slaves, e.g. general purpose peripherals or application specific circuits. But also multi-master systems with arbitration and collision detection are possible.

Data on the I2C-bus can be transferred at rates of up to 100 kbit/s in standard mode, up to 400 kbit/s in fast mode and up to 3.4 Mbit/s in high-speed mode. A slow slave may stretch the clock period. The number of devices connected to the same I2C-bus is limited only by a maximum bus capacitance of 400 pF.

The module relieves the CPU from many time-critical tasks.

The clock for the kernel of the I2C module is derived from the system clock via a prescaler. An additional fractional divider generates the desired bit rate. The exact timing of the I2C-bus signals can be adjusted.

A FIFO is used for data transfer between CPU and I2C module during transmission and reception. This allows writing and reading of multiple bytes. Data alignment and data sizes can be configured.

Six separate interrupt requests are available: for filling or emptying the FIFO, for reacting on certain protocol events and for handling of errors.

34.3 Functional Description

Description of the kernel, module implementation and integration.

34.3.1 I2C Kernel Description

Functional description of the I2C kernel.

34.3.1.1 I2C Protocol

Data is transmitted bit-by-bit on line SDA in conjunction with the clock on line SCL. To start communication, a master device generates a so called start condition. Subsequently data transfer starts. Therefore the data on the SDA line must be stable during the high period of the clock and may only change during SCL low phase. After all data have been transferred, the master closes transmission with a stop condition (see **Figure 325**).

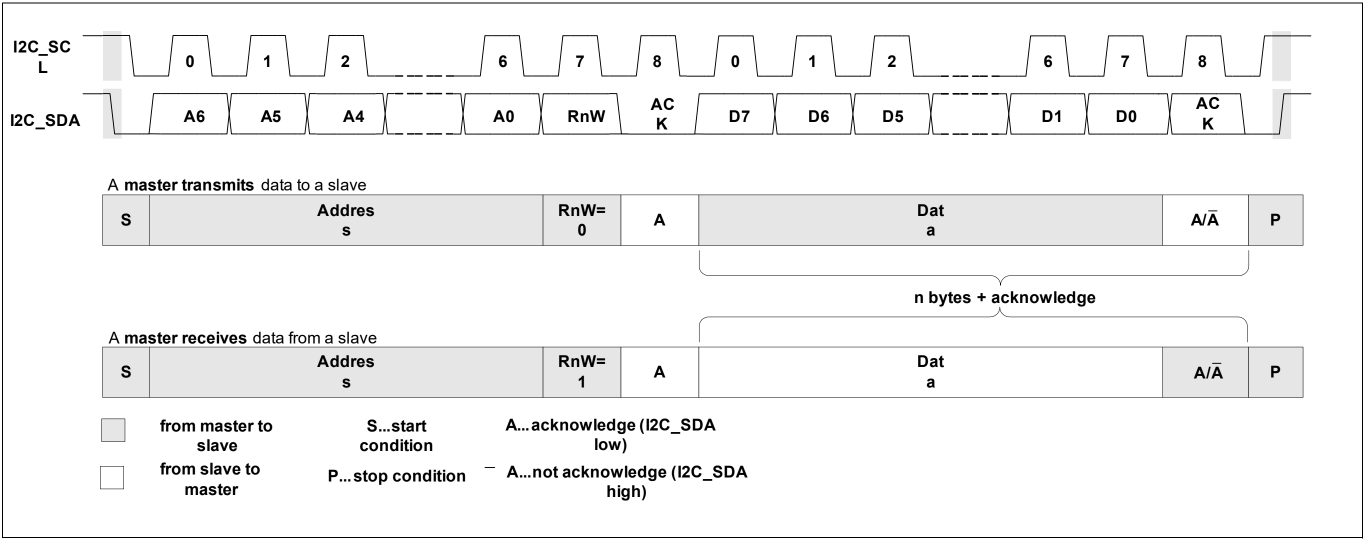

Figure 325 A Complete I2C Data Transfer

Start, Restart and Stop Conditions

A start condition is indicated by a high to low transition on line SDA while SCL remains high, a stop condition is defined by low to high transition under the same condition of high level on line SCL. The bus is considered to be busy after a start condition and to be free again after a stop condition. To allow continuous data transmission or reception without first generating a stop condition, a third condition has been introduced - the restart condition.

The bus stays busy if this condition is generated and the same or another slave can be addressed. Start and restart condition are functionally identical.

Address Transmission

After the start condition a slave address is sent. Normally the address is 7 bits long followed by an 8th bit that is a data direction bit (Read/not Write see **Figure 326**). A zero indicates a transmission (write operation); a one indicates a request to data (read operation). Following the address transmission the addressed device responds with an acknowledge (low on line SDA). If no acknowledge is returned, the master can then generate either a stop condition to abort the transfer or a repeated start condition to start another new transfer.

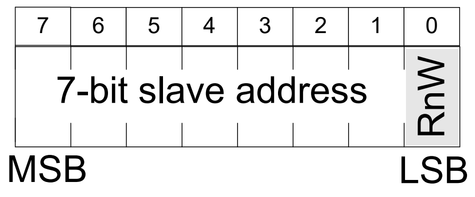

Figure 326 Address Byte Composition

Two groups of eight addresses are reserved for purposes, as shown in **Table 296**. For example the bit combination “11110XX” is reserved for 10-bit addressing mode.

Table 296 Reserved I2C-bus Addresses

Address	RnW	Description
0000000	0	General call address
0000000	1	Start byte - no device is allowed to acknowledge
0000001	x	CBUS address - I2C-bus devices may not respond
0000010	x	Reserved for different bus format
0000011	x	Reserved for future use
00001XX	x	High-speed mode master code
11111XX	x	Reserved for future use
11110XX	x	10-bit slave addressing mode

General Call

A general call is characterized by transmission of address “0000000” and RnW bit = 0. It is used for addressing every device connected to the I2C-bus. However, if a device doesn’t need any of the data supplied within the general call structure, it can ignore this address by not issuing an acknowledgement. Otherwise it acknowledges this general call and the following data bytes (if required) and behaves like a slave-receiver. The received data (e.g. programmable part of slave address, address of I2C-bus master) has to be handled by software. For further information the reader is referred to the I2C-bus specification version 2.1 [1].

Data Transmission

Each byte transferred over the I2C-bus has to be 8 bits long whereby the number of bytes transmitted per transfer is unrestricted. Bytes are transferred MSB (Most Significant Bit) first. If the slave cannot receive or transmit a data byte until it has performed another function, it can hold the clock line SCL low (wired AND connection) to force the master into a wait state. Data transfer continues when the slave releases the clock line.

Acknowledge

Each transferred respectively received byte has to be followed by an acknowledge bit that is set by the recipient (master or slave depending on the transmission direction) onto the data line SDA. The acknowledge-related clock pulse is generated by the master. The device that has to set the acknowledge bit must pull down the SDA line during the acknowledge clock pulse so that it remains stable low during the high period of this clock pulse. Therefore in case of a slave the clock line may be held low until an acknowledge is released.

When the recipient of data doesn't acknowledge the data, for example because it was unable to receive the data or to transmit any data, the data line must be left high. A master-receiver that does not acknowledge the data from a transmitting slave device thereby tells the slave to stop transmission. The master then generates a stop condition to abort transfer or a repeated start condition to continue.

Clock Synchronization

To transfer messages a master generates its own clock on the SCL line. Data is only valid during the high period of the clock. Therefore data on line SDA has to be stable during this period and may only be changed during the low period of clock.

Two special cases must be considered. First, slave devices (or master devices operating as slaves) never generate clocks but are permitted to delay them. According to **Figure 327**, if device 1 as master device for example sets line SCL to low, device 2 as slave device may extend the low phase by setting its SCL output to low. A device such as a microcontroller with limited hardware for the I2C-bus can thereby slow down the bus clock and adapt the master to the internal operating rate of this device.

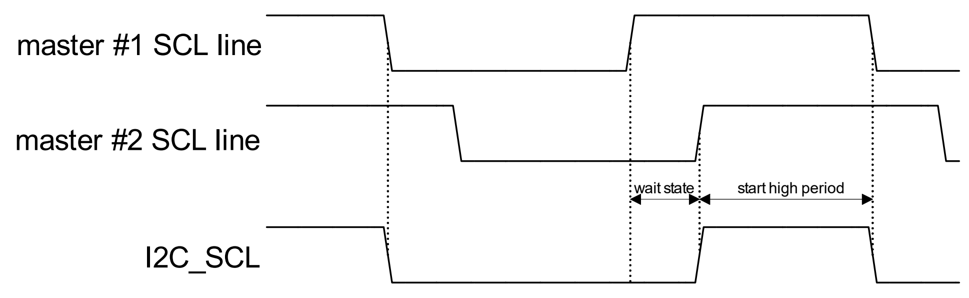

Figure 327 Clock Synchronization

Second, if several master devices access the I2C-bus at the same time, the SCL line is influenced by more than one device. As shown in the previous example, the SCL line will be held low by the device with the longest SCL low period. Devices with shorter periods enter a wait state during this time. In this way a synchronized SCL clock is generated.

Arbitration

To allow multi-master mode operation, bus arbitration is required. A master may start a transfer only if the bus is free. Two or more masters may generate a defined start condition simultaneously within a minimum hold time.

Arbitration takes place on the SDA line while SCL line is high, when one master transmits a high level while another master transmits a low level. Because of the wired AND functionality, the low level dominates the high level. As result the device that was trying to transmit a high level recognizes a different level on the SDA line and switches off its SDA and SCL output (see **Figure 328**).

