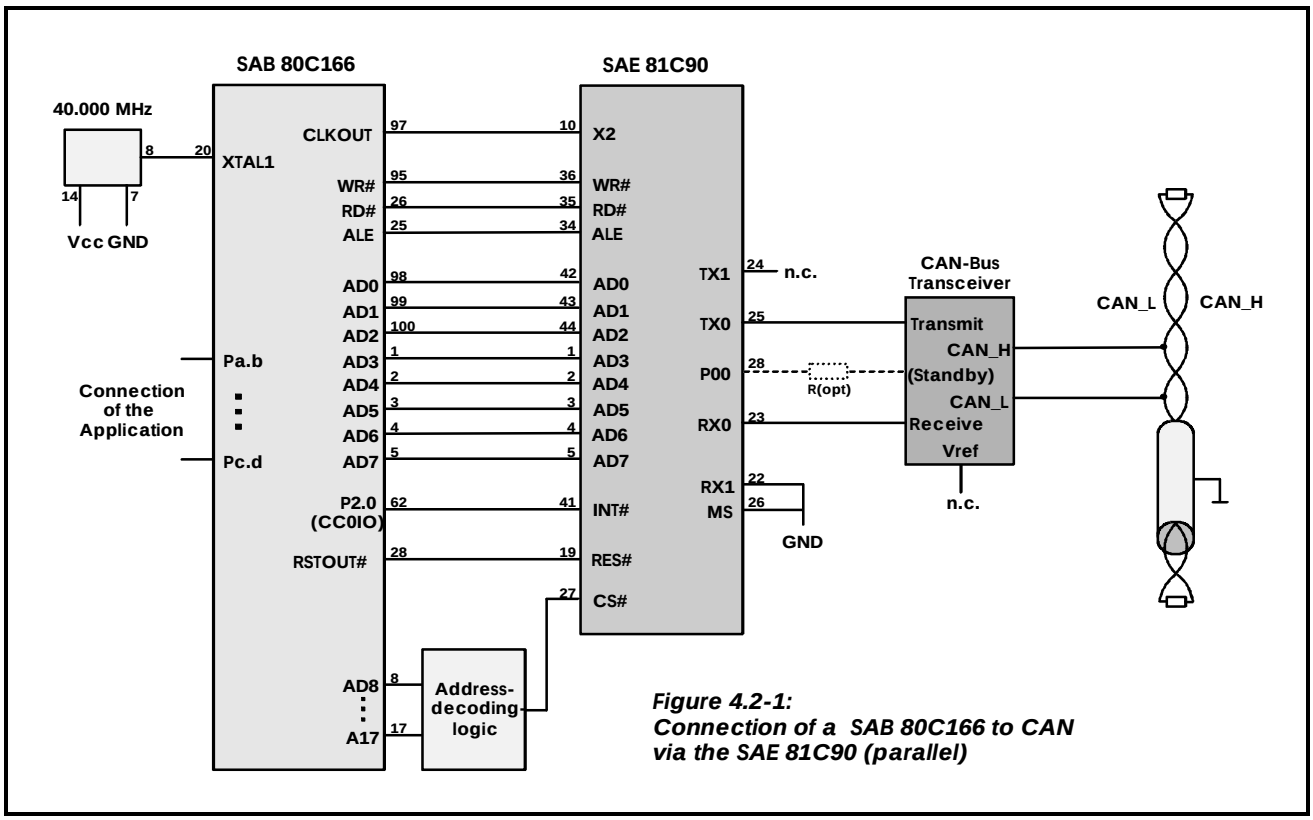

Figure 4.2-1:
Connection of a SAB 80C166 to CAN via the SAE 81C90 (parallel)

The SAE 81C90 obtains its defined reset via the RSTOUT# pin of the SAB 80C166. The interrupt output INT# of the CAN controller is routed to a capture input of the host controller (P2.0 in this case). Via this line, the CAN controller can report incoming messages. A 40 MHz quartz oscillator feeds the SAB 80C166, whose CLKOUT pin (which must be enabled in the SYSCON register) in turn supplies the CAN controller with 20 MHz clock information.

Like the C167CR / C515C, the SAE 81C90 contains no implementation of the physical layer and therefore a transceiver must be used. This driver module is connectd to the SAE 81C90 via pins TX0 (Transmitter Output 0, set as a push/pull output by software here) and RX0 (Comparator Input 0 / Digital input). Neither the second transmit pin TX1 nor the analog input RX1 is required here. RX1 is at 0V because the signal shall be evaluated digitally. Two of the transceiver pins are again directly connected to the CAN_H and CAN_L lines of the CAN bus. If the transceiver has a pin for slope control or standby, this can be driven by a pin of the two 8-bit I/O ports of the SAE 81C90 (e.g. pin P00). The application driven by the SAB 80C166, e.g . a robot controller, can now interchange data with other nodes in the same Controller Area Network.

Note: If a complete external CAN transceiver chip is used, the best way is to evaluate the signal digitally as shown in this application note. But the SAE 81C90/91 also has an input comparator implemented on-chip in case no such transceiver IC is used. However, if the bus lines work according to the ISO specification, additional circuitry is necessary for the interconnection of the input comparator to the bus lines.

4.3 Connecting the C511 / C513 to CAN using the serial interface of the SAE 81C*91*

The 8-bit microcontrollers C511 / C513 are new low-cost members of the C500 family. The SAE 81C90/91 is an obvious choice for using one of these devices in a Controller Area Network, just as it is for the SAB 80C166. In contrast to the the example shown in figure 4.2-1, however, the SAE 81C91 (P-LCC 28, without I/O ports) was selected in the circuit shown in figure 4.3-1. It is connected to the C511 / C513 and its Synchronous Serial Channel (SSC) via the Serial Synchronous Interface (SI), which is selected by applying a high level to the MS pin. In this example, the SSC should be configured as follows: master mode, clock idle level = low, first edge of clock (= rising edge) = data shift, second edge (= trailing edge) = data shift. The data width is 8 bits, MSB is sent / received first.

