APPLICATION NOTE

SJA1000
Stand-alone CAN controller

AN97076

Philips
Semiconductors

PHILIPS

Abstract

The Controller Area Network (CAN) is a serial, asynchronous, multi-master communication protocol for connecting electronic control modules, sensors and actuators in automotive and industrial applications.

With the SJA1000, Philips Semiconductors provides a stand-alone CAN controller which is more than a simple replacement of the PCA82C200.

Attractive features are implemented for a wide range of applications, supporting system optimization, diagnosis and maintenance.

APPLICATION NOTE

SJA1000
Stand-alone CAN controller

AN97076

Author(s):
Peter Hank, Egon Jöhnk
Systems Laboratory Hamburg
Germany

Keywords

SJA1000
Stand-alone CAN controller
CAN2.0B
PeliCAN

Date: 1997-12-15

Summary

This application note focuses on the description of the SJA1000 as part of a system. Diagrams illustrate the interface capability of the SJA1000 for the connection to a variety of microcontrollers and CAN transceiver circuits. General flow diagrams for programming the device in different modes are shown in detail. Configuration, Transmission and Reception program examples are attached. Special emphasis has been placed on the description of the SJA1000 PeliCAN features including useful examples, e.g., for automatic bit-rate detection, global clock synchronization and system self test.

CONTENTS

(this page has intentionally been left lank)

1. INTRODUCTION

The SJA1000 is a stand-alone CAN Controller product with advanced features for use in automotive and general industrial applications. It is intended to replace the PCA82C200 because it is hardware and software compatible. Due to an enhanced set of functions this device is well suited for many applications especially when system optimization, diagnosis and maintenance are important.

This report is intended to guide the user in designing complete CAN nodes based on the SJA1000. The report provides typical application circuit diagrams and flow charts for programming.

2. OVERVIEW

The stand-alone CAN controller SJA1000 [1] has two different Modes of Operation:

- BasicCAN Mode (PCA82C200 compatible)
- PeliCAN Mode

Upon Power-up the BasicCAN Mode is the default mode of operation. Consequently, existing hardware and software developed for the PCA82C200 can be used without any change. In addition to the functions known from the PCA82C200 [7], some extra features have been implemented in this mode which make the device more attractive. However, they do not influence the compatibility to the PCA82C200.

The PeliCAN Mode is a new mode of operation which is able to handle all frame types according to CAN specification 2.0B [8]. Furthermore it provides a couple of enhanced features which makes the SJA1000 suitable for a wide range of applications.

2.1 SJA1000 Features

The features of the SJA1000 can be clustered into three main groups:

Well-established PCA82C200 Functions

Features of this group have already been implemented in the PCA82C200.

Improved PCA82C200 Functions

Partly these functions have already been implemented in the PCA82C200. However, in the SJA1000 they have been improved in terms of speed, size or performance.

Enhanced Functions in PeliCAN Mode

In PeliCAN Mode the SJA1000 offers a couple of Error Analysis Functions supporting diagnosis, system maintenance and optimization. Furthermore functions for general *CPU* support and System Self Test have been added in this mode.

In the following table all SJA1000 features are listed including their main benefits for the application.

Table 1: SJA1000 Features with benefits for the application

Well-established PCA82C200 Functions

Flexible microprocessor interface	Allows interfacing most microprocessors or microcontrollers.
Programmable CAN output driver	Interface to all kind of physical layers.
CAN bit-rates up to 1Mbit/s	The SJA1000 covers the whole range of bit-rates, including high speed applications.

Improved PCA82C200 Functions

CAN 2.0B (passive)	The CAN 2.0B passive characteristics of the SJA1000 allows the CAN controller to tolerate CAN messages with 29-bit identifiers.
64 byte Receive FIFO	Up to 21 messages can be stored in the Receive FIFO, this lengthens the max. interrupt service time and avoids data overrun conditions.
24 MHz Clock frequency	Faster microprocessor access and more CAN bit-timing options.
Receive Comparator Bypass	Shortens the internal delays, resulting in a much higher CAN bus length due to an improved bit-timing programming.

Enhanced Functions in PeliCAN Mode

CAN 2.0B (active)	CAN 2.0B active support extends application field to networks with 29-bit identifiers.
Transmit Buffer	Single message transmit buffer for messages with 11-bit or 29-bit identifiers.
Enhanced Acceptance Filter	Two acceptance filter modes supporting both 11-bit and 29-bit identifier filtering.
Readable Error Counters	Supports error analysis which can be used for:
Programmable Error Warning Limit	- diagnostics, system maintenance and system optimization
Error Code Capture Register	during the prototype phase and during normal operation.
Error Interrupts	
Arbitration Lost Capture Interrupt	Supports system optimization including message latency time analysis.
Single Shot Transmission	Minimizes software commands and allows fast reloading of transmit buffer.
Listen Only Mode	SJA1000 can operate as a passive CAN monitor which can be used for analyzing the CAN bus traffic or for automatic bit-rate detection.
Self Test Mode	Supports functional self tests of complete CAN nodes or self reception in a system.

2.2 CAN Node Architecture

Generally each CAN module can be divided into different functional blocks. The connection to the CAN bus lines is usually built with a **CAN Transceiver** optimized for the applications [3], [4], [5]. The transceiver controls the logic level signals from the CAN controller into the physical levels on the bus and vice versa.

The next upper level is a **CAN Controller** which implements the complete CAN protocol defined in the CAN Specification [8]. Often it also covers message buffering and acceptance filtering.

All these CAN functions are controlled by a **Module Controller** which performs the functionality of the application. For example, it controls actuators, reads sensors and handles the man-machine interface (MMI).

As shown in Figure 1 the SJA1000 stand-alone CAN controller is always located between a microcontroller and the transceiver, which is an integrated circuit in most cases.

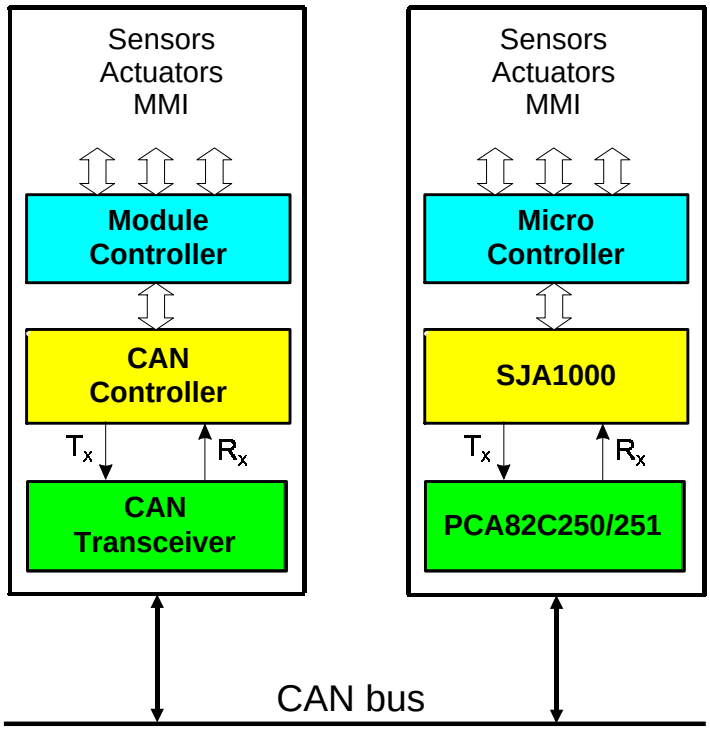

Figure 1: CAN Module Set-up

2.3 Block Diagram

The following figure shows the block diagram of the SJA1000.

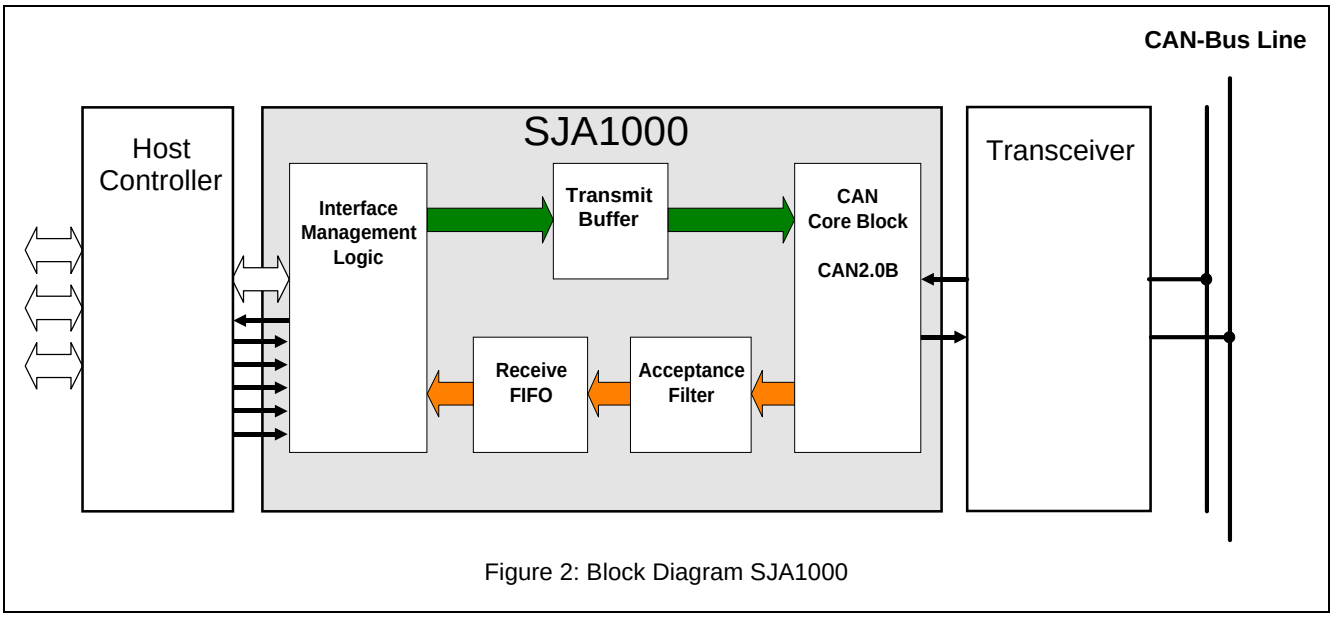

Figure 2: Block Diagram SJA1000

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

The **Interface Management Logic** block performs a link to the external host controller which can be a microcontroller or any other device. Every register access via the SJA1000 multiplexed address/data bus and controlling of the read/write strobes is handled in this unit. Additionally to the BasicCAN functions known from the PCA82C200, new PeliCAN features have been added. As a consequence of this, additional registers and logic have been implemented mainly in this block.

The **Transmit Buffer** of the SJA1000 is able to store one complete message (Extended or Standard). Whenever a transmission is initiated by the host controller the Interface Management Logic forces the CAN Core Block to read the CAN message from the Transmit Buffer.

When receiving a message, the CAN Core Block converts the serial bit stream into parallel data for the **Acceptance Filter**. With this programmable filter the SJA1000 decides which messages actually are received by the host controller.

All received messages accepted by the acceptance filter are stored within a **Receive FIFO**. Depending on the mode of operation and the data length up to 32 messages can be stored. This enables the user to be more flexible when specifying interrupt services and interrupt priorities for the system because the probability of data overrun conditions is reduced extremely.

3. SYSTEM

For connection to the host controller, the SJA1000 provides a multiplexed address/data bus and additional read/write control signals. The SJA1000 could be seen as a peripheral memory mapped I/O device for the host controller.

3.1 SJA1000 Application

Configuration Registers and pins of the SJA1000 allow to use all kinds of integrated or discrete CAN transceivers. Due to the flexible microcontroller interface applications with different microcontrollers are possible.

In Figure 3 a typical SJA1000 application diagram including 80C51 microcontroller and PCA82C251 transceiver is shown. The CAN controller functions as a clock source and the reset signal is generated by an external reset circuitry. In this example the chip select of the SJA1000 is controlled by the microcontroller port function P2.7. Instead of this, the chip select input could be tied to VSS. Control via an address decoder is possible, e.g., when the address/data bus is used for other peripherals.

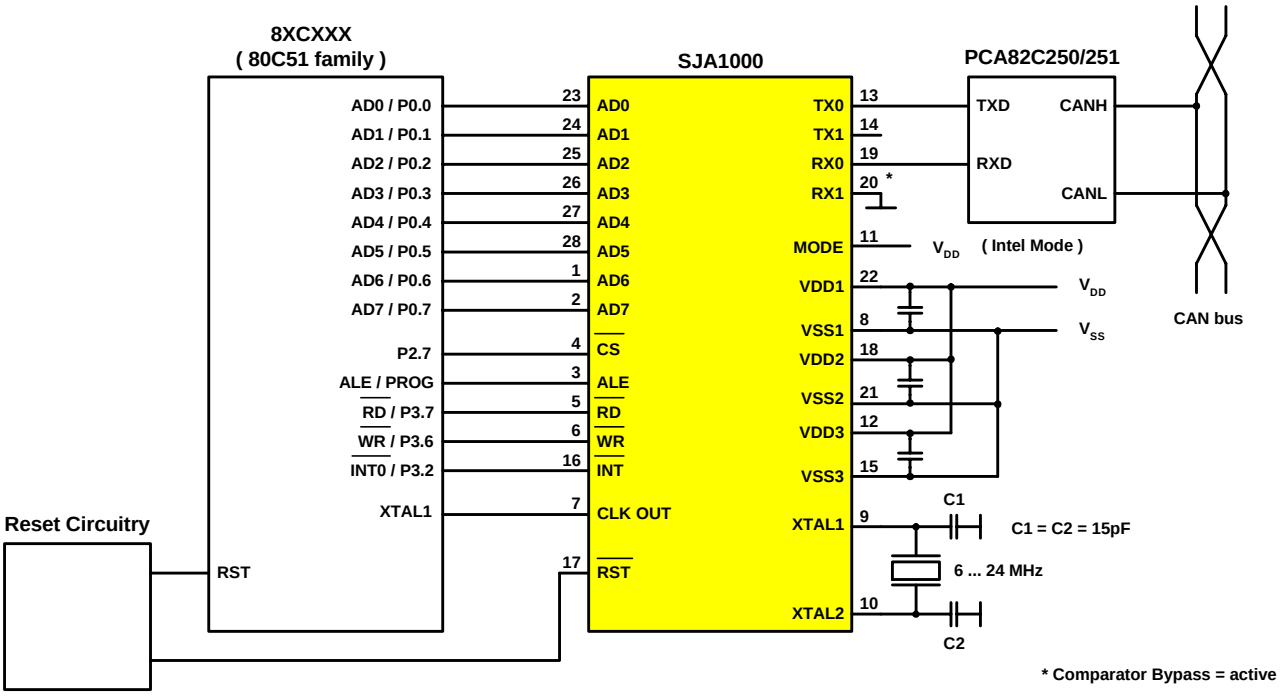

Figure 3: Typical SJA1000 Application

3.2 Power Supply

The SJA1000 has three pairs of voltage supply pins which are used for different digital and analog internal blocks of the CAN controller.

VDD1 / VSS1: internal logic (digital)
VDD2 / VSS2: input comparator (analog)
VDD3 / VSS3: output driver (analog)

The supply has been separated for better EME behaviour. For instance the VDD2 can be de-coupled via an RC filter for noise suppression of the comparator.

3.3 Reset

For a proper reset of the SJA1000 a stable oscillator clock has to be provided at XTAL1 of the CAN controller, see also chapter 3.4. An external reset on pin 17 is synchronized and internally lengthened to 15 t_{XTAL}. This guarantees a correct reset of all SJA1000 registers (see [1]). Note that an oscillator start-up time has to be taken into account upon power-up.

3.4 Oscillator and Clocking Strategy

The SJA1000 can operate with the on-chip oscillator or with external clock sources. Additionally the CLK OUT pin can be enabled to output the clock frequency for the host controller. Figure 4 shows four different clocking principles for applications with the SJA1000. If the CLK OUT signal is not needed, it can be switched off with the Clock Divider register (Clock Off = 1). This will improve the EME performance of the CAN node.
The frequency of the CLK OUT signal can be changed with the Clock Divider Register:

$f_{CLK\ OUT} = f_{XTAL}$ / Clock Divider factor (1,2,4,6,8,10,12,14).

Upon power up or hardware reset the default value for the Clock Divider factor depends on the selected interface mode (pin 11). If a 16 MHz crystal is used in Intel mode, the frequency at CLK OUT is 8 MHz. In Motorola mode a Clock Divider factor of 12 is used upon reset which results in 1,33 MHz in this case.

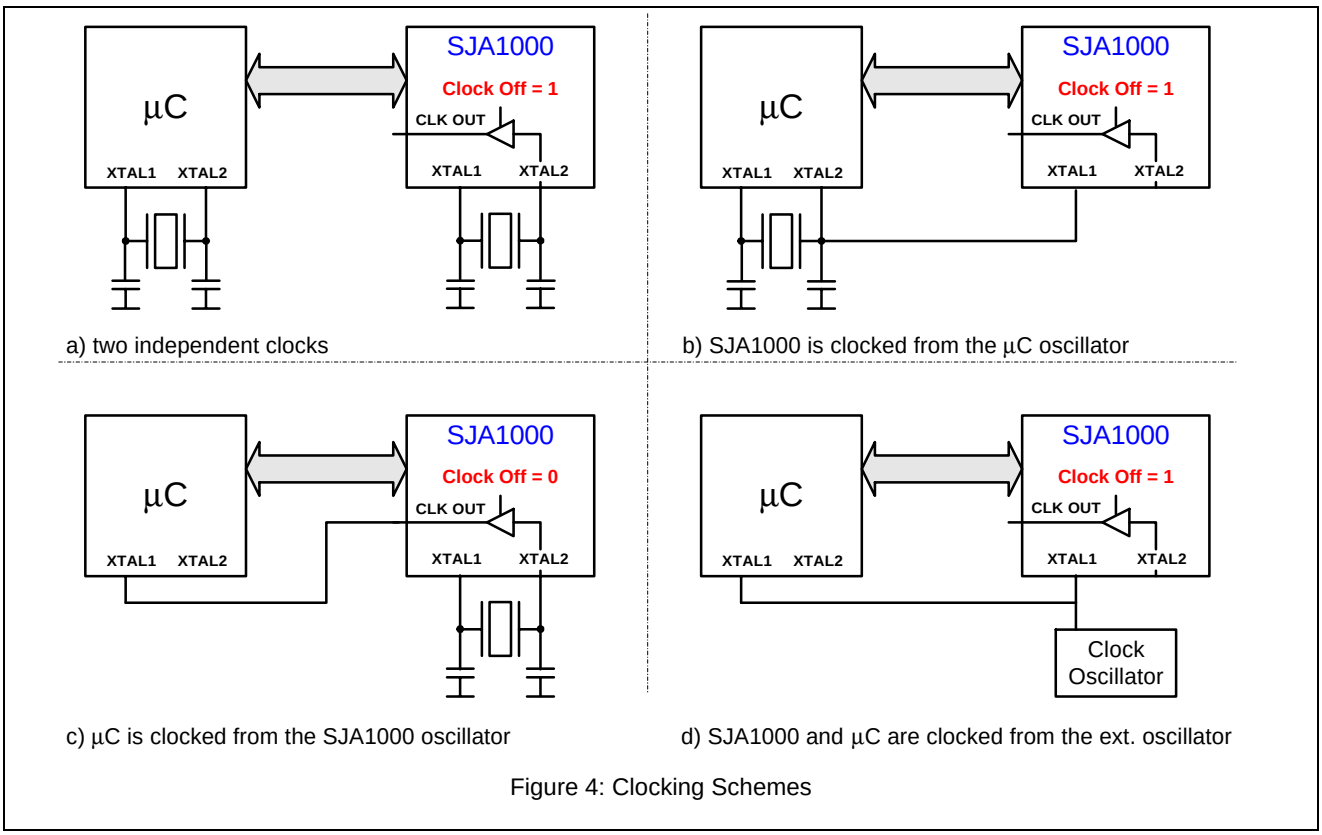

Figure 4: Clocking Schemes

3.4.1 Sleep and Wake-up

Upon setting the Go To Sleep bit in the Command Register (BasicCAN mode) or the Sleep Mode bit in the Mode Register (PeliCAN mode) the SJA1000 will enter Sleep Mode if there is no bus activity and no interrupt is pending. The oscillator keeps on running until 15 CAN bit times have been passed. This allows a microcontroller clocked with the CLK OUT frequency to enter its own low power consumption mode.
If one of three possible wake-up conditions [1] occurs the oscillator is started again and a Wake-up interrupt is generated. As soon as the oscillator is stable the CLK OUT frequency is active.

