AVR316: SMbus Slave Using the TWI Module

Features

- **Supports 9 different SMBus protocols**
- **Packet error checking (PEC)**
- **Interrupt-driven SMBus slave driver**
- **Sample implementation with demonstration of all supported protocols**

1 Introduction

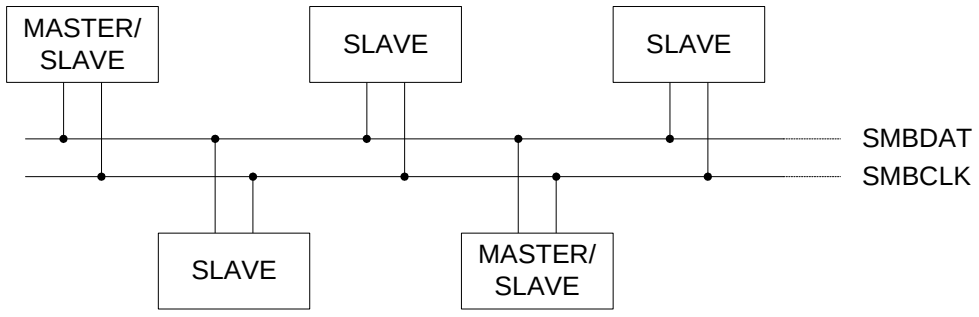
The System Management Bus (SMBus) uses the principles of the I^2C bus to make a bus where devices can communicate to exchange system and power management information.

SMBus is commonly used in personal computers for smart batteries, temperature control and other low bandwidth system management communication.

The SMBus specification revision 2.0 lists 11 protocols for device-to-device communication, in addition to the Address Resolution Protocol (ARP) which is used to dynamically assign addresses to devices. Each protocol is basically a sequence of I^2C commands. Slave devices can implement the full set, or a subset of the defined protocols, allowing for implementations scaled for the task at hand.

This application note provides background information on the SMBus specification and the AVR TWI module, an interrupt-driven SMBus slave driver and a sample implementation.

Figure 1-1. SMBus system overview.



8-bit AVR® Microcontrollers

Application Note

Rev. 2583A-AVR-10/05

2 The SMBus specification

The SMBus specification builds on the principles of I^2C to enable communication between many devices on a two-wire bus, but there are also some small but important differences both in electrical and timing parameters.

2.1 Differences between SMBus and I2C

The following comparison is for SMBus 2.0 low power devices.

First of all, I^2C defines input voltage levels as percentages of V_{CC}, while SMBus operates with fixed input voltage levels. Input voltage level minimum and maximum ratings for standard mode I2C and SMBus are shown in Table 1.

Table 1. Input voltage levels for I2C and SMBus

	I2C (Standard mode)		**SMBus**	
	Min.	**Max.**	**Min.**	**Max.**
V_{IL}	-0.5V	$0.3V_{CC}$	-	0.8V
V_{IH}	$0.7V_{CC}$	V_{CCmax}+0.5V	2.1V	5.5V

AVRs comply best with SMBus V_{IH} Min. at a supply voltage of 3V instead of 5V.

I^2C places restrictions on the maximum bus capacitance. The SMBus specification has no such restriction, but requires the pull-down current to be in the range of 100-350µA.

The rise and fall times of bus signals are defined in the SMBus specification. The I^2C specification defines only maximum bus capacitance, but not rise and fall times.

The maximum leakage current of each device connected to an I^2C bus is specified as 10µA, while the corresponding requirement for SMBus is 5µA.

SMBus requires a minimum bus clock frequency of 10 kHz (Except when the bus is idle). I^2C has no such requirement. The maximum frequency of SMBus is 100 kHz, which equals the maximum speed of I^2C operating in standard mode.

SMBus requires SMBDAT (SDA) to remain unchanged for 300ns after the falling edge of SMBCLK (SCL). No such requirements are imposed by the I^2C specification.

SMBus has several limitations on the maximum extension of a clock low time. I^2C allows the clock low time to be stretched for an arbitrary length of time, allowing a faulty device to block the entire bus.

SMBus requires devices to always acknowledge it's slave address, regardless of what other task the device is performing. I^2C has no such requirement.

SMBus uses the NACK signal to indicate that a slave is busy, or that an error occurred. Slave devices must be able to generate the NACK signal after the transfer of each byte, even if it is the last byte in the transfer.

2.2 Packet Error Code (PEC)

The SMBus specification includes a CRC based error detection algorithm called Packet Error Code (PEC). An SMBus device is not required to use PEC, but any device with PEC capability must be able to communicate with other SMBus devices that do not implement PEC.

The PEC is calculated as an 8-bit CRC with polynomial $X^8+X^2+X^1+1$. The PEC CRC is computed in the order the bits are received and transmitted. All bytes of a protocol, including SLA+R/W must be included in the calculation, while the control signals START, Repeated START, STOP, ACK and NACK are left out.

There are several ways to implement the CRC calculation used for PEC. The CRC can be calculated, using shifts and xor operations, or a lookup table can be utilized. A full lookup table takes up 256 bytes of flash memory, but the CRC algorithm is executed in only a few clock cycles. The calculation method has very low memory requirements, but executes much slower and has a larger implementation than the lookup method.

2.3 The SMBus protocols

The SMBus specification 2.0 defines 11 protocols for device to device communication. Each device can implement the full set, or a subset, of the protocols. If a device does not implement a protocol, the result of the operation is undefined.

Each protocol is composed of a strict sequence of I^2C data transfer format(s). The SMBus protocols will be presented by sequence diagrams. The meaning of the different symbols in the diagrams is listed in Table 2.

Table 2. SMBus protocol diagram legend

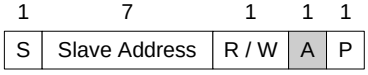
Symbol	Meaning
S	Start condition
Sr	Repeated start condition
R	Read (1)
W	Write (0)
$A / \overline{A}$	Acknowledge (0) or Not Acknowledge (1)
P	Stop condition
PEC	Packet Error Code
	Master to slave
	Slave to master

Additionally, a number placed directly above a field specifies the bit width of the field, and a number placed directly below a field specifies that the field is required to have this value.

2.3.1 Quick command

The “Quick command” protocol can be used to send 1 bit through the R/W bit to a slave device.