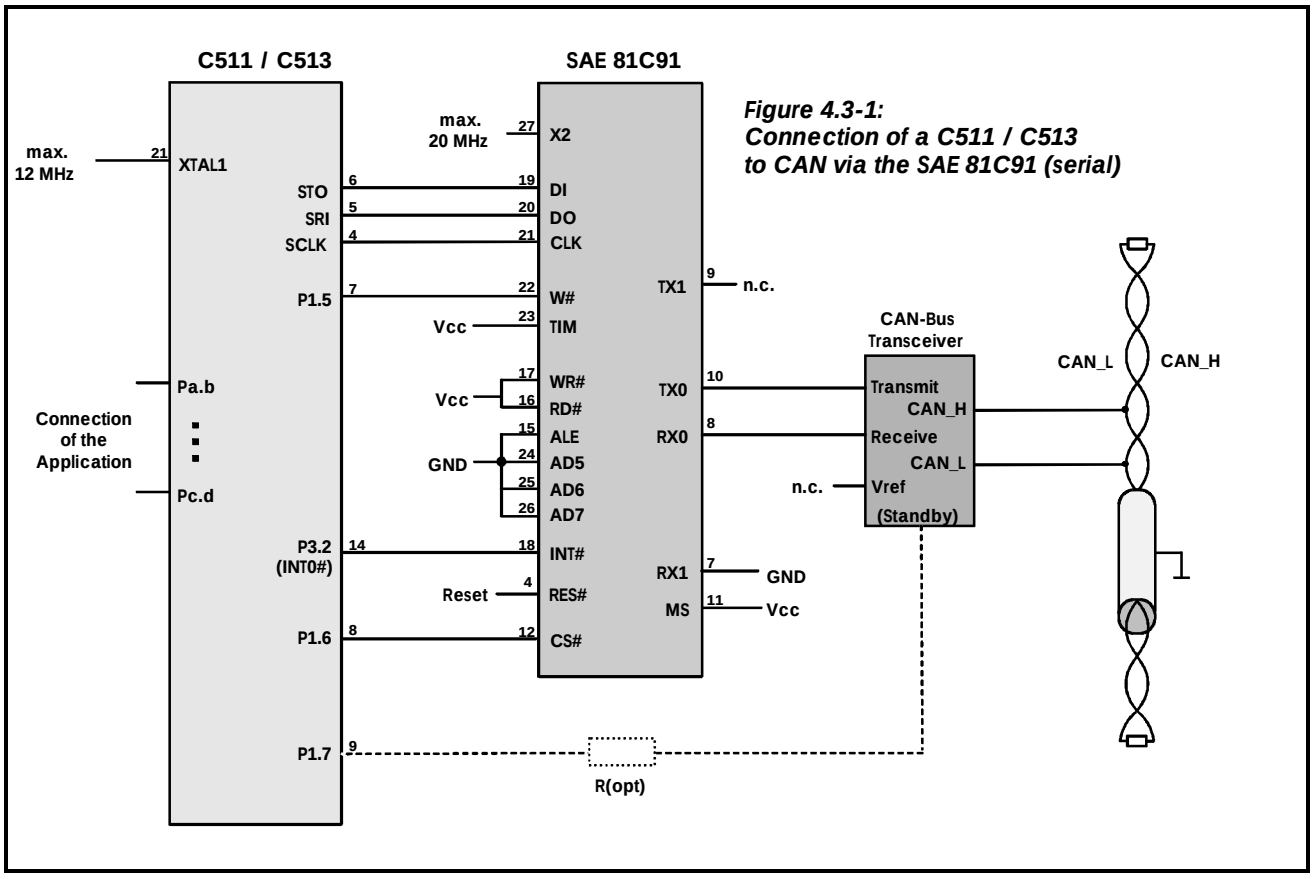

Figure 4.3-1:
Connection of a C511 / C513 to CAN via the SAE 81C91 (serial)

Every access to the SAE 81C91 is started by activating the device (CS# = 0), performed by a port pin (e.g. P1.6). Pin P1.5 of the C511 / C513 then selects a read (W# = 1) or a write (W# = 0) operation via the W# input of the CAN controller. Then the address of the register to be read or written is sent to the SAE 81C91 via the line connecting pin STO (SSC Transmit Output) with pin DI (Data Input). Depending on the operation, one or more (if the automatic decrementing of the address is activated in the CAN controller by setting

bit ADE in the MOD register) data bytes can be written to the SAE 81C91 or can be read via the line between pin SRI (SSC Receive Input) and pin DO (Data Output). Finally, the CAN controller has to be deactivated again (CS# = 1).

The synchronization of the controllers is done via the line SCLK - CLK. Data from pin DI are always transferred into the internal shift register with the rising edge of the clock. The level applied to the timing pin (TIM) of the CAN controller decides whether data is output at the DO pin with the rising edge (TIM = 0) or with the falling edge (TIM = 1, see figure 4.3-1) of the CLK signal. Pin P1.7 of the C511 / C513 controls the slope control pin of the CAN transceiver, when available. On the CAN controller side, an inactive level is applied to pins WR#, RD#, and ALE, and to pins AD5 to AD7, which are not required when the serial connection between the SAE 81C91 and the host controller is used.

4.4 A proposal for the CAN Bus Cables

The CAN standard does not include the connectors with which an application is connected to the bus lines. A proposal of the CAN user group "CAN in Automation" (CiA) uses 9-pole SUB-D connectors with the following pin usage:

Pin	Signal	Description
1	—	(reserved)
2	CAN_L	CAN_L bus line
3	GND	ground
4	—	(reserved)
5	—	(reserved)
6	(GND)	optional ground
7	CAN_H	CAN_H bus line
8	—	(reserved)
9	(V+)	opt. power supply

5. Ways of handling the SAE 81C90/91 and the CAN Module on the C167CR / C515C

5.1 Notes on the following Sections

The following sections contain important hints necessary for establishing a communication via the CAN bus using the devices described above. The program parts are mainly written in "C". They show just *one* way to solve a certain problem and they don't claim for optimized code. For better understanding the reader should also refer to the Siemens information "Description of the On-Chip CAN-Module", the SAE 81C90/91 Data Sheet, and the C515C User's Manual.

5.2 Accessing the Registers of the CAN Module and the SAE 81C90/91

To access the registers of the CAN module and the SAE 81C90/91, the registers have been given names that show the task of the respective register. Via pointer these names have been connected with the respective address in three include files. Therefore, in the software hints, not "address EF04h of the C167CR" is accessed but "directly" the register "BTR" (the Bit Timing Register). Those registers, which appear more than once (e.g. in each message object) are given the name adder "_M1", "_M2" etc. for message object 1, 2 etc. Another possibility would be to define the message objects as "structures" ("MOBJ1", "MOBJ2",...) to then access for example the Message Control Register of message object 5 with "MOBJ5.MCR".

Extract from the include file for the C167CR:

```
/*          Register                          Address */
#define     CSR         *(unsigned int *)     0xef00
#define     IR          *(unsigned char*)     0xef02
#define     BTR         *(unsigned int *)     0xef04
#define     GMS         *(unsigned int *)     0xef06
#define     UGML        *(unsigned int *)     0xef08
#define     LGML        *(unsigned int *)     0xef0a
#define     UMLM        *(unsigned int *)     0xef0c
#define     LMLM        *(unsigned int *)     0xef0e
#define     MCR_M1      *(unsigned int *)     0xef10
#define     UAR_M1      *(unsigned int *)     0xef12
#define     LAR_M1      *(unsigned int *)     0xef14
#define     MCFG_M1     *(unsigned char*)     0xef16
#define     DB0_M1      *(unsigned char*)     0xef17
#define     DB1_M1      *(unsigned char*)     0xef18
#define     DB2_M1      *(unsigned char*)     0xef19
#define     DB3_M1      *(unsigned char*)     0xef1a
#define     DB4_M1      *(unsigned char*)     0xef1b
#define     DB5_M1      *(unsigned char*)     0xef1c
#define     DB6_M1      *(unsigned char*)     0xef1d
#define     DB7_M1      *(unsigned char*)     0xef1e
#define     MCR_M2      *(unsigned char *)    0xef20
#define  ...
...
```

Extract from the include file for the C515C:

```
/*          Register                        Address */
#define     CR          *(unsigned char*)       0xf700
#define     SR          *(unsigned char*)       0xf701
#define     IR          *(unsigned char*)       0xf702
#define     BTR0        *(unsigned char*)       0xf704
#define     BTR1        *(unsigned char*)       0xf705
#define     GMS0        *(unsigned char*)       0xf706
#define     GMS1        *(unsigned char*)       0xf707
#define     UGML0       *(unsigned char*)       0xf708
#define     UGML1       *(unsigned char*)       0xf709
#define     LGML0       *(unsigned char*)       0xf70a
#define     LGML1       *(unsigned char*)       0xf70b
#define     UMLM0       *(unsigned char*)       0xf70c
#define     UMLM1       *(unsigned char*)       0xf70d
#define     LMLM0       *(unsigned char*)       0xf70e
#define     LMLM1       *(unsigned char*)       0xf70f

#define     MCR0_M1     *(unsigned char*)       0xf710
#define     MCR1_M1     *(unsigned char*)       0xf711
#define     UAR0_M1     *(unsigned char*)       0xf712
#define     UAR1_M1     *(unsigned char*)       0xf713
#define     LAR0_M1     *(unsigned char*)       0xf714
#define     LAR1_M1     *(unsigned char*)       0xf715
#define     MCFG_M1     *(unsigned char*)       0xf716
#define     DB0_M1      *(unsigned char*)       0xf717
#define     DB1_M1      *(unsigned char*)       0xf718
#define     DB2_M1      *(unsigned char*)       0xf719
#define     DB3_M1      *(unsigned char*)       0xf71a
#define     DB4_M1      *(unsigned char*)       0xf71b
#define     DB5_M1      *(unsigned char*)       0xf71c
#define     DB6_M1      *(unsigned char*)       0xf71d
#define     DB7_M1      *(unsigned char*)       0xf71e

#define     MCR0_M2     *(unsigned char*)       0xf720
#define ...
...
```

Extract from the include file for the SAE 81C90/91 (according to the specification 01/97):

```
/*          Register                          Addresse */
#define    BL1         *(unsigned char far*)   0x...00
#define    BL2         *(unsigned char far*)   0x...01
#define    OC          *(unsigned char far*)   0x...02
#define    BRPR        *(unsigned char far*)   0x...03
#define    RRR1        *(unsigned char far*)   0x...04
#define    RRR2        *(unsigned char far*)   0x...05
#define    RIMR1       *(unsigned char far*)   0x...06
#define    RIMR2       *(unsigned char far*)   0x...07
#define    TRSR1       *(unsigned char far*)   0x...08
#define    TRSR2       *(unsigned char far*)   0x...09
#define    IMSK        *(unsigned char far*)   0x...0a
#define    MOD         *(unsigned char far*)   0x...10
#define    INT         *(unsigned char far*)   0x...11
#define    CTRL        *(unsigned char far*)   0x...12

...

#define    DR0H        *(unsigned char far*)   0x...40
#define    DR0L        *(unsigned char far*)   0x...41

#define    DR1H        *(unsigned char far*)   0x...42
#define    DR1L        *(unsigned char far*)   0x...43

...

#define    BYTE0MSG0   *(unsigned char far*)   0x...80
#define    BYTE1MSG0   *(unsigned char far*)   0x...81
#define    BYTE2MSG0   *(unsigned char far*)   0x...82
#define    BYTE3MSG0   *(unsigned char far*)   0x...83
#define    BYTE4MSG0   *(unsigned char far*)   0x...84
#define    BYTE5MSG0   *(unsigned char far*)   0x...85
#define    BYTE6MSG0   *(unsigned char far*)   0x...86
#define    BYTE7MSG0   *(unsigned char far*)   0x...87

#define    BYTE0MSG1   *(unsigned char far*)   0x...88
...
```

The absolute addresses have to be chosen according to the address range the controller is supposed to be located in.