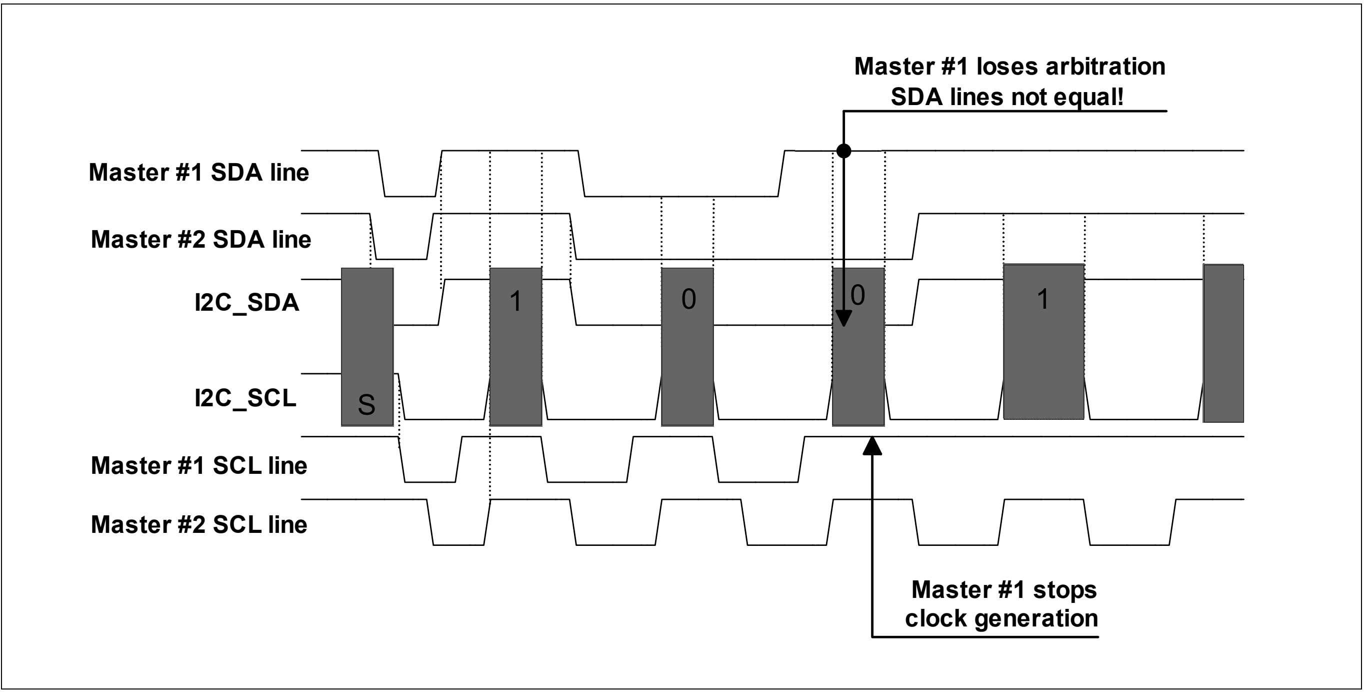

Figure 328 Example of an Arbitration Procedure

Arbitration may occur in different stages of transmission. A different level may potentially first appear during comparison of the address bits. If the masters are each trying to address the same device, arbitration continues with comparison of the RnW bit and subsequent data bits (if the masters are transmitting) or acknowledge bits (if the masters are receiving). Because information on the I2C-bus is determined by the winning master, no information is lost during the arbitration process. Since control of the I2C-bus is decided only by comparison of the data transmission stream over line SDA, there is no central master or any order of priority on the bus.

I2C-bus Formats

With the described rule types the formats shown in **Figure 329** are possible.

- Master-transmitter transmits to slave-receiver and stops after not-acknowledge.
- Master reads slave immediately after transmission of the RnW bit (high level) and the acknowledge from the slave, whereas the master becomes master-receiver and the slave becomes slave-transmitter.
- In combined format the just described regular sequences are handled, except for the repeated start condition instead of the stop condition between two sequences.

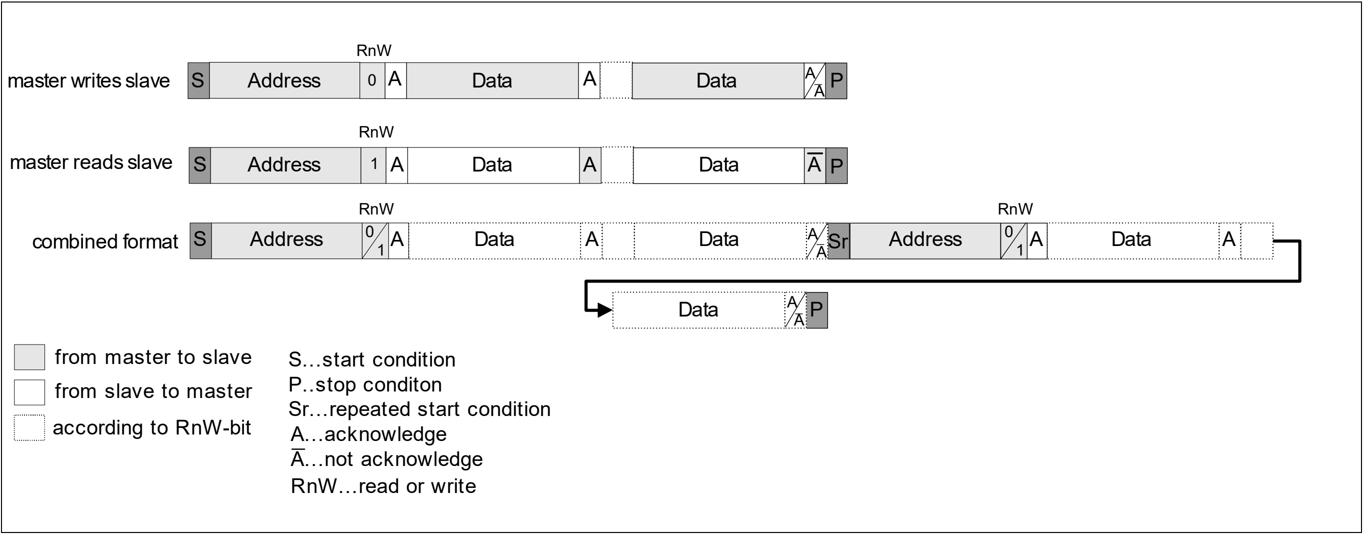

Figure 329 Types of I2C-bus Formats

10-bit Addressing Mode

The I2C address area is restricted to 112 applicable addresses with the 7-bit address mechanism described above. It turned out, that more combinations were required to prevent problems with the allocation of slave addresses for new devices. This problem was resolved with the 10-bit addressing scheme that is compatible with 7-bit addressing. Both modes may be used simultaneously. The following formats are possible:

To use 10 bits for addressing, the preamble address "11110XX" with RnW bit = 0 has to be used, including the two most significant bits "XX" of the 10-bit address, followed by an acknowledge from at least one 10-bit address matching slave and the second address byte containing the eight least significant bits. Only the (combined) 10-bit address matching slave now acknowledges. Data packages may now be transmitted to the addressed slave until stop condition or restart condition. If the master needs to read the slave, a combined format with a restart condition is necessary to

- First transmit the 10-bit address (first data package after preamble address contains 2nd address byte, RnW bit is 0) and (after the restart condition)
- Receive the requested data packages. Note that after the restart condition the preamble address "11110XX" has to be sent one more time, now with RnW bit = 1; the matching slave remembers that it was addressed before (see **Figure 330**). In case of a non matching preamble address, a not-acknowledge is returned.

The slave remains active as slave-transceiver until a stop condition or a restart condition occurs. According to the value of the following RnW bit a new slave may be addressed or the same slave is requested to transmit more data. Additionally the following combined formats (and mixtures of these formats) are allowed (**Figure 330**):

- A master transmits data to a slave and then reads back data from the same slave. The transfer direction is changed after the restart condition.
- A master transmits data to one slave and then transmits data to another slave.
- 10-bit and 7-bit addressing combined in one serial transfer. After restart condition the new addressing mode is selected by the address itself.

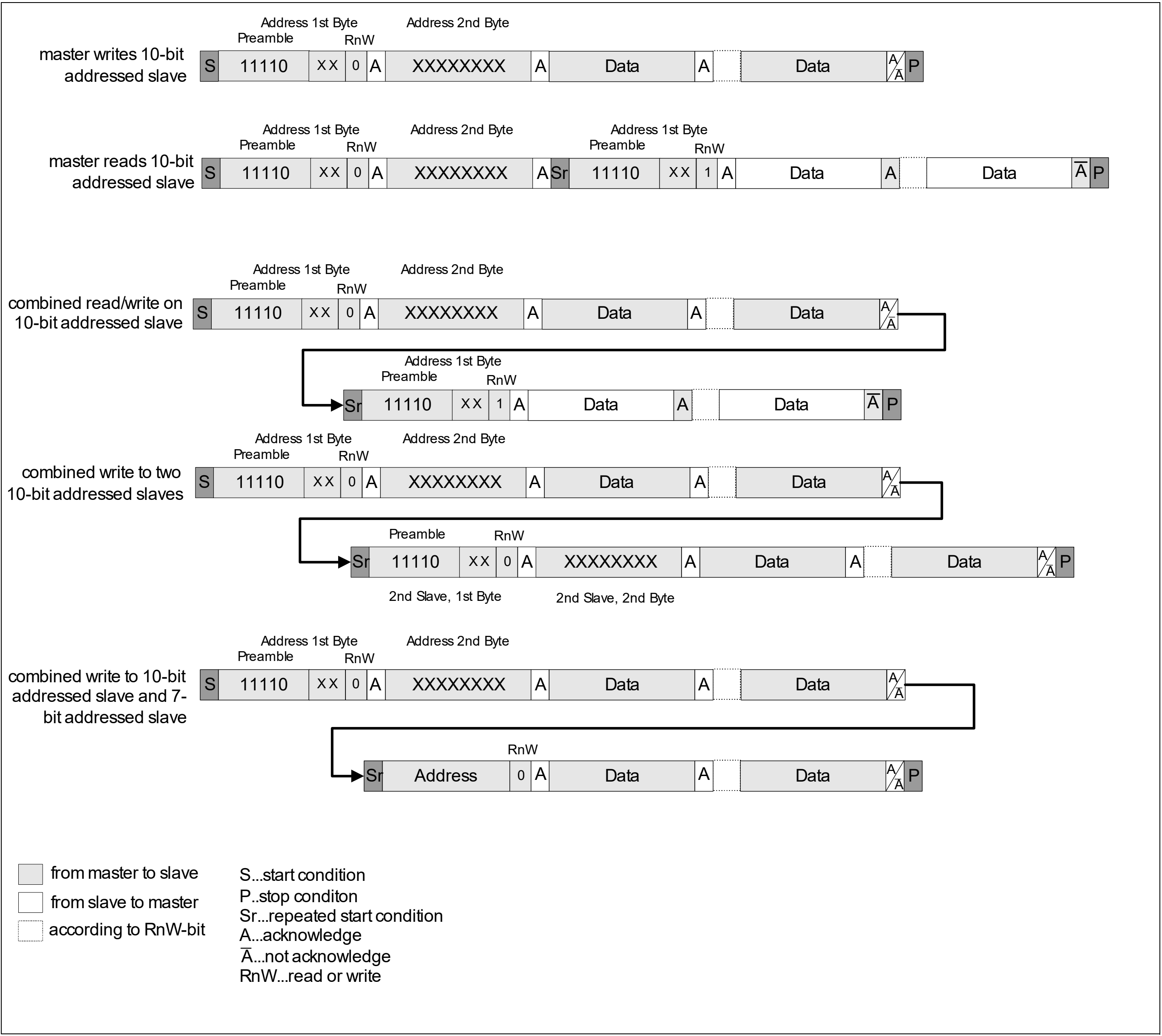

Figure 330 Types of I2C-bus 10-bit Address Formats

High-speed Mode

To support higher baud rates on the I2C-bus (up to 3.4 MBaud) the specification introduces the high-speed mode. To communicate at this higher baud rate the protocol uses a so called master code in place of the address byte to indicate that a master wants to change to the high-speed mode.

Special attention must be paid to meet the requirements of rise and fall times of the signal edges and other timing characteristics when working in high-speed mode. Also it is necessary to disconnect I2C-bus devices which are not able to follow the fast communication.

After the master sends this master code, no device is allowed to acknowledge it. The involved devices change to high-speed mode and the master starts the communication. The high-speed mode transmission behavior is the same as at lower baud rates with the exception that only restart conditions and stop conditions appear (no start condition). Since the master code determines a unique master device to control the bus, the arbitration process is finished after the master code sequence is put on the bus and must not be continued when working at the new communication speed.

Improvements to the regular I2C-bus specification for high-speed mode are given in section 13.1 of the I2C-bus specification version 2.1 **[1]**.