3.5 CPU Interface

The SJA1000 supports the direct connection to two famous microcontroller families: 80C51 and 68xx. With the MODE pin of the SJA1000 the interface mode is selected.

Intel Mode: MODE = high
Motorola Mode: MODE = low

The connection for the address/data bus and the read/write control signals in both Intel and Motorola mode is shown in Figure 5. For Philips 8-bit microcontrollers based on the 80C51 family and the 16-bit microcontrollers with XA architecture the Intel Mode is used.

For other controllers additional glue logic is necessary for adaptation of the address/data bus and the control signals. However, it has to be made sure that no write pulses are generated during power-up. Another possibility is to disable the CAN controller with a high-level on the chip select input in this time.

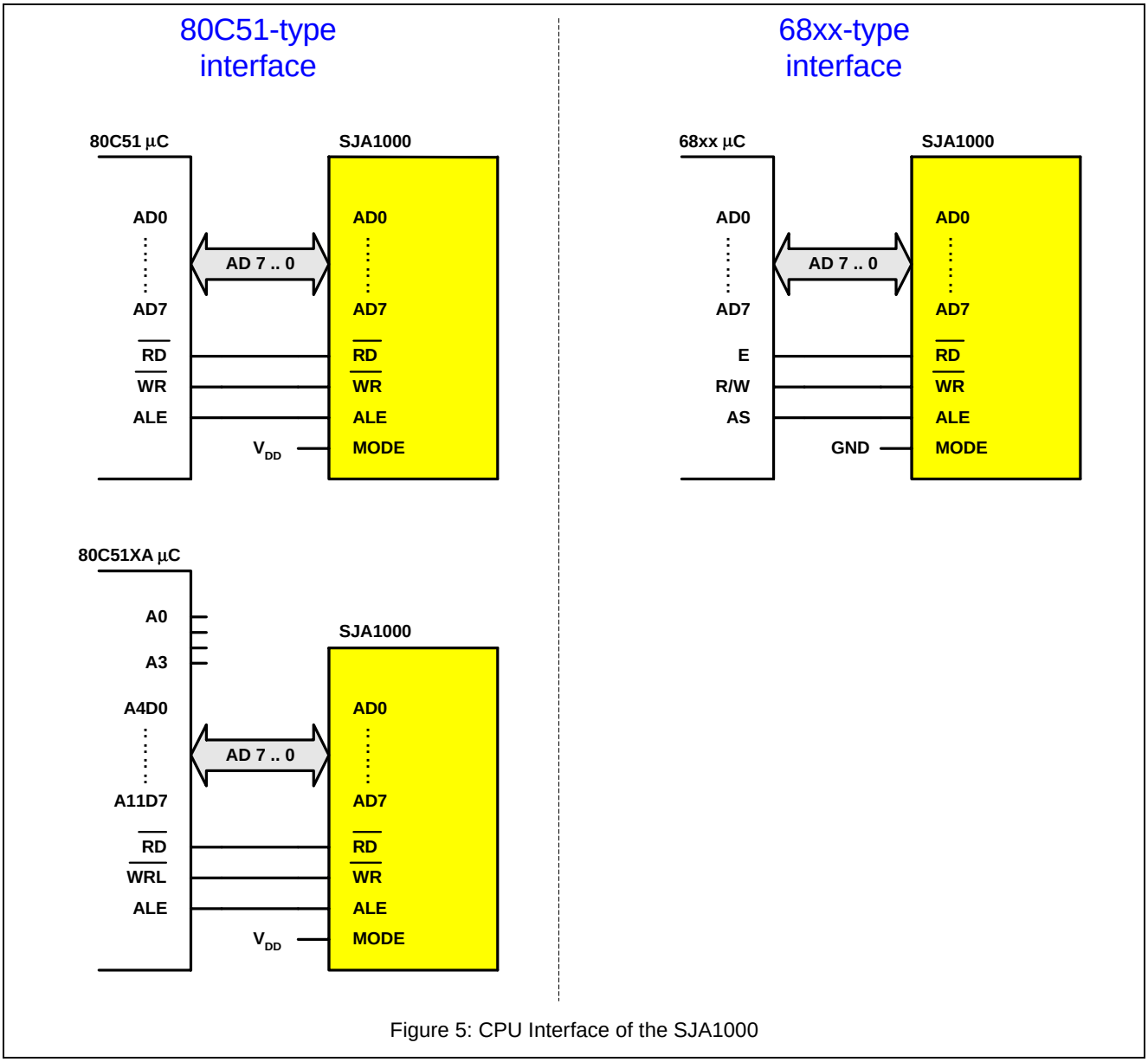

Figure 5: CPU Interface of the SJA1000

3.6 Physical Layer Interface

For compatibility purposes with the PCA82C200, the SJA1000 includes an analog receive input comparator circuit. This integrated comparator can be used if the transceiver function is realized with discrete components.

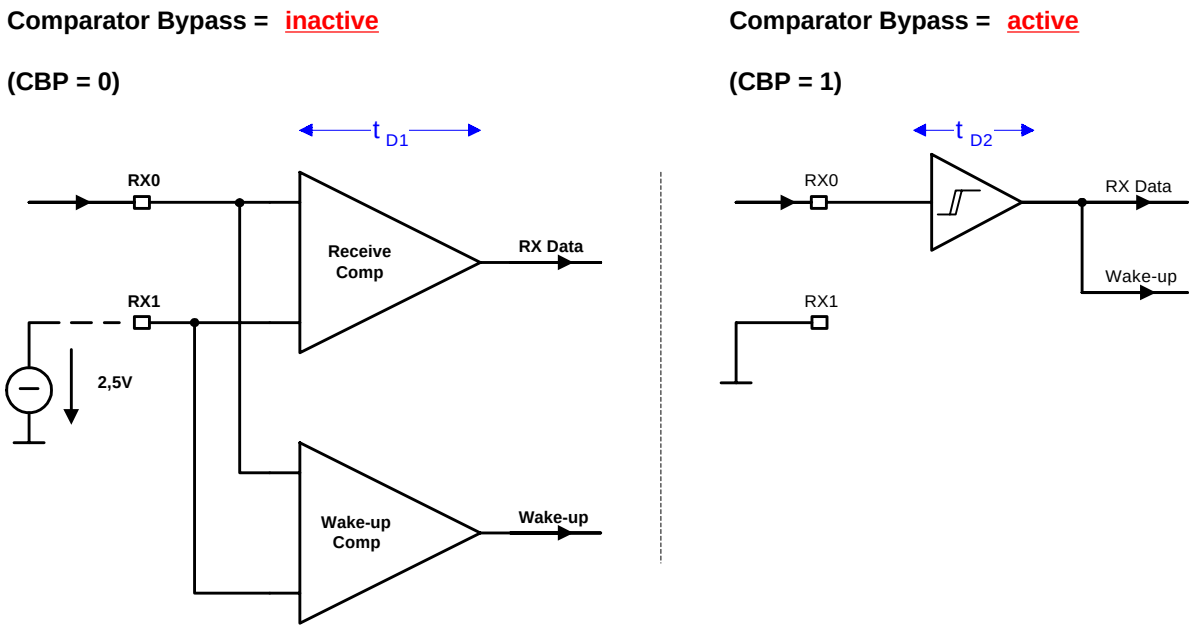

Figure 6: SJA1000 Receive Input Comparator

If an external integrated transceiver circuit is used and the comparator bypass function is not enabled in the Clock Divider Register, the RX1 input has to be connected to a reference voltage of 2.5V (reference voltage output of existing transceiver circuits). Figure 6 shows the equivalent circuits for both configurations: CBP = active and CBP = inactive. Additionally the path for the wake-up signal is drawn.
For all new applications where an integrated transceiver circuit is used, it is recommended to activate the comparator bypass function of the SJA1000 (Figure 7). If this function is enabled, a schmitt-trigger input is used and the internal propagation delay t_{D2} is much shorter as the delay t_{D1}. of the receive comparator. This has a positive impact on the maximum bus length [6]. Additionally, it will reduce the supply current in sleep mode significantly.

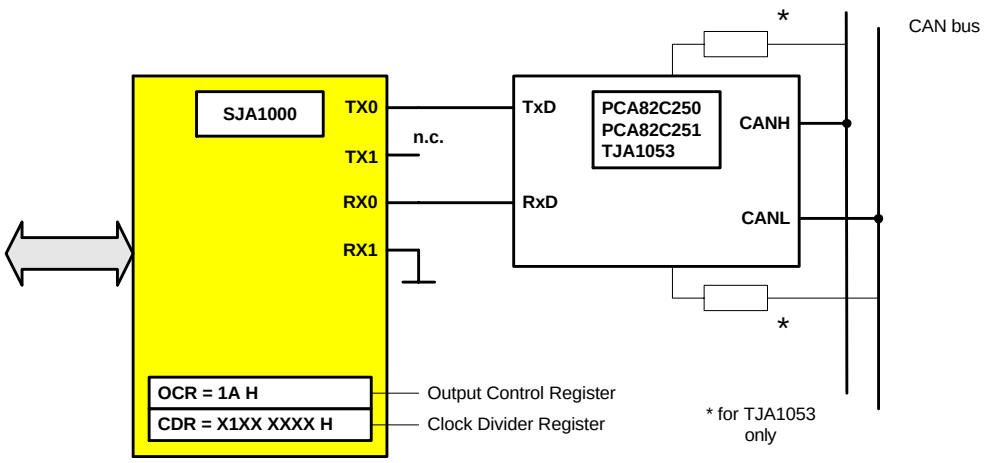

Figure 7: Standard application with integrated transceiver circuit

4. CONTROL OF CAN COMMUNICATION

4.1 Basic Functions and Registers for Controlling the SJA1000

The functionality with respect to configuration and activities of the SJA1000 is given by the program of the host controller. Thus the SJA1000 is tailored to meet the requirements of CAN-bus systems with different properties. The data exchange between the host controller and the SJA1000 is done via a set of registers (control segment) and a RAM (message buffer). The registers and an address window to a part of the RAM, making up the Transmit and Receive Buffers, appear to the host controller as peripheral registers.

Table 2 lists these registers grouped according to their usage in a system.

Note, that some registers are available in PeliCAN mode only and that the Control Register is available in BasicCAN mode only. Furthermore some registers are read only or write only and some can be accessed during Reset Mode only.

More information about the registers with respect to read and/or write access, bit definition and reset values, can be found in the data sheet [1].

Table 2: Classification of the internal registers of the SJA1000

Type of Usage	Register Name	(Symbol)	Register Address: PeliCAN mode	Register Address: BasicCAN mode	Functionality
elements for selecting different operation modes	Mode	(MOD)	0	—	Sleep-. Acceptance Filter-, Self Test-, Listen Only- and Reset-Mode selection
	Control	(CR)	—	0	Reset Mode selection in BasicCAN mode
	Command	(CMR)	—	1	Sleep mode command in BasicCAN mode
	Clock Divider	(CDR)	31	31	set-up of clock signal at CLKOUT (pin 7) selection of PeliCAN Mode, Comparator Bypass Mode, TX1 (pin 14) Output Mode
elements for setting up the CAN communication	Acceptance Code, Mask	(ACR) (AMR)	16-19 20-23	4, 5	selection of bit patterns for Acceptance Filtering
	Bus Timing 0 1	(BTR0) (BTR1)	6 7	6 7	set-up of Bit Timing Parameters
	Output Control	(OCR)	8	8	selection of Output Driver properties

Table 2: Classification of the internal registers of the SJA1000 (continued)

Type of Usage	Register Name (Symbol)	Register Address:		Functionality
		PeliCAN mode	BasicCAN mode	
basic elements for the CAN communication	Command (CMR)	1	1	commands for Self Reception, Clear Data Overrun, Release Receive Buffer, Abort Transmission and Transmission Request
	Status (SR)	2	2	status of message buffers, status of CAN Core Block
	Interrupt (IR)	3	3	CAN Interrupt flags
	Interrupt Enable (IER)	4	—	enable/disable of interrupt events in PeliCAN mode
	Control (CR)	—	0	enable/disable of interrupt events in BasicCAN mode
elements for a comprehensive error detection and analysing	Arbitration Lost Capture (ALC)	11	—	shows bit position, where arbitration was lost
	Error Code Capture (ECC)	12	—	shows last error type and location
	Error Warning Limit (EWLR)	13	—	selection of threshold for generating an Error Warning Interrupt
	RX Error Counter (RXERR)	14	—	reflects the current value of the Receive Error Counter
	TX Error Counter (TXERR)	14, 15	—	reflects the current value of the Transmit Error Counter
	Rx Message Counter (RMC)	29	—	number of messages in the Receive FIFO
	Rx Buffer Start Addr. (RBSA)	30	—	shows the current internal RAM address of the message available in the Receive Buffer
message buffers	Transmit Buffer (TXBUF)	16-28	10-19	
	Receive Buffer (RXBUF)	16-28	20-29	

4.1.1 Transmit Buffer / Receive Buffer

The data to be transmitted on the CAN bus is loaded into the memory area of the SJA1000, called “Transmit Buffer”. The data received from the CAN bus is stored in the memory area of the SJA1000, called “Receive Buffer”. These buffers contains 2, 3 or 5 bytes for the identifier and frame information (dependent on mode and frame type) and up to 8 data bytes. For further information about the definition and composition of the bits in the message buffers see the data sheet [1].

- BasicCAN mode: The buffers are 10-bytes long (see Table 3).
  - 2 identifier bytes
  - up to 8 data bytes.
- PeliCAN mode: The buffers are 13 bytes long (see Table 4).
  - 1 byte for Frame Information
  - 2 or 4 identifier bytes (Standard Frame or Extended Frame)
  - up to 8 data bytes.

Table 3: Layout of Rx- and Tx-Buffer in BasicCAN mode

CAN Addr. (dec.)	Name	Composition and Remarks
Tx-Buffer: 10 Rx-Buffer: 20	Identifier Byte 1	8 Identifier bits
Tx-Buffer: 11 Rx-Buffer: 21	Identifier Byte 2	3 Identifier bits, 1 Remote Transmission Request bit, 4 bits for the Data Length Code, indicating the amount of data bytes
Tx-Buffer: 12-19 Rx-Buffer: 22-29	Data Byte 1 - 8	up to 8 data bytes as indicated by the Data Length Code

Table 4: Layout of Rx-[1.] (read access) and Tx-Buffer (write access[2.]) in PeliCAN mode

CAN Addr. (dec.)	Name	Composition and Remarks
16	Frame Information	1 bit indicating, if the message contains a Standard or Extended frame 1 Remote Transmission Request bit 4 bits for the Data Length Code, indicating the amount of data bytes
17, 18	Identifier Byte 1, 2	Standard Frame: 11 Identifier bits Extended Frame: 16 Identifier bits
19, 20	Identifier Byte 3, 4	Extended Frame only: 13 Identifier bits
Frame type Standard: 19 - 26 Extended: 21 - 28	Data Byte 1 - 8	up to 8 data bytes as indicated by the Data Length Code

1. The whole Receive FIFO (64 bytes) can be accessed using the CAN addresses 32 to 95 (see also chapter 5.1).
2. A read access of the Tx-Buffer can be done using the CAN addresses 96 to 108 (see also chapter 5.1)

4.1.2 Acceptance Filter

The stand-alone CAN controller SJA1000 is equipped with a versatile acceptance filter, which allows an automatic check of the identifier and data bytes. Using these effective filtering methods, messages or a group of messages not valid for a certain node can be prevented from being stored in the Receive Buffer. Thus it is possible to reduce the processing load of the host controller.

The filter is controlled by the acceptance code and mask registers according to the algorithms given in the data sheet [1]. The received data is compared bitwise with the value contained in the Acceptance Code register. The Acceptance Mask Register defines the bit positions, which are relevant for the comparison (0 = relevant, 1 = not relevant). For accepting a message all **relevant** received bits have to match the respective bits in the Acceptance Code Register.

Acceptance Filtering in BasicCAN Mode

This mode is implemented in the SJA1000 as a plug-and-play replacement (hardware and software) for the PCA82C200. Thus the acceptance filtering corresponds to the possibilities, which were found in the PCA82C200 [7]. The filter is controlled by two 8-bit wide registers – Acceptance Code Register (ACR) and Acceptance Mask Register (AMR). The 8 most significant bits of the identifier of the CAN message are compared to the values contained in these registers, see also Figure 8. Thus always groups of eight identifiers can be defined to be accepted for any node.

Example:

The Acceptance Code register (ACR) contains:

The Acceptance Mask register (AMR) contains:

Messages with the following 11-bit identifiers are accepted

(x = don't care)

	MSB							LSB			
ACR	**0**	**1**	1	1	0	**0**	**1**	**0**			
AMR	0	0	***1***	***1***	***1***	0	0	0			
	0	**1**	x	x	x	**0**	**1**	**0**	x	x	x
	ID.10										ID.0

At the bit positions containing a "1" in the Acceptance Mask register, any value is allowed in the composition of the identifier. The same is valid for the three least significant bits. Thus 64 different identifiers are accepted in this example. The other bit positions must be equal to the values in the Acceptance Code register.