Figure 2. Quick command



2.3.2 Send byte

The “Send byte” protocol can be used to send 1 byte of data to a slave device.

Figure 3. Send byte

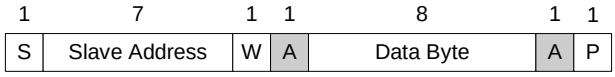
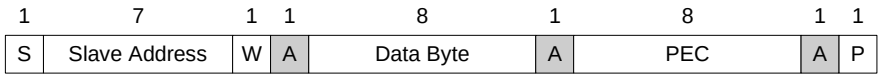


Figure 4. Send byte with PEC



2.3.3 Receive byte

The “Receive byte” protocol can be used to request 1 byte of data from a slave.

Figure 5. Receive byte

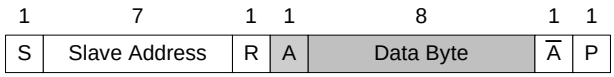
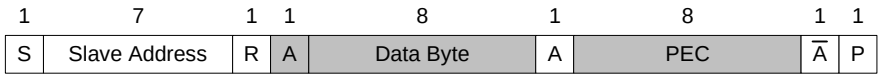


Figure 6. Receive byte with PEC



2.3.4 Write byte/word

The “Write byte/word” protocol can be used to send one command code, and one byte/word of data. The command code can for instance specify the memory location to place the data.

Figure 7. Write byte

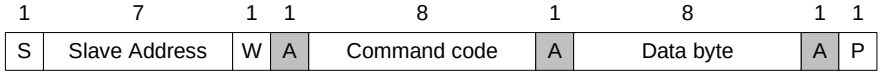


Figure 8. Write byte with PEC

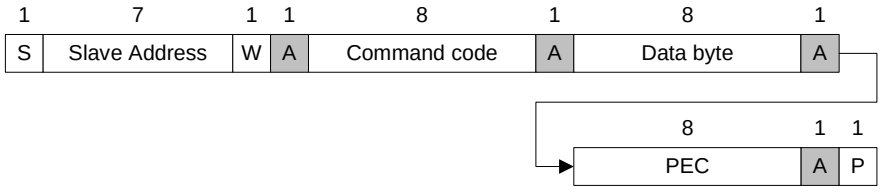


Figure 9. Write word

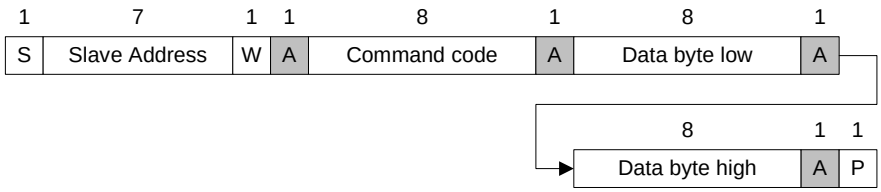
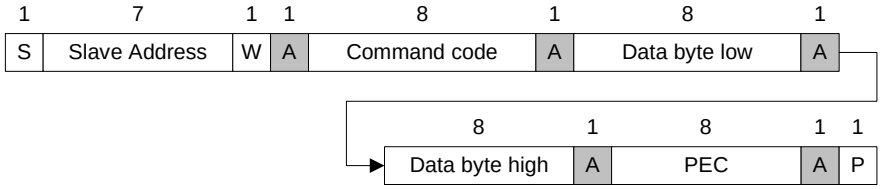


Figure 10. Write word with PEC



2.3.5 Read byte/word

The “Read byte/word” protocol can be used to read one byte/word from a slave device. The data read can be controlled by the command code, for instance by addressing internal registers of the slave device.

Figure 11. Read byte

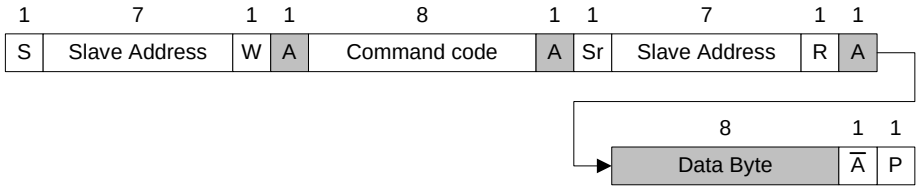


Figure 12. Read byte with PEC

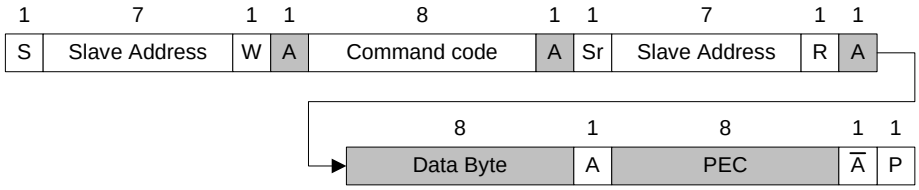


Figure 13. Read word

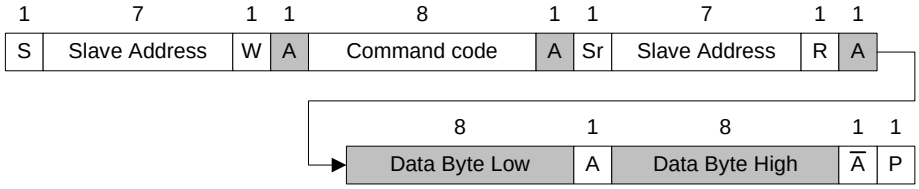
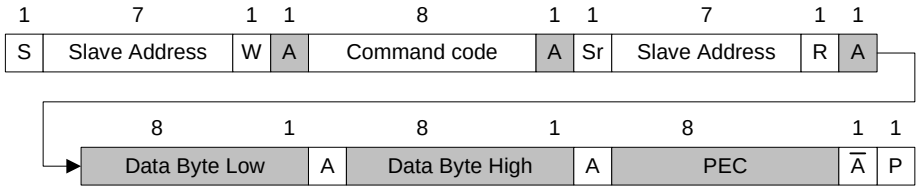


Figure 14. Read word with PEC



2.3.6 Process call

The “Process call” protocol allows the master to send both a command code and two bytes of data to a slave and receive two bytes of data. This can for instance be used to call a function with argument(s) on a slave device and receive the return value, in one command.