5.3 Configuration of the Bit Timing

Before initializing the CAN controllers, one has to think about the internal frequencies of the devices as well as the baudrate of the CAN bus. The registers that are responsible for the bit timing contain the parameters TSEG1, TSEG2, SJW and BRP which have to be programmed accordingly.

In the following section, a way to determine these parameters for a special controller frequency (e.g. 20 MHz) and a certain CAN baudrate (e.g. 125 kbit/s) shall be presented. The calculations are based on the following (rough) structure of one bit time (figure 5.3-1):

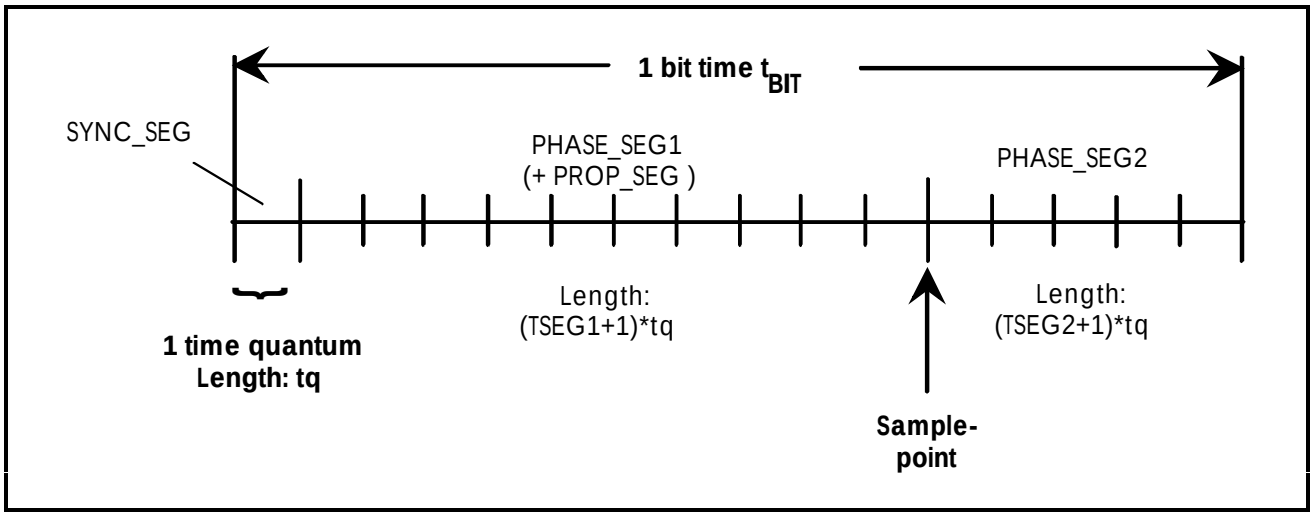

Figure 5.3-1:
Rough Structure of one Bit Cell in the CAN Module of the C167CR / C515C and in the SAE 81C90/91

- The following equations apply to figure 5.3-1:

$$t_{BIT} = t_{SYNC\_SEG} + t_{PHASE\_SEG1} + t_{PHASE\_SEG2} \quad (1)$$

$$t_{SYNC\_SEG} = 1 * t_q \quad (2)$$

$$t_{PHASE\_SEG1} = (TSEG1 + 1) * t_q \quad (3)$$

$$t_{PHASE\_SEG2} = (TSEG2 + 1) * t_q \quad (4)$$

$$t_q = (BRP + 1) * t_{CAN\_CLOCK} \quad (5)$$

$$t_{SJW} = (SJW + 1) * t_q \quad (6)$$

Please note that TSEG1, TSEG2, BRP and SJW are **the numerical values of the respective fields to be programmed in the respective registers of the controllers**, t_q represents one BTL cycle (see section 2.4).

The following equation applies to the C167CR: $t_{CAN\_CLOCK} = 2 * t_{XCLK} = \frac{2}{f_{CPU}}$.

The following equation applies to the C515C: $t_{CAN\_CLOCK} = 1\ CLP = \frac{1}{f_{CPU}}$.

The following equation applies to SAE 81C90/91: $t_{CAN\_CLOCK} = 2 * t_{OSC} = \frac{2}{f_{OSC}}$.

- Equations (2), (3) and (4) inserted into (1) results in:

$$t_{BIT} = 1*t_q + (TSEG1 + 1)*t_q + (TSEG2 + 1)*t_q$$

which is equal to

$$t_{BIT} = 3*t_q + (TSEG1 + TSEG2)*t_q.$$

- This equation solved to (TSEG1 + TSEG2) results in:

$$(TSEG1 + TSEG2) = \frac{t_{BIT} - 3 * t_q}{t_q}.$$

- Inserting (5) results in

$$(TSEG1 + TSEG2) = \frac{t_{BIT}}{(BRP + 1) * t_{CAN\_CLOCK}} - 3. \qquad (7)$$

- The following applies to (TSEG1 + TSEG2):

$$1 \leq (TSEG1 + TSEG2) \leq 22 \quad \text{and} \quad (TSEG1 + TSEG2) \in N,$$

because the possible values for TSEG1 are between 0 and 15 and for TSEG2 are between 0 and 7 and both parameters have to be integer. Now certain values of t_{BIT}, t_{CAN_CLOCK}, and BRP (possible values for BRP are 0 to 63) result in different possible values for (TSEG1 + TSEG2).

Example: C167CR CPU clock: 20 MHz ⇒ t_{CAN_CLOCK} = 100 ns
Desired CAN baudrate: 125 kBit/s ⇒ t_{BIT} = 1 / baudrate = 8 µs

Using equation (7) results in the following table with the valid values of (TSEG1 + TSEG2). Additionally, t_q is calculated using equation (5).

BRP (0 ≤ BRP ≤ 63)	...	3	4	...	7	...	9	...
TSEG1 + TSEG2 (1 ≤ x ≤ 22)	...	17	13	...	7	...	5	...
t_q [ns]	...	400	500	...	800	...	1000	...

When choosing one possibility for BRP, TSEG1, and TSEG2, some rules have to be obeyed so that the CAN specification is fulfilled. Signal delays by the bus lines, input comparators and output drivers have to be taken into account.

General rules for TSEG1 and TSEG2 depending on BRP:

if BRP = 0:	if BRP ≥ 1:
TSEG1 ≥ 2 TSEG2 ≥ 1	TSEG1 ≥ 1 TSEG2 ≥ 0

As a general rule, the sampling of the bit should take place at about 60-70% of the total bit time. Nevertheless, for each system the delays of bus drivers, transmitter / receiver circuits and the bus lines have to be taken into account when configuring the sample point. Examples can be found in the "Description of the on-chip CAN Module", page N-32 / N33.

If BRP is chosen to be 4, then TSEG1 + TSEG2 = 13 and t_q is 500 ns. Therefore the total bit time of 8 µs is devided into 16 t_q. One t_q is needed for the Synchronization Segment which leaves 15 t_q left for Phase Buffer Segment 1 and Phase Buffer Segment 2 (see figure 5.3-1 again).

60% of 16 t_q are about 10 t_q which results in the following configuration:

- 1 t_q for the Synchronization Segment
- 9 t_q for Phase Buffer Segment 1
- 6 t_q for Phase Buffer Segment 2.

This means that TSEG1 and TSEG2 should be programmed to the following values:

$$TSEG1 = 8; \quad TSEG2 = 5.$$

General rules for the SJW:

$$0 \leq SJW \leq 3;$$
$$SJW \leq TSEG2.$$

This means that SJW could be programmed to the maximum value of 3. But normally such a big SJW is only necessary when the clock generation of the different nodes is quite inaccurate, e.g. if ceramic resonators are used. So a SJW of 1 should be enough.