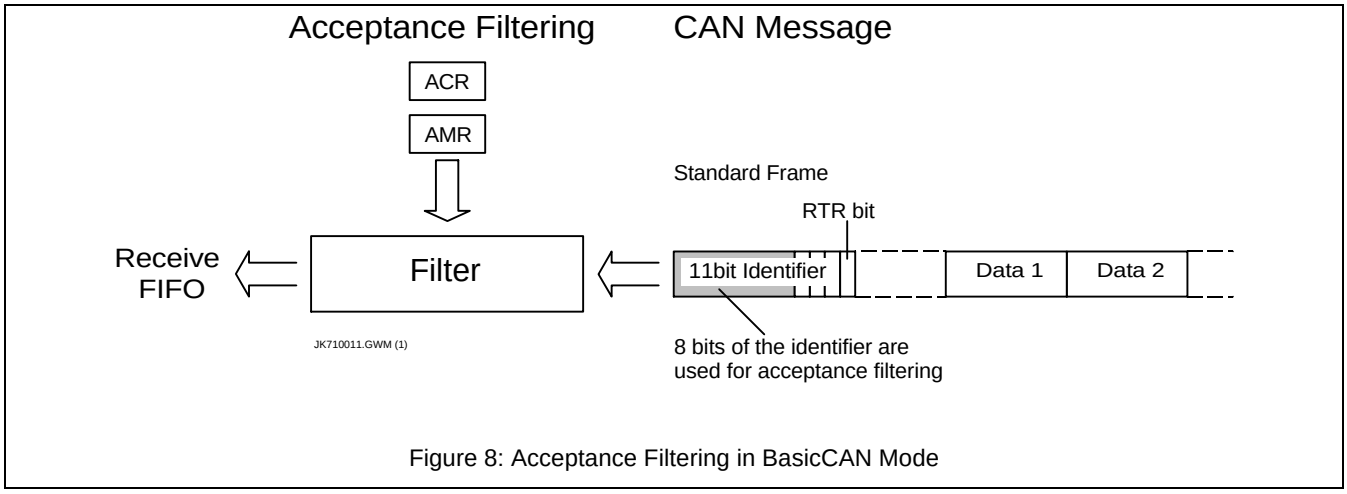

Figure 8: Acceptance Filtering in BasicCAN Mode

Acceptance Filtering in PeliCAN Mode

The acceptance filtering has been expanded for the PeliCAN mode: Four 8-bit wide Acceptance Code registers (ACR0, ACR1, ACR2 and ACR3) and Acceptance Mask registers (AMR0, AMR1, AMR2 and AMR3) are available for a versatile filtering of messages. These registers can be used for controlling a single long filter or two shorter filters, as shown in Figure 9 and Figure 10. Which bits of the message are used for the acceptance filtering, depend on the received frame (Standard or Extended) and on the selected filter mode (single or dual filter). Table 5 gives more information about which bits of the message are compared with the Acceptance Code and Mask bits. As it is seen from the figures and the table, it is possible to include the RTR bit and even data bytes in the acceptance filtering for Standard Frames. In any case for all message bits, which shall **not** be included in the acceptance filtering (e.g. if groups of messages are defined for acceptance), the Acceptance Mask Register must contain a “1” at the corresponding bit position.

If a message doesn’t contain data bytes (e.g. in a Remote Frame or if the Data Length Code is zero) but data bytes are included in the acceptance filtering, such messages are accepted, if the identifier up to the RTR bit is valid.

Example 1:

Let us assume, that the same 64 Standard Frame messages as described in the example on page 18 have to be filtered in PeliCAN mode.

This can be done using one long filter (Single Filter Mode).

The Acceptance Code Registers (ACRn) and Acceptance Mask Registers (AMRn) contain:

n	0	1 (upper 4 bits)	2	3
ACRn	0 1 X X X 0 1 0	X X X X	X X X X X X X X	X X X X X X X X
AMRn	0 0 1 1 1 0 0 0	1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
accepted messages (ID.28..ID.18, RTR)	0 1 x x x 0 1 0 x x x x			

(“X” = irrelevant, “x” = don’t care, only the upper 4 bits of ACR1 and AMR1 are used)

At the bit positions containing a “1” in the Acceptance Mask registers, any value is allowed in the composition of the identifier, for the Remote Transmission Request bit and for the bits of data byte 1 and 2.

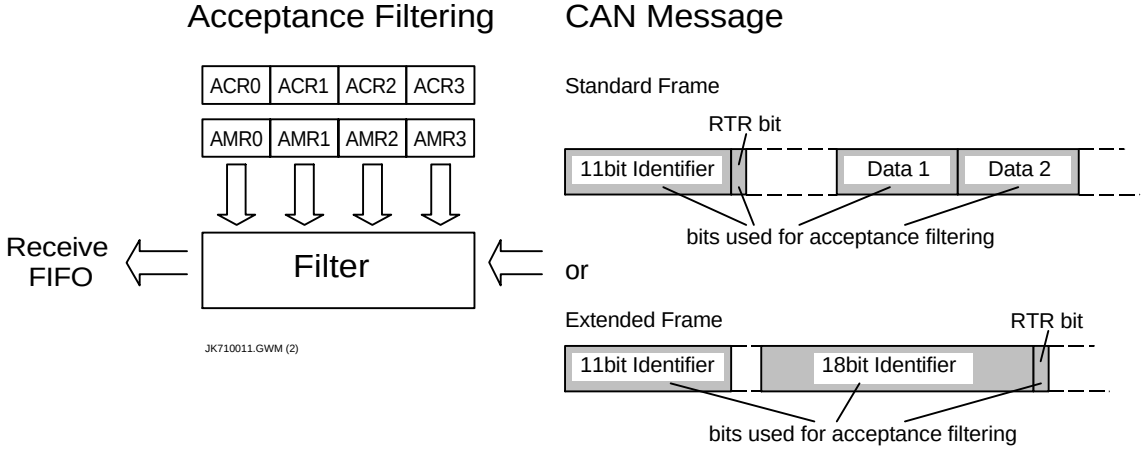

Figure 9: Acceptance Filtering in PeliCAN Mode (Single Filter Mode)

Example 2:

Suppose the following 2 messages with a Standard Frame Identifier have to be accepted without any further decoding of the identifier bits. Data and Remote Frames have to be received correctly. Data bytes are not involved in the acceptance filtering.

message 1: (ID.28) 1***0***11 ***1***100 101 (ID.18)
message 2: (ID.28) 1***1***11 ***0***100 101 (ID.18)

Using the Single Filter Mode results in accepting four messages and not only the requested two:

n	0	1 (upper 4 bits)	2	3
ACRn	1 X 1 1 X 1 0 0	1 0 1 X	X X X X X X X X	X X X X X X X X
AMRn	0 1 0 0 1 0 0 0	0 0 0 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
accepted messages (ID.28..ID.18, RTR)	1 0 1 1 0 1 0 0 1 0 1 x 1 1 1 1 0 1 0 0 1 0 1 x (message 2) 1 0 1 1 1 1 0 0 1 0 1 x (message 1) 1 1 1 1 1 1 0 0 1 0 1 x			

("X" = irrelevant, "x" = don't care, only the upper 4 bits of ACR1 and AMR1 are used)

This result does **not** meet the request for receiving 2 messages without any further decoding.

Using the Dual Filter mode gives the correct result:

	Filter 1			Filter 2	
n	0	1	3 lower 4 bits	2	3 upper 4 bits
ACRn	1 0 1 1 1 1 0 0	1 0 1 X X X X X	... X X X X	1 1 1 1 0 1 0 0	1 0 1 X ...
AMRn	0 0 0 0 0 0 0 0	0 0 0 1 1 1 1 1	... 1 1 1 1	0 0 0 0 0 0 0 0	0 0 0 1 ...
accepted messages (ID.28..ID.18, RTR)	1 0 1 1 1 1 0 0 1 0 1 x (message 1)			1 1 1 1 0 1 0 0 1 0 1 x (message 2)	

("X" = irrelevant, "x" = don't care)

Message 1 is accepted by Filter 1 and message 2 by Filter 2. As messages are accepted and stored into the Receive FIFO if they are accepted at least by one of the two filters, this solution meets the request.

Example 3:

In this example a group of messages with an Extended Frame Identifier are filtered using a long single acceptance filter.

n	0	1	2	3 (upper 6 bits)
ACRn	1 0 1 1 0 1 0 0	1 0 1 1 0 0 0 X	1 1 0 0 X X X X	0 0 1 1 0 X X X
AMRn	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 0 0 0 1 1 1 1	0 0 0 0 0 1 1 1
accepted messages (ID.28..ID.0, RTR)	1 0 1 1 0 1 0 0 1 0 1 1 0 0 0 x 1 1 0 0 x x x x 0 0 1 1 0 x			

("X" = irrelevant, "x" = don't care, only the upper 6 bits of ACR3 and AMR3 are used)

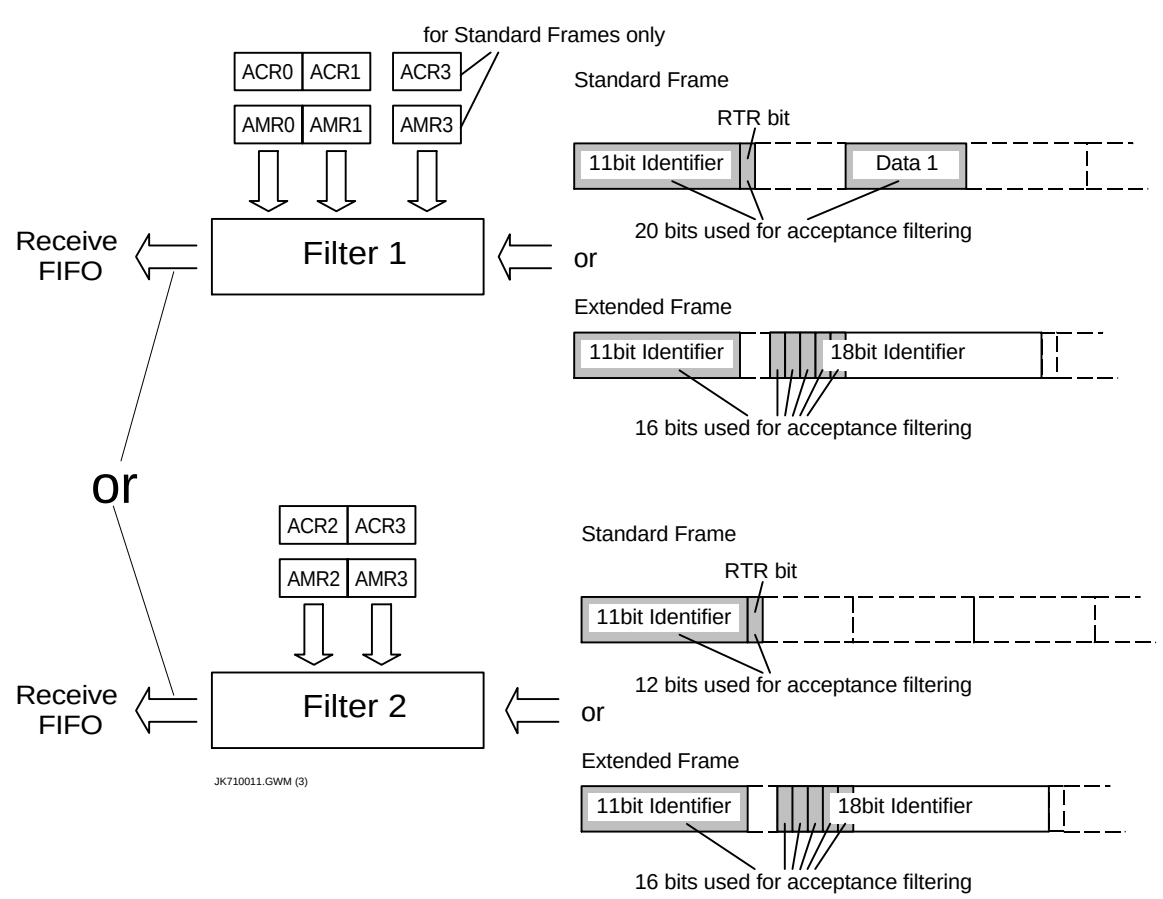

Figure 10: Acceptance Filtering in PeliCAN Mode (Dual Filter Mode)

Example 4:

There are systems, which use Standard Frames only and identify messages by the 11-bit identifier and the first two data bytes. Such a protocol is used, e.g., in the DeviceNet, where the first two data bytes define a message header and the fragmentation protocol, if messages contain more than 8 data bytes. For this system type the SJA1000 can filter two data bytes in single filter mode and one data byte in dual filter mode in addition to the 11-bit identifier and the RTR-bit.

Using the Dual Filter mode, the following example shows effective filtering of messages in such a system:

	Filter 1			Filter 2	
n	0	1	3 lower 4bits	2	3 upper 4 bits
ACRn	1 1 1 0 1 0 1 1	0 0 1 0 1 1 1 1	... 1 0 0 1	1 1 1 1 0 1 0 0	X X X 0 ...
AMRn	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	... 0 0 0 0	0 0 0 0 0 0 0 0	1 1 1 0 ...
accepted messages	1 1 1 0 1 0 1 1 ID + RTR	0 0 1 0 1 1 1 1 first data byte	... 1 0 0 1	1 1 1 1 0 1 0 0 ID	x x x 0 RTR

(“X” = irrelevant, “x” = don’t care)

Filter 1 is used for filtering messages with
- the identifier “1 1 1 0 1 0 1 1 0 0 1”
- RTR = “0” i.e. Data Frames only and
- the data byte “1 1 1 1 1 0 0 1” (this means e.g. for the DeviceNet: all fragments for one message are filtered).

Filter 2 is used for filtering a group of 8 messages with

- the identifiers “1 1 1 1 0 1 0 0 0 0 0” through “1 1 1 1 0 1 0 0 1 1 1” and
- RTR = “0”, i.e. Data Frames only.

Table 5: Summary of Acceptance Filter in PeliCAN mode

Frame Type	Single Filter mode (Figure 9)	Dual Filter mode (Figure 10)
Standard	message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) - 2nd data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/upper 4 bits of ACR1/ACR2/ACR3 - AMR0/upper 4 bits of AMR1/AMR2/AMR3 (unused bits of the Acceptance Mask Register should be set to “1”)	<u>Filter 1</u> message bits used for acceptance: - 11 bit identifier - RTR Bit - 1st data byte (8 bit) Acceptance Code & Mask registers used: - ACR0/ACR1/lower 4 bits of ACR3 - AMR0/AMR1/lower 4 bits of AMR3 <u>Filter 2</u> message bits tested for acceptance: - 11 bit identifier - RTR Bit Acceptance Code & Mask registers used: - ACR2/upper 4 bits of ACR3 - AMR2/upper 4 bits of AMR3
Extended	message bits used for acceptance: - 11 bit basic identifier - 18 bit extended identifier - RTR Bit Acceptance Code & Mask registers used: - ACR0/ACR1/ACR2/upper 6 bits of ACR3 - AMR0/ AMR1/ AMR2/ upper 6 bits of AMR3 (unused bits of the Acceptance Mask Register should be set to “1”)	<u>Filter 1</u> message bits used for acceptance: - 11 bit basic identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR0/ACR1 and AMR0/AMR1 <u>Filter 2</u> message bits tested for acceptance: - 11 bit basic identifier - 5 most significant bits of extended identifier Acceptance Code & Mask registers used: - ACR2/ACR3 and AMR2/AMR3

4.2 Functions for CAN Communications

The steps to be taken for establishing communication via the CAN bus are:

- after power-on of the system
  - setting up the host controller with respect to hardware and software links to the SJA1000
  - setting up the CAN controller for the communication with respect to the selection of mode, acceptance filtering, bit timing etc. – to be done also after a hardware reset of the SJA1000
- during the main process of the application
  - prepare messages to be transmitted and activate the SJA1000 to transmit them
  - react on messages received by the CAN controller
  - react on errors occurred during communication

Figure 11 shows the general flow of a program. In the following paragraphs the flows, which refer directly to controlling the SJA1000, are described in more detail.

4.2.1 Initialization

As mentioned before, the stand-alone CAN controller SJA1000 has to be set up for CAN communication after power-on or after a hardware reset. Furthermore the SJA1000 may be re-configured (re-initialized) during operation by the host controller, which may send a (software) reset request. The flow is given in Figure 12. A programming example using an 80C51 microcontroller derivative is given in this chapter.

After power-on the host controller runs through its own special reset routine and then it enters the set-up routine for the SJA1000. As the part “configure control lines...” of Figure 11 is specific to the used microcontroller, it can not be discussed in general in this place. However, the example in this chapter shows, how to configure an 80C51 derivative.

For the following description of the initialization processing see Figure 12. It is assumed, that after power-on also the stand-alone CAN controller gets a reset pulse (LOW level) at the pin 17, enabling it to enter the reset mode. Before setting up registers of the SJA1000, the host controller should check by reading the reset mode/request flag, if the SJA1000 has reached the reset mode, because the registers, which get the configuration information, can be written only during reset mode.

The host controller has to configure the following registers of the control segment of the SJA1000 in reset mode:

- Mode Register (in PeliCAN mode only), selecting the following modes of operation for this application
  - Acceptance Filter mode
  - Self Test mode
  - Listen Only mode
- Clock Divider Register, defining
  - if the BasicCAN or the PeliCAN mode is used
  - if the CLKOUT pin is enabled
  - if the CAN input comparator is bypassed
  - if the TX1 output is used as a dedicated receive interrupt output
- Acceptance Code and Acceptance Mask Registers
  - defining the acceptance code for messages to be received
  - defining the acceptance mask for relevant bits of the message to be compared with corresponding bits of the acceptance code

- Bus Timing Registers, see also [6]
  - defining the bit-rate on the bus
  - defining the sample point in a bit period (bit sample point)
  - defining the number of samples taken in a bit period
- Output Control Register
  - defining the used output mode of the CAN bus output pins TX0 and TX1
    Normal Output Mode, Clock Output Mode, Bi-Phase Output Mode or Test Output Mode
  - defining the output pin configuration for TX0 and TX1
    Float, Pull-down, Pull-up or Push/Pull and polarity

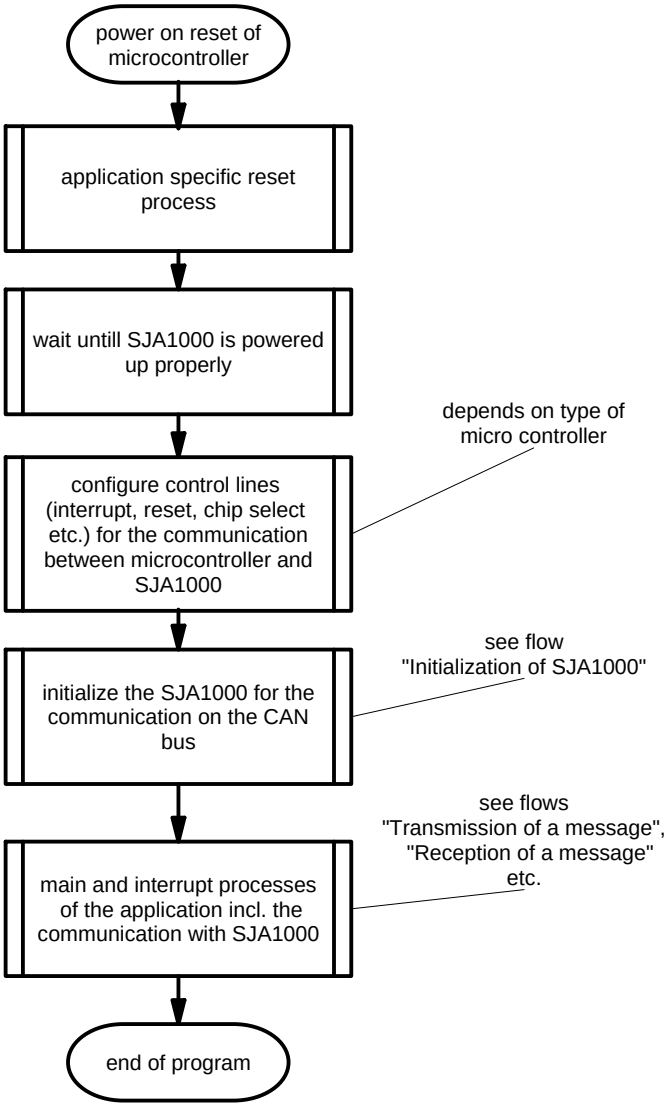

Figure 11: General program flow

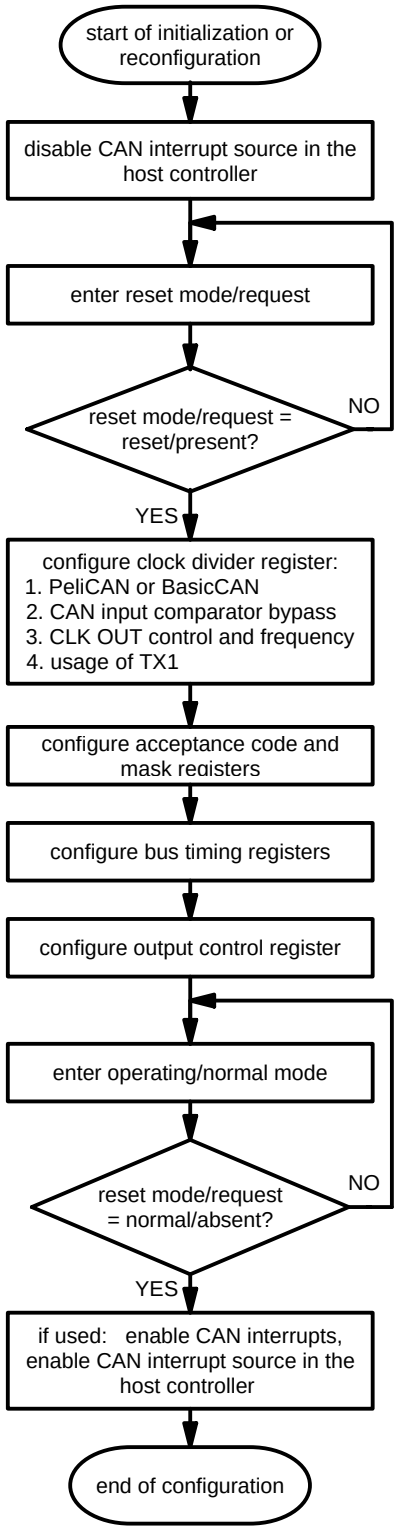

Figure 12: Flow Diagram “Initialization of SJA1000”

After having transferred this information to the control segment of the SJA1000, it is switched into operation mode by clearing the reset mode/request flag. It has to be checked, if the flag is really cleared and the operation mode is entered before going on further. This is done by reading the flag in a loop.