Figure 15. Process call

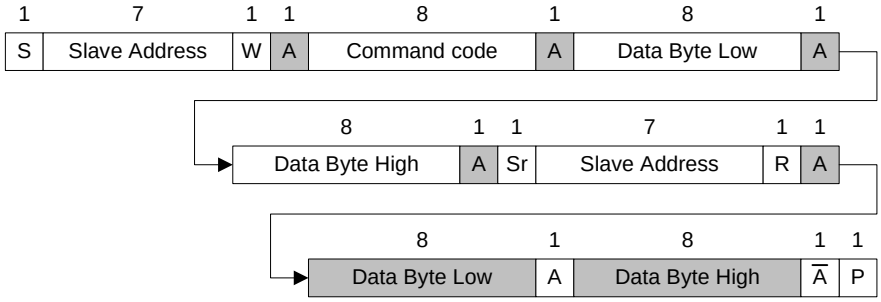
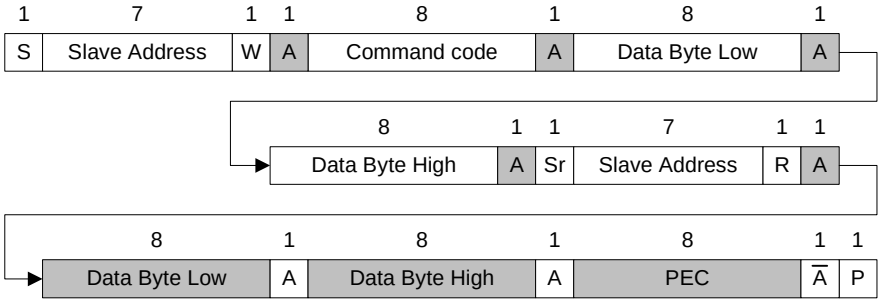


Figure 16. Process call with PEC



2.3.7 Block write/read

The “Block write/read” protocol can be used to send/receive up to 32 bytes of data to/from a slave device. A command code and the number of bytes to write/read is transmitted first. The byte count must be in the range 1 to 32 and does not include the PEC.

Figure 17. Block Write

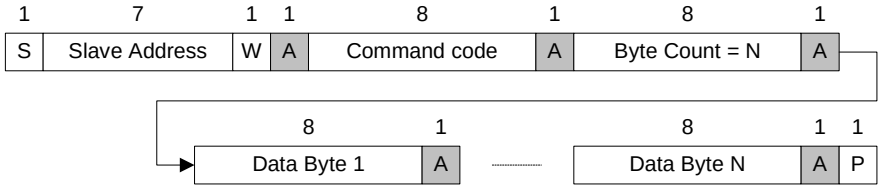


Figure 18. Block Write with PEC

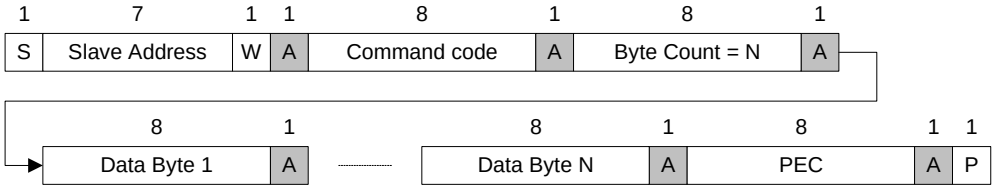


Figure 19. Block Read

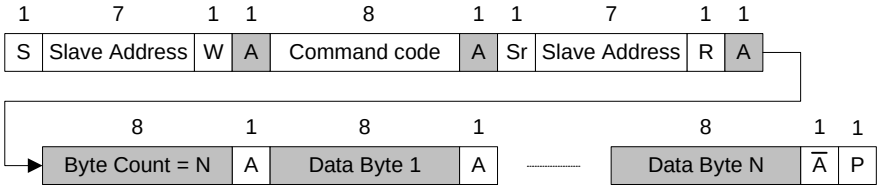
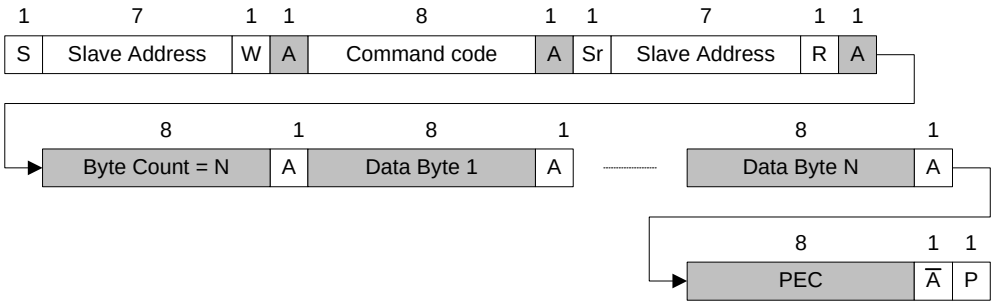


Figure 20. Block Read with PEC



2.3.8 Block write-block read process call

The “Block write-block read process call” protocol was introduced with SMBus version 2.0 and can be used to send a command code and 1 to 32 bytes to a slave, and receive 1 to 32 bytes from the slave in one operation. This can be used in the same way as the Process call protocol, but the length of parameters and return values are variable.