For high-speed mode there exists a separate configuration register **FDIVHIGHCFG** to program the baud rate; see also **Baudrate Generation for Master Mode**.

34.3.1.2 References

For more information, see the following documentation:

[1] I2C-bus specification version 2.1 standard (released January-2000)

34.3.1.3 Functional restrictions

There are some functional restrictions in this I2C module.

Multi-master mode - Master collision

The Multi-master mode is supported with a limitation/restriction in the following situation:

- The I2C module operates in master mode

- Another master wants to access/address the I2C module as slave

The resulting "master collision (problem)" cannot be handled correctly because the incoming address sent by the other master cannot be evaluated. So, if collision occurs second master has to restart access and retransmit the data.

Multi-master mode - Hold Time for the (repeated) Start condition

In a special situation this design step of the I2C module violates this timing specification by ~30% for Standard mode and by ~18,6% for Fast mode (min . 2.8 µs instead 4.0 µs for Standard mode and min. 0,488 µs instead 0,6 µs for Fast mode).

This occurs when data is written into TX FIFO, the I2C module is ready to start transmission an another master starts driving its startbit shortly before the I2C module starts driving. In this case the I2C Finite State Machine tries to win arbitration by reloading approximately half period in baudrate generator, so next SCL clock edge of the I2C module comes earlier than defined by t_7.

High-speed mode

The standard for the clock duty cycle ratio of the I2C high speed mode is 1:2. See **Chapter 34.3.1.4** in order to achieve the best duty cycle ratio for high speed mode. Note that not for every f_{I2C} frequencies 0% duty cycle ratio deviation can be achieved

34.3.1.4 Clock and Timing Control

The I2C module offers the following features to control clock and timing:

- Module clock control for integration logic clock and I2C kernel clock (see **Section 34.3.2.2**)
- Baudrate generation
- I2C signal timing adjustment

34.3.1.4.1 Baudrate Generation for Master Mode

A baudrate generation unit in the I2C kernel generates the SCL signal from the kernel_clk, controlled by settings of parameters **INC**, **DEC**, and **FS_SCL_LOW** in registers **FDIVCFG** (normal and fast mode), **FDIVHIGHCFG** (high speed mode), and **TIMCFG**.

The I2C standard has some special requirements for the clock, like

- asymmetric duty cycle
- slave or other masters are allowed to delay the clock by extending the low period. For this reason, a master must check, if any slave holds the clock line low after the master set the clock line to high. This implies that the state machine of a master needs at least 1 internal clock cycle delay between setting the clock to high and reading it back to check if it actually reached high level already. In addition more delay cycles can be introduced by the time constant of the pull up resistor and the line capacity. In any case, this protocol has the consequence that the actual baud rate will always be lower than the nominal baud rate.

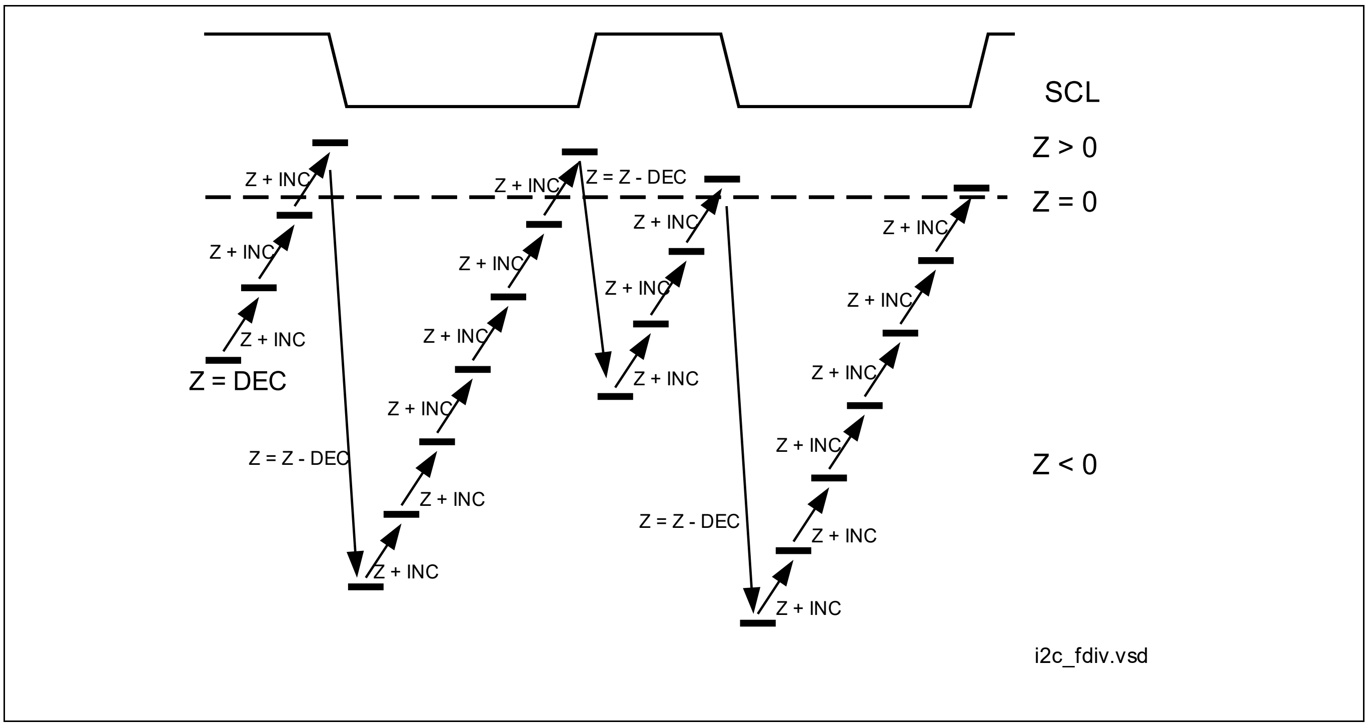

Figure 331 FDIV Principle - example for high speed mode

The operation principle of I2C fractional divider is shown in **Figure 331**. A counter Z is set to value **DEC** and is incremented using value **INC**. When counter reaches Z=0 it will be decremented using value **DEC**. With each decrementing operation SCL is toggled to achieve the required SCL waveform. Correction values are also applied to this mechanism but not shown above.

The formulas in the **Table 297** show how to calculate **DEC** and **INC** for a configured f_{SCL} frequency and a given f_{I2C}. **EN_SCL_LOW_LEN**, **FS_SCL_LOW** and **SCL_LOW_LEN** are bit-fields in register **TIMCFG**.

Table 297 I2C Baudrate Generation Configuration for Master Mode

MODE	EN_SCL_LOW_LEN	FS_SCL_LOW	f_{SCL}	Low Time Correction	Duty Cycle
Standard	0	0	$$fSCL = \frac{INC}{2DEC + 3INC} \times fI2C \quad (34.1)$$	-	-
Fast	0	1		-	-
	1	X		[1)] $$SCL\_LOW\_LEN = \frac{3INC}{2} \quad (34.2)$$	-
High speed	X	X	$$fSCL = \frac{INC}{5DEC + 2INC} \times fI2C \quad (34.3)$$	-	1:2

1) This formula can be use to achieve a 50% duty cycle. **SCL_LOW_LEN** can be program with other values, but not greater than **DEC**.

In **Table 298** some example settings to program **INC**, **DEC** and LTC (bit-field **SCL_LOW_LEN**) are given and information about the duty cycle is shown.

Table 298 I2C INC/DEC Settings for Master Mode

I2C MODE	f_{I2C} [MHz]	INC	DEC	SCL_LOW_LEN	Duty Cycle [%]	f_{SCL} [MHz]
Standard	66.6	1	332	-	56.3	0.099
Fast	66.6	2	170	3	50	0.384
High speed	66.6	46	162	-	30.6	3.399

If a slave device is sensitive for violation of max frequency according I2C standard (eg. 3.4 MHz in High-speed mode) it is recommended to select a target SCL frequency with some margin to this limit.

34.3.1.4.2 Baudrate Generation for Slave Mode

Baudrate generation is mainly used for master mode, but there is one corner case where a support of the baudrate generation is required in slave mode: when I2C is in slave mode with a pending transmit data request and FIFO is currently empty then SCL is kept low until FIFO is filled after some time. Then transmit data is sent out immediately. SCL is kept low for a min. time of t_4.

In order to keep the set-up time,t_4 , according to I2C standard, the baudrate generation is used to guarantee the delay on SCL. The formulas in the 3rd and the 4th column of **Table** show how to configure **DEC** and **INC** for a given f_{I2C} and the intended set-up time. The formulas in column **Resulting t4 [µs]** define the real value of t_4 with the selected **DEC** and **INC** values.

The same value for **DEC** and **INC** as in master mode is not recommended and would lead to additional delay, as requested by the standard, in most cases.

Table 299 I2C Baudrate Generation Configuration for Slave Mode

MODE	min t_4[µs]	DEC / INC	Resulting t_4 [µs] [1][2]
Standard	0.25	$$\mathrm{DEC} = \frac{8}{9}((\mathrm{fI2C}) \times (t_4) \times \mathrm{INC} + \mathrm{INC}) \quad (34.4)$$	$$C = \mathrm{INT}\left(\frac{\mathrm{DEC} + \mathrm{DEC(div)8}}{\mathrm{INC}} - 1\right) \quad (34.5)$$ $$\frac{C}{\mathrm{fI2C}} = t_4 \quad (34.6)$$
Fast	0.1	$$\mathrm{DEC} = \frac{4}{5}((\mathrm{fI2C}) \times (t_4) \times \mathrm{INC} + \mathrm{INC}) \quad (34.7)$$	[3] $$C = \mathrm{INT}\left(\frac{\mathrm{DEC} + \mathrm{DEC(div)4}}{\mathrm{INC}} - 1\right) \quad (34.8)$$ $$\frac{C}{\mathrm{fI2C}} = t_4 \quad (34.9)$$
High speed	0.01	$$\mathrm{DEC} = \mathrm{INC} + 1 \quad (34.10)$$	$$C = \mathrm{INT}\left(\frac{\mathrm{DEC}}{\mathrm{INC}} + 1\right) \quad (34.11)$$ $$\frac{C}{\mathrm{fI2C}} = t_4 \quad (34.12)$$

1) The used abbreviation INT is the integer function.
2) The operator div denotes the integer division: a div b = greatest integer not greater than a/b.
3) SCL_LOW_LEN can not be greater than DEC.

In the following table some example settings are given to program **INC** / **DEC**.The used abbreviation INT is the integer function.

Table 300 I2C INC/DEC Settings for Slave Mode

I2C MODE	f_{I2C} [MHz]	INC	DEC	t_4 [ns]
Standard	66.6	20	314	250
Fast	66.6	25	160	105.1
High speed	66.6	250	251	30

The minimum kernel frequency is 8MHz for Standard and Fast Mode and 55MHz for High Speed Mode.

34.3.1.4.3 I2C Timing Adjustment

The I2C standard includes a number of requirements for the SCL and the SDA timing. These cannot always be fulfilled by the timing of the pads. Therefore, additional timing adjustment options are provided in the controller logic.