With other baudrates and clock frequencies the calculation can be done in the same way. On request three EXCEL tools (CP_167CR.XLS, CP_81C90.XLS, CP_C515C.XLS) are available which calculate proposals for the different parameters from the used frequency and baudrate.

5.4 Ways of Handling the CAN Module

5.4.1 The Initialization of the CAN Module on the C167CR / C515C

- The initialization starts with setting bits INIT and CCE in the Control Register.

C167CR:

```
CSR = 0x0041;     /* Contr./Stat.Register(Adr.EF00h)   */

/* 0 0 0 0  0 0 0 0 <STATUS CONTROL> 0 1 0 0  0 0 0 1 */

/* - - - R  T  LEC                   0 C - -  E S I I */
/*        X  X                       C      I I E N */
/*        O  O                       E      E E   I */
/*        K  K                                    T */
```

C515C:

```
CR = 0x41;   /* Control Register (Adr. F700h)  */

/* 0 1 0 0  0 0 0 1 */

/* T C - -  E S I I */
/* E C      I I E N */
/* S E      E E   I */
/* T             T */
```

- In the Bit Timing Register, the baudrate then can be configured (e.g. to 125 kbit/s) (see also section 5.3).

```
/* load Bit Timing Register */
```

C167CR:

```
BTR = 0x4944; /* = 125 kbit/s (@ fCPU = 20 Mhz) */

/* 0 1 0 0  1 0 0 1  0 1 0 0  0 1 0 0  */
/* - TSEG2   TSEG1   SJW|<--  BRP -->| */
```

C515C:

```
BTR0   = 0x41;
BTR1   = 0x6B;   /* = 125 kbit/s (@ fCPU = 5 MHz)*/

/* 0 1 1 0  1 0 1 1 <BTR1 BTR0> 0 1 0 0  0 0 0 1  */
/* - TSEG2   TSEG1              SJW|<--  BRP -->| */
```

- If Standard CAN is used then the "Global Mask Short" has to be initialized according to the identifier bits to be used for acceptance filtering (application specific). Shall all bits of the 11-bit identifier be evaluated, then all respective bits of the "Global Mask Short" are set to "1":

  C167CR:

```
GMS = 0xE0FF; /* Global Mask Short (Adr. EF06h) */

/* 1 1 1 0  0 0 0 0  1 1 1 1  1 1 1 1 */

/* 2 1 1            2 2 2 2  2 2 2 2   (Ident.-Bits */
/* 0 9 8 -  - - - -  8 7 6 5  4 3 2 1     18 - 28)   */
```

  C515C:

```
GMS0   = 0xFF;   /* Global Mask Short Low (Adr. F706h) */
GMS1   = 0xE0;   /* Global Mask Short High (Adr. F707h) */

/* 1 1 1 0  0 0 0 0 <GMS1 GMS0> 1 1 1 1  1 1 1 1 */

/* 2 1 1            (Id.-Bits  2 2 2 2  2 2 2 2 */
/* 0 9 8 -  - - - -    18-28)   8 7 6 5  4 3 2 1 */
```

- If Extended CAN is used, then the "Upper Global Mask Long" and the "Lower Global Mask Long" have to be initialized according to the identifier bits to be used for acceptance filtering (application specific). Shall all bits of the 29-bit identifier be evaluated, then all respective bits of the "Global Mask Short" are set to "1":

  C167CR:

```
UGML = 0xFFFF;  /* Upper Global Mask Long (Adr.EF08h) */

/* 1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1 */

/* 2 1 1 1  1 1 1 1  2 2 2 2  2 2 2 2   (Ident.-Bits */
/* 0 9 8 7  6 5 4 3  8 7 6 5  4 3 2 1     13 - 28)   */

LGML = 0xF8FF;  /* Lower Global Mask Long (Adr.EF0Ah) */

/* 1 1 1 1  1 0 0 0  1 1 1 1  1 1 1 1 */

/*                  1 1 1              (Ident.-Bits */
/* 4 3 2 1  0 - - -  2 1 0 9  8 7 6 5      0 - 12)   */
```

C515C:

```
UGML0 = 0xFF;  /*Upper Gl. Mask Long Low (Adr.F708h)*/
UGML1 = 0xFF;  /*Upper Gl. Mask Long High (Adr.F709h)*/

/* 1 1 1 1  1 1 1 1 <UGML1 UGML0> 1 1 1 1  1 1 1 1 */

/* 2 1 1 1  1 1 1 1   (Id.-Bits   2 2 2 2  2 2 2 2 */
/* 0 9 8 7  6 5 4 3     13 - 28)  8 7 6 5  4 3 2 1 */

LGML0 = 0xFF;  /*Lower Gl. Mask Long Low (Adr.F70Ah)*/
LGML1 = 0xF8;  /*Lower Gl. Mask Long High (Adr.7F0Bh)*/
/* 1 1 1 1  1 0 0 0 <LGML1 LGML0> 1 1 1 1  1 1 1 1 */

/*                   (Id.-Bits    1 1 1           */
/* 4 3 2 1  0 - - -    0 - 12)    2 1 0 9  8 7 6 5 */
```

- If the Basic CAN feature shall be used (message object 15), also the "Upper Mask of Last Message" and the "Lower Mask of Last Message" have to be initialized according to the identifier bits to be used for acceptance filtering of message object 15 (application specific). Shall all frames be received in MO 15 that cannot be stored in any other message object, all bits of the Mask of Last Message have to be set to 0 (= don't care):

C167CR:

```
UMLM = 0x0000; /* Upper Mask of Last Mess. (Adr.EF0Ch) */
/* 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0 */

/* 2 1 1 1  1 1 1 1  2 2 2 2  2 2 2 2   (Ident.-Bits */
/* 0 9 8 7  6 5 4 3  8 7 6 5  4 3 2 1     13 - 28)   */

LMLM = 0x0000; /* Lower Mask of Last Mess. (Adr.EF0Eh) */

/* 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0 */

/*                   1 1 1              (Ident.-Bits */
/* 4 3 2 1  0 - - -  2 1 0 9  8 7 6 5      0 - 12)   */
```

C515C:

```
UMLM0 = 0x00; /*Upper Mask of Last Mess. Low (F70Ch)*/
UMLM1 = 0x00; /*Upper Mask of Last Mess. High (F70Dh)*/

/* 0 0 0 0  0 0 0 0 <UMLM1 UMLM0> 0 0 0 0  0 0 0 0 */

/* 2 1 1 1  1 1 1 1 (Ident.-Bits  2 2 2 2  2 2 2 2 */
/* 0 9 8 7  6 5 4 3    13 - 28)   8 7 6 5  4 3 2 1 */

LMLM0 = 0x00; /*Lower Mask of Last Mess. Low (F70Eh)*/
LMLM0 = 0x00; /*Lower Mask of Last Mess. High (F70Fh)*/

/* 0 0 0 0  0 0 0 0 <LMLM1 LMLM0> 0 0 0 0  0 0 0 0 */
```

```
/*                  (Ident.-Bits  1 1 1           */
/* 4 3 2 1  0 - - -    0 - 12)    2 1 0 9  8 7 6 5 */
```

Please also see section 5.6 for the use of the Basic CAN feature.

- Then the message objects are configured. For this reason,
  - Message Configuration Register,
  - Arbitration Registers and
  - Message Control Register

  of the respective message have to be initialized.

  In the Message Configuration Register, the following configurations have to be made:
  - Length of the Data Field of the message (DLC = 0000b - 1000b),
  - Used protocol:
    XTD = 0: Standard CAN,
    XTD = 1: Extended CAN
  - Direction of the object (please also see table below):
    DIR = 1: transmit Data Frames; receive and answer Remote Frames
    (in the following called "transmit object"),
    DIR = 0: transmit Remote Frames; receive Data Frames
    (in the following called "receive object")

CAN Module DIR-BIT Behaviour	Transmission of this message object will generate...	If a Data Frame with a matching identifier is received...	If a Remote Frame with a matching identifier is received...
DIR Bit = 0 =Receive Object (*receives Data Frames, transmits Remote Frames)*	... a Remote Frame. The corresponding Data Frame is stored in this MO on reception.	... the Data Frame is stored.	... the Remote Frame is NOT answered.
DIR Bit = 1 =Transmit Object (*transmits Data Frames, receives Remote Frames)*	... a Data Frame.	... the Data Frame is NOT stored.	... the Remote Frame is answered with the corresponding Data Frame.