Figure 21. Block write - block read process call

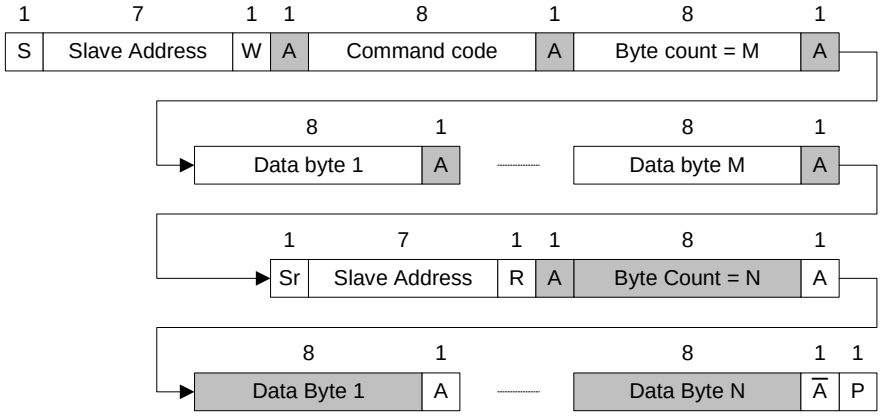
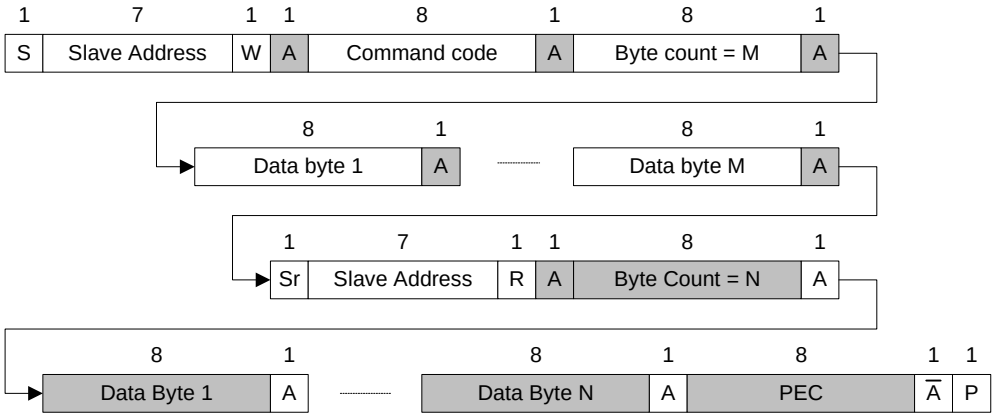


Figure 22. Block write - block read process call with PEC



2.3.9 SMBus host notify protocol

The “SMBus host notify” protocol is used when a device wants to become a master. Only a slave implementation is considered here, so the SMBus host notify protocol is not needed.

2.4 The Address Resolution Protocol (ARP)

The Address Resolution Protocol was introduces with SMBus version 2.0 and is used to dynamically assign unique addresses to each slave device on the bus. ARP requires the slave device to listen to both its own slave address and an SMBus device default address. The SMBus device default address is different from the general call address. The AVR TWI module cannot be set up to listen to two different slave addresses, unless one is the general call address. ARP is thus not supported.

3 Implementing SMBus on AVR microcontrollers

The TWI module found on many AVRs will be used to implement the SMBus driver in this application note. An introduction to the operation of the TWI module is given here. For a thorough description of the TWI module, consult the device data sheet.

3.1 The TWI module

The basic operation of the TWI module in slave mode is as follows:

The slave address of the AVR is stored in the TWI Address Register (TWAR). The TWI logic compares addresses transmitted on the bus to the address stored in the device to determine whether the address of the ongoing communication matches TWAR.

Whenever a decision has to be made to continue the ongoing communication, the TWI interrupt flag is set. The TWI status register (TWSR) must then be examined to determine the reason of the interrupt. The program can then take some action based on what has happened. Finally, the TWI control register (TWCR) must be manipulated to continue or stop the communication. This happens over and over again until the communication is finished. The TWSR status codes relevant for slave modes are listed in Table 3.

All information about which SMBus protocol is being transmitted must be derived from the sequence of TWSR status codes and the data received.

Table 3. TWSR status codes

Status code	TWI status
0x60	Own SLA+W has been received; ACK has been returned
0x68	Arbitration lost in SLA+R/W as master; own SLA+W has been received; ACK has been returned
0x70	General call address has been received; ACK has been returned
0x78	Arbitration lost in SLA+R/W as master; General call address has been received; ACK has been returned
0x80	Previously addressed with own SLA+W; Data has been received; ACK has been returned
0x88	Previously addressed with own SLA+W; Data has been received; NACK has been returned
0x90	Previously addressed with general call; data has been received; ACK has been returned
0x98	Previously addressed with general call; data has been received; NACK has been returned
0xa0	A STOP or repeated START condition has been received while still addressed as slave
0xa8	Own SLA+R has been received; ACK has been returned
0xb0	Arbitration lost in SLA+R/W as master; own SLA+R has been received; ACK has been returned
0xb8	Data byte in TWDR has been transmitted; ACK has been received
0xc0	Data byte in TWDR has been transmitted; NACK has been received

Status code	TWI status
0xc8	Last data byte in TWDR has been transmitted (TWEA = “0”); ACK has been received
0xf8	No relevant state information
0x00	Bus error due to an illegal START or STOP condition

There are some implementation details to be aware of when implementing SMBus with the TWI module:

- When the AVR acts as a slave receiver, one must decide before receiving data whether to ACK or NACK that data. If data is received when e.g. a stop or repeated start condition was expected, the ACK has already been sent when the slave has detected the error. Similarly, this makes it impossible for an AVR to use ACKs and NACKs to signal PEC errors. The SMBus specification does not require the slave to do this, and suggests that verification of PECs can be done in higher network protocol layers.
- When TWSR indicates that a STOP or repeated START condition has been received, there is no way of knowing which one occurred. This makes it difficult to know whether a Send byte or Read byte, Read word or a Block read protocol is in progress.
- If it is not known in advance whether PEC is to be used when acting as slave transmitter, there is no way of knowing when to expect a NACK in reply to a data byte.
- The AVR TWI module does not support the Quick command very well.

None of the above points is a major problem when implementing an SMBus slave on the AVR. The SMBus host should not rely only on ACK/NACK to decide whether a communication completed without errors. The single bit ACK/NACKs are susceptive to noise. PEC does not need to be supported, or handling can be accomplished in higher level protocols. A slave implementation is free to choose which protocols to implement. The master is responsible for issuing the right commands and the result of an unsupported protocol is undefined according to the SMBus specification.

3.2 Proposed implementation

Based on the above discussion, a versatile implementation is proposed here. The implementation avoids the protocols that can yield ambiguities on the AVR and minimizes the code needed to determine what kind of protocol is in progress. Furthermore, PEC can be supported if needed. The proposed implementation supports the following SMBus protocols:

- Receive byte
- Write byte
- Write word
- Read byte
- Read word
- Block write
- Block read

- Process call
- Block write – Block read Process call

This reduced set of protocols makes it very easy to determine what protocol is in progress. Receive byte is the only protocol that starts with a SLA+R. All the other protocols include a command code that is sent as the first data byte to the slave. The command code could be utilized in several ways. One way is to have a unique command code for every operation, in such a way that the protocol can always be derived from the command code. Alternatively, the command code could specify the address of a register or memory location.