Various timings of signals can be shifted by multiples of kernel_clk cycles, as defined in register **TIMCFG**.

These settings offer a wide range of timing compensation. Not all of the values can be used and will result in a valid I2C timing, even clock or data hang-up is possible. Detailed analyses should be done - eg. during chip evaluation - taking actual kernel_clk and external timing into account.

SDA_DEL_HD_DAT

- Delays SDA output data starting from SCL falling edge for a defined number of kernel_clks.
- Must not exceed SCL low period to keep setup time for collision detection at next SCL rising edge.

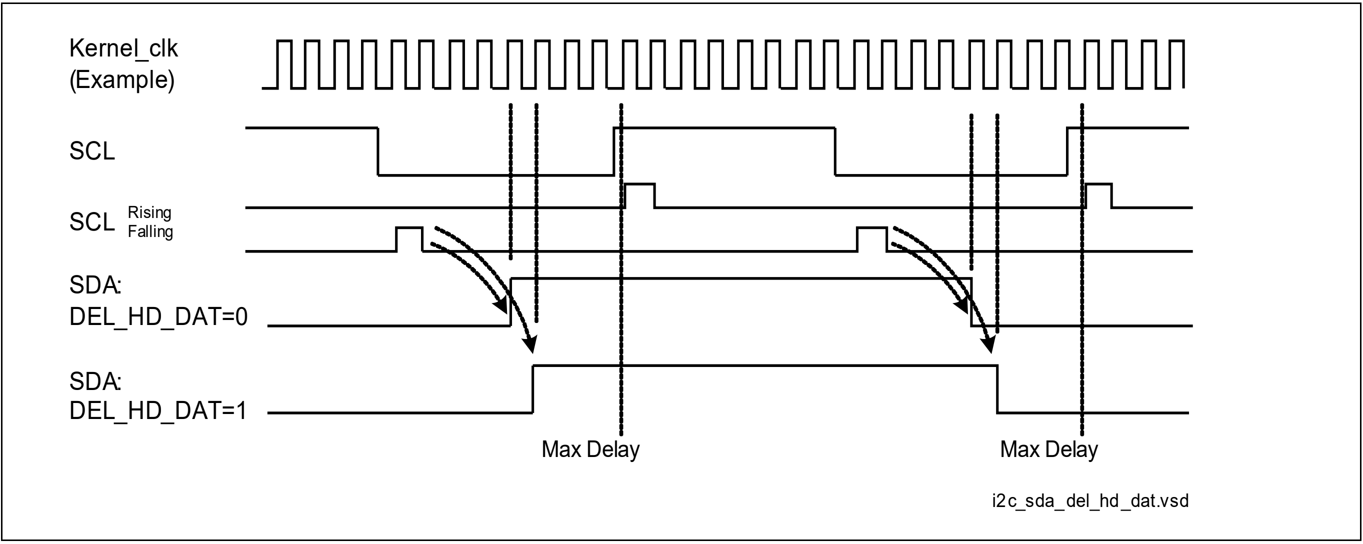

Figure 332 SDA_DEL_HD_DAT

High Speed Mode corrections

In **Figure 333**, the base period and delay corrections are shown for the entering and stop phases of HS mode.

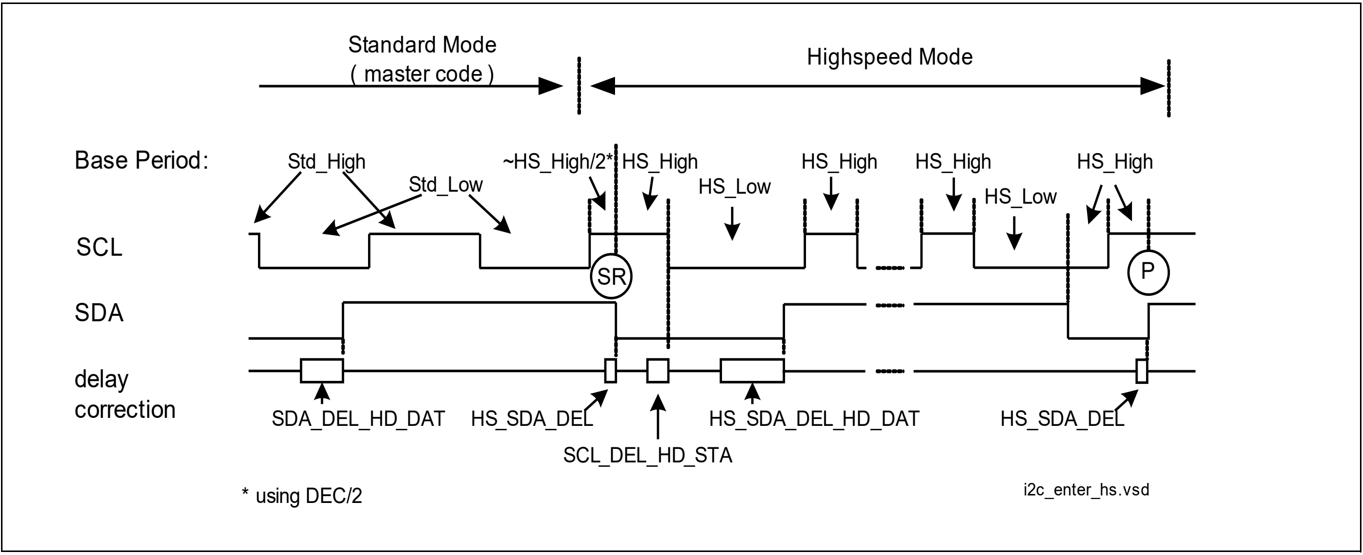

Figure 333 High Speed Mode corrections

34.3.1.5 I2C Kernel Control Logic

The I2C kernel can work in four different modes:

- Master-transmitter
- Master-receiver
- Slave-transmitter
- Slave-receiver

A state machine description can give a good and fast overview of the functionality. Also nearly all possible conditions can be rebuilt by stepping through the state machine and most of the error handling is covered by it. So this method is used to describe the I2C kernel.

Beginning with the top-level state machine, the starting up procedure is covered and together with the slave process and the master process sub-state machines, the active operations are comprehensible. These three state machines and their state transitions are described in the following.

As the I2C-bus working principle is packet orientated, the FIFO is typically configured as flow controller and so it issues the burst- and single-requests. This configuration is established by programming the bits **RXFC** and **TXFC** of the **FIFOCFG** register. Writing to the **TPS** bit-field of the **TPSCTRL** register starts the transfer.

Notes

1. *Peripheral is flow controller. As the I2C working principle is packet oriented, the FIFO is configured as flow controller and with this issues the burst- and single-requests. This configuration is established by programming the bit FC of the FIFOCFG register. Writing to the TPS-bitfield of the TPS_CTRL register starts the transfer.*
2. *Peripheral is not flow controller. In principle it is possible to use the FIFO in non flow controller mode. When the FIFO is not configured to operate as flow controller, the behavior of the FIFO operation has to be considered. The update of the FIFO fill level is only done when a stage is filled completely. In case of a (not protocol compliant) stopped transfer of the transmitter, the device is not able to evaluate the valid bytes since the FIFO issues interrupts/updates fill level only when a stage is filled completely. To prevent this behavior, the FIFO can be, for example, programmed to word alignment, so every received byte issues an interrupt/update. Anyway the software has to take care of such upcoming not protocol compliant data transmission stops, and has to handle this for example by programming an appropriate time-out.*

The Top-level State Machine

Figure 334 shows the top-level state machine. There are four states in the top-level state machine:

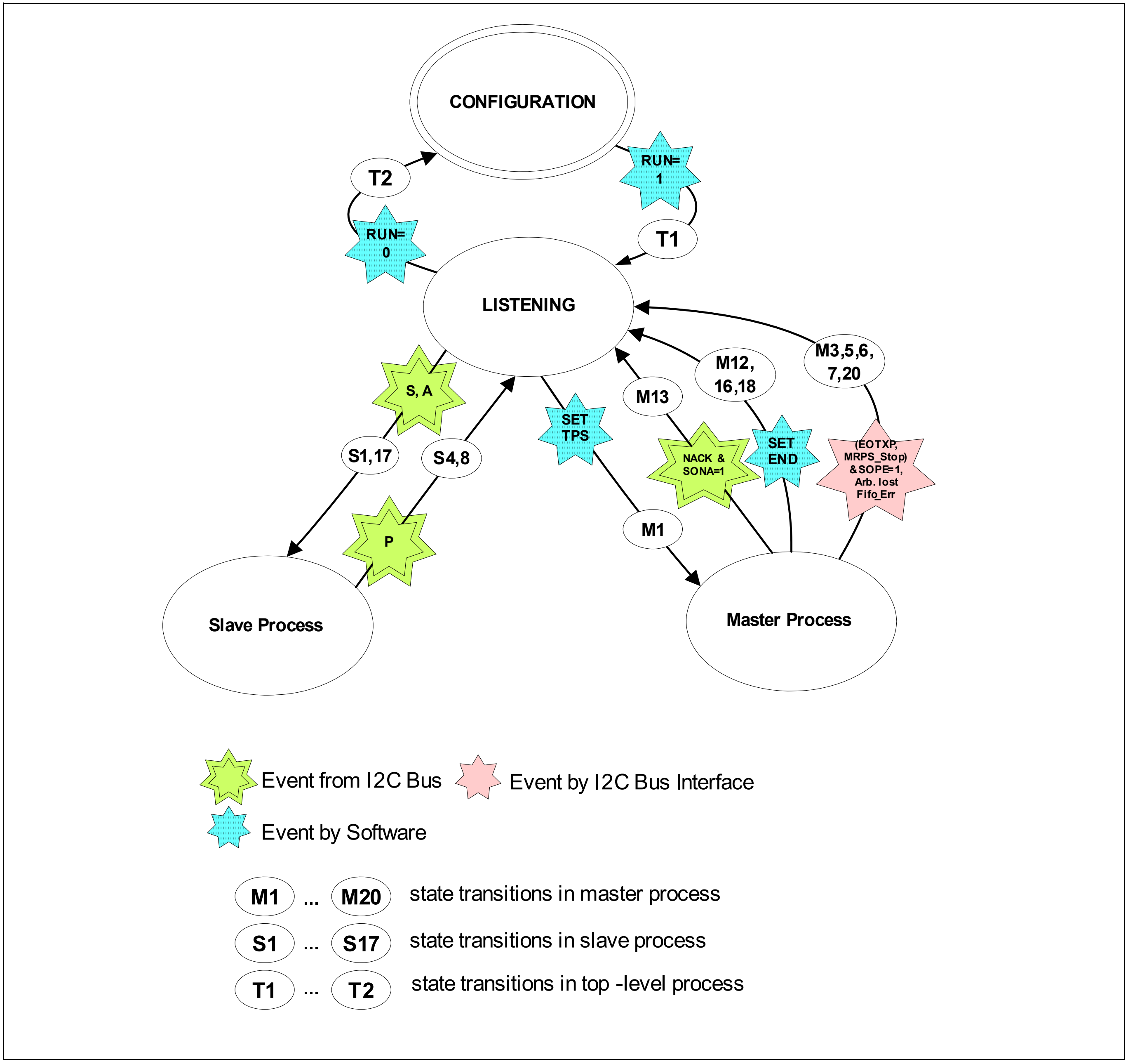
CONFIGURATION
T2
RUN=1
RUN=0
T1
LISTENING
S, A
M12, 16,18
M3,5,6, 7,20
M13
SET TPS
S1,17
S4,8
NACK & SONA=1
SET END
(EOTXP, MRPS_Stop) &SOPE=1, Arb. lost Fifo_Err
M1
P
Slave Process
Master Process
Event from I2C Bus
Event by I2C Bus Interface
Event by Software
M1 ... M20 state transitions in master process
S1 ... S17 state transitions in slave process
T1 ... T2 state transitions in top -level process

Figure 334 Top-level State Machine

CONFIGURATION: The peripheral is inactive; this is indicated by the **RUN** bit in **RUNCTRL** register, which is set to 0. In this state the peripheral should be configured by software by writing appropriate data words to the **ADDRCFG**, **FDIVCFG**, and **FIFOCFG** registers. This determines the following features:

- Peripheral is master or slave
- Address when accessed by another device
- Answering behavior to master calls and general calls
- Operating speed on the I2C-bus
- FIFO behavior

LISTENING: The peripheral has been activated by software. This state is entered in master and slave mode. The peripheral has to observe the bus for a certain time[1] to ensure that there is no running transmission on the bus after this state has been entered. If no transitions on the I2C-bus are detected during this time period, the bus is supposed to be free. Otherwise the bit-field **BS** (bus status) in register **BUSSTAT** is set to 01_B (bus busy). Further functionality depends on the configuration: If the peripheral is a slave, only the slave process can be entered. If the device works as master, slave and master processes can be entered.