Example for a Standard CAN transmit object with 8 bytes data length:

C167CR / C515C:

```
MCFG_Mn = 0x88; /*Mess. Configur. Reg. n (EFn6h/F7n6h)*/

/* 1 0 0 0  1 0 0 0 */

/*|<-DLC->| D X - - */
/*          I T     */
/*          R D     */
```

Example for an Extended CAN receive object with 3 bytes data length:

C167CR / C515C:

```
MCFG_Mn = 0x34; /*Mess. Configur. Reg. n (EFn6h/F7n6h)*/

/* 0 0 1 1  0 1 0 0 */

/*|<-DLC->| D X - - */
/*          I T     */
/*          R D     */
```

- Then the identifier of the message is configured in the Arbitration Registers corresponding to the chosen CAN protocol. Please mind the identifier bit positions in these registers:

```
/* Configure identifier: */
/* (example for Standard CAN (11-bit identifier) */
```

C167CR:

```
UAR_Mn = 0xE068; /* Upper Arbitr. Reg. n (Adr. EFn2h) */

/* 1 1 1 0  0 0 0 0  0 1 1 0  1 0 0 0 */

/* 2 1 1 1  1 1 1 1  2 2 2 2  2 2 2 2  (Identifier-Bits */
/* 0 9 8 7  6 5 4 3  8 7 6 5  4 3 2 1       13 - 28 )   */

LAR_Mn = 0x0000; /* Lower Arbitr. Reg. n (Adr. EFn4h) */

/* 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0 */

/* 4 3 2 1  0 |res|  1 1 1 9  8 7 6 5  (Identifier-Bits */
/*                  2 1 0                 0 - 12)     */
```

C515C:

```
UAR0_Mn = 0x68; /* Upper Arbitr. Reg. n Low (F7n2h) */
UAR1_Mn = 0xE0; /* Upper Arbitr. Reg. n High (F7n3h) */

/* 1 1 1 0  0 0 0 0 <UAR1_Mn UAR0_Mn> 0 1 1 0  1 0 0 0 */

/* 2 1 1 1  1 1 1 1    (Id.-Bits      2 2 2 2  2 2 2 2 */
/* 0 9 8 7  6 5 4 3     13 - 28)      8 7 6 5  4 3 2 1 */

LAR0_Mn = 0x00; /* Lower Arbitr. Reg. n Low (F7n4h) */
LAR1_Mn = 0x00; /* Lower Arbitr. Reg. n High (F7n5h) */

/* 0 0 0 0  0 0 0 0 <LAR1_Mn LAR0_Mn> 0 0 0 0  0 0 0 0 */

/* 4 3 2 1  0 |res|    (Id.-Bits      1 1 1 9  8 7 6 5 */
/*                       0 - 12)      2 1 0           */

/* ⇒ results in the 11-bit identifier "0110 1000 111" */
```

- In the respective Message Control Register, the object is declared valid (MSGVAL = set) and the programmer can decide whether he wants to get an interrupt on successful reception (RXIE = set) and / or transmission (TXIE = set) of a frame. As the data bytes of the message object contain unpredictable data at this time, CPUUPD is set in the transmit objects in order to prevent the object from being transmitted at this time (e.g. by an incoming Remote Frame). CPUUPD has to be kept set until the data bytes have been filled with real data. Receive objects do not have a CPUUPD field. This field is called MSGLST in these objects and is set by the CAN controller if a new message comes in before the previously received message has been read out.

  Example for a transmit object which shall not generate interrupts:

  C167CR:

```
MCR_Mn = 0x5995; /* Configure Transmit Object n */
```

  C515C:

```
MCR0_Mn = 0x95;  /* Configure Transmit Object n */
MCR1_Mn = 0x59;


/* RMTPND = reset;      TXRQ   = reset */
/* CPUUPD = set;     NEWDAT = reset */
/* MSGVAL = set;     TXIE   = reset */
/* RXIE   = reset;      INTPND = reset */
```

Example for a receive object with interrupt generation on successful reception of a Data Frame (RXIE = set):

C167CR:

```
MCR_Mn = 0x5599; /* Configure Receive Object n */
```

C515C:

```
MCR0_Mn = 0x99       /* Configure Receive Object n */
MCR1_Mn = 0x55;


/* RMTPND = reset;      TXRQ   = reset */
/* MSGLST = reset;      NEWDAT = reset */
/* MSGVAL = set;    TXIE   = reset */
/* RXIE   = set;    INTPND = reset */
```

- If the programmer wishes the data bytes to be in a defined state, he can set all data bytes of the message object to zero now:

C167CR / C515C:

```
DB0_Mn = 0x00;       /* set data bytes to 0 */
DB1_Mn = 0x00;       /* (Addr. EFn7h - EFnEh (C167CR)  */
DB2_Mn = 0x00;       /* (Addr. F7n7h - F7nEh (C515C)   */
DB3_Mn = 0x00;
DB4_Mn = 0x00;
DB5_Mn = 0x00;
DB6_Mn = 0x00;
DB7_Mn = 0x00;
```

Finally, all message objects that will not be used have to be declared as not valid:

C167CR:

```
MCR_Mn = 0x5555; /* reset all functions incl. MSGVAL */
```

C515C:

```
MCR0_Mn = 0x55;  /* reset all functions incl. MSGVAL */
MCR1_Mn = 0x55;
```

- Clearing bits INIT and CCE ends the initialization. Setting the bits IE (Interrupt Enable), SIE (Status Change Interrupt Enable), and EIE (Error Interrupt Enable) at the same time enables all interrupts of the CAN module to the CPU.

C167CR:

```
CSR = 0x000E;     /* End initialization */

/* 0 0 0 0  0 0 0 0 <STATUS CONTROL> 0 0 0 0  1 1 1 0 */

/* - - - R  T  LEC                   0 C - -  E S I I */
/*        X  X                         C      I I E N */
/*        O  O                         E      E E   I */
/*        K  K                                      T */
```

C515C:

```
CR = 0x0E;       /* End initialization */

/* 0 0 0 0  1 1 1 0 */

/* T C - -  E S I I */
/* E C      I I E N */
/* S E      E E   I */
/* T                T */
```

Now the CAN module waits for 11 consecutive recessive bits on the CAN bus (bus idle) and can then participate in the bus communication.

5.4.2 The Transmission of a Data Frame with the CAN Module

- Make sure that you have configured a valid message object as transmit object (DIR=1). Before the next transmission, the TXOK bit in the Control Register which may be still set from the last successful transmission can be reset.

  C167CR:

```
CSR &= 0xFFF7;          /* Reset TXOK */
```

  C515C:

```
SR &= 0xF7;          /* Reset TXOK */
```

- Then "CPUUPD" (CPU Update) and "NEWDAT" (New Data) of the transmit object's corresponding Message Control Register should be set to show that the CPU now wants to work on the data bytes of this transmit object.

  C167CR:

```
MCR_Mn = 0xFAFF; /* Set CPUUPD and NEWDAT */
```

  C515C:

```
MCR1_Mn = 0xFA; /* Set CPUUPD and NEWDAT */


/* RMTPND = unchanged;  TXRQ   = unchanged */
/* CPUUPD = set;     NEWDAT = set       */
/* MSGVAL = unchanged;  TXIE   = unchanged */
/* RXIE   = unchanged;  INTPND = unchanged */
```

- Now the data byte(s) (1-8 referring to the DLC field in the Message Configuration Register) can be filled with the respective message (DLC = 4 here):

  C167CR / C515C:

```
DB0_Mn = 0x00;       /* load data byte 0 with 00H */
DB1_Mn = 0x11;       /* load data byte 1 with 11H */
DB2_Mn = 0x22;       /* load data byte 2 with 22H */
DB3_Mn = 0x33;       /* load data byte 3 with 33H */
```

- By resetting "CPUUPD" and setting the Transmission Request field "TXRQ" in the Message Control Register, the CAN controller resets NEWDAT and transmits the Data Frame:

  C167CR:

```
MCR_Mn = 0xE7FF; /* send Data Frame */
```

  C515C:

```
MCR1_Mn = 0xE7;  /* send Data Frame */


/* RMTPND = unchanged;  TXRQ   = set       */
/* CPUUPD = reset;      NEWDAT = unchanged */
/* MSGVAL = unchanged;  TXIE   = unchanged */
/* RXIE   = unchanged;  INTPND = unchanged */
```

  If the data bytes of the message object are not to be changed, setting TXRQ is enough to transmit the message. If the transmission was successful and was acknowledged by at least one other node, and if NEWDAT is still 0 after the transmission, the CAN controller resets TXRQ and RMTPND, sets bit TXOK in the Status Register and generates a status change interrupt (if enabled in the Control Register (IE=1; SIE=1)). If TXIE is set in the respective Message Control Register, a transmit interrupt is generated (if enabled in the Control Register (IE=1)).