3.3 Description of the included driver and sample application

In the sample driver and application included with this application note, the command code uniquely identifies both the SMBus protocol and the required slave action.

A very general SMBus implementation could just take care of transmission and reception from buffers and use flags to communicate with the main program. However, in typical SMBus applications, the SMBus commands are used to set register values or read back status information. The exact meaning of each command is well defined in advance. With this in mind, the SMBus driver included is implemented in the TWI interrupt routine, making the driver totally autonomous. The functionality of the sample application is shown in Table 4.

Table 4. Command code assignment

Command code	Action	Protocol
None	Read switches pressed (PIND)	Receive byte
0x10	Read vendor id string	Block read
0x20	Read switches pressed (PIND)	Read byte
0x30	Set EEPROM pointer	Write word
0x40	Read EEPROM data byte	Read byte
0x41	Read EEPROM data word	Read word
0x50	Output byte to LEDs (PORTB)	Write byte
0x51	Output word as two alternating patterns to LEDs (PORTB)	Write word
0x52	Output sequence of patterns to LEDs (PORTB)	Block write
0x60	Return received word multiplied by 2	Process call
0x70	Return sum of received bytes as a word	Block write - block read process call

The included example is written with the STK500 development board in mind, but this is not a prerequisite for use. When the STK500 is used, the “PORTB” header should be connected to the “LEDS” header and the “PORTD” header should be connected to the SWITCHES header. “Read switches pressed” returns the ones complement of PIND. The commands that output data to PORTB/LEDS, places the data in an array. A timer overflow interrupt outputs the sequence in the array one by one.

Optional support for PEC is included in the driver. The CRC calculation needs to be done several places within the TWI interrupt routine. By default, the lookup table

method is used, since it has the smallest and fastest implementation. It is possible to change to CRC calculation, if the 256-byte table in flash is too big. If PEC is not needed, it can be disabled, resulting in a smaller and faster implementation.

Since PEC support is optional, every operation relevant only to PEC is marked with gray in the flowcharts. If PEC is not enabled, program flow will go straight through, or in the “Yes” direction through decision blocks.

The implementation of the CRC lookup table method works as follows:

1. If this is the beginning of a new CRC calculation, set CRC value to 0.
2. Xor the old CRC value with the new data.
3. Use the value from 2 as an index into the lookup table.
4. The value retrieved at this index is the new CRC.
5. Repeat steps 2-4 until finished.

When the CRC of a data stream is appended to the data stream itself, the CRC of the total data stream should be zero when no error is detected. This has been used in the included driver. When a PEC is received from a master, the PEC is also fed through the CRC calculation. In this way, PEC verification is performed by checking that the PEC variable is zero.

To be able to track the state of the ongoing communication, a set of variables is needed (Name in parenthesis):

- Transmit length (`txLength`)
- Transmit counter (`txCount`)
- Receive counter (`rxCount`)
- SMBus state flag (`state`)
- SMBus enable flag (`enable`)
- SMBus error flag (`error`)
- PEC (`pec`) (If enabled)
- Receive buffer (`rxBuffer`)
- Transmit buffer (`txBuffer`)

The receive and transmit buffers are used only by the SMBus driver as temporary storage. Once a protocol has completed successfully, the contents of the receive buffer is copied to the intended location. This prevents a failed communication to cause data corruption. No other parts of the application should access the SMBus receive and transmit buffers. The length of these buffers must be set to accommodate the longest protocol supported.

The receive and transmit counters keep track of the progress of the current protocol. The receive counter is increased every time one data byte is received and acts as a pointer into the receive buffer. Similarly, the transmit counter is increased every time one data byte is transmitted and acts as a pointer into the transmit buffer. Transmit length is set to the total number of bytes to be transmitted.

The SMBus state flag is used in combination with the command code to make decisions regarding the protocol being used. The flag can be set to one of the following four values:

- IDLE – Waiting for an SMBus protocol to start.
- READ REQUESTED – SLA+R has been received as the first command.
- WRITE_REQUESTED – SLA+W has been received as the first command.
- WRITE_READ_REQUESTED – SLA+W has been received as the first command, repeated start, then SLA+R has been received.

The SMBus enable flag can be used by other parts of the application to enable or disable the SMBus interface. When this flag is set, the SMBus driver will try to complete any transmission initiated by the master. When the flag is cleared, TWI interrupts will still be generated, but the SMBus driver will only answer to the SLA+R/W and then attempt to abort the communication by signaling to the host that it is busy. Note that the SMBus specification demands that any SMBus slave should be able to reply to it's own address at all times.

To disable the SMBus driver completely and prevent TWI interrupts from being generated, the TWEN flag in the TWCR register must be cleared. If the TWI module is disabled, the SCL and SDA pins will be configured as standard I/O pins. To avoid conflict with the SMBus, these must be configured as inputs, with internal pull-ups disabled. Note that currently all AVR devices connected to an SMBus need to be powered to allow any bus operation.

The SMBus error flag will be set whenever an error occurs during SMBus communication. The flag is never used for any useful purpose in the example implementation. This must be handled in higher-level protocols.

If PEC is enabled, the pec variable holds the current PEC value calculated for this transmission so far. Two CRC routines are included in the example. Both operates directly on this variable and must therefore only be used by the SMBus driver.

Figure 23 shows the general program flow of the SMBus driver during the execution of one SMBus protocol. Note that this is not the flowchart of the TWI interrupt routine, several TWI interrupts will be triggered to complete one cycle of the flowchart in Figure 23. At every point where a TWI interrupt is expected, a dashed box shows the TWSR status code that corresponds to program flow in that direction.

Figure 24 and Figure 25 shows the TWI interrupt routine. These flowcharts perform the same as the one in Figure 23, broken down to iterations of the TWI interrupt routine.