SLAVE PROCESS: A Start condition has been detected. The device is waiting for commands from the I2C-bus. This state represents the operation when acting as slave-transmitter or slave-receiver and is itemized in an own sub state machine.

MASTER PROCESS: The device is able to control the bus and work either as master-receiver or master-transmitter. Before the master can control the bus it has to set its clock frequency by programming its fractional divider with the appropriate value by writing it to the **FDIVCFG** register (when the I2C-bus is also operating in high-speed mode the appropriate values for INC and DEC have to be written to). The master process state is itemized in an own sub state machine.

The transition between the three states on the top-level are:

T1. The **RUN** bit has been set to 1 by software. The peripheral is activated and ready to observe the I2C-bus or react on transmission start commands.

T2. The **RUN** bit has been set to 0 again. I.e., because of new configuration effort, the peripheral is switched back to CONFIGURATION mode.

IMPORTANT: When “turning off” the peripheral it must be assured that no I2C-bus control responsibility is owned by the interface! Therefore the shut off process has to change the peripheral into a “secure” state and free the I2C-bus if necessary (generate a stop condition if in master mode). There are signals provided by the interface block which can be used to guarantee a secure shut off when turning off the I2C kernel.

1) One period of 1/100 kHz should be sufficient.

Slave Process Sub-state Machine

Figure 335 shows the slave process sub-state machine. There are 5 plus 1 (the listening state belongs to master and slave) different states in this sub-state machine:

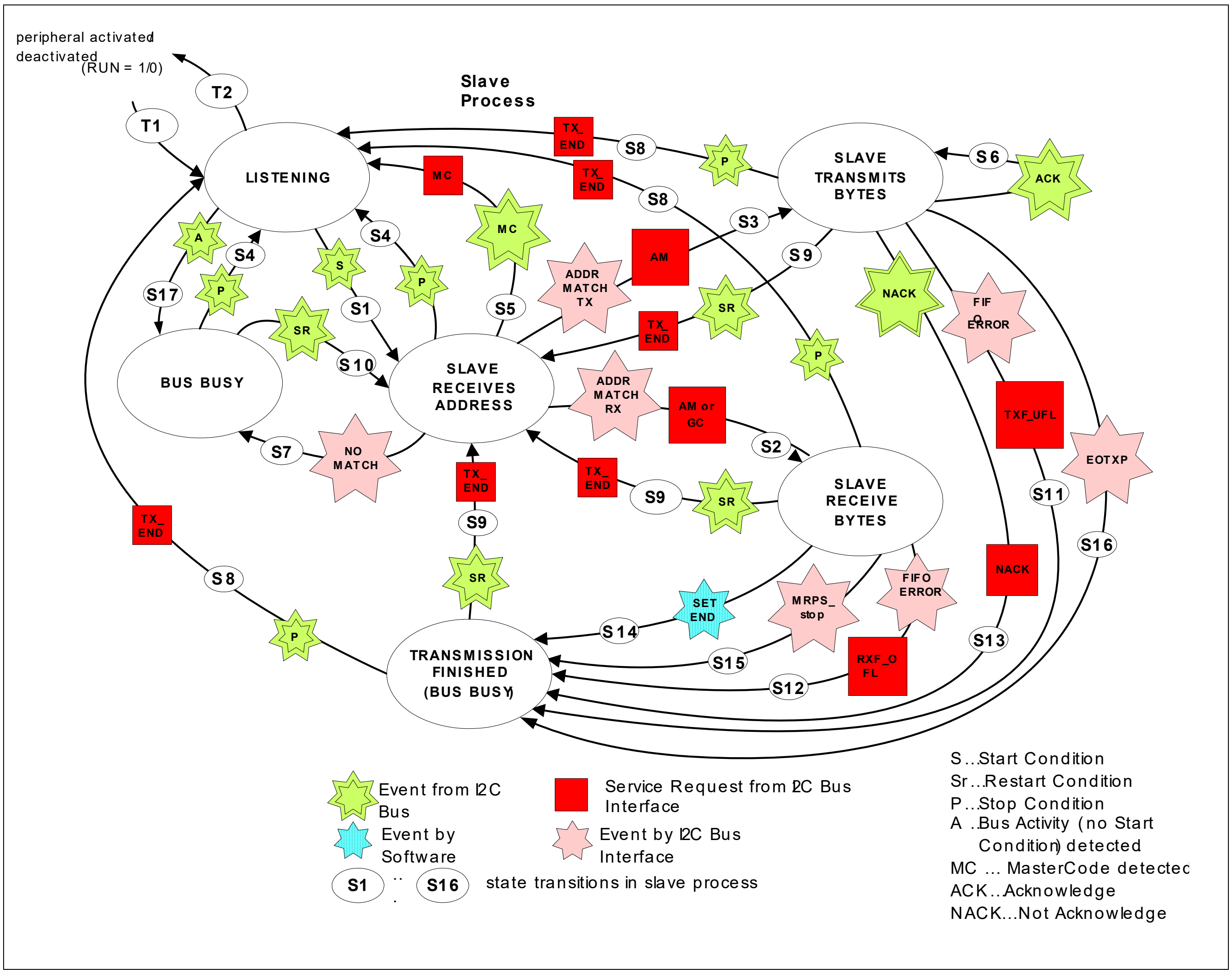

Figure 335 Slave Process Sub-state Machine

LISTENING: This state has been previously described (**Figure 334**). If the device is configured as slave, the module is listening only to the bus activity.

SLAVE RECEIVES ADDRESS: The module is shifting in the serial bits from the I2C-bus and acknowledges if the received byte(s) (first one in 7-bit address mode, first and second one in 10 bit address mode) match the adjusted address. During address matching the slave-receiver has the possibility to slow down the master clock by holding the SCL line low after each bit, since this can be used as a bit-by-bit handshake procedure.

Note: The address byte(s) is (are) not shifted into the FIFO.

SLAVE RECEIVES BYTES: The module is shifting in the serial bits from the I2C-bus and acknowledges when the whole data byte has been transported to the FIFO. The slave-receiver has the possibility to slow down the master clock by holding the SCL line low after each bit since this can be used as a bit-by-bit handshake procedure.

SLAVE TRANSMITS BYTES: The slave is allowed to hold down the SCL line when preparing the data for transmission. When valid data is available the slave releases the SCL line and waits for the clock cycles from the master. The interrupt routine which proceeds when entering this state may write to the **TPSCTRL** register to initiate a transmission. This is only allowed if the bit-field **BS** (bus status) in register **BUSSTAT** in combination with the **RnW** clearly identifies the slave-transmitter mode of the peripheral. Otherwise writing to this register has no effect. First the I2C kernel gets a byte from the FIFO and puts it into a shift register. The module is then putting single data bits onto the I2C-bus and releases the SDA line after the 8th bit is sent.

*Note: Since the data flow control responsibility is task of the master device the slave should not stop transmitting bytes until the master tells it to do so. Therefore the **TPS** value in register **TPSCTRL** should be large enough to not run out of data bytes. In case of running out of data, the TRANSMISSION FINISHED state is reentered. The I2C data line is released to high level. The master continues receiving wrong values (0xFF).*

BUS BUSY: The bus is busy and the module is not involved in any data transfer. This is also the entry state when the bus is not free when starting up procedure is taking place (see state S17).

TRANSMISSION FINISHED (BUS BUSY): The module was involved in any data transfer. In distinction to the BUS BUSY state, the slave is still addressed by a master. Every time this state is left, caused due to a stop/restart condition, a TX_END request is generated as indication that the previous transmission has been closed.

The transitions which are connecting the several states are in detail:

S1: The module is detecting a start condition on the bus. It has to set the bit-field **BS** (bus status) in register **BUSSTAT** to 01_B. When this situation is occurring the I2C kernel is initialized since this indicates a new data transmission.

S2: The address which has been received matches the one stored in the address field in the **ADDRCFG** register. The **RnW** bit (8th bit in the first received byte) was set to 0. This means that the master wants to write to the slave. The corresponding bit **RnW** in register **BUSSTAT** is set to 1, i.e. the device continues working as slave-receiver. Also an acknowledge has to be set on the I2C-bus in the appropriate place (9th clock cycle). The I2C kernel sets the bit-field **BS** (bus status) in register **BUSSTAT** to 11_B and generates an AM (address match) request.

If the general call (GC) address matching is enabled by setting bit **GCE** in the **ADDRCFG** register, the device has to react also when it receives a general call (0x00). The GC request is issued and also an acknowledge has to be set on the I2C-bus when the GC is detected.

S3: The address which has been received matches the one stored in the address field in the **ADDRCFG** register and the SDA line was at a high level when the **RnW** bit was sent. This means that the remote master wants to read from the slave. The corresponding bit **RnW** in the **BUSSTAT** register has to be set to 0 (write). Also an acknowledge has to be set on the I2C-bus in the appropriate place (9th clock cycle). The I2C kernel sets the bit-field **BS** (bus status) in register **BUSSTAT** to 11_B and generates an AM (address match) request. The device is now working as a slave-transmitter.

S4: The device detects a stop condition on the I2C-bus and switches back to SLAVE LISTENING state. No interrupt is generated after the device was not addressed. The bit-field **BS** (bus status) in register **BUSSTAT** is set to 00_B (bus free).

S5: If the device is configured via bit MCE (master code enable) in the **ADDRCFG** register to react on a master code, the device can change baud rate to high-speed (Hs) mode when a remote master sends a master code. No device is allowed to acknowledge this master call. The interrupt routine which is called (MC request issued), has to handle the clearing of the interrupt bit in register **PIRQSC**. The LISTENING state is reentered. From now on the device works as slave in high-speed mode until a stop condition occurs.