5.4.3 The Transmission of a Remote Frame with the CAN Module

- Make sure that you have configured a valid message object as receive object (DIR=0). Before the transmission of the Remote Frame, check NEWDAT first. If NEWDAT = 1, then the last received message for this object has not yet been read out, which should be done now together with a reset of NEWDAT. After another check of NEWDAT (which should be 0 by then), a Remote Frame can be generated by setting TXRQ in the Message Control Register of this message object.

  C167CR:

```
MCR_Mn = 0xEFFF; /* send Remote Frame */
```

  C515C:

```
MCR1_Mn = 0xEF;  /* send Remote Frame */

/* RMTPND = unchanged;  TXRQ   = set       */
/* MSGLST = unchanged;  NEWDAT = unchanged */
/* MSGVAL = unchanged;  TXIE   = unchanged */
/* RXIE   = unchanged;  INTPND = unchanged */
```

- Then the node which is supposed to provide the requested data will receive the Remote Frame and answer it with the corresponding Data Frame. If this Data Frame is received, the CAN controller sets NEWDAT in the Message Control Register. So the CPU could poll NEWDAT to wait for this event:

  C167CR:

```
while ((MCR_Mn & 0x0200) == 0);
```

  C515C:

```
while ((MCR1_Mn & 0x02) == 0);

/* wait until Data Frame arrives (until NEWDAT = 1) */
```

  Another possibility is to react on the receive interrupt or the status change interrupt of the CAN controller which are generated on the reception of the Data Frame (if enabled in the Control Register (IE=1; SIE=1) or in the Message Control Register (RXIE=set) resp.).

- Now the requested data can be read from the data bytes of the message object.

- NEWDAT should then be reset:

C167CR:

```
MCR_Mn = 0xFDFF; /* reset NEWDAT */
```

C515C:

```
MCR1_Mn = 0xFD;  /* reset NEWDAT */

/* RMTPND = unchanged;  TXRQ   = unchanged */
/* MSGLST = unchanged;  NEWDAT = reset     */
/* MSGVAL = unchanged;  TXIE   = unchanged */
/* RXIE   = unchanged;  INTPND = unchanged */
```

5.4.4 Evaluation of a received Message with the CAN Module

- If the C167CR or the C515C receive a message with a matching identifier this can either be a data or a Remote Frame.

  If a *Remote Frame* is received in one of the message objects 1-14 configured as transmit object (DIR=1), only the data length code is stored in the corresponding object. The identifier and the data bytes remain unchanged. The CAN controller sets TXRQ and RMTPND in the corresponding Message Control Register. The set TXRQ bit field will cause the Remote Frame to be immediately and automatically answered by the CAN controller with the corresponding Data Frame (if the CPUUPD field is not set!). If RXIE in the Message Control Register and the global Interrupt Enable bit (bit IE in the Control Register) is set, the CAN controller also sets INTPND (INterrupt PeNDing) in the Message Control Register which generates an receive interrupt to the CPU

  If a *Data Frame* is received in a receive object (DIR=0) and NEWDAT=0 in the corresponding Message Control Register, the CAN controller sets NEWDAT and the data in the Data Field is written into the data bytes of the corresponding receive object. Additionally, the Arbitration Registers and the data length code of the receive object are updated with the identifier and the data length code of the received Data Frame. If RXIE in the corresponding Message Control Register and the global Interrupt Enable bit (bit IE in the Control Register) is set, the CAN controller sets INTPND (INterrupt PeNDing) in the Message Control Register which generates an receive interrupt to the CPU.

  In the C167CR, CAN interrupts are controlled via the corresponding Interrupt Control Register XP0IC. In the C515C, CAN interrupts can be activated / deactivated via the bit ECAN of the Special Function Register IEN2.

- In the interrupt service routine, the reason for the interrupt (status change interrupt or interrupt of one of the message objects) can be read from the contents of the Interrupt Register (the so-called INTID (=INTerrupt IDentifier)). If bit SIE in the Control Register has been set and the CAN controller updated (not necessarily changed) the Status Register (which it always does on transmission or reception of a frame), INTID will have the value "01". If a message has been received, RXOK is set, if a message has been successfully transmitted, TXOK is set. Read the Status Register to service the status change interrupt and to update INTID. If a message was received by message object "n", INTID will then have the value "n+2".

  After now the message object that has caused the interrupt has been identified, the data bytes of the message objects can be read and evaluated. Then INTPND and NEWDAT in the corresponding Message Control Register can be reset:

C167CR:

```
MCR_Mn = 0xFDFD; /* reset INTPND,NEWDAT of obj. n */
```

C515C:

```
MCR0_Mn = 0xFD;  /* reset INTPND,NEWDAT of obj. n */
MCR1_Mn = 0xFD

/* RMTPND = unchanged;  TXRQ   = set       */
/* MSGLST = unchanged;  NEWDAT = reset     */
/* MSGVAL = unchanged;  TXIE   = unchanged */
/* RXIE   = unchanged;  INTPND = reset     */
```

General hints concerning the interrupt handling:

1. When resetting INTPND, also the corresponding value in the INTID in the interrupt register is cleared. If further interrupts are pending, the interrupt with the next highest priority will appear in INTID. If no more interrupt is pending, INTID will get 00h. A CAN interrupt service routine must not be left until INTID has the value 00h. Otherwise the interrupt line to the stays active and no further interrupt generation from the CAN module to the CPU occurs.
2. Please note that reading the status partition (= high byte) of the Control/Status Register of the C167CR will clear the Status Change INTID in the Interrupt Register. Also note that this also happens when reading the whole 16-bit Control/Status Register because this word access also reads the status partition of this register. If you want to read the control partition without the clearing of the Status Change INTID, use byte access to the low byte of the Control/Status Register. - Reading the Status Register in the C515C has the same effect of clearing the Status Change INTID.
3. The Interrupt with the lowest INTID has the highest priority. If an interrupt with a higher priority occurs, before a pending interrupt with lower priority is serviced, the INTID is updated accordingly. So the servicing of the lower priority interrupt has to be postponed.

5.5 Ways of Handling the SAE 81C90/91

5.5.1 The Initialization of the SAE 81C90/91

- The initialization starts with setting both the bits IM and RES in the Mode/Status Register MOD. The Control Register CTRL activates some special functions, but clearing it to 00h is fine for the first approach. Then also the Interrupt Register is cleared.

```
MOD = 0x03;   /* load MOD (Addr. 10h) */

 /*   0   0   0   0     0   0   1   1    */
 /* ADE  RS  TC  TWL   RWL BS  RES  IM   */

CTRL = 0x00;  /* clear CTRL (Addr. 12h) */

/*  0   0   0   0     0   0   0   0    */
/* RX TST TSP1 TSP0 TSOV SME TCE  MM   */

INT = 0x00;   /* clear INT (Addr. 11h) */

/*  0   0   0   0     0   0   0   0    */
/* TCI EPI BOI WUPI RFI  WLI  TI  RI   */
```

- Now the baudrate is configured via the registers BRPR, BL1, and BL2. The values for the parameters TSEG1, TSEG2, SJW in the registers BL1 and BL2 as well as the value of the baudrate prescaler in the register BRPR are equivalent to those in the Bit Timing Register of the C167CR (not C515C because of the missing prescaler!) and can also be calculated the same way (see section 5.3). If the input signal is evaluated digitally (only RX0 used), bit DI in BL2 has to be set. If the input signal is applied to the input comparator (RX0 and RX1 used), DI has to be cleared. According to the CAN specification, the Speed Mode bit SM should be cleared (only the recessive to dominant edge is used for synchronization). Example for 125 kBit/s (81C90/91 external clock = 20 MHz):

```
BRPR = 0x04;  /* load BRPR (Addr. 03h) */
BL1 = 0x49;   /* load BL1 (Addr. 00h) */
BL2 = 0x41;   /* load BL2, Digital Input (Addr. 01h) */

/* 0 1 0 0  1 0 0 1 <BL1   BL2> 0 1 0 0  0 0  0 1 */
/* S TSEG2   TSEG1              I D - -  - S  SJW */
/* A                            P I        M      */
/* M                            O                 */
/*                              L                 */
```