The reset mode/request flag cannot be cleared as long as a hardware reset still is pending (LOW-level at pin 17), because this will force the reset mode/request flag to “reset/present” (see the data sheet for further information [1]). Thus this loop is used to continuously trying to clear the flag **and** checking if the reset mode was left successfully.

After having entered the operation mode, the interrupts from the CAN controller may be enabled, if appropriate.

Example: Configuration and Initialization of SJA1000

This example is based on the application example given in Figure 3 on page 11. In the following programming examples a micro controller S87C654 is assumed as host controller. It is clocked by the clock output from the SJA1000. During power-on a reset circuit delivers the hardware reset for both the micro controller and the CAN controller. The Clock Divider Register of the SJA1000 is cleared during reset [1]. Thus the CAN controller comes up in BasicCAN mode with the clock output enabled, being able to deliver the clock for the S87C654 as soon as the crystal oscillator is running. The frequency of this clock is $f_{CLK}/2$ as pin 11 is connected to support controllers of the 80C51-family. Upon receiving the clock the micro controller starts its own reset process as shown in Figure 11.

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., “InterruptEnReg” points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language “C” is used for programming.

In this example it is assumed, that the CAN controller has to be initialized for being used in PeliCAN mode. It should be easy to derive the corresponding initialization for the BasicCAN mode.

The first step must be to set up a communication link (chip select, interrupts, etc.) between the host controller and the SJA1000 (“configure Control lines...” in Figure 11).

```
/* define interrupt priority & control (level-activated, see chapter 4.2.5)  */
 PX0 = PRIORITY_HIGH;  /* CAN HAS A HIGH PRIORITY INTERRUPT                   */
 IT0 = INTLEVELACT;    /* set interrupt0 to level activated                   */

/* enable the communication interface of the SJA1000                         */
 CS  = ENABLE_N;       /* Enable the SJA1000 interface                        */

/*- end of the definition of the communication link -------------------------*/
```

The second step is to initialize all internal registers of the SJA1000. As some registers can be written to during reset mode only, this has to be checked before writing. After power-on the SJA1000 is set into reset mode, but in a loop it can be checked, if the reset mode has been set.

```
/* disable interrupts, if used (not necessary after power-on)                */
 EA       = DISABLE;   /* disable all interrupts                            */
 SJAIntEn = DISABLE;   /* disable external interrupt from SJA1000           */

/* set reset mode/request (Note: after power-on SJA1000 is in BasicCAN mode)
   leave loop after a time out and signal an error                           */
 while((ModeControlReg & RM_RR_Bit ) == ClrByte)
 {
   /* other bits than the reset mode/request bit are unchanged               */
   ModeControlReg = ModeControlReg | RM_RR_Bit ;
 }
/* set the Clock Divider Register according to the given hardware of Figure 3
     select PeliCAN mode
     bypass CAN input comparator as external transceiver is used
     select the clock for the controller S87C654                             */
 ClockDivideReg = CANMode_Bit  | CBP_Bit  | DivBy2;
```

```
/* disable CAN interrupts, if required (always necessary after power-on)
   (write to SJA1000 Interrupt Enable / Control Register)                  */
InterruptEnReg = ClrIntEnSJA;

/* define acceptance code and mask                                         */
 AcceptCode0Reg = ClrByte;
 AcceptCode1Reg = ClrByte;
 AcceptCode2Reg = ClrByte;
 AcceptCode3Reg = ClrByte;
 AcceptMask0Reg = DontCare;   /* every identifier is accepted              */
 AcceptMask1Reg = DontCare;   /* every identifier is accepted              */
 AcceptMask2Reg = DontCare;   /* every identifier is accepted              */
 AcceptMask3Reg = DontCare;   /* every identifier is accepted              */

/* configure bus timing                                                    */
/* bit-rate = 1 Mbit/s @ 24 MHz, the bus is sampled once                   */
 BusTiming0Reg = SJW_MB_24 | Presc_MB_24;
 BusTiming1Reg = TSEG2_MB_24 | TSEG1_MB_24;

/* configure CAN outputs: float on TX1, Push/Pull on TX0,
                          normal output mode                             */
 OutControlReg = Tx1Float | Tx0PshPull | NormalMode;

/* leave the reset mode/request i.e. switch to operating mode,
   the interrupts of the S87C654 are enabled
   but not the CAN interrupts of the SJA1000, which can be done separately
   for the different tasks in a system                                    */

/* clear Reset Mode bit, select dual Acceptance Filter Mode,
   switch off Self Test Mode and Listen Only Mode,
   clear Sleep Mode (wake up)                                             */
 do                                 /* wait until RM_RR_Bit  is cleared    */
/* break loop after a time out and signal an error                        */
 {
   ModeControlReg  = ClrByte;
 } while((ModeControlReg & RM_RR_Bit ) != ClrByte);

 SJAIntEn = ENABLE;   /* enable external interrupt from SJA1000            */
 EA       = ENABLE;   /* enable all interrupts                             */

/*----- end of Initialization Example of the SJA1000 ----------------------*/
```

4.2.2 Transmission

A transmission of a message is done autonomously by the CAN controller SJA1000 according to the CAN protocol specification [8]. The host controller has to transfer the message to be transmitted into the Transmit Buffer of the SJA1000 and set the flag “Transmit Request” in the command register. The transmission process can be controlled either by an interrupt request from the SJA1000 or by polling status flags in the control segment of the SJA1000.

Interrupt Controlled Transmission

According to the main processing of the controller as given in Figure 13, the transmit interrupt of the CAN controller and the external interrupt used by the host controller for the communication with the SJA1000 are enabled prior to the start of a transmission, which is controlled by interrupt. The interrupt enable flags are located in the Control Register for the BasicCAN mode and in the Interrupt Enable Register for the PeliCAN mode (see Table 2 and [1]).

As long as the SJA1000 is transmitting a message, the Transmit Buffer is locked for writing. Thus the host controller has to check the “Transmit Buffer Status” flag (TBS) of the Status Register (see [1]), if a new message can be placed into the Transmit Buffer.

- The Transmit Buffer is locked:

  The host controller stores the new message temporarily in its own memory and sets a flag, indicating that a message is waiting for being transmitted. It is up to the software designer how to handle this temporary storage, which may be designed to store several messages to be transmitted. The start of a transmission of the message will then be handled during the interrupt service routine, which is initiated at the end of the current running transmission.

  Upon reception of an interrupt from the CAN controller (see the interrupt processing of Figure 13), the host controller checks the type of interrupt. If it was a Transmit Interrupt, it checks, whether further messages have to be transmitted or not. A waiting message is copied from the temporary store into the Transmit Buffer and the flag indicating further messages to be transmitted is cleared. The flag “Transmission Request” (TR) of the Command Register (see [1]) is set, which will cause the SJA1000 to start the transmission.

- The Transmit Buffer is released:

  The host controller writes the new message into the Transmit Buffer and sets the flag “Transmission Request” (TR) of the Command Register (see [1]), which will cause the SJA1000 to start the transmission. At the end of a successful transmission, a Transmit Interrupt is generated by the CAN controller.

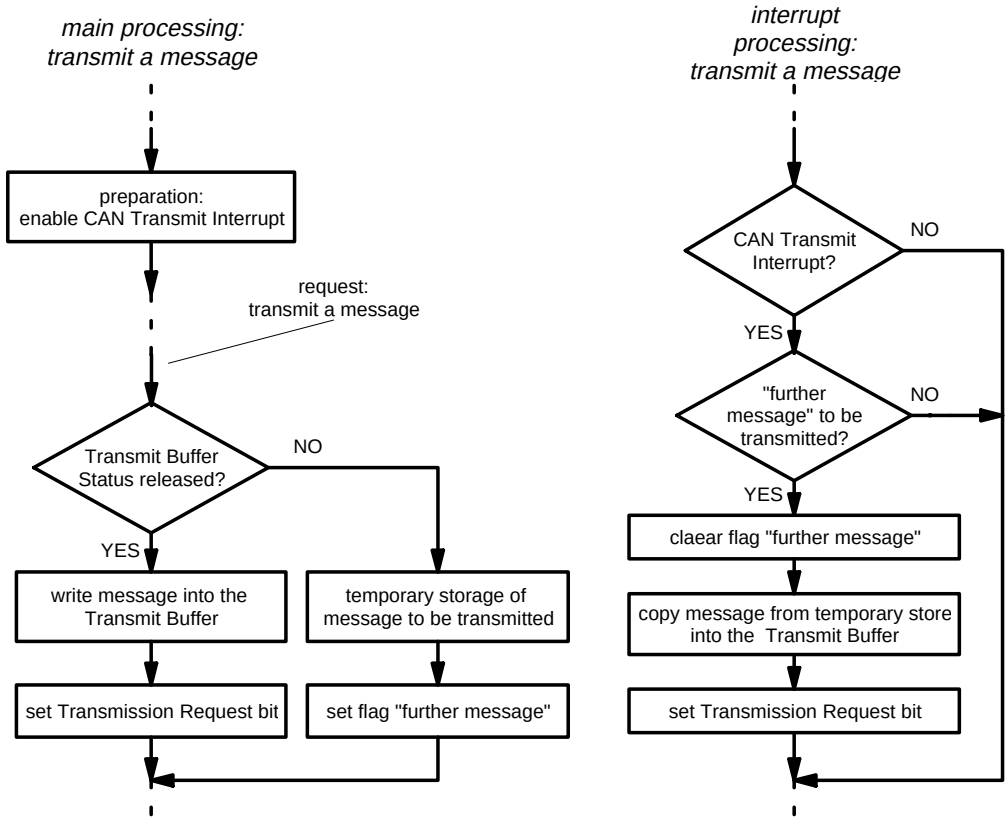

Figure 13: Flow Diagram “Transmission of a message” (interrupt controlled)

Polling Controlled Transmission

The flow is shown in Figure 14. The transmission interrupt of the CAN controller is disabled for this type of transmission control.

As long as the SJA1000 is transmitting a message, the Transmit Buffer is locked for writing. Thus the host controller has to check the “Transmit Buffer Status” flag (TBS) of the Status Register (see [1]), if a new message can be placed into the Transmit Buffer.

- The Transmit Buffer is locked:

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

- The Transmit Buffer is released:

  The host controller writes the new message into the Transmit Buffer and sets the flag “Transmission Request” (TR) of the Command Register (see [1]), which will cause the SJA1000 to start the transmission.

Example for the PeliCAN mode:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., “InterruptEnReg” points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language “C” is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

```
.
.
/* wait until the Transmit Buffer is released                              */
 do
 {
   /* start a polling timer and run some tasks while waiting
      break the loop and signal an error if time too long                  */
 } while((StatusReg & TBS_Bit  ) != TBS_Bit  );

/* Transmit Buffer is released, a message may be written into the buffer   */
/* in this example a Standard Frame message shall be transmitted           */
 TxFrameInfo = 0x08;      /* SFF (data), DLC=8                             */
 TxBuffer1   = 0xA5;      /* ID1   = A5, (1010 0101)                       */
 TxBuffer2   = 0x20;      /* ID2   = 20, (0010 0000)                       */
 TxBuffer3   = 0x51;      /* data1 = 51                                    */
 .
 .
 TxBuffer10  = 0x58;      /* data8 = 58                                    */

/* Start the transmission                                                  */
 CommandReg = TR_Bit  ;      /* Set Transmission Request bit                */
.
.
```

The TS and RS flags in the Status Register can be used for detecting, that the CAN controller has reached the idle-state. The TBS- and TCS-flags can be checked for a successful transmission.

Example for the BasicCAN mode:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., “InterruptEnReg” points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language “C” is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

```
.
/* wait until the Transmit Buffer is released                          */
 do
 {
   /* start a polling timer and run some tasks while waiting
      break the loop and signal an error if time too long              */
 } while((StatusReg & TBS_Bit  ) != TBS_Bit  );

/* Transmit Buffer is released, a message may be written into the buffer  */
/* only Standard Frame messages are possible in BasicCAN mode             */
  TxBuffer1   = 0xA5;      /* ID1 = A5, (1010 0101)                      */
  TxBuffer2   = 0x28;      /* ID2 = 28, (0010 1000) (DLC=8)              */
  TxBuffer3   = 0x51;      /* data1 = 51                                 */
.
  TxBuffer10  = 0x58;      /* data8 = 58                                 */

/* Start the transmission                                               */
 CommandReg = TR_Bit  ;      /* Set Transmission Request bit              */
.
```

The TBS- and TCS-flags can be checked for a successful transmission.

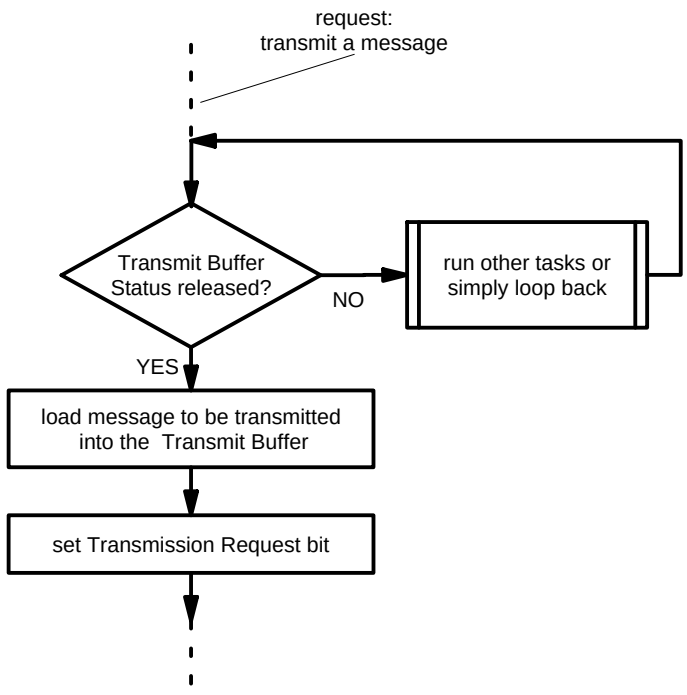

Figure 14: Flow Diagram “Transmission of a message” (polling controlled)

4.2.3 Abort Transmission

The transmission of a message, which was requested, may be aborted using the “Abort Transmission” command by setting the corresponding bit in the Command Register [1]. This feature may be used e.g. for transmitting an urgent message prior to the message, which has been written into the transmit buffer previously, but which was not transmitted successfully until now.

Figure 15 shows a flow using the transmit interrupt. The flow illustrates the situation, where a message has to be aborted in order to transmit a message with a higher priority. Other reasons for aborting a message may require a different interrupt flow.

A corresponding flow can be derived for the polling controlled transmission handling.

In case a message is still waiting for being served due to different reasons, the Transmit Buffer is locked (see the main flow part in Figure 15). If a transmission of an urgent message is requested, the Abort Transmission bit is set in the Command Register. When the message waiting to be served has either been transmitted successfully or aborted, the Transmit Buffer is released and a Transmit Interrupt is generated. During the interrupt flow the Transmission Complete flag of the Status Register has to be checked, if the previous transmission has been successful or not. The status “incomplete” indicates, that the transmission was aborted. In this case the host controller can run through a special routine dealing with a strategy for aborted transmissions, e.g., repeat the transmission of the aborted message after having checked, if it is still valid.

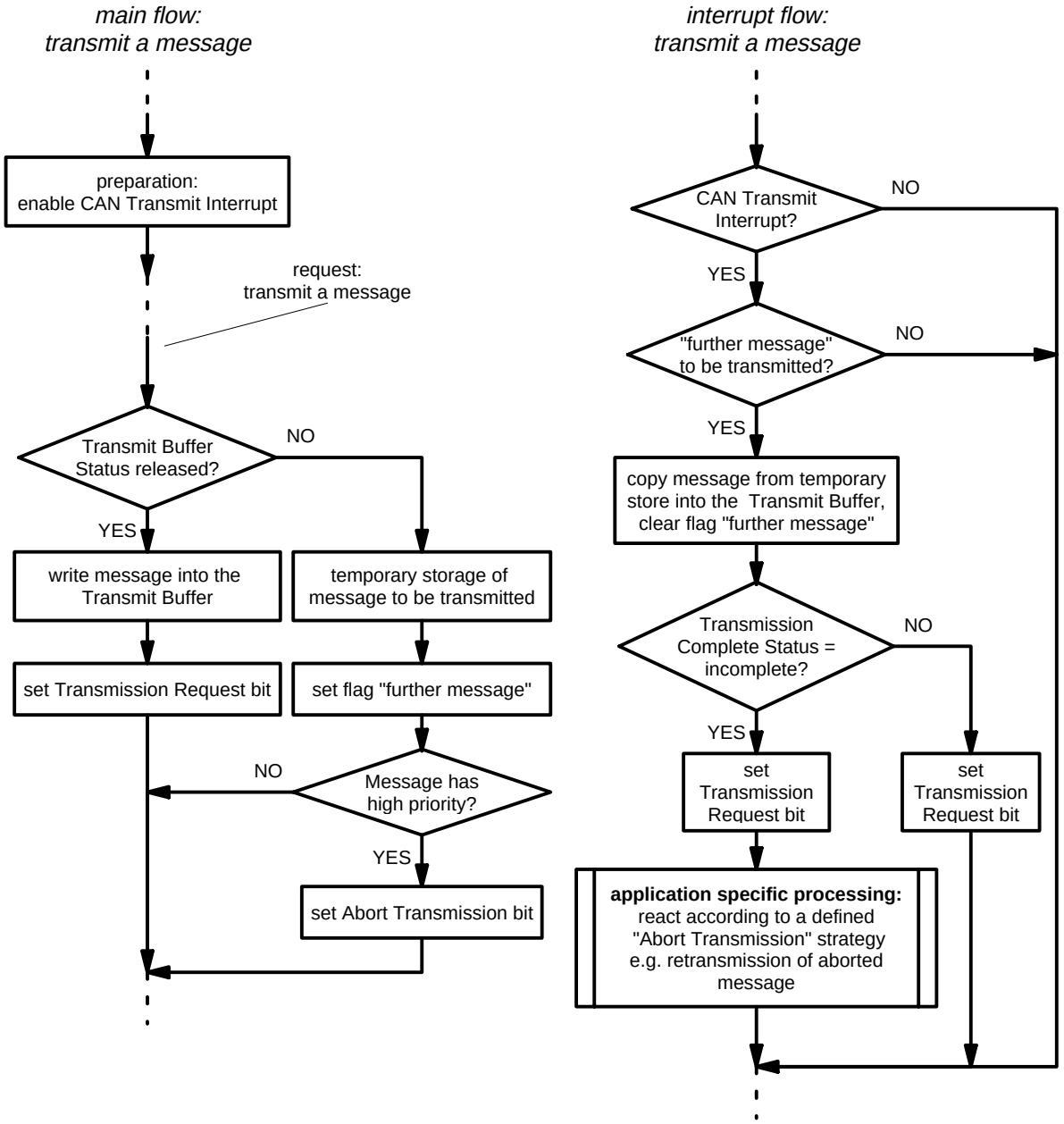

Figure 15: Flow Diagram “Abort Transmission of a message” (interrupt controlled)

4.2.4 Reception

The reception of messages is done autonomously by the CAN controller SJA1000 according to the CAN protocol specification [8]. Received messages are placed into the Receive Buffer (see chapter 4.1.1 and 5.1). A message, ready to be transferred to the host controller, is signalled by the Receive Buffer Status flag “RBS” (see [1]) of the Status Register and by a Receive Interrupt flag “RI” (see [1]), if enabled. The host controller has to

transfer the message to its local message memory, release the Receive Buffer and react on the content of the message. The transfer process can be controlled either by an interrupt request from the SJA1000 or by polling status flags in the control segment of the SJA1000.