S6: The slave-transmitter has received an acknowledge from the master-receiver and continues transmission.

S7: After the reception of the address bytes has been finished and no address match has taken place, the device changes to BUS BUSY state.

S8: The device has detected a stop condition on the I2C-bus after/while it was addressed. A TX_END (transmission end) request is generated, the bus is free again respectively the bit-field BS (bus status) in register **BUSSTAT** has to be set to 00_B again. The slave is now in LISTENING state.

S9: The slave has received a restart condition (repeated start condition) after/while it was addressed. A TX_END (transmission end) request is generated, the state switches back to SLAVE RECEIVES ADDRESS. The slave is reset again to a new transmission pending state (bit-field BS in register **BUSSTAT** = 01_B.). When the TBAM is activated (configuration register) the slave has to remember that it was addressed before because this event may be followed by a read access of the master to this device. So the address procedure after reset can be shortened and the transition S3 occurs one byte earlier (master sends 10-bit address mode byte with RnW bit = 1 (read)).

S10: A running transmission is ended by a remote master and this device wants to continue without losing control of the I2C-bus. The slave was not in transmission and resets to receive address bytes again since the new data transmission is concerning all connected slaves.

S11: The peripheral has problems to read data bytes from the FIFO (underflow/not ready). This is indicated by generating a TXF_UFL (TX FIFO underflow) request. It also changes to TRANSMISSION FINISHED state.

Note: The device is still addressed by a master (bit-field BS in register ***BUSSTAT*** *= 11_B.) until a stop or restart condition occurs.*

S12: The peripheral has problems to write the received byte into the FIFO (overflow/not ready). This is indicated by generating a RXF_OFL (RX FIFO overflow) request. The slave-transmitter puts a not-acknowledge on the I2C-bus to indicate its situation to the controlling master. This byte is not valid and will be discarded.

Note: The device is still addressed by a master (bit-field BS in register ***BUSSTAT*** *= 11_B.) until a stop or restart condition occurs.*

S13: The remote master wishes to close the connection to this slave and sets a not-acknowledge on the I2C-bus. The device generates a NACK (not-acknowledge) request and changes to TRANSMISSION FINISHED state (bit-field BS in register **BUSSTAT** = 01_B.). The interrupt routine can clear the appropriate bit via the corresponding bit in the **PIRQSC** register.

S14: The slave wants to stop the transmission and sets the **SETEND** bit in register **ENDDCTRL**. It puts a not-acknowledge on the I2C-bus after the next received byte. With this the slave gets the possibility to cancel the transmission via software intervention.

S15: The value of received bytes is equal to bit-field MRPS (maximum receive packet size) of the FIFO register **MRPSCTRL**. The FIFO part of the interface generates the signal MRPS_stop to the I2C kernel, which indicates the end of the received packet capacity. The slave produces a not-acknowledge to the master and changes to TRANSMISSION FINISHED. For more details on FIFO MRPS operation see **Section 34.3.1.6.5**.

S16: The number of transmitted bytes is equal to the TPS bit-field of the FIFO register **TPSCTRL**. The FIFO controller generates the EOTXP (end of transmit packet) signal to the I2C kernel, which indicates the last byte to transmit. After the last byte the device changes to TRANSMISSION FINISHED. For more details on FIFO EOTXP operation see **Section 34.3.1.6.2**.

Note: The master is not informed about this situation. Since the data flow control responsibility is task of the master device, it may continue receiving wrong values (0xFF).

S17: The module detects bus activity; no start condition has been detected before. A transmission may be currently in progress. The device changes to BUS BUSY state. This transition is performed if a start condition has been missed because the device was not activated (CONFIGURATION state).

Master Process Sub-state Machine

Figure 336 shows the master process sub-state machine. There are four different states in this sub-state machine:

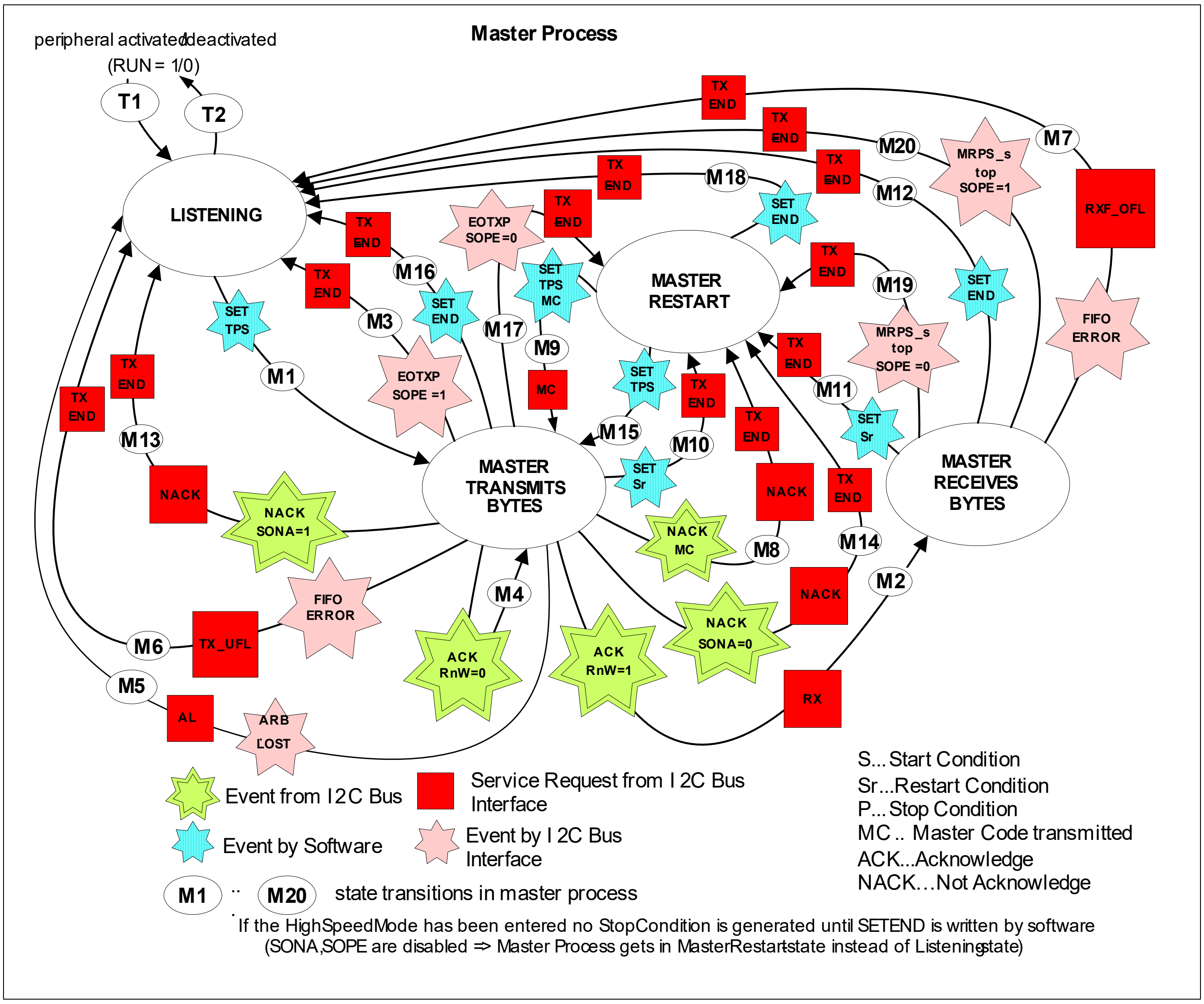

Figure 336 Master Process Sub-state machine

LISTENING: This state has been previously described (**Figure 334**). If the device is configured as master, it is listening to events either from software (writing to **TPSCTRL** register) or from the I2C-bus.

MASTER TRANSMITS BYTES: The device has started a transmission on the I2C-bus by software. The first byte after a start condition contains the address byte including the RnW bit. According to this bit (RnW bit in **BUSSTAT** register is set by hardware) the direction of the following transfer is determined. Bytes are transmitted on the I2C-bus until the FIFO is empty or a stop is initiated by software.

If bit MCE (master code enable) is set in the **ADDRCFG** register, the device has to check if the transmitted address byte is a master code.

MASTER RESTART: The device wants to restart a transmission without losing the control over the I2C-bus. The clock SCL stays low as long as no further communication is requested. The only event that should occur in this state is writing to the **TPSCTRL** register. Nevertheless the device can give up the I2C-bus control by writing to bit

SETEND in the **ENDDCTRL** register. Of course, when this situation occurs, the peripheral produces a stop condition on the I2C-bus.

If bit MCE (master code enable) is set in the **ADDRCFG** register, the device has to check if the transmitted address byte is a master code.

MASTER RECEIVES BYTES: The master has ordered bytes from the slave and handles the data transfer from the I2C-bus to the FIFO. This means that the bits are shifted into a buffer register in the I2C kernel and afterwards are passed to the FIFO. Every time a byte was transferred to the FIFO successfully the master-receiver produces an acknowledge in the according position.

The state transitions in the master process are:

M1: The software has written the number of bytes which have to be transmitted over the I2C-bus into the control register **TPSCTRL**. The FIFO generates appropriate data transfer requests which transfers valid data into the FIFO. A start condition is set on the bus and the bit-field BS (bus status) in register **BUSSTAT** is set to 10_B (busy master). The first (the first and the second) byte is (are) the 7-bit (10-bit) address of the slave which shall be accessed.

If the master wishes to initiate a transition to high-speed mode, the transition is only valid in combination with the MCE (master code enable) bit in the **ADDRCFG** register. The software has to write the master code into the FIFO, this means also the value of the **TPSCTRL** is predetermined with 1.

M2: The master receives an acknowledge from slave and the bit RnW in the **BUSSTAT** register was set to 1 during the addressing phase. A slave has been addressed successfully as transmitter, the state moves to MASTER RECEIVES BYTES. An RX (Receive) request is generated to distinguish between transmit- and receive-FIFO requests in non flow controller mode.

M3: The valid data packet from the FIFO was transferred successfully to the I2C-bus and the FIFO indicates this by the EOTXP (end of transmit packet) signal. If the RnW bit was set to 0 (writing to a slave) and bit SOPE (stop on packet end) in register **ADDRCFG** was set to 1, the transfer is closed by putting a stop condition on the bus. After this stop condition has been detected the peripheral indicates this successful transfer to the software via the TX_END (transmission end) request. The bus status bit-field is set to 0x0 (bus free).

M4: After the master transmits a byte it releases the SDA line to allow the addressed slave to pull down the SDA line (acknowledge). If this event occurs (and RnW in register **BUSSTAT** was set to 0), the master can continue to transfer bytes on the I2C-bus.

M5: The SDA line level is not equal to the one that is set by the master. This means that another master also tries to control the I2C-bus. This is indicated by the AL (arbitration lost) request and the device switches to the LISTENING state. The interrupt routine has to clear bit AL in the **PIRQSC** register. The Arbitration lost indication is not executable when the device works in high-speed mode (no arbitration procedure in high-speed mode since there is only one master remaining). The arbitration process must be operating at each bit which is sent by the master! Before interrupting the CPU, the bit-field BS (bus status) in register **BUSSTAT** is set to 01_B.