- In the Interrupt Mask Register IMSK, bit ERI is set to globally enable interrupts to the host controller generated by incoming messages. Other interrupts (on completed transmissions, on reception of a Remote Frame, on entering the bus-off state etc.) can be enabled as well if desired. Afterwards, in the Receive Interrupt Registers RIMR1 and

RIMR2 the message objects that shall generate an interrupt on the reception of a message have to be specified individually (here: message objects 1 to 5).

```
IMSK = 0x01;  /* (Addr. 0Ah) interrupt on reception */

/*  0    0    0    0     0    0    0    1   */
/*ETCI EEPI EBOI EWUPI  ERFI EWLI ETI  ERI  */

RIMR1 = 0x3E; /* (Addr. 06h) recv. int. for MO 1 to 5 */
RIMR2  = 0x00;   /* (Addr. 07h) */

/* 0 0 1 1  1 1 1 0 <RIMR1 RIMR2> 0 0 0 0  0 0 0 0     */
/* Messages                                            */
/* 7 6 5 4  3 2 1 0              1 1 1 1  1 1 9 8      */
/*                               5 4 3 2  1 0          */
```

- The Output Control Register, which specifies the connection to the physical layer, has to be configured according to the application. In this example, it is configured to output a low level at pin TX0 if a dominant bit is to be sent. It outputs a high level, if the bit to be sent is recessive.

```
OC = 0x18;/* (Addr. 02h) DOM=LOW, RECESSIVE=HIGH */

/*   0 0 0  1 1 0  0 0   */
/*    Tx1    Tx0   OCM   */
```

- The Clock Control Register is used to determine the output frequency of the SAE 81C90/91's CLK pin. The output frequency may be the oscillator frquency divided by 1, 2, 4, 6, 8, 10, 12 or 14 or may be switched off completely. Writing to this register requires a special protocol in order to avoid erroneous writing. In the following example, the output frequency shall be programmed to $f_{osc}/2$:

```
CCR = 0x80;   /* Prepare writing to CCR (Addr. 14h) */
CCR = 0x01;   /* Switch output frequency to fosc/2 */
```

Please note that this must be two consecutive accesses to the SAE 81C90/91.

- In the descriptor bytes, the identifier of the message objects and the length of the Data Field is configured. The values have to be chosen according to the message objects in other CAN nodes the SAE 81C90/91 shall communicate with. With the RTR bit, the SAE 81C90/91 distinguishes between message objects handling *only Data Frames* (RTR=0) or handling *only Remote Frames* (RTR=1). Please mind the following table:

SAE 81C90/91 RTR-BIT Behaviour	Transmission of this message object will generate...	If a Data Frame with a matching identifier is received...	If a Remote Frame with a matching identifier is received...
RTR Bit = 0 = Object that handles *Data Frames*	... a Data Frame.	... the Data Frame is stored.	... the Remote Frame is NOT answered.
RTR Bit = 1 = Object that handles *Remote Frames*	...a Remote Frame. The corresponding Data Frame is NOT stored in this MO! The Data Frame has to be received by another MO	... the Data Frame is NOT stored.	... the Remote Frame is answered with the corresponding Data Frame.

- Please note that after a Remote Frame has been sent out (RTR=1), the corresponding Data Frame is not stored in the same message object. The SAE 81C90/91 could be used as follows:
- One message object is used for the transmission of Data Frames (RTR=0; e.g. MO 1), whose identifier and DLC is always configured new before each transmission.
- One message object is used for the transmission of Remote Frames (RTR=1, DLC=0; e.g. MO 2), whose identifier is always configured new before each transmission.
- Some other message objects are used for the reception of Data Frames (certain identifier, certain DLC; RTR=0). In these message objects, also the Data Frames that correspond to Remote Frames sent by MO 1 are received. This corresponds to a receive object in the CAN module of the C167CR / C515C (without the possibility of sending its own Remote Frames, though).
- The remaining message objects are used for the automatic answering of incoming Remote Frames (certain identifier, certain DLC; RTR=1). This corresponds to a transmit object in the CAN module of the C167CR / C515C (without the possibility of sending its own Data Frames, though).

```
/* load descriptor bytes of MO n */
/* Example: Identifier 1001 0110 000, RTR = 0, DLC = 8 */

DRnH   = 0x96;   /* load descriptor high byte object n */
DRnL   = 0x08;   /* load descriptor low byte object n */

/*Byte H> 1 0 0 1  0 1 1 0 | 0 0 0  0  1 0 0 0 <Byte L */
/*       |<-     IDENTIFIER     ->|RTR|<-DLC->|        */
```

- The descriptor bytes that are not going to be used should be configured with an unused identifier (as there's no way to completely disable a message object). Not initialized descriptor bytes have unknown values after reset and can lead to unexpected behavior. The data bytes can be cleared to 00h. Please note that writing to the data bytes must always start with one of the data bytes 7 to 1 and *must end with writing of data byte 0*, because on writing of data byte 0, the contents of the the shadow register is transferred in parallel into the RAM of the respective message. Please also take care that a read operation to the data bytes of a message object must not be interrupted by a write operation and vice versa. This ensures that the SAE 81C90/91 always works on consistent data.

```
/* invalidate unused descriptor bytes: */

DRmH = DRmL = 0x..;     /* unused identifier */
...

/* clear corresponding data bytes: */

BYTE7MSGm = 0x00;
...
BYTE0MSGm = 0x00;       /* end with writing to data byte 0! */
```

- After resetting the bits RES and IM, the initialization of the device is completed.

```
MOD = 0x00;   /* end initialization */

 /*   0   0   0   0     0   0   0   0    */
 /* ADE  RS  TC  TWL   RWL BS  RES  IM   */
```

5.5.2 The Transmission of a Data Frame / Remote Frame with the SAE 81C90/91

- To transmit a ***Data Frame***, make sure that you have configured a message object "n" (e.g. MO 1) that handles Data Frames (RTR = 0) and that carries the right identifier and the right DLC. Then the data bytes (1 to 8 depending on the configuration in descriptor byte 0) can be loaded with the desired message (DLC = 4 here):

```
BYTE3MSGn = 0x33;       /* load data byte 3 obj.n with 33h */
BYTE2MSGn = 0x22;       /* load data byte 2 obj.n with 22h */
BYTE1MSGn = 0x11;       /* load data byte 1 obj.n with 11h */
BYTE0MSGn = 0x00;       /* load data byte 0 obj.n with 00h */
```

  Please take care that writing must end with data byte 0 as mentioned in the previous section about the initialization of the device.

- The message is transmitted by setting the bit that corresponds to message object n in the registers TRSR1 (messages 0 - 7) or TRSR2 (messages 8 - 15), resp.:

```
TRSRy = 0xdd; /* send mess.; y = 1..2; dd = 01h..FFh */

/*   ?     ?     ?     ?     ?     ?    ?    ?  < Reg. TRSR1*/
/* TRS7  TRS6  TRS5  TRS4  TRS3  TRS2  TRS1 TRS0            */

/*   ?     ?     ?     ?     ?     ?    ?    ?  < Reg. TRSR2*/
/* TRS15 TRS14 TRS13 TRS12 TRS11 TRS10 TRS9 TRS8           */
```

  Several bits may be set simultaneously which results in the transmission of all selected messages. The message with the highest message object number (e.g. message object 15, if selected) is sent first. Message object 0 is transmitted at last.
- To transmit a ***Remote Frame***, make sure that you have configured a message object "n" (e.g. MO 2) that handles Remote Frames (RTR = 1) and that carries the right identifier and "0" in the DLC. Like the Data Frame, the Remote Frame is transmitted by setting the bit that corresponds to message object n in the registers TRSR1 (messages 0 - 7) or TRSR2 (messages 8 - 15), respectively. Several bits may be set simultaneously which results in the transmission of all selected messages starting with the one with the highest message index.

- Please note that the corresponding Data Frame is not stored into this message object n from which the Remote Frame was sent. Another message object configured to handle Data Frames (RTR=0) is needed.

  A possibility to avoid this:

  - Configure a message object "n" that handles Remote Frames (RTR = 1) and that carries the right identifier and "0" in the DLC.
  - Transmit the Remote Frame.
  - Wait until bit TC in the MOD register is set on the successful transmission of the frame.
  - Now change bit RTR in the descriptor bytes to zero and adjust the DLC.
  - When now the corresponding Data Frame arrives, it will be stored in this message object (as long as there's no other message object with RTR=0, the same identifier, and a higher message object priority).