Figure 26 shows the flowchart for the block called "Process message" in Figure 24. This is where most of the application specific parts of the SMBus slave are implemented. Note that while this runs the SMBCLK is held low, and it is thus important to keep the functions short enough not to violate the SMBus $T_{LOW:SEXT}$ of 25ms. If this happens, the master will time out and drop the lines, and since the AVR's TWI module is I2C compatible there is no timeout in the TWI module. The AVR can then hold the SMBDAT line low waiting for clock, and thus block the bus indefinitely. This implementation has no provisions for correcting this situation, as this should be handled according to the application.

In addition Timer/Counter1 is set up to produce interrupts at a fixed rate. This interrupt is used to display data sequentially at the LEDs of the STK500 development board. A sequence of up to 32 values can be put in the global array 'leds'. The global variable 'ledLength' defines the number of values to display from this array, while 'ledIndex' controls the current displayed value.

Figure 23. General SMBus driver flow

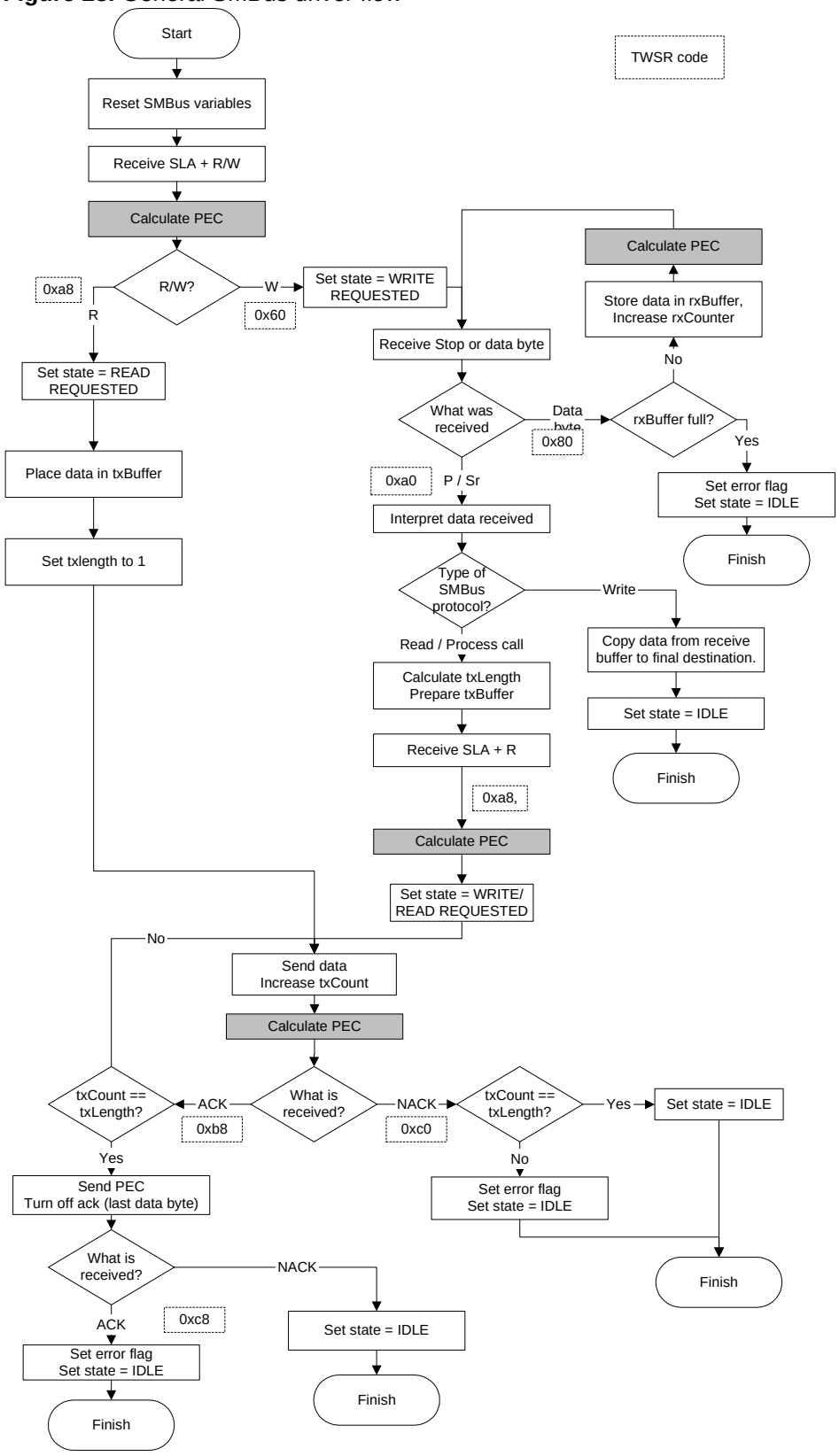


Figure 24. TWI ISR part 1

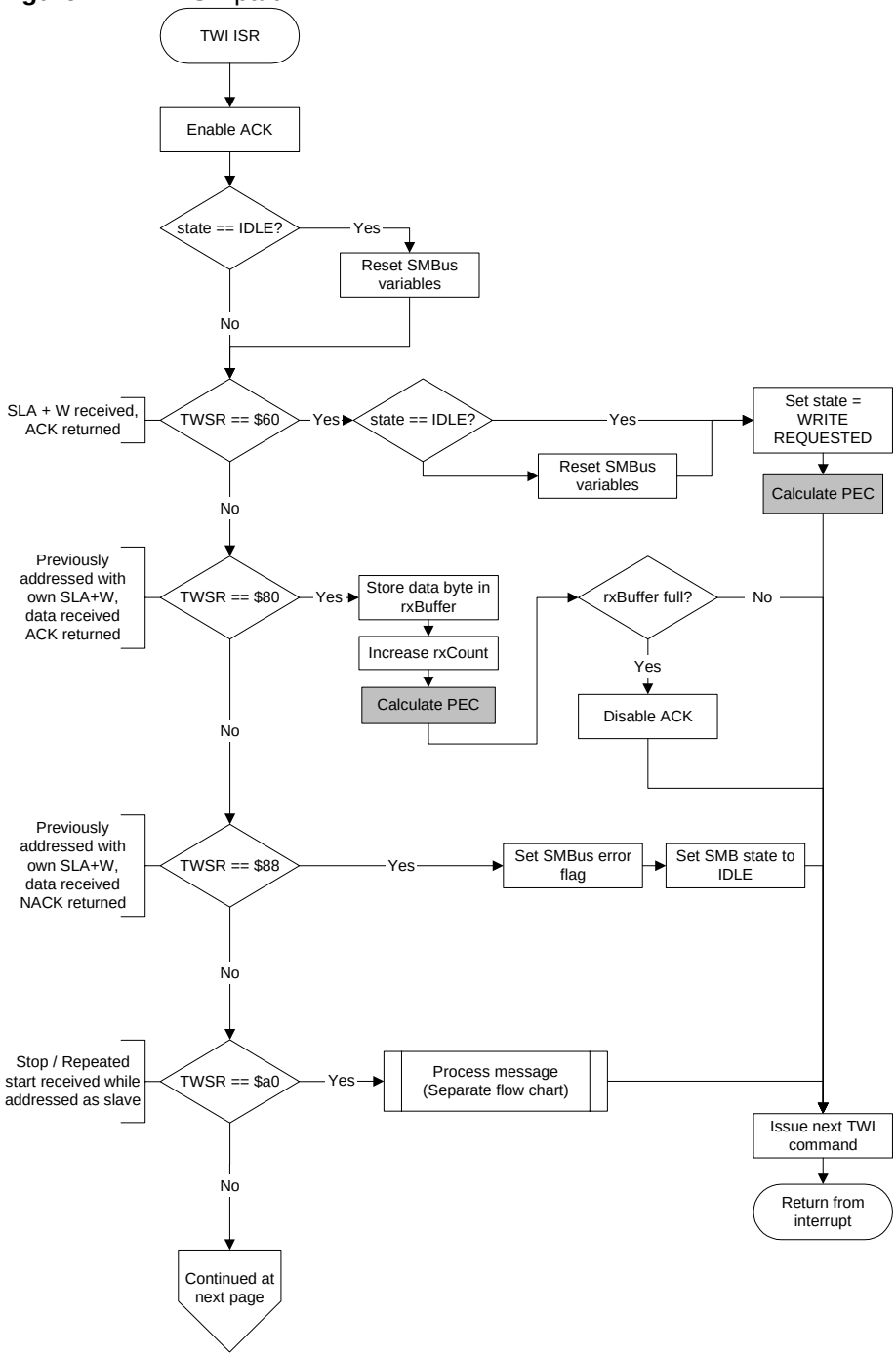