M6: If the master wants to transmit bytes but the FIFO is empty (underflow), it has to generate a TXF_UFL (TX FIFO underflow) request. The bit-field BS (bus status) in register **BUSSTAT** is reset to 00_B, and a TX_END (transmission end) request is generated after the master has put a stop condition on the bus.

M7: The I2C kernel wants to transfer bytes to the FIFO but without success. Then it has to produce an RXF_OFL (RX FIFO overflow) request. The current byte is not-acknowledged and a stop condition is put on the bus (bit-field BS in register **BUSSTAT** is set to 00_B and a TX_END (transmission end) request is generated). Requests (SREQ, BREQ, LSREQ or LBREQ) are generated for the outstanding data, the target is to empty the FIFO. (

M8: If master code has been transmitted by the device and a NACK (not-acknowledge) has been received, the MASTER RESTART state is entered. The clock SCL stays low as long as no further communication is requested. This is done by writing bit-field TPS in register **TPSCTRL** (see transition M9).

M9: A new communication in high-speed mode is started by writing bit-field TPS in register **TPSCTRL** when a master code has been previously transmitted (M8). The switch into high-speed mode (MC request) is done as soon as clock and data are released to high (see **Figure 337**). State MASTER TRANSMITS BYTES is entered. During high-speed mode, all fast and standard mode devices must be disconnected from the bus. This can be done in the MC request.

M10: Bit SETRSC has been set during the transmission of the data (the software wants to change direction/slave and therefore writes to the **ENDDCTRL** register). The I2C kernel resets and changes to MASTER RESTART state and a TX_END (transmission end) request is generated. Therefore, the master does not lose the control of the I2C-bus after the transmission has ended. The master stops transmission after the current byte was acknowledged/not acknowledged. Valid data in the FIFO are discarded. In this situation the FIFO is flushed to produce an initial situation.

M11: If the master wants to change direction/slave immediately in this state, it has to write to bit SETRSC in the **ENDDCTRL** register. The master puts a not-acknowledge on the appropriate place during the current transmitted byte. This indicates the slave-transmitter to stop transmission. The master switches to MASTER RESTART state; a TX_END (transmission end) request is generated.

M12: By sending a not-acknowledge on the bus, the master indicates the slave an upcoming stop of the transmission. This is established by writing to bit SETEND in register **ENDDCTRL**. Stop condition and TX_END (transmission end) request are generated. Requests (SREQ, BREQ, LSREQ or LBREQ) are generated for the outstanding data, the target is to empty the FIFO. The LISTENING state is entered; the bit-field BS (bus status) in register **BUSSTAT** is set to 00_B.

M13: If the master is receiving a not-acknowledge and bit SONA (stop on not-acknowledge) in the **ADDRCFG** register is 1, it produces a stop condition, resets the bit-field BS (bus status) in register **BUSSTAT** and issues a NACK (not-acknowledge) request and a TX_END (transmission end) request. The FIFO is flushed; remaining data in the FIFO stages are discarded.

M14: If bit SONA (stop on not-acknowledge) in the **ADDRCFG** register is 0, the peripheral wants to remain the controlling master of the I2C-bus after receiving a not-acknowledge. The next state is MASTER RESTART. A NACK (not-acknowledge) request and a TX_END (transmission end) request indicate this event to the software. The FIFO transfers the valid data to the memory.

M15: The master decided to transfer data on the bus again and writes to the **TPSCTRL** register. The first byte is interpreted as address byte (the 8th bit indicates RnW). The appropriate data transfer requests are activated by the FIFO controller. After the SCL line is high again (slave can stretch the low period) the master is allowed to put a restart condition onto the bus and is again in the MASTER TRANSMITS BYTES state starting to transmit bytes again.

M16: The software has written the SETEND bit in the **ENDDCTRL** register. The peripheral stops transmission of bytes after the current byte has been written on the bus. The I2C kernel flushes the FIFO since the remaining data in the FIFO is invalid. A TX_END (transmission end) request is generated.

M17: The FIFO indicates the end of the data packet and the peripheral was configured to go into MASTER RESTART state after that event. Since the data transfer has ended as intended a TX_END (transmission end) request is produced.

M18: The master has no data to send anymore but still controls the I2C-bus. Therefore it can use the SETEND bit to give up the I2C-bus control by setting a stop condition. A TX_END (transmission end) request is generated.

M19: A value different to zero is programmed to bit-field MRPS (maximum receive packet size) of the **MRPSCTRL** register. If the number of received bytes is equal to the MRPS bit-field, the FIFO produces the MRPS_stop signal. When bit SOPE (stop on packet end) in the **ADDRCFG** register is set to 0, the master generates a not-acknowledge and changes to MASTER RESTART state, so the master does not lose the control over the I2C-bus. A TX_END (transmission end) request is generated.

M20: (See also M19.) The FIFO produces the MRPS_stop signal which indicates that the desired amount of received data bytes is reached. The peripheral generates a not-acknowledge and changes to LISTENING state because bit SOPE (stop on packet end) in the **ADDRCFG** register is set to 1. Of course, the interface produces a stop condition; so the I2C-bus is free again (change also the bus status bit-field!). A TX_END (transmission end) request is also generated. Requests (SREQ, BREQ, LSREQ or LBREQ) are generated for the outstanding data, the target is to empty the FIFO. The LBREQ or the LSREQ indicate the completed reception of data. For more details on FIFO MRPS operation see **Section 34.3.1.6.5**.

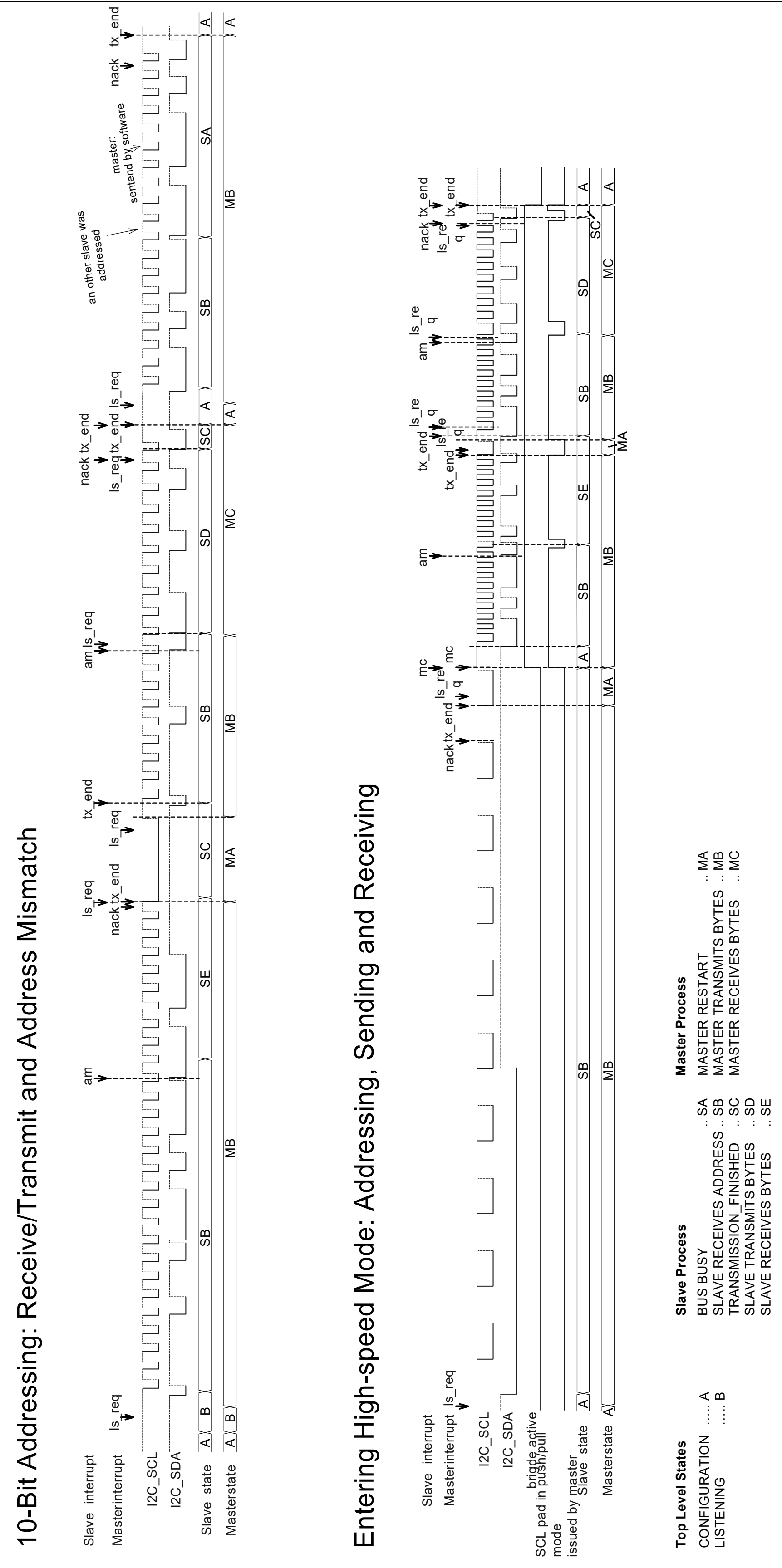

Figure 337 Connection Examples including Requests in Master and Slave Mode

34.3.1.6 FIFO Operation

The I2C module uses a FIFO between its kernel and the bus, in order to adapt the character processing speed of the peripheral to the transfer rate of the bus system. The FIFO has a transmission state and a reception state. **Figure 338** shows the bidirectional half duplex FIFO unit with the input and output lines and with the corresponding FIFO registers.