You have to make sure, though, that the reprogramming of the RTR bit is done before the corresponding Data Frame is transmitted. Be also aware of the fact that the requested Data Frame is not necessarily the frame that directly follows the transmitted Remote Frame.

5.5.3 Evaluation of a received Message with the SAE 81C90/91

- If the SAE 81C90/91 receives a ***Remote Frame*** with an identifier that matches with the identifier of a message object configured to handle Remote Frames (RTR=1), then the DLC, the data bytes and the identifier remain unchanged. The Remote Frame is automatically answered with the corresponding Data Frame.

- If the SAE 81C90/91 receives a ***Data Frame*** with an identifier that matches with the identifier of a message object configured to handle Data Frames (RTR=0), then the data bytes in the Data Field are copied into the data bytes of this message object. The DLC is updated while The RTR bit and the identifier remain unchanged. The device sets the corresponding receive-ready bit in the RRx register and bit RI in the register INT and generates an interrupt to the host controller (if enabled). The interrupt service routine then first has to read INT to determine the reason for the interrupt.

- Then, via the Receive Ready Registers RRR1 and RRR2, the ISR can trace for which message object new data has been received (eg. MO 5).

```
{
   if ((INT & 0x01) != 0)

   /* if bit INT.RI is set (= a mess. has been received) */

   {
      if ((RRR1 & 0x20) != 0)
   /* if message for object 5...*/

      {
       /* now the data bytes can be read */

       ...
```

When accessing one of the data bytes MSGn0 to MSGn7, automatically all ather data bytes of this message are moved to the shadow RAM and can be accessed.

- At last, the software should clear the corresponding bit in the Receive Ready Register (RRR1.5 here):

```
RRR1 = 0xDF;          /* clear bit RRR1.5 */
```

INT.RI will be cleared by the SAE 81C90/91 if all receive interrupts have been serviced and no more receive-ready bits are set.

5.6 How to use the Basic CAN Features of the CAN Module and the SAE 81C90/91

5.6.1 The Basic-CAN Feature of the CAN Module

The Basic-CAN Feature of the CAN module works with message object 15 in combination with the "Mask of Last Message" Register(s). MO 15 is a double-buffered receive-only object. No messages can be sent with this MO (this is prevented by hardware). It depends on the configuration of Message Configuration Register 15 which kind of frames will be stored in MO 15:

- XTD = 0, DIR = 0: Standard Data Frames
- XTD = 0, DIR = 1: Standard Remote Frames
- XTD = 1, DIR = 0: Extended Data Frames
- XTD = 1, DIR = 1: Extended Remote Frames

It is therefore not possible to configure MO 15 to receive both Data Frames and Remote Frames! The DLC and the contents of the Arbitration Registers is "don't care".

Which kind of identifiers will be stored in MO 15 depends on the configuration of the Mask of Last Message (MOLM) and the Global Mask (GM). For the acceptance filtering of MO 15, the GM is ANDed with the MOLM. Identifier bits that are set to "0" in one of these masks will be treated as "don't care". If all identifiers that cannot be stored in any other MO shall be received in MO 15 (= Basic-CAN receive register), then all identifier bits in the MOLM should be set to "0".

Example for standard frames:

```
Global Mask Short:          1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1
ANDed with                  & & & &  & & & &  & & & &  & & & &
Upper Mask of               | | | |  | | | |  | | | |  | | | |
Last Message :              0 0 0 x  x x x x  0 0 0 0  0 0 0 0
=> bits to be masked        | | | |  | | | |  | | | |  | | | |
(to be "don't care" (d)):   d d d -  - - - -  d d d d  d d d d
                            | | | |  | | | |  | | | |  | | | |
Upper Arbitration             | | | |  | | | |  | | | |  | | | |
Register of MO 15:          x x x x  x x x x  x x x x  x x x x
                            | | | |  | | | |  | | | |  | | | |
Identifiers stored:         d d d x  x x x x  d d d d  d d d d

= Identifiers ddddddddddd (00000000000 .. 11111111111)
```

Scenario for MO 15:

Assumptions:
- SIE = 0, IE = 1 (Control Register)
- RXIE = set (Message Control Register 15)

1.) If a frame arrives that fits into MO 15 (and in none of the other objects), the identifier and (in case of a Data Frame) the data bytes are stored into one of the two buffers of MO 15:

- If Buffer1 = released and Buffer2 = released: store into Buffer1.
- If Buffer1 = allocated and Buffer2 = released: store into Buffer2.
- If Buffer1 = released and Buffer2 = allocated: store into Buffer1.
- If Buffer1 = allocated and Buffer2 = allocated: store into the buffer that was filled with the previous data; set MSGLST.

2.) The CAN controller then sets

- INTPND and NEWDAT
  (if MO 15 is configured to receive Data Frames (DIR = 0)) or
- INTPND and RMTPND
  (if MO 15 is configured to receive Remote Frames (DIR = 1))

in MO 15 and bit RXOK in the Control Register.

3.) An interrupt is generated to the CPU. Now the CPU can read the interrupt register and will detect "02" (message 15 Interrupt).

4.) Now the CPU can read (and store elsewhere if necessary)

- the identifier of the received frame (via access to the Arbitration Registers),
- additionally the new data bytes (if a Data Frame was received and the identifier tells the CPU that the received data has to be evaluated).

and then reset in MO 15 (the momentarily accessed buffer):

- INTPND and NEWDAT (if a Data Frame was received)
- INTPND, NEWDAT and RMTPND (if a Remote Frame was received).

This will release the momentarily accessed buffer.

5.) The CPU now has to check INTID in the interrupt register again.

- If INTID is no longer "02" but has been updated by the CAN controller to "00" or an other value, both buffers of MO 15 are released.
- If INTID is still "02", then the other buffer of MO15 is still allocated. The CPU then has to continue with step 4, then automatically accessing the other (still allocated) buffer.

6.) Steps 4 and 5 are to be repeated until both buffers are released.

7.) If both buffers of MO 15 are released and no other interrupt is pending, INTID should then have the value 00h. The interrupt service routine can be left.
Otherwise (if INTID <> 0), the pending interrupt sources have to be serviced until INTID is 00h.

8.) If a Remote Frame was received and is to be answered by this node, this can be done by loading a transmit object with the identifier of the received Remote Frame and the requested data. Then this object must be transmitted.

5.6.2 The Monitor Mode of the SAE 81C90/91

The Monitor Mode of the SAE 81C90/91 works with message object 0 and is enabled by setting bit MM in the CTRL Register (CTRL.0) to "1" (otherwise MO 0 responds like all other MOs). Now MO 0 receives all identifiers that are not accepted by other memory locations, which corresponds to a Basic-CAN receive register. The identifier, the data length code and the RTR-bit in the descriptor bytes of MO 0 are "don't care".

If a ***Data Frame*** arrives (that does not match with one of the other MOs (1..15)), the Data Frame is stored in MO 0:

- The descriptor bytes are updated with the identifier and the data length code (DLC) of the Data Frame. The RTR bit is set to 0.
- The data bytes of MO 0 are overwritten by the data bytes of the Data Frame.
- Bit RR0 in Receive Ready Register 1 (RRR1) is set.
- If bit ERI in the IMSK register and bit RIM0 in the RIMR1 register both are set, an interrupt is generated.

If a ***Remote Frame*** arrives[2] (that does not match with one of the other MOs (1..15)), the Remote Frame is stored in MO 0:

- The descriptor bytes are updated with the identifier and the data length code (DLC) of the Remote Frame (= 0). The RTR bit is set to 1.
- The data bytes of MO 0 are not overwritten by the Remote Frame because the Remote Frame contains no data bytes.
- Bit RR0 in Receive Ready Register 1 (RRR1) is set.
- If bit ERI in the IMSK register and bit RIM0 in the register RIMR1 both are set, an interrupt is generated.

Note:
Bit RRP0 in the Remote Request Pending Register and bit RFI in the Interrupt Register are ***not*** set (as it is normally done when a Remote Frame has been received).

Remote frames received in MO 0 in Monitor Mode should be read out by the CPU (identifier) and if necessary should be answered by another MO configured for the transmission of Data Frames.

Please note that in Monitor Mode, MO 0 acts like a Basic CAN receive register. You shouldn't transmit messages with MO 0 if the Monitor Mode is enabled (although it is still possible).

[2] Please note: The D13 Step of the SAE 81C90/91 is not yet able to receive Remote Frames in Monitor Mode