Polling Controlled Reception

The flow is shown in Figure 16. The Receive Interrupt of the CAN controller is disabled for this type of reception control.

The host controller reads the Status Register of the SJA1000 on a regular basis, checking if the Receive Buffer Status flag (RBS) indicates, that at least one message has been received. For the definition of the flags located in the registers of the control segment see [1].

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

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

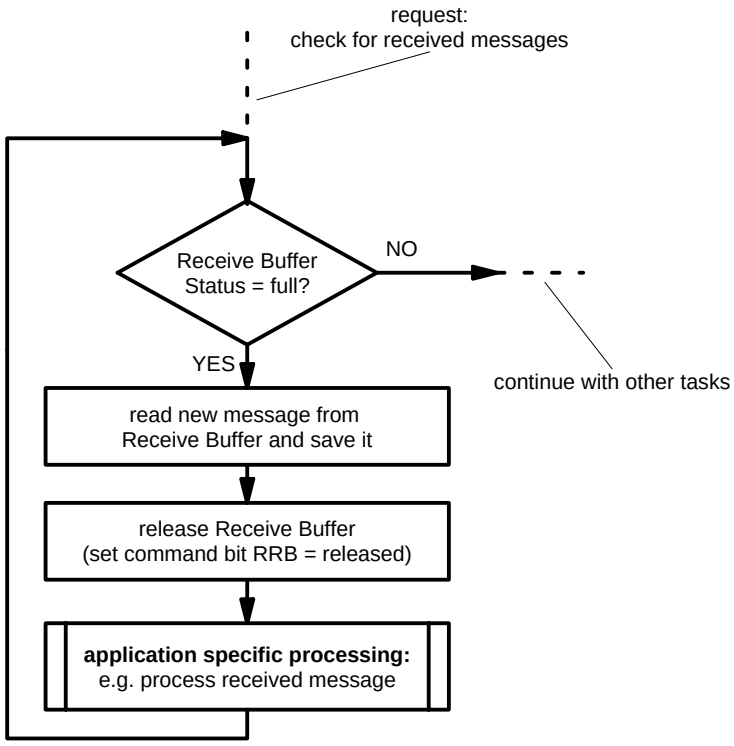

Figure 16: Flow Diagram “Reception of a message” (polling controlled)

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

  The host controller gets the first message from the SJA1000 and sends a Release Receive Buffer command afterwards by setting the corresponding flag in the Command Register. The host controller can process each received message before checking for further messages, as indicated in Figure 16. But it is also possible to check at once for further messages by polling the Receive Buffer Status bit again and process the received messages all together later. In this case the local message memory of the host controller has to be large enough to store more than one message before they are processed. After having transferred and processed one or all messages, the host controller can continue with other tasks.

Interrupt Controlled Reception

According to the main processing of the controller as given in Figure 17, the receive interrupt of the CAN controller and the external interrupt used by the host controller for the communication with the SJA1000 are enabled prior to an interrupt controlled reception of messages. The interrupt enable flags are located in the Control Register (for the BasicCAN mode) or in the Interrupt Enable Register (for the PeliCAN mode) - see Table 2 and [1].

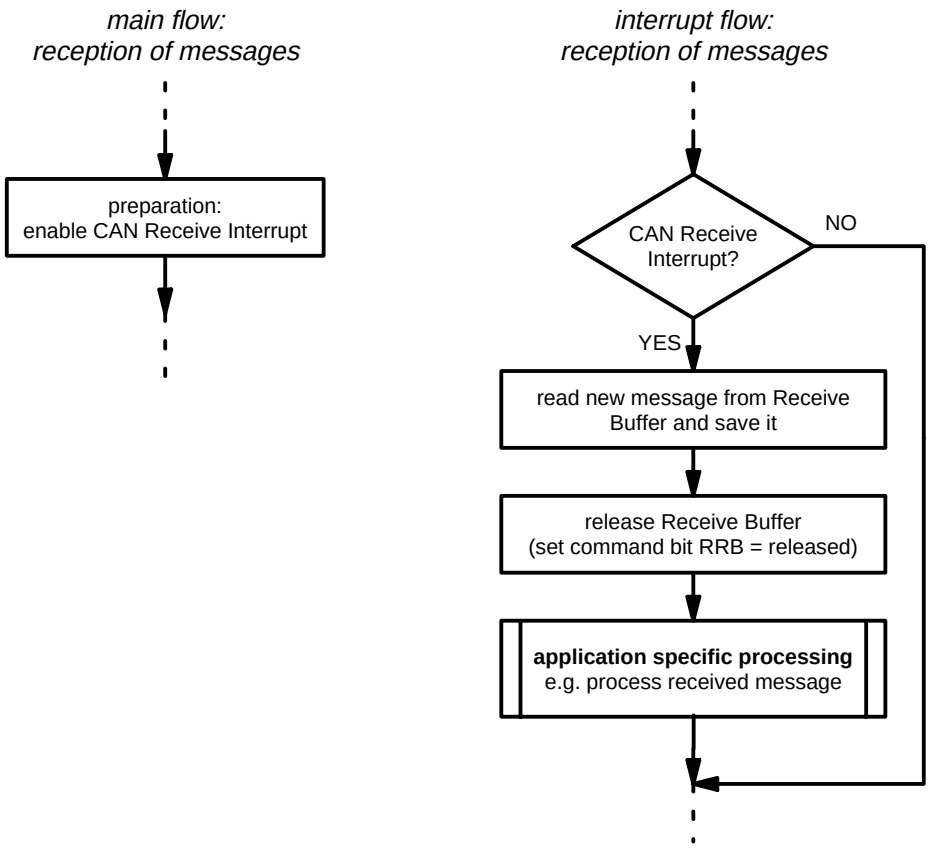

Figure 17: Flow Diagram “Reception of a message” (interrupt controlled)

If the SJA1000 has received a message, which has passed the acceptance filter and has been placed into the Receive FIFO, a receive interrupt is generated. Thus the host controller can react immediately, transferring the received message into its message memory and send a Release Receive Buffer command afterwards by setting the corresponding flag “RRB” (see [1]) in the Command Register. Further messages in the Receive FIFO will

generate a new receive interrupt, so it is not necessary to read all messages available in the Receive FIFO during one interrupt. Contrary to this solution the procedure for reading all available messages at once is used in Figure 18. After having released the Receive Buffer, the Receive Buffer Status (RBS) in the Status Register is checked for further messages and all available are read in a loop.

As given in Figure 17, the whole reception process may be done during the interrupt routine, without interaction with the main program. If feasible, even the reaction on messages can be done in the interrupt too.

Example:

Definitions for the different constants and variables, etc., are given in the Appendix. Variables may be interpreted different in BasicCAN and PeliCAN mode, e.g., “InterruptEnReg” points to the Control Register in BasicCAN mode but to the Interrupt Enable Register in PeliCAN mode. The language “C” is used for programming.

After having initialized the CAN controller according to the example given in chapter 4.2.1, normal communication can be started.

1. part of the main processing

```
.
.
/* enable the receive interrupt                                          */
 InterruptEnReg = RIE_Bit;
.
.
```

2. part of the interrupt 0 service routine

```
.
/* read the Interrupt Register content from SJA1000 and save temporarily
   all interrupt flags are cleared (in PeliCAN mode the Receive
   Interrupt (RI) is cleared first, when giving the Release Buffer command)
*/
CANInterrupt = InterruptReg;
.
.
/* check for the Receive Interrupt and read one or all received messages     */
 if (RI_VarBit == YES)                  /* Receive Interrupt detected       */
 {
    /* get the content of the Receive Buffer from SJA1000 and store the
       message into internal memory of the controller,
       it is possible at once to decode the FrameInfo and Data Length Code
       and adapt the fetch appropriately                                     */
    .
    .
    /* release the Receive Buffer, now the Receive Interrupt flag is cleared,
       further messages will generate a new interrupt                        */
    CommandReg = RRB_Bit;                /* Release Receive Buffer          */
 }
.
.
```

Data Overrun Handling

In case the Receive FIFO is full but another message is being received, a Data Overrun is signalled to the host controller by setting the Data Overrun Status in the Status Register and, if enabled, a Data Overrun Interrupt is generated by the SJA1000.

Running into a Data Overrun situation states, that the host controller is extremely overloaded, as it did not have enough time to fetch received messages from the Receive Buffer in time. A Data Overrun signals, that data are lost, possibly causing inconsistencies in the system. Normally a system should be designed in such a way, that the received messages are transferred and processed fast enough to avoid a Data Overrun condition. An exception handler dealing with an application specific processing should be implemented in the host controller, if Data Overrun situations cannot be avoided.

Figure 18 illustrates the program flow, in case a Data Overrun Interrupt has to be handled.

After having transferred the message, which caused the receive interrupt, and released the Receive Buffer, it is checked, if further messages are available in the Receive FIFO by reading the Receive Buffer Status. Thus all messages can be fetched from the Receive FIFO before going on further. Of course reading a message and perhaps processing it already during the interrupt, should be done faster, than it takes the SJA1000 to receive a new message. Otherwise it could happen, that the host controller stays in the interrupt forever reading messages.

Detecting a Data Overrun starts an exception handling according to a “Data Overrun” strategy. This strategy can decide between two situations:

– A Data Overrun occurred together with a Receive Interrupt:
  Messages may have been lost.

– A Data Overrun occurred, but no Receive Interrupt was detected:
  Messages may have been lost. The Receive Interrupt may have been disabled.

It is up to the system designer how the host controller should react on these situations.

An equivalent handling can also be done during a polling controlled reception of messages.

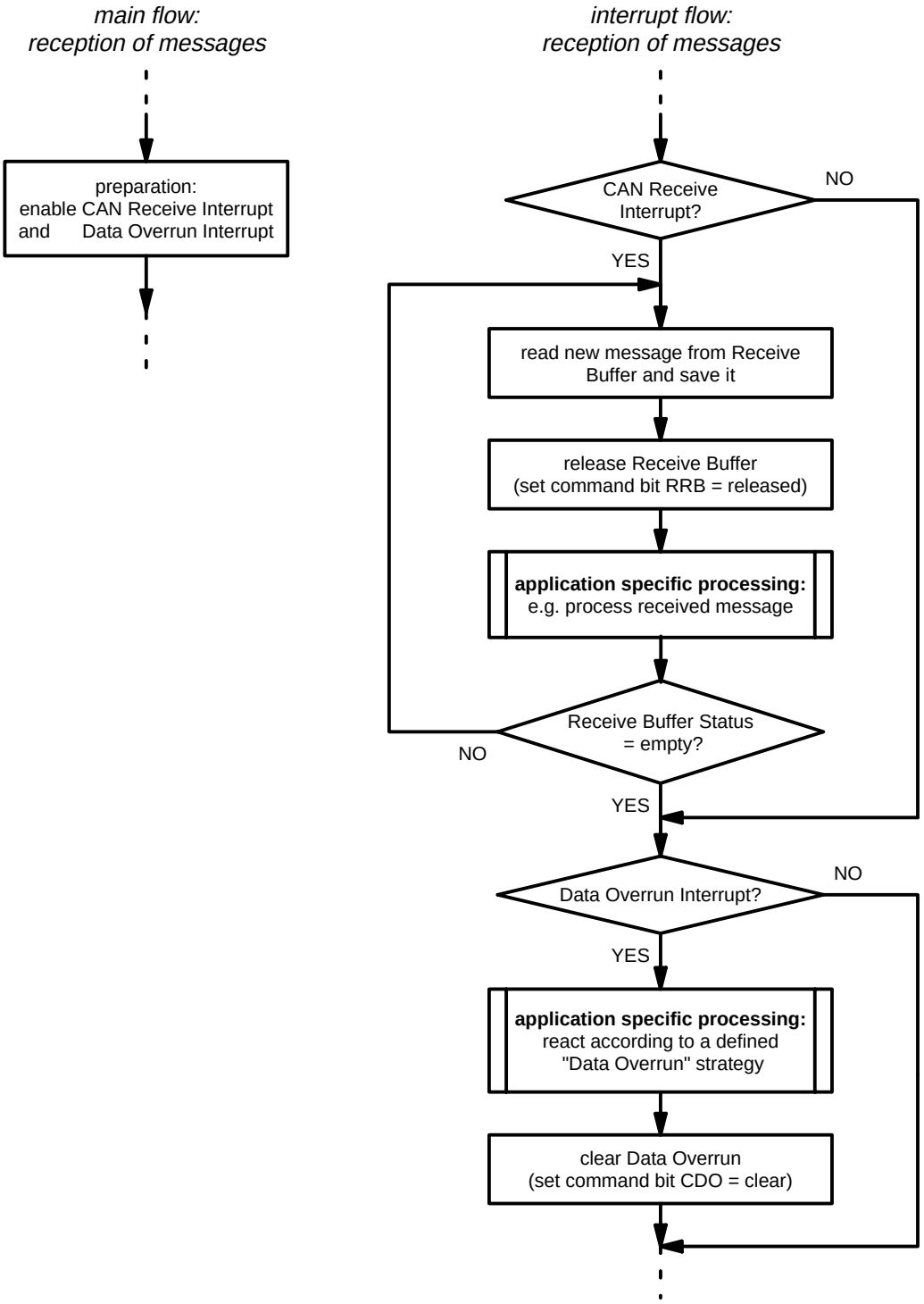

Figure 18: Flow Diagram “Data Overrun and reception of messages” (interrupt controlled)

4.2.5 Interrupts

In PeliCAN mode the SJA1000 has 8 different interrupts (in BasicCAN mode there are only 5), which may be used for causing immediate actions by the host controller on certain states of the CAN controller.

In case a CAN interrupt is present, the SJA1000 sets the interrupt output (pin 16) to LOW-level. The output stays at LOW-level, until the host controller reacts on the interrupt by reading the Interrupt Register of the SJA1000; – in case of a receive interrupt in PeliCAN mode upon releasing the Receive Buffer. After this reaction from the host controller the SJA1000 switches the interrupt output back to HIGH-level. In case further interrupts did arrive in the meantime, or further messages are available in the Receive FIFO, the SJA1000 at once sets the interrupt output to LOW-level again. Thus the output may stay HIGH for a very short time only. Both the handshaking during serving the interrupt request and the possible short HIGH-level pulse during two interrupts require, that the interrupt of the host controller must be level-activated.

The flow in Figure 19 gives an overview of all possible interrupts and references to more detailed descriptions in this Application Note. The order, in which the different interrupts are handled in this flow, is one possible solution only. It depends very much on the system and the requested behaviour of it, in which order the interrupts have to be served. This has to be decided by the designer of the overall system.

The reactions on the Transmission, Receive and Data Overrun Interrupts are already discussed in the previous paragraphs.

The flows after a Wake Up Interrupt, Arbitration Lost Interrupt and three different error interrupts are given in more detail in Figure 20, Figure 21 and Figure 22. All error interrupts may be used for implementing a versatile error strategy in the system. This strategy should deal with system optimization in the development phase and automatic system optimization and system maintenance in the operational phase. Also the Arbitration Lost Interrupt may be used for system optimization and maintenance. See also the following chapters and the data sheet [1] for more details on the different error signals, arbitration lost handling and related information.