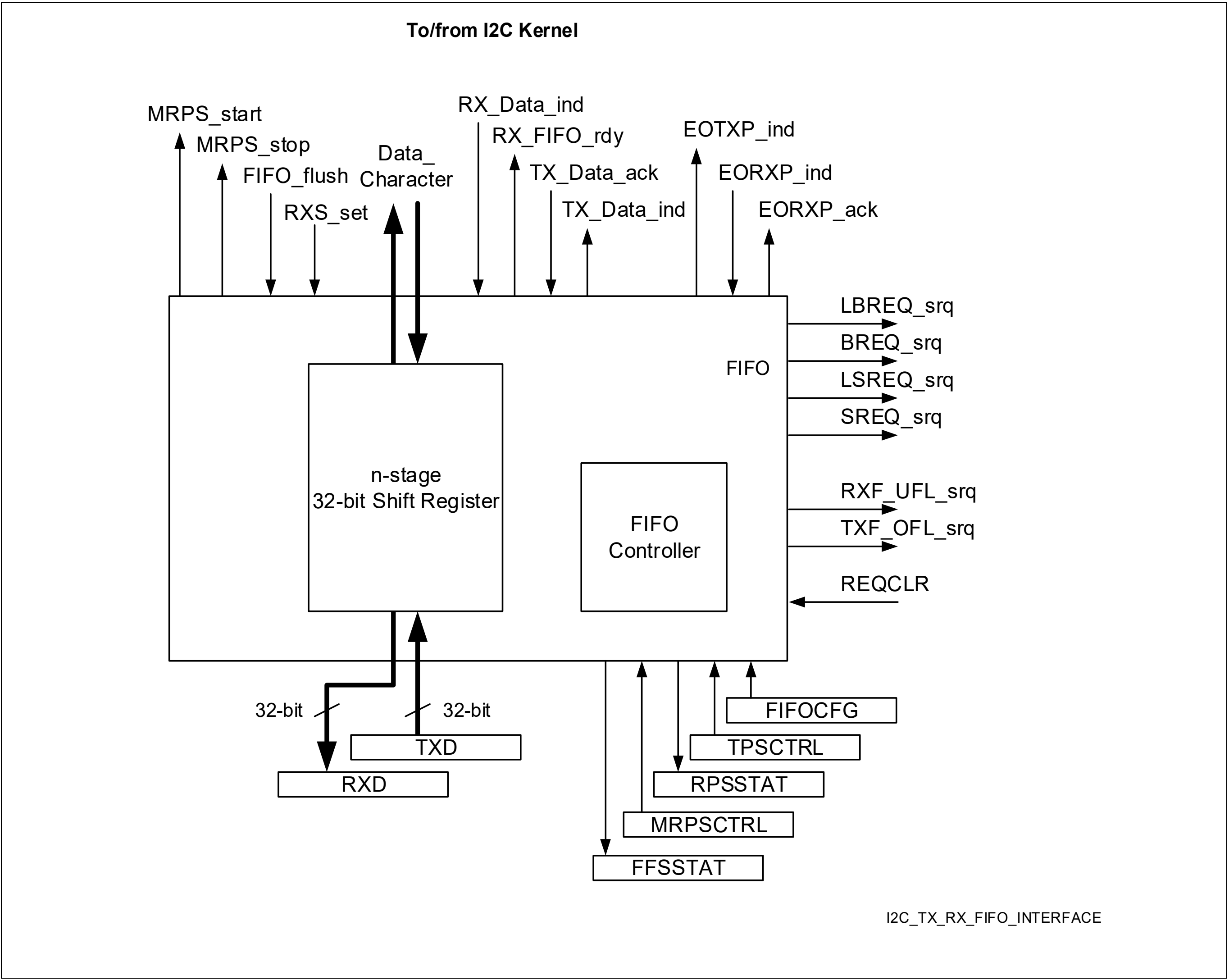

Figure 338 Bidirectional Half Duplex FIFO

Notes

1. *When the FIFO isn't serviced in time (transmission: filled with transmit data, reception: read the received data), an underflow/overflow event will lead to abortion of the transaction.To avoid this behaviour the software shall be configured to transfer a maximum of 32 bytes per transfer. If more than 32 bytes should be transmitted, the transmissions should be divided in maximum 32 data bytes per transfer*
2. *If bit-field **CRBC** in register **FIFOCFG** is disable, a linked list should be used to avoid that the CPU has to get active to clear the request. The DMA channel that serves I2C has to be configured with a linked list that first makes the FIFO-TX/RX transfer and second clears the interrupt by writing 0x0 to register ICR (Interrupt Clear Register) .*

34.3.1.6.1 Data Transmission

The software can move transmit data to the FIFO by writing to the Transmission Data Register **TXD**. Moving data to the FIFO can be done according to the data transfer request signals generated by the FIFO controller as described in **Section 34.3.1.6.2**. But with the register **FFSSTAT**, which indicates the number of full FIFO stages, the data transfer to the FIFO can be handled completely without data transfer requests. The data alignment within the FIFO is described in **Section 34.3.1.6.3**. The FIFO shifts data to the I2C kernel by means of the handshake lines tx_data_ind and tx_data_ack.

With the signal FIFO_flush the I2C kernel is able to clear the FIFO content. But if a data transfer request (see **Section 34.3.1.6.2**) is pending, then the FIFO is not cleared until the corresponding signal REQCLR has been cleared.

In bit-field **CRBC** in the register **FIFOCFG**can be configured how to clear the FIFO request.

The I2C-bus interface should know when it is receiving the last character of a data packet from the FIFO. Therefore the software should define a transmit packet size in bit-field TPS of register **TPSCTRL** and then the FIFO controller indicates the last character of the packet to the I2C kernel by setting the EOTXP (end of transmit packet) signal.

Since TPS is double buffered the software can write the size of the next packet into **TPS** as soon as the **TPS** value of the current packet has been loaded into the TPS counter, i.e. reading **TPS** returns 0. It loads the new TPS value as soon as ithas room in the FIFO.

Also the characters of the next packet can be moved to the FIFO immediately after the current packet characters as indicated in **Figure 339**. If byte or half word alignment is used as described in **Section 34.3.1.6.3**, then the characters of two different packets must not be aligned in a single stage.

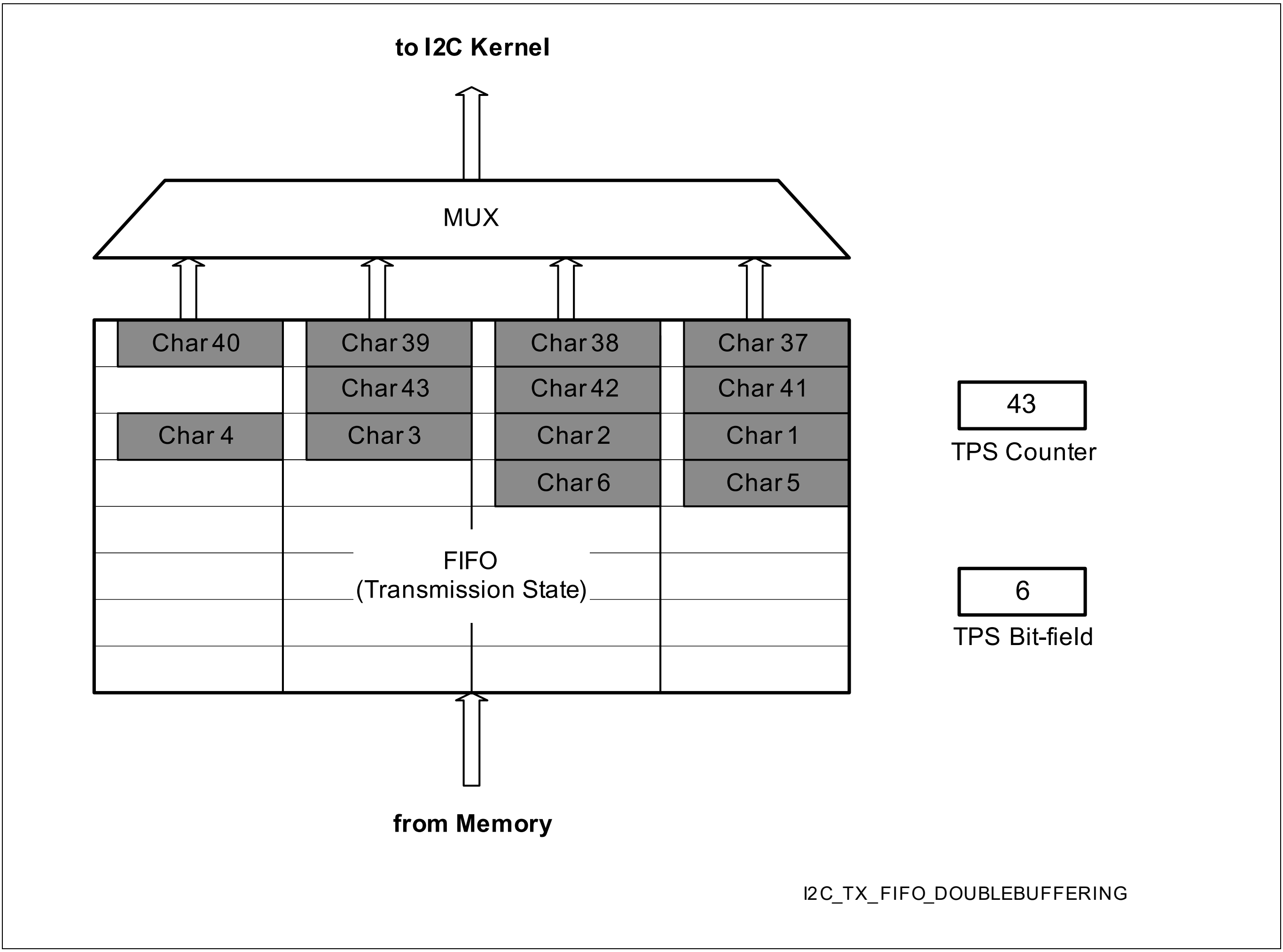

Figure 339 FIFO Containing Data of Two Different Transmit Packets

Error Handling

- If a TX FIFO overflow occurs (i.e. the software writes data too fast to the FIFO), then the new incoming character is discarded and a TX FIFO overflow interrupt request is generated.
- If a TX FIFO underflow occurs (i.e. the I2C kernel tries to read from an empty FIFO), then the I2C kernel generates a TX FIFO underflow interrupt request.

34.3.1.6.2 Transmit Request Generation

For the following it is assumed that the FIFO is used as flow controller, selected via bit **TXFC** of the **FIFOCFG** register.

In this case the bit-field **TPS** of the **TPSCTRL** register is not only used for the generation of the EOTXP (end of transmit packet) signal (see **Section 34.3.1.6.1**), but also to initiate the whole data transfer. I.e. the software indicates that it wants to transmit a data packet by writing the packet size to the bit-field **TPS**. Then the FIFO unit asserts burst requests BREQ until the amount of data still to be transferred is less than or equal to the burst size set in bit-field **TXBS** of register **FIFOCFG**. At this point, if the remaining data is equal to the burst size, then a last burst request LBREQ is issued. Otherwise, single requests SREQ and a last single request LSREQ will be issued.

If a data transfer (BREQ, LSBREQ, SREQ, LSREQ) request is pending, then no more request/interrupt will be done until the corresponding signal REQCLR has been cleared. The CPU (software) or DMA have to transfer the data to the FIFO and clear the generated interrupt.

Notes

1. *Pre-loading* ***TPS*** *with the packet size of the next frame, before the current packet has been transmitted completely, enables a continuous data throughput (e.g. of audio data).*
2. *If the FIFO is not used as flow controller, the I2C-bus interface should use bit-field* ***TPS*** *of the* ***TPSCTRL*** *register to set the EOTXP signal. A burst request BREQ is generated each time the number of empty FIFO stages is equal or greater than the burst size set in bit-field* ***TXBS*** *and the previous request was cleared. No single requests SREQ and LSREQ will be issued.*

34.3.1.6.3 Transmit Data Alignment

Depending on bit-field **TXFA** (TX FIFO Alignment) in register **FIFOCFG**, the FIFO can deal with byte aligned, half word aligned or word aligned characters (or even more) as it is described in the following.

Note: Byte alignment is synonymous with character alignment, half word alignment with character alignment of two data characters, and word alignment with character alignment of four characters.

Byte Aligned TX Characters

If the TX characters are byte aligned, then the FIFO extracts the bytes and shifts them to the I2C kernel via a multiplexer as it is shown in **Figure 340**.

Since the number of characters of the packet to be transmitted is not necessarily a multiple of 4, the last word must be filled up with dummy bytes if necessary. The FIFO transmits only the valid bytes of the last word depending on the setting of the bit-field **TPS** of the **TPSCTRL** register.