Figure 25 TWI ISR part 2

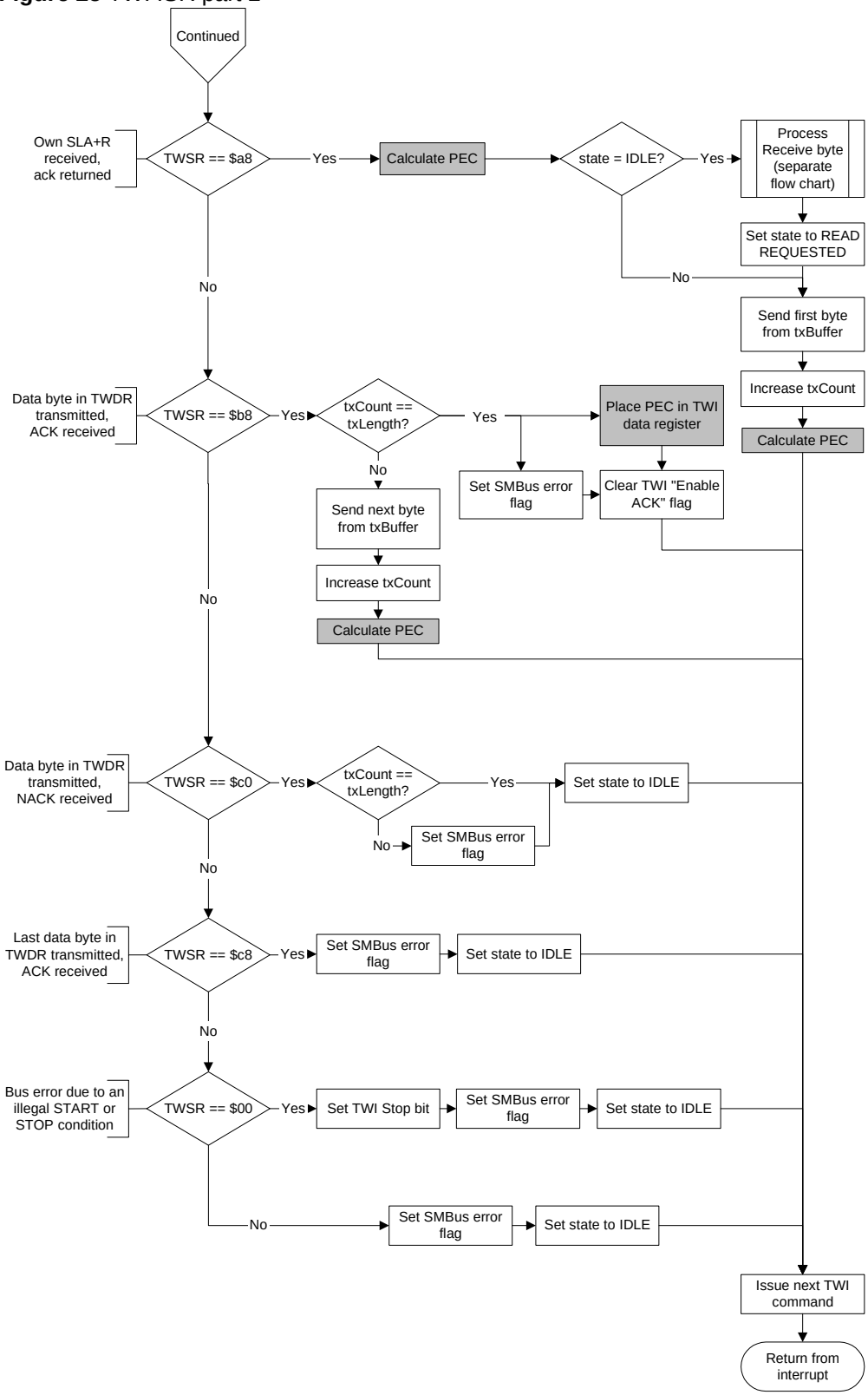
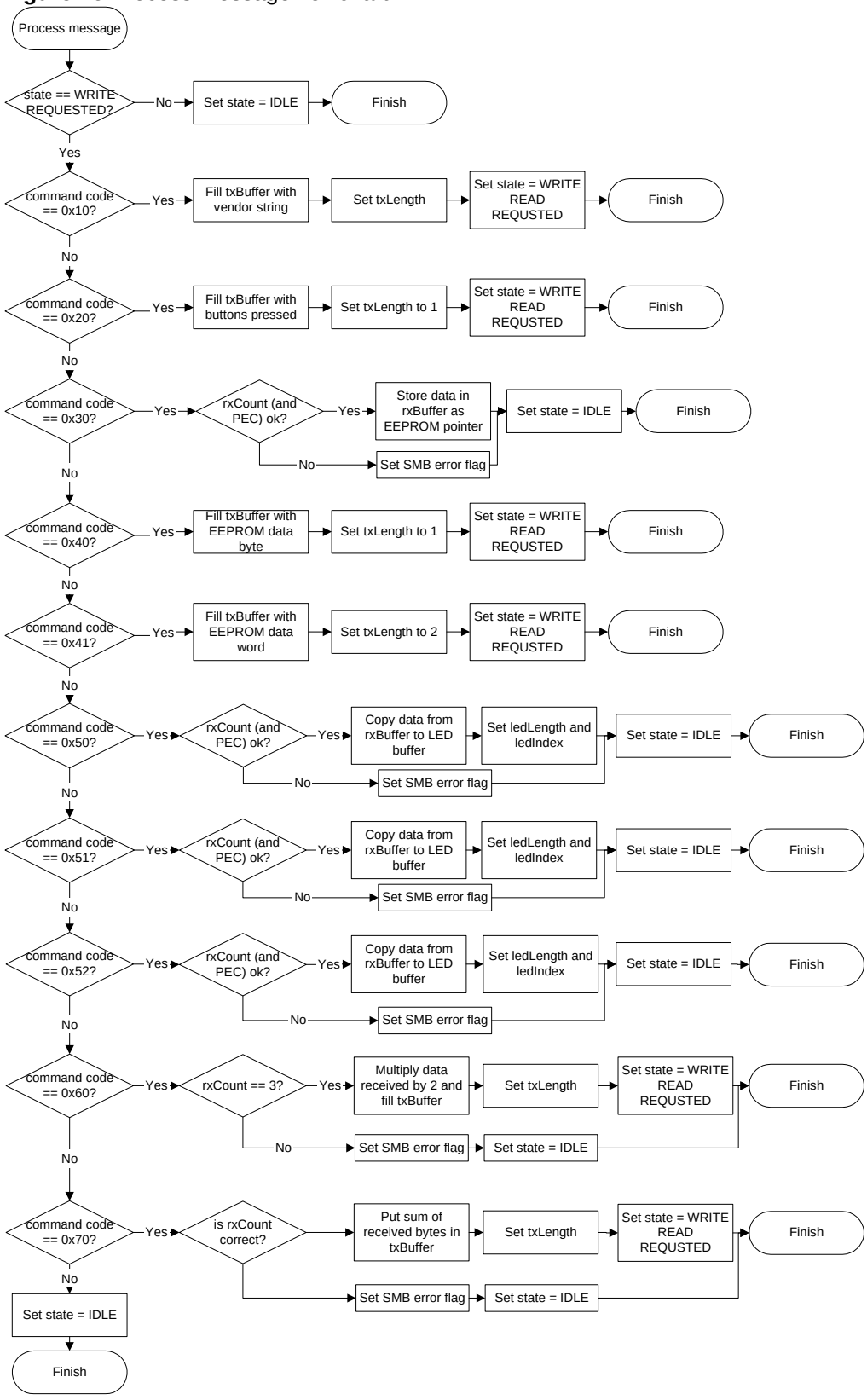


Figure 26 Process message flowchart



3.4 Source code

The source code is available for download as a zip-file. It has been documented with Doxygen-compatible comments, and the compiled documentation is included with the source in the file ‘readme.html’. It also contains info about supported compiler(s) and on how to compile the files.

3.5 Connections

Connect the ‘SDA’ pin of the AVR to ‘SMBDAT’ and ‘SCL’ on the AVR to ‘SMBCLK’. If an STK500 development board is used, connect ‘PORTB’ to ‘LEDS’ and ‘PORTD’ to ‘SWITCHES’. The connection of ‘PORTB’ and ‘PORTD’ is not necessary to run the example, but the functionality will be limited.

3.6 Adapting the SMBus example

The sample implementation demonstrates how the different supported protocols can be implemented. In order to adapt the sample application to your own needs, the functions ‘`ProcessReceiveByte`‘ and ‘`ProcessMessage`’ must be altered. This is documented through comments in the source code or the doxygen documentation.

Note that the clock is held low by the TWI module during execution of ‘`ProcessReceiveByte`‘ and ‘`ProcessMessage`’. Please make sure that these functions execute fast enough to be in conformance with the ‘Clock low extending’ section of the SMBus specification.

3.7 Adding support for new devices

If an error message is shown during compilation, saying that the device is not supported by the SMBus driver, support for the device can be added manually. The device needs a TWI module, and a 16 bit Timer/Counter1 (only needed for the LEDS). This can be done by editing the ‘device_specific.h’ file.

‘device_specific.h’ contains a long list of device specific #defines. The Atmega16 definition looks like this:

```
#elif defined(__ATmega16__)
#define TIMER_OVF_VECT    TIMER1_OVF_vect
#define TWI_VECT          TWI_vect
#define TIMSK1            TIMSK
```

Adding support for a new device only requires a new entry like the one above. The meaning of each symbol is described in Table 5.

Table 5. Device specific symbols

Symbol	Meaning
TIMER_OVF_VECT	Name of Timer/Counter1 overflow interrupt vector.
TWI_VECT	Name of TWI interrupt vector
TIMSK1	Name of register containing the Timer/Counter1 overflow interrupt enable bit (TOIE1).

3.8 Code size

See documentation for code sizes.

4 Table of Contents

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature

2583A-AVR-10/05