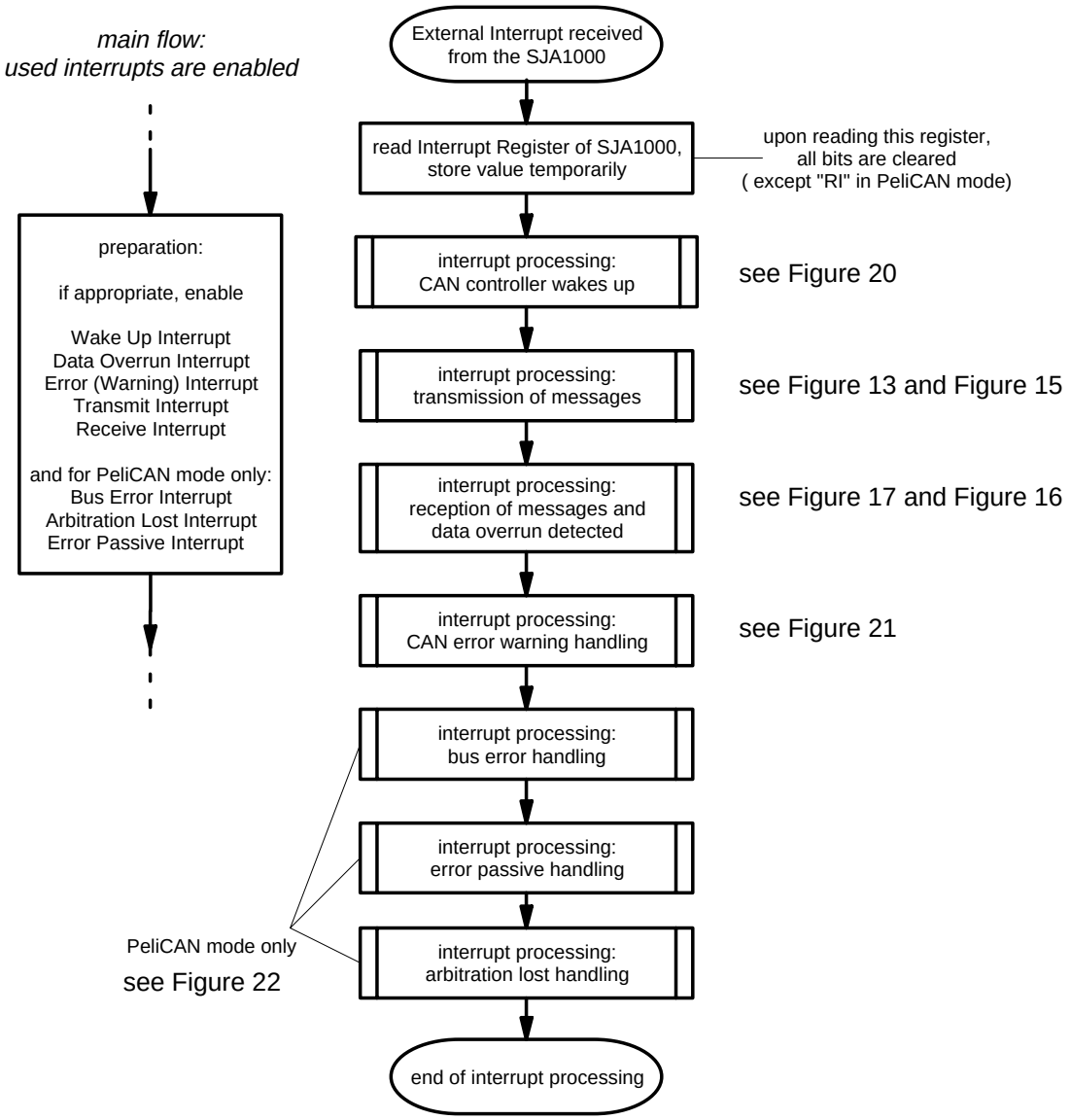

Figure 19: General interrupt flow

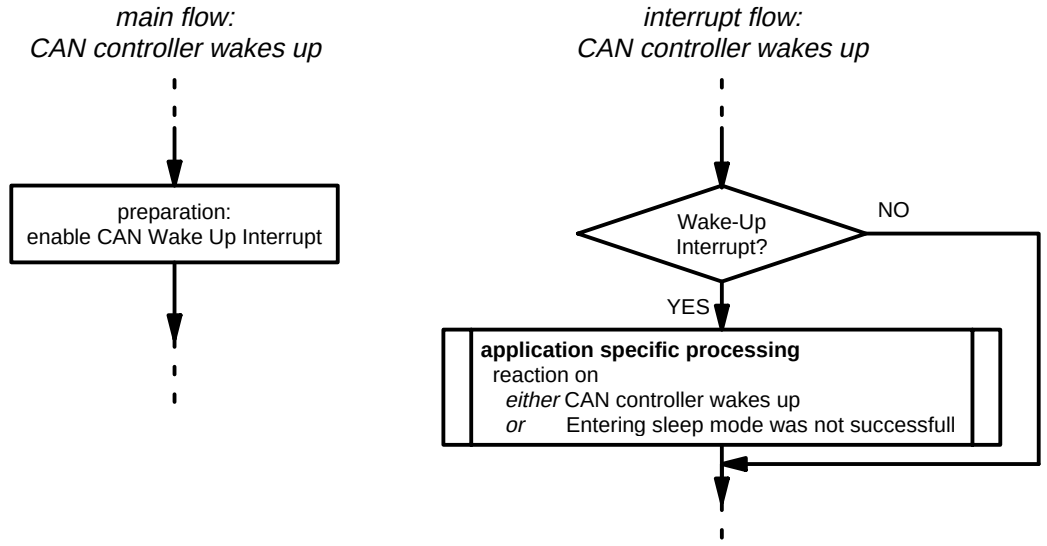

Figure 20: Flow Diagram "CAN controller wakes up"

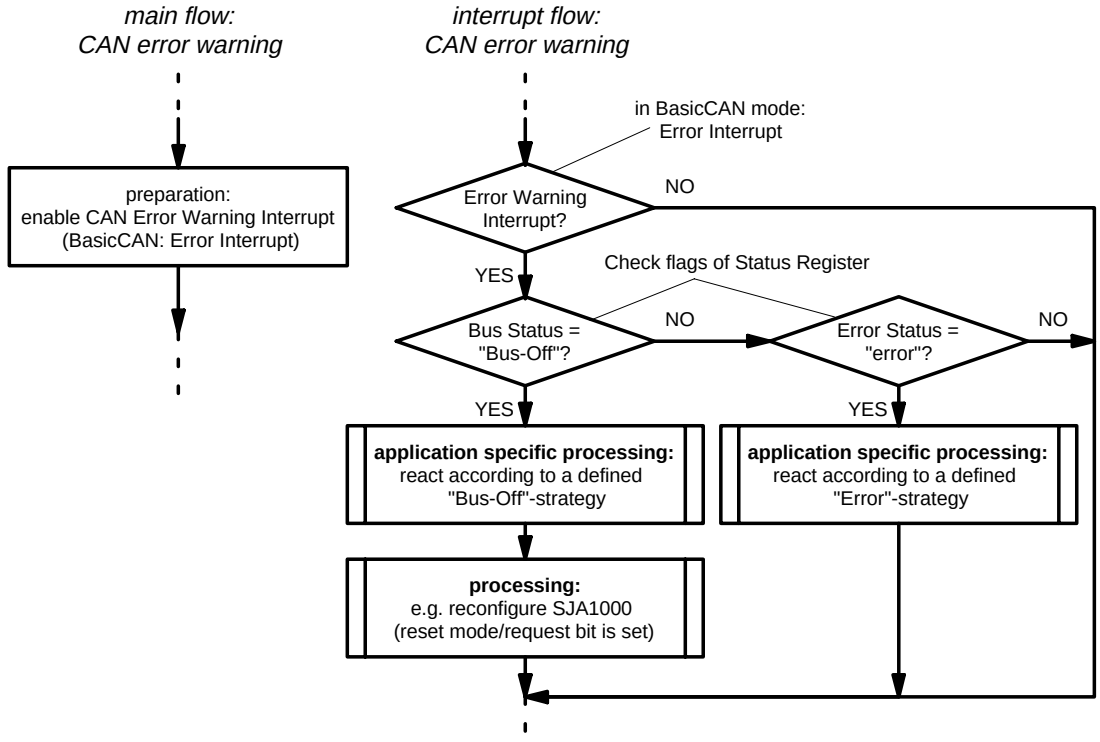

Figure 21: Flow Diagram "Error Warning Interrupt"

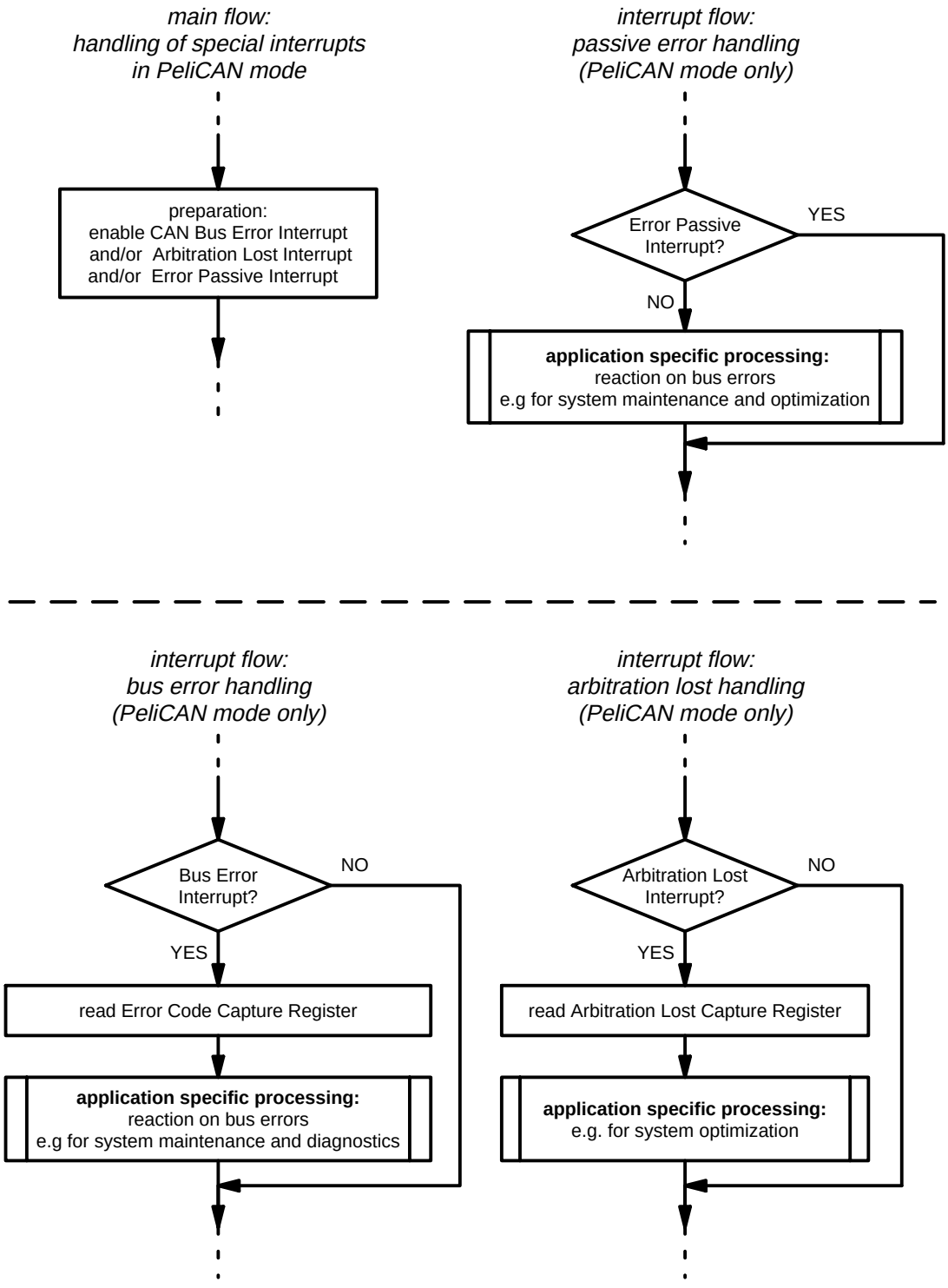

Figure 22: Flow Diagram “Processing of special PeliCAN interrupts”

5. PELICAN MODE FUNCTIONS

5.1 Receive FIFO / Message Counter / Direct RAM Access

The SJA1000 registers and message buffers appear to the host controller as peripheral registers which can be addressed via the multiplexed address/data bus. Depending on the selected mode (Operating or Reset) different registers are accessible. The address range for normal operation is: Address 0 .. 31. It contains registers for initialization, status and control purposes. Furthermore the CAN message buffers are allocated between address 16 and 28. With a host controller write access the user can address the CAN controller's Transmit Buffer and with a read access the Receive Buffer contents is read.

Additionally to the range described above the whole Receive FIFO is mapped between CAN address 32 and 95, see also Figure 23. Furthermore the Transmit Buffer of the SJA1000 which is also part of the internal 80 byte RAM is available between CAN address 96 and 108.

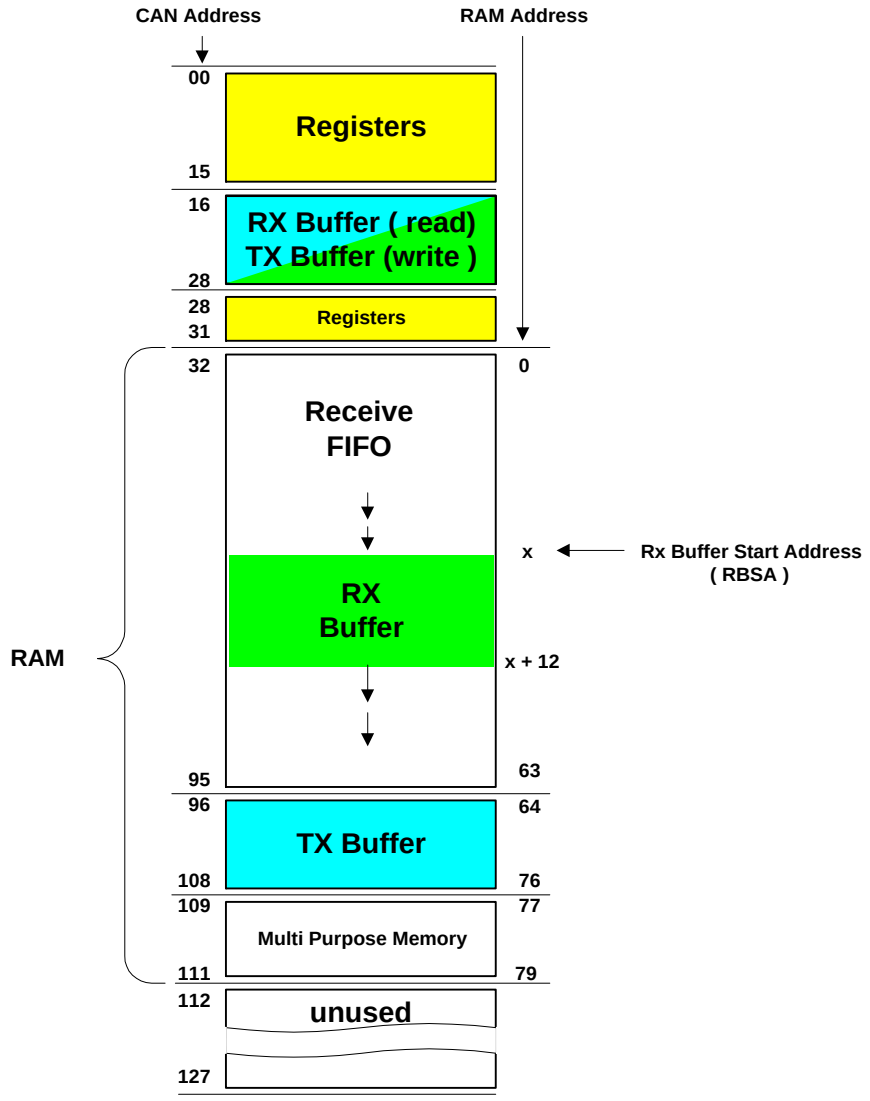

Figure 23: Register and RAM Address Allocations

With the described direct RAM access it is possible to read the Transmit Buffer and the complete Receive FIFO.

In PeliCAN mode the Receive FIFO is able to store up to $n = 21$ messages. With the help of the following equation it is possible to calculate the maximum number of messages:

$$n = \frac{64}{3 + data\_length\_code}$$

The Receive Buffer is defined as a 13 byte window always containing the current receive message of the Receive FIFO. As shown in Figure 24 it could happen that parts or the complete following message is already available in the Receive Buffer window.

However, upon command ´Release Receive Buffer´ the next receive message in the Receive FIFO will become completely visible in the Receive Buffer window starting at CAN address 16.

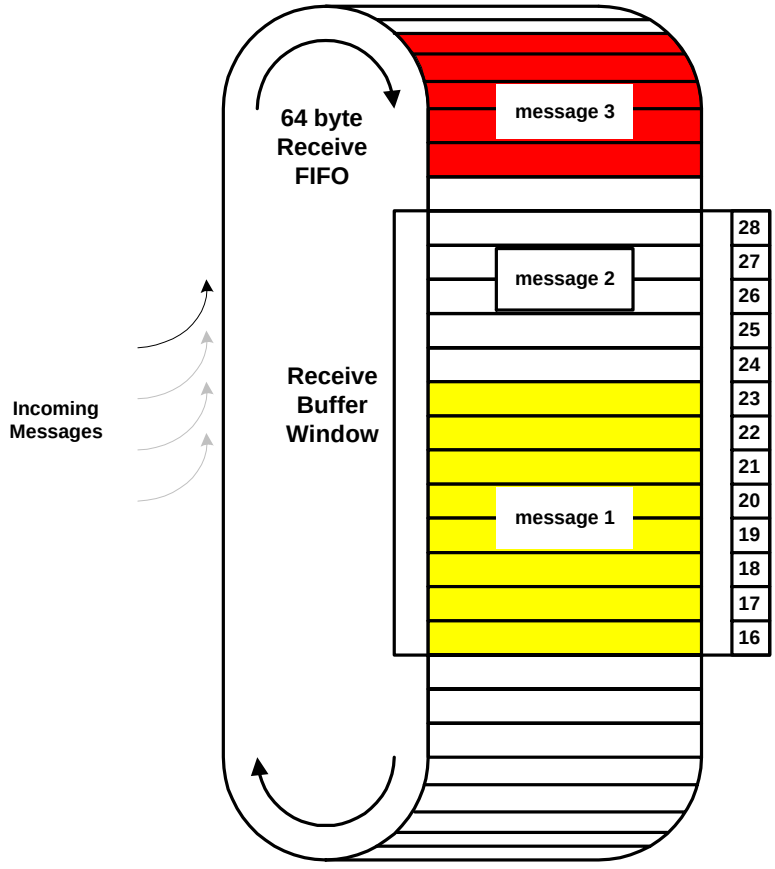

Figure 24: Receive FIFO

Mainly for analysis purposes the SJA1000 provides two additional registers supporting receive message handling:

- Rx Buffer Start Address Register (RBSA) allows identification of single CAN messages in the Receive FIFO range.
- RX Message Counter Register which contains the current number of stored messages in the Receive FIFO.

Figure 23 shows the relation between the physical RAM address and the CAN address.

5.2 Error Analysis Functions

Depending on the value of the error counters each CAN controller can operate in one of three possible error states: error active, error passive or bus-off. The CAN controller is error active if both error counters are between 0... 127 . In case of an error condition an active error flag (6 dominant bits) is generated. The SJA1000 is error passive if one of the error counters is between 128 and 255. A passive error flag (6 recessive bits) is generated upon detection of an error condition in this case. If the Transmit Error Counter is greater than 255 the bus-off status is reached. In this state the reset request bit is set automatically and the SJA1000 can not influence the bus. As shown in Figure 25 bus-off can only be terminated with the host controller command ´Reset Request = 0´. This will start the bus-off recovery time where the Transmit Error Counter is used to count 128 occurrences of a bus free signal. At the end of this time both error counters are 0 and the device is error active again.

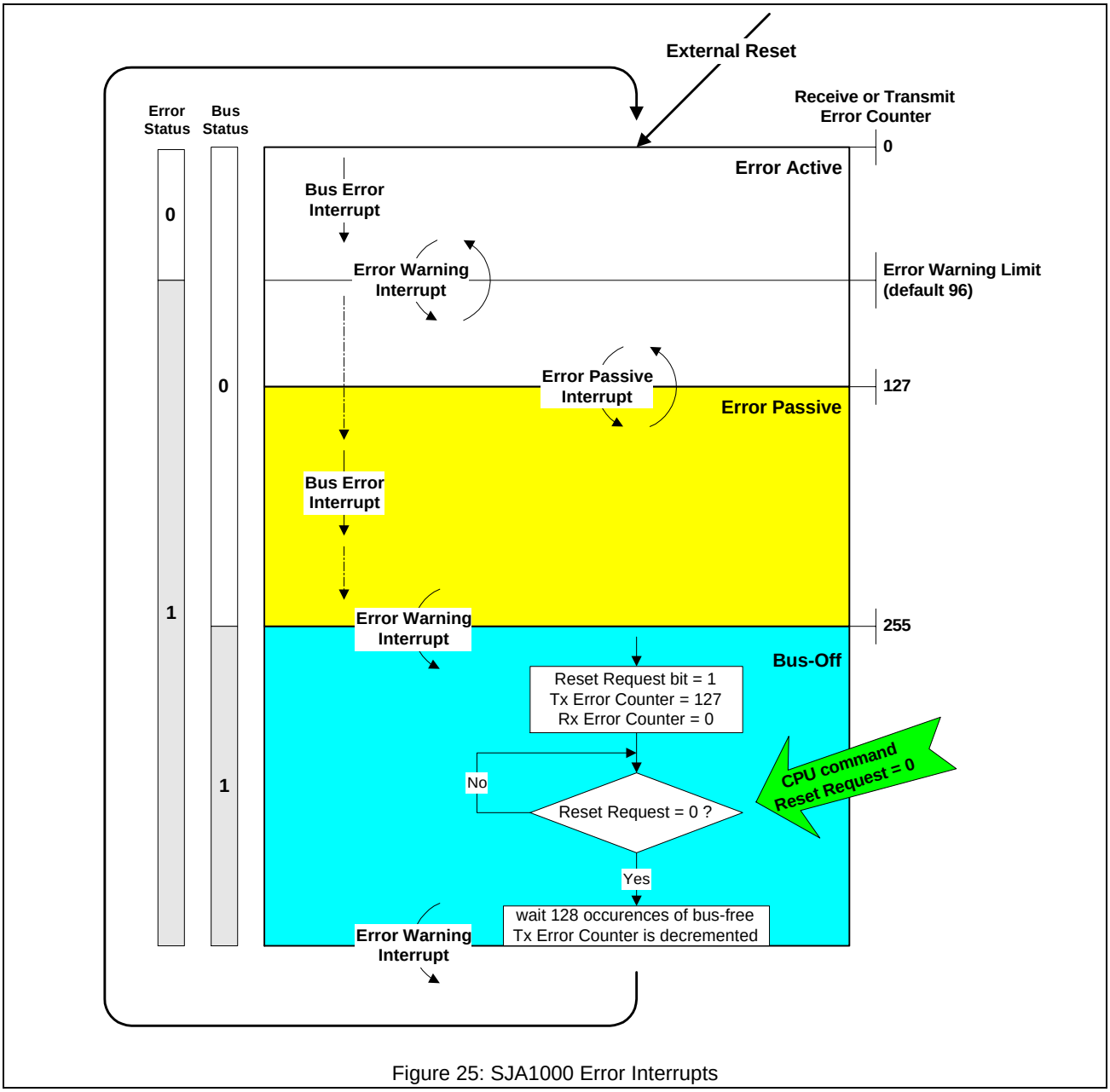

Figure 25: SJA1000 Error Interrupts

Furthermore the figure shows the value for both Error and Bus status at different error states.

5.2.1 Error Counters

As described above the error states of the CAN controller are directly related to the values of the Transmit and Receive Error Counters.
To allow a deep look inside into the error confinement and to support an enhanced error analysis with the SJA1000 the CAN controller provides readable error counters. Additionally, in Reset Mode a write access to both error counters is allowed.

5.2.2 Error Interrupts

Three interrupt sources have been implemented to signal error conditions to the host controller, see Figure 25. Each interrupt can be enabled separately in the Interrupt Enable Register.

Bus Error Interrupt:

This interrupt is generated upon any error condition on the CAN bus.

Error Warning Interrupt:

The Error Warning Interrupt is generated if the error warning limit is passed. Furthermore it is generated if the CAN controller enters the bus-off state and upon re-entry into error active state. The error warning limit of the SJA1000 is programmable in reset mode. The default value upon reset Is 96.

Error Passive Interrupt:

If the error status changes from error active to error passive or vice versa an error passive interrupt is signalled.

5.2.3 Error Code Capture

As described in the previous section the SJA1000 performs the full error confinement specified in the CAN2.0B specification [8]. As in every CAN controller the whole process of handling errors is executed fully automatically. However, to provide the user with additional details about a certain error condition the SJA1000 contains the Error Code Capture function. Whenever a CAN bus error occurs, the corresponding bus error interrupt is forced. At the same time, the current bit position is captured into the Error Code Capture Register. The captured data is fixed until the host controller has read it. From now on the capture mechanism is activated again. The register contents distinguishes four different types of errors: form,-stuff,- bit and other errors. As shown in Figure 26 the register additionally indicates whether the error occurred during reception or transmission of a message. Five

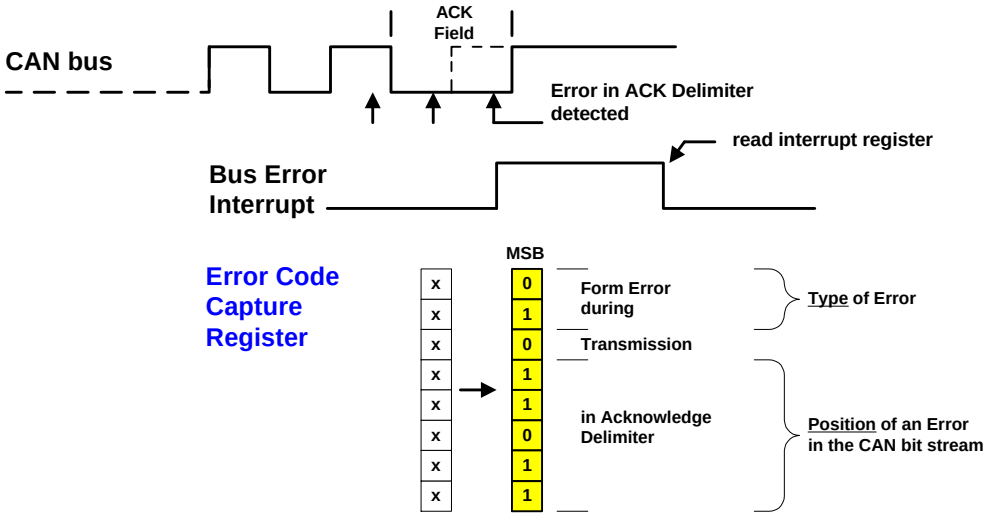

Figure 26: Example for the Error Code Capture Function

bits in this register indicate the erroneous bit position in the CAN frame, see also the following tables and the data sheet for more details.
As defined in the CAN specification, every single bit on the CAN bus can only have special types of errors. The next two tables show all possible errors during transmission and reception of CAN messages. The left part contains the position and the type of an error, captured by the Error Code Capture Register. The right part of each table is a translation into an upper level error description and can be derived directly from the register contents. With the help of these tables further information concerning error counter change and the erroneous state at the transmit and receive pins of the device can be derived. While using this table, e.g., in the error analysis software it is possible to analyze every single error situation in detail. The information about type and position of CAN errors can be used for error statistics and system maintenance or for corrective actions during system optimization.

Table 6: Possible errors during reception

Error Code Capture		RX Error Count	Description	
Position of an Error in the CAN bit stream	**Type of Error**			
Identifier SRR, IDE and RTR bit Reserved Bits Data Length Code Data Field CRC Sequence	Stuff	+ 1	more than 5 consecutive bits with same level received	- -
CRC Delimiter	Form Stuff	+ 1 + 1	Rx = dominant more than 5 consecutive bits with same level received	bit has to be recessive
Acknowledge Slot	Bit	+ 1	Tx = dominant but Rx = recessive	can't write dominant bit
Acknowledge Delimiter[1]	Form	+ 1	Rx = dominant or CRC error detected[1]	critical bus timing or bus length CRC sequence not correct
End of Frame	Form Other	+ 1 ± 0	Rx = dominant in first 6 bits Rx = dominant in last bit	- - reaction: overload flag will be sent, data duplication is possible if transmitter starts re-transmission
Intermission	Other	± 0	Rx = dominant	reaction: overload flag will be sent by receiver
Active Error Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Tolerate Dominant Bits	Other	+ 8	Rx = dominant in first bit upon error flag Rx = dominant for more than 7 bits upon error or overload flag	
Error Delimiter	Form Other	+ 1 ± 0	Rx = dominant within first 7 bits Rx = dominant in last bit of delimiter	- - overload flag will be sent
Overload Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit

[1] if the CRC is not o.k., then the error is processed in the Acknowledge Delimiter resulting in a ´Form Error´.

Table 7: Possible errors during transmission

Error Code Capture				
Position of an Error in the CAN bit stream	**Type of Error**	**TX Error Count**	**Description**	
Start Of Frame	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Identifier	Bit Stuff	+ 8 ± 0	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
SRR Bit	Bit Stuff	+ 8 ± 0	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
IDE and RTR Bit	Bit Stuff	+ 8 + 8	Tx = dominant but Rx = recessive Tx = recessive but Rx = dominant	can't write dominant bit - -
Reserved Bits, Data Length Code, Data Field, CRC Sequence,	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
CRC Delimiter	Form	+ 8	Rx = dominant	bit has to be recessive
Acknowledge Slot	Other Other	+ 8 ± 0	Rx = recessive (error active) Rx = recessive (error passive)	no acknowledge no acknowledge, node is probably alone on the bus
Acknowledge Delimiter	Form	+ 8	Rx = dominant	critical bus timing or bus length
End of Frame	Form Other	+ 8 + 8	Rx = dominant within first 6 bits Rx = dominant in last bit	- - frame has already been received by some nodes, re-transmission may result in data duplication in receivers
Intermission	Other	± 0	Rx = dominant	overload flag from ´old´ CAN controllers
Active Error Flag Overload Flag	Bit	+ 8	Tx = dominant but Rx = recessive	can't write dominant bit
Tolerate Dominant Bits	Form	+ 8	Rx = dominant for more than 7 bit times after active error flag or overload flag	- -
Error Delimiter	Form Other	+ 8 ± 0	Rx = dominant within first 7 bits Rx = dominant in last bit of delimiter	- - overload flag from ´old´ CAN controller
Passive Error Flag	Other	+ 8	Rx = dominant (error passive)	no acknowledge received, node is not alone on the bus

5.3 Arbitration Lost Capture

The SJA1000 is able to identify the exact CAN bit stream position where the arbitration has been lost. Immediately upon this an ´Arbitration Lost Interrupt´ is generated. Furthermore the bit number is captured in the Arbitration Lost Capture Register. As soon as the host controller has read the contents of this register, the capture function is activated for the next arbitration lost situation.

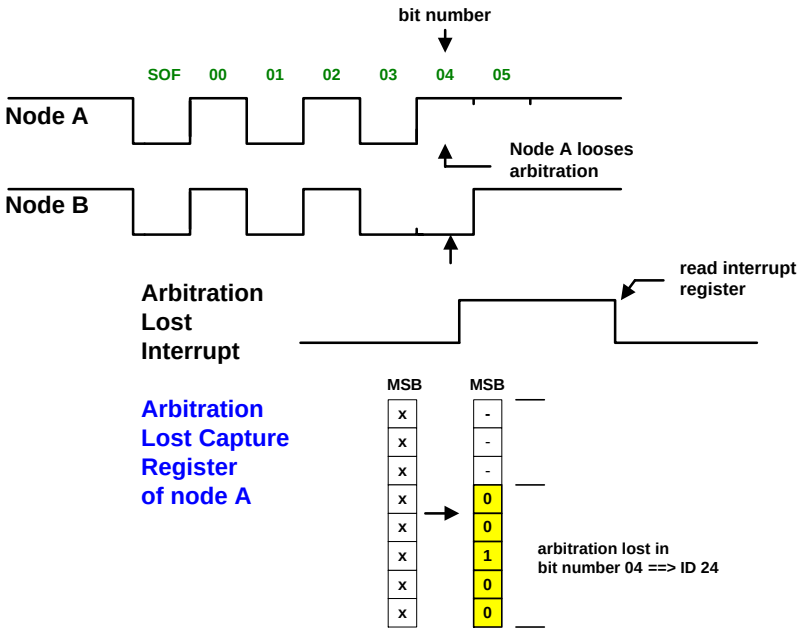

Figure 27: Arbitration Lost Capture Function

With the help of this feature the SJA1000 is able to monitor each CAN bus access. For diagnostics or during system configuration it is possible to identify every situation where the arbitration was not successful.

The next example shows how the arbitration lost function can be used.

First the Arbitration Lost Interrupt is enabled in the Interrupt Enable Register. Upon interrupt the contents of the Interrupt Register is saved. If the arbitration lost interrupt flag is set, the contents of the Arbitration Lost Capture Register is analyzed.

Example: **Arbitration Lost**

```
....
InterruptEnReg = ALIE_Bit;          /* Enable Arbitration Lost Interrupt       */
....

/* ----- Interrupt Service Routine ------------------------------------------ */
....
int_reg_copy = InterruptReg;        /* save interrupt register contents        */
....
if (int_reg_copy & ALIE_Bit)
   candat = ArbLostCapReg;          /* read arbitration lost capture register  */
...
...
```

5.4 Single Shot Transmission

In some applications the automatic re-transmission of CAN messages does not make sense: it could happen that a CAN node looses arbitration several times and the data have become obsolete.

In order to request a ´Single Shot Transmission´ previous CAN controllers have to perform the following steps:

1. Transmission Request

2. Wait for transmission status

3. Abort Transmission

The software necessary to process this can be minimized to a single command with the ´Single Shot Transmission´ option, which is initiated by setting the command bits CMR.0 and CMR.1 simultaneously.

In this case no status bit polling is needed and the host controller can concentrate on other tasks. The described `Single Shot Transmission´ function can be combined perfectly with the arbitration lost and the error code capture functions of the SJA1000.
In case of arbitration lost or if an error condition occurs the message is not re-transmitted by the SJA1000. As soon as the Transmit Status bit is set within the Status Register, the internal Transmission Request Bit is cleared automatically.
With the additional information from both capture registers it is under the control of the user whether a message is re-transmitted or not.

As described in chapter 5.7 the Single Shot Transmission can also be used together with the Self Test Mode.

5.5 Listen Only Mode

In Listen Only Mode the SJA1000 is not able to write dominant bits onto the CAN bus. Neither active error flags or overload flags are written nor a positive acknowledge is given upon successful reception.

Errors are treated like in error passive mode. The error analysis functions, e.g., error code capture and error interrupts are working as known from normal operating mode.

However, the status of the error counters is frozen.

Reception of messages is possible, transmission is not possible. Therefore, this mode can be used for automatic bit-rate detection, see also chapter 5.6, and other applications with monitor characteristics.

Note, before entering the Listen Only Mode the Reset Mode has to be entered.

Example: **Listen Only Mode**

```
....
ModeControlReg = RM_RR_Bit;       /* Enter Reset Mode       */
ClockDivideReg = CANMode_Bit;     /* PeliCAN Mode           */
....
ModeControlReg = LOM_Bit;         /* Enter Listen Only Mode */
                                  /* and leave Reset Mode   */
...
```

5.6 Automatic Bit-Rate Detection

The major drawback of existing trial and error concepts for automatic bit-rate detection is the generation of CAN error frames which is not acceptable. The SJA1000 supports the requirements for an automatic bit-rate detection with new features of the PeliCAN mode. This section briefly describes an application example without influencing running operations on the network.
In Listen Only Mode, the SJA1000 is neither able to transmit messages nor to generate error frames. Only message reception is feasible in this mode. A pre-defined table within the software contains all possible bit-rates including their bit-timing parameters. Before starting message reception with the highest possible bit-rate, the SJA1000 enables both receive and error interrupts.
In case of one or more errors on the CAN bus, the software switches to the next lower bit-rate.
Upon successful reception of a message, the SJA1000 has detected the right bit-rate and can switch to normal operating mode. From now on this node is able to operate as any other active CAN node in the system.

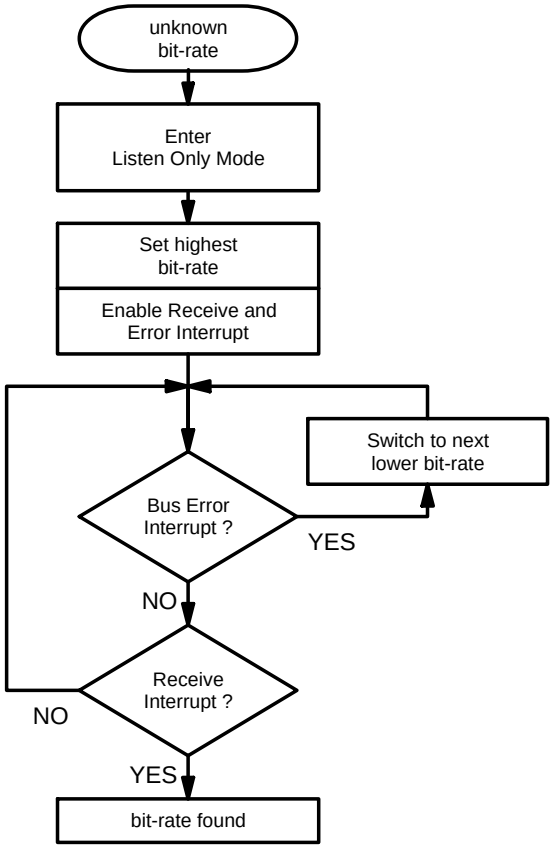

Figure 28: Algorithm of Bit-Rate Detection

5.7 CAN Self Tests

The SJA1000 supports two different options for self tests:

- Local Self Test
- Global Self Test

A *Local Self Test*, e.g., can be used perfectly for single node tests because an acknowledge from other nodes is not needed. In this case the SJA1000 has to be put into ´Self Test Mode´ (Mode register) and the command ´Self Reception Request´ is given.

For a *Global Self Test* the SJA1000 performs the same command in Operating Mode. However, for a Global Self Test in a running system a CAN acknowledge is needed.

Note that in both cases a physical layer interface must be available including CAN bus lines with a termination. A transmission or self reception is initiated by setting the appropriate bits in the Command Register.

The SJA1000 provides three command bits for the initiation of CAN transmissions and self receptions . Table 8 shows all possible combinations depending on the selected mode of operation.

Table 8: CAN Transmission Request Commands

Command	CMR =	Interrupt(s) upon successful operation	Self Test Mode	Operating Mode
Self Reception Request	0x10	RX and TX	local self test	global self test
Transmission Request	0x01	TX	normal transmission[1]	normal transmission
Single Shot	0x03	TX	transmission without re-transmission[1]	transmission without re-transmission
Single Shot and Self Reception Request	0x12	RX and TX	local self test without re-transmission	global self test without re-transmission

The following example presents basic programming elements for the initiation of a local self test.

Example: **Local Self Test**

```
....
ModeControlReg = RM_RR_Bit;       /* Enter Reset Mode       */
ClockDivideReg = CANMode_Bit;     /* PeliCAN Mode           */
ModeControlReg = STM_Bit;         /* Enter Self Test Mode   */
                                  /* and leave Reset Mode   */
TxFrameInfo = 0x03;               /* Fill Transmit Buffer   */
TxBuffer1   = 0x53;               /*                        */
...
TxBuffer5   = 0xAA;               /* Last Transmit Byte     */

CommandReg = SRR_Bit;             /* Self Reception Request */
.........................
if (RxBuffer1 != TxBufferRd1) comparison = false;
if (RxBuffer2 != TxBufferRd2) comparison = false;
```

[1] A normal transmission with or without re-transmission is usually performed in Operating Mode

5.8 Receive Sync Pulse Generation

The SJA1000 allows the generation of a pulse on the TX1 pin upon successful reception of a message. It is generated if the message is completely stored within the Receive FIFO. The pulse can be enabled in the Clock Divider Register and is active for the duration of the 6th bit in ´End Of Frame´.
Therefore, it can be used for versatile event triggered tasks, e.g., as a dedicated receive interrupt source or for a global clock synchronization in a distributed system which is briefly described in the following section.

In distributed systems it is difficult to implement a system wide clock without having an extra synchronization line [9]. All nodes connected onto the bus have local clocks with clock drifts. Lets assume that one CAN node in the network is assumed to operate as a ´master´ clock and the remaining clocks in the network are synchronized to the value of the master clock.

The Self Reception Request feature together with the fact that each SJA1000 is able to generate a pulse at a definite time upon message reception, can be used to support clock synchronization in distributed systems.

In Figure 29 a system master transmits a ´Self Reception Message´ onto the CAN bus. After message reception, each node, including the master, generates a Receive Sync Pulse. In every slave node it is used, i.e., to reset the timers. Simultaneously the master node uses this pulse to capture the master clock value $\mathbf{t_M}$.

In a next step the $\mathbf{t_M}$ value is sent as a ´Reference Time Message´ to all slaves by the master. A simple adder function in every slave, followed by reloading all timers with $\mathbf{t_S}$ synchronizes the system wide clock.

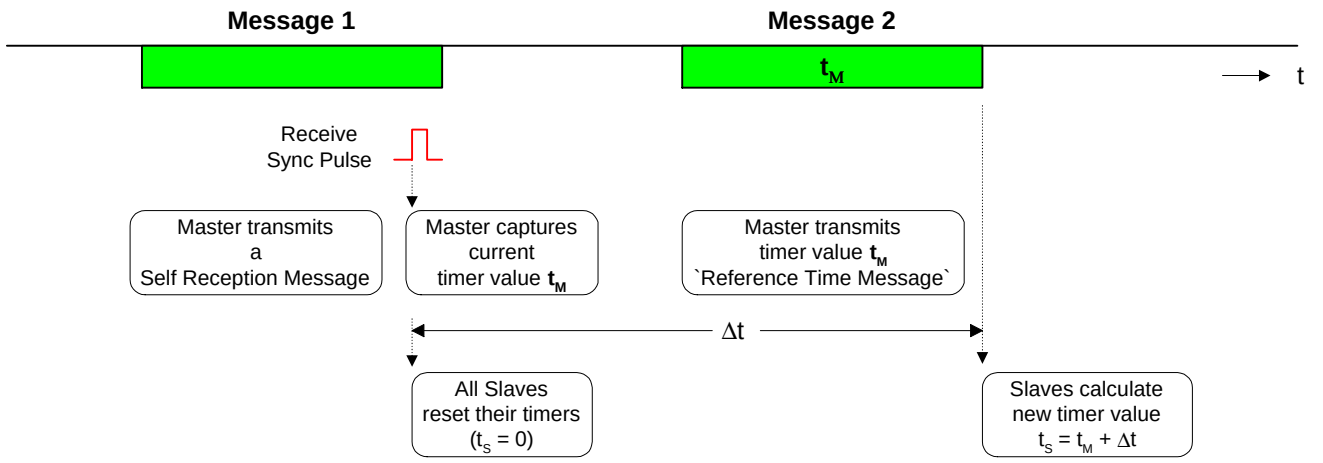

Figure 29: Timing Diagram during System Synchronization

The major advantage of this concept is the simplicity of implementation without complicated time stamp handling. No software cycle count is necessary because critical paths are hardware controlled and therefore deterministic. Furthermore it is independent of network parameters. Interrupt events may happen during the complete period without influencing the synchronization process.

6. REFERENCES

[1] Data Sheet SJA1000, Philips Semiconductors

[2] Eisele, H. and Jöhnk, E.: PCA82C250/251 CAN Transceiver, Application Note AN96116, Philips Semiconductors, 1996

[3] Data Sheet PCA82C250, Philips Semiconductors, September 1994

[4] Data Sheet PCA82C251, Philips Semiconductors, October 1996

[5] Data Sheet TJA1053, Philips Semiconductors,

[6] Jöhnk, E. and Dietmayer, K.: Determination of Bit Timing Parameters for the CAN Controller SJA1000, Application Note AN97046, Philips Semiconductors, 1997

[7] Data Sheet PCx82C200, Philips Semiconductors, November 1992

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

[9] Hank, P.: PeliCAN: A New CAN Controller Supporting Diagnosis and System Optimization, 4th International CAN Conference, Berlin, Germany, October 1997

7. APPENDIX

For the examples in the Application Note the C-language (C-compiler from Keil) is used to describe possible flows for the programming of the SJA1000. The target controller in these examples is the S87C654 from Philips Semiconductors, but any other derivative of the 80C51 family can be used. Be sure to include the correct register declaration for the targeted derivative in the main program.

Register and bit definitions for the SJA1000

```
/* definition for direct access to 8051 memory areas              */

#define XBYTE ((unsigned char volatile xdata *) 0)


/* address and bit definitions for the Mode & Control Register    */

#define ModeControlReg  XBYTE[0]

#define RM_RR_Bit    0x01   /* reset mode (request) bit           */
#if defined (PeliCANMode)
#define LOM_Bit      0x02   /* listen only mode bit               */
#define STM_Bit      0x04   /* self test mode bit                 */
#define AFM_Bit      0x08   /* acceptance filter mode bit         */
#define SM_Bit       0x10   /* enter sleep mode bit               */
#endif


/* address and bit definitions for the
                          Interrupt Enable & Control Register   */

#if defined (PeliCANMode)
#define InterruptEnReg  XBYTE[4] /* PeliCAN mode                 */

#define RIE_Bit      0x01   /* receive interrupt enable bit       */
#define TIE_Bit      0x02   /* transmit interrupt enable bit      */
#define EIE_Bit      0x04   /* error warning interrupt enable bit */
#define DOIE_Bit     0x08   /* data overrun interrupt enable bit  */
#define WUIE_Bit     0x10   /* wake-up interrupt enable bit       */
#define EPIE_Bit     0x20   /* error passive interrupt enable bit */
#define ALIE_Bit     0x40   /* arbitration lost interr. enable bit*/
#define BEIE_Bit     0x80   /* bus error interrupt enable bit     */
#else  /* BasicCAN mode */
#define InterruptEnReg  XBYTE[0] /* Control Register             */

#define RIE_Bit      0x02   /* Receive Interrupt enable bit       */
#define TIE_Bit      0x04   /* Transmit Interrupt enable bit      */
#define EIE_Bit      0x08   /* Error Interrupt enable bit         */
#define DOIE_Bit     0x10   /* Overrun Interrupt enable bit       */
#endif


/* address and bit definitions for the Command Register           */

#define CommandReg      XBYTE[1]

#define TR_Bit       0x01   /* transmission request bit           */
#define AT_Bit       0x02   /* abort transmission bit             */
#define RRB_Bit      0x04   /* release receive buffer bit         */
#define CDO_Bit      0x08   /* clear data overrun bit             */
#if defined (PeliCANMode)
#define SRR_Bit      0x10   /* self reception request bit         */
```

```
#else  /* BasicCAN mode */
#define GTS_Bit      0x10   /* goto sleep bit (BasicCAN mode)     */
#endif


/* address and bit definitions for the Status Register            */

#define StatusReg        XBYTE[2]

#define RBS_Bit      0x01   /* receive buffer status bit          */
#define DOS_Bit      0x02   /* data overrun status bit            */
#define TBS_Bit      0x04   /* transmit buffer status bit         */
#define TCS_Bit      0x08   /* transmission complete status bit   */
#define RS_Bit       0x10   /* receive status bit                 */
#define TS_Bit       0x20   /* transmit status bit                */
#define ES_Bit       0x40   /* error status bit                   */
#define BS_Bit       0x80   /* bus status bit                     */


/* address and bit definitions for the Interrupt Register         */

#define InterruptReg    XBYTE[3]

#define RI_Bit       0x01   /* receive interrupt bit              */
#define TI_Bit       0x02   /* transmit interrupt bit             */
#define EI_Bit       0x04   /* error warning interrupt bit        */
#define DOI_Bit      0x08   /* data overrun interrupt bit         */
#define WUI_Bit      0x10   /* wake-up interrupt bit              */
#if defined (PeliCANMode)
#define EPI_Bit      0x20   /* error passive interrupt bit        */
#define ALI_Bit      0x40   /* arbitration lost interrupt bit     */
#define BEI_Bit      0x80   /* bus error interrupt bit            */
#endif


/* address and bit definitions for the Bus Timing Registers       */

#define BusTiming0Reg    XBYTE[6]
#define BusTiming1Reg    XBYTE[7]

#define SAM_Bit      0x80   /* sample mode bit
                               1 == the bus is sampled 3 times
                               0 == the bus is sampled once       */


/* address and bit definitions for the Output Control Register    */

#define OutControlReg    XBYTE[8]

                            /* OCMODE1, OCMODE0                  */
#define BiPhaseMode  0x00   /* bi-phase output mode              */
#define NormalMode   0x02   /* normal output mode                */
#define ClkOutMode   0x03   /* clock output mode                 */

                            /* output pin configuration for TX1  */
#define OCPOL1_Bit   0x20   /* output polarity control bit       */
#define Tx1Float     0x00   /* configured as float               */
#define Tx1PullDn    0x40   /* configured as pull-down           */
#define Tx1PullUp    0x80   /* configured as pull-up             */
#define Tx1PshPull   0xC0   /* configured as push/pull           */

                            /* output pin configuration for TX0  */
#define OCPOL0_Bit   0x04   /* output polarity control bit       */
#define Tx0Float     0x00   /* configured as float               */
#define Tx0PullDn    0x08   /* configured as pull-down           */
```

```
#define Tx0PullUp    0x10   /* configured as pull-up              */
#define Tx0PshPull   0x18   /* configured as push/pull            */


/* address definitions of Acceptance Code & Mask Registers       */

#if defined (PeliCANMode)
#define AcceptCode0Reg   XBYTE[16]
#define AcceptCode1Reg   XBYTE[17]
#define AcceptCode2Reg   XBYTE[18]
#define AcceptCode3Reg   XBYTE[19]
#define AcceptMask0Reg   XBYTE[20]
#define AcceptMask1Reg   XBYTE[21]
#define AcceptMask2Reg   XBYTE[22]
#define AcceptMask3Reg   XBYTE[23]
#else  /* BasicCAN mode */
#define AcceptCodeReg    XBYTE[4]
#define AcceptMaskReg    XBYTE[5]
#endif


/* address definitions of the Rx-Buffer                          */

#if defined (PeliCANMode)
#define RxFrameInfo      XBYTE[16]
#define RxBuffer1        XBYTE[17]
#define RxBuffer2        XBYTE[18]
#define RxBuffer3        XBYTE[19]
#define RxBuffer4        XBYTE[20]
#define RxBuffer5        XBYTE[21]
#define RxBuffer6        XBYTE[22]
#define RxBuffer7        XBYTE[23]
#define RxBuffer8        XBYTE[24]
#define RxBuffer9        XBYTE[25]
#define RxBuffer10       XBYTE[26]
#define RxBuffer11       XBYTE[27]
#define RxBuffer12       XBYTE[28]
#else  /* BasicCAN mode */
#define RxBuffer1        XBYTE[20]
#define RxBuffer2        XBYTE[21]
#define RxBuffer3        XBYTE[22]
#define RxBuffer4        XBYTE[23]
#define RxBuffer5        XBYTE[24]
#define RxBuffer6        XBYTE[25]
#define RxBuffer7        XBYTE[26]
#define RxBuffer8        XBYTE[27]
#define RxBuffer9        XBYTE[28]
#define RxBuffer10       XBYTE[29]
#endif


/* address definitions of the Tx-Buffer                          */

#if defined (PeliCANMode)
/* write only addresses                                          */
#define TxFrameInfo      XBYTE[16]
#define TxBuffer1        XBYTE[17]
#define TxBuffer2        XBYTE[18]
#define TxBuffer3        XBYTE[19]
#define TxBuffer4        XBYTE[20]
#define TxBuffer5        XBYTE[21]
#define TxBuffer6        XBYTE[22]
#define TxBuffer7        XBYTE[23]
#define TxBuffer8        XBYTE[24]
#define TxBuffer9        XBYTE[25]
```

```
#define TxBuffer10       XBYTE[26]
#define TxBuffer11       XBYTE[27]
#define TxBuffer12       XBYTE[28]
/* read only addresses                                          */
#define TxFrameInfoRd    XBYTE[96]
#define TxBufferRd1      XBYTE[97]
#define TxBufferRd2      XBYTE[98]
#define TxBufferRd3      XBYTE[99]
#define TxBufferRd4      XBYTE[100]
#define TxBufferRd5      XBYTE[101]
#define TxBufferRd6      XBYTE[102]
#define TxBufferRd7      XBYTE[103]
#define TxBufferRd8      XBYTE[104]
#define TxBufferRd9      XBYTE[105]
#define TxBufferRd10     XBYTE[106]
#define TxBufferRd11     XBYTE[107]
#define TxBufferRd12     XBYTE[108]
#else  /* BasicCAN mode */
#define TxBuffer1        XBYTE[10]
#define TxBuffer2        XBYTE[11]
#define TxBuffer3        XBYTE[12]
#define TxBuffer4        XBYTE[13]
#define TxBuffer5        XBYTE[14]
#define TxBuffer6        XBYTE[15]
#define TxBuffer7        XBYTE[16]
#define TxBuffer8        XBYTE[17]
#define TxBuffer9        XBYTE[18]
#define TxBuffer10       XBYTE[19]
#endif


/* address definitions of Other Registers                       */

#if defined (PeliCANMode)
#define ArbLostCapReg    XBYTE[11]
#define ErrCodeCapReg    XBYTE[12]
#define ErrWarnLimitReg  XBYTE[13]
#define RxErrCountReg    XBYTE[14]
#define TxErrCountReg    XBYTE[15]
#define RxMsgCountReg    XBYTE[29]
#define RxBufStartAdr    XBYTE[30]
#endif


/* address and bit definitions for the Clock Divider Register   */

#define ClockDivideReg   XBYTE[31]

#define DivBy1       0x07   /* CLKOUT = oscillator frequency      */
#define DivBy2       0x00   /* CLKOUT = 1/2 oscillator frequency  */

#define ClkOff_Bit   0x08   /* clock off bit,
                                    control of the CLK OUT pin  */
#define RXINTEN_Bit  0x20   /* pin TX1 used for receive interrupt */
#define CBP_Bit      0x40   /* CAN comparator bypass control bit  */
#define CANMode_Bit  0x80   /* CAN mode definition bit            */
```

Register and bit definitions for the Micro Controller S87C654

```
/*  Port 2 Register  “P2”                                              */
sfr P2     = 0xA0;

sbit P2_7  = 0xA7;   /* MSB of port 2, used for chip select for SJA1000 */
.

/* alternate functions of port 3 “P3”                                  */
sfr P3     = 0xB0;
.
sbit int0  = 0xB2;
.

/*  Timer Control Register   “TCON”  */
sfr TCON   = 0x88;

.
sbit IE0   = 0x89;   /* external interrupt 0 edge flag                 */
sbit IT0   = 0x88;   /* interrupt 0 type control bit
                        (edge or low-level triggered                   */
.

/*  Interrupt Enable Register   “IE” */
sfr IE     = 0xA8;

sbit EA    = 0xAF;   /* overall interrupt enable/disable flag          */
.
sbit EX0   = 0xA8;   /* enable or disable external interrupt 0         */
.

/*  Interrupt Priority Register   “IP” */
sfr IP     = 0xB8;

.
sbit PX0   = 0xB8;   /* external interrupt 0 priority level control    */
.
```

Definitions of Variables and Constants for the Examples

```
/*- definition of hardware / software connections ----------------------*/

/* controller: S87C654; CAN controller: SJA1000(see Figure 3 on page 11)*/

#define CS        P2_7   /* ChipSelect for the SJA1000                  */

#define SJAIntInp int0   /* external interrupt 0 (from SJA1000)         */

#define SJAIntEn  EX0    /* external interrupt 0 enable flag            */


/*- definition of used constants ---------------------------------------*/

#define YES           1
#define NO            0

#define ENABLE        1
#define DISABLE       0
#define ENABLE_N      0
#define DISABLE_N     1

#define INTLEVELACT   0
#define INTEDGEACT    1

#define PRIORITY_LOW  0
#define PRIORITY_HIGH 1

/* default (reset) value for register content, clear register     */
#define ClrByte      0x00

/* constant: clear Interrupt Enable Register                      */
#if defined (PeliCANMode)
#define ClrIntEnSJA ClrByte
#else
#define ClrIntEnSJA ClrByte | RM_RR_Bit /* preserve reset request */
#endif

/* definitions for the acceptance code and mask register          */
#define DontCare     0xFF


/*- definition of bus timing values for different examples -------*/

/* bus timing values for the example given in (AN97046)
   - bit-rate                          : 250 kBit/s
   - oscillator frequency              : 24 MHz, 1,0%
   - maximum propagation delay         : 1630 ns
   - minimum requested propagation delay :  120 ns                */
#define PrescExample     0x02 /* baud rate prescaler : 3          */
#define SJWExample       0xC0 /* SJW                 : 4          */
#define TSEG1Example     0x0A /* TSEG1               : 11         */
#define TSEG2Example     0x30 /* TSEG2               : 4          */

/* bus timing values for
   - bit-rate                          : 1 MBit/s
   - oscillator frequency              : 24 MHz, 0,1%
   - maximum tolerated propagation delay : 747 ns
   - minimum requested propagation delay :  45 ns                 */
#define Presc_MB_24      0x00 /* baud rate prescaler : 1          */
#define SJW_MB_24        0x00 /* SJW                 : 1          */
```

```
#define TSEG1_MB_24      0x08 /* TSEG1               : 9          */
#define TSEG2_MB_24      0x10 /* TSEG2               : 2          */

/* bus timing values for
   - bit-rate                          : 100 kBit/s
   - oscillator frequency              : 24 MHz, 1,0%
   - maximum tolerated propagation delay : 4250 ns
   - minimum requested propagation delay :  100 ns               */
#define Presc_kB_24      0x07 /* baud rate prescaler : 8          */
#define SJW_kB_24        0xC0 /* SJW                 : 4          */
#define TSEG1_kB_24      0x09 /* TSEG1               : 10         */
#define TSEG2_kB_24      0x30 /* TSEG2               : 4          */

/* bus timing values for
   - bit-rate                          : 1 MBit/s
   - oscillator frequency              : 16 MHz, 0,1%
   - maximum tolerated propagation delay : 623 ns
   - minimum requested propagation delay :  23 ns                */
#define Presc_MB_16      0x00 /* baud rate prescaler : 1          */
#define SJW_MB_16        0x00 /* SJW                 : 1          */
#define TSEG1_MB_16      0x04 /* TSEG1               : 5          */
#define TSEG2_MB_16      0x10 /* TSEG2               : 2          */

/* bus timing values for
   - bit-rate                          : 100 kBit/s
   - oscillator frequency              : 16 MHz, 1,0%
   - maximum tolerated propagation delay : 4450 ns
   - minimum requested propagation delay :  500 ns               */
#define Presc_kB_16      0x04 /* baud rate prescaler : 5          */
#define SJW_kB_16        0xC0 /* SJW                 : 4          */
#define TSEG1_kB_16      0x0A /* TSEG1               : 11         */
#define TSEG2_kB_16      0x30 /* TSEG2               : 4          */

/*- end of definitions of bus timing values ----------------------*/


/*- definition of used variables ---------------------------------*/

/* intermediate storage of the content of the Interrupt Register  */
BYTE bdata CANInterrupt;              /* bit addressable byte     */
sbit RI_BitVar     = CANInterrupt ^ 0;
sbit TI_BitVar     = CANInterrupt ^ 1;
sbit EI_BitVar     = CANInterrupt ^ 2;
sbit DOI_BitVar    = CANInterrupt ^ 3;
sbit WUI_BitVar    = CANInterrupt ^ 4;
sbit EPI_BitVar    = CANInterrupt ^ 5;
sbit ALI_BitVar    = CANInterrupt ^ 6;
sbit BEI_BitVar    = CANInterrupt ^ 7;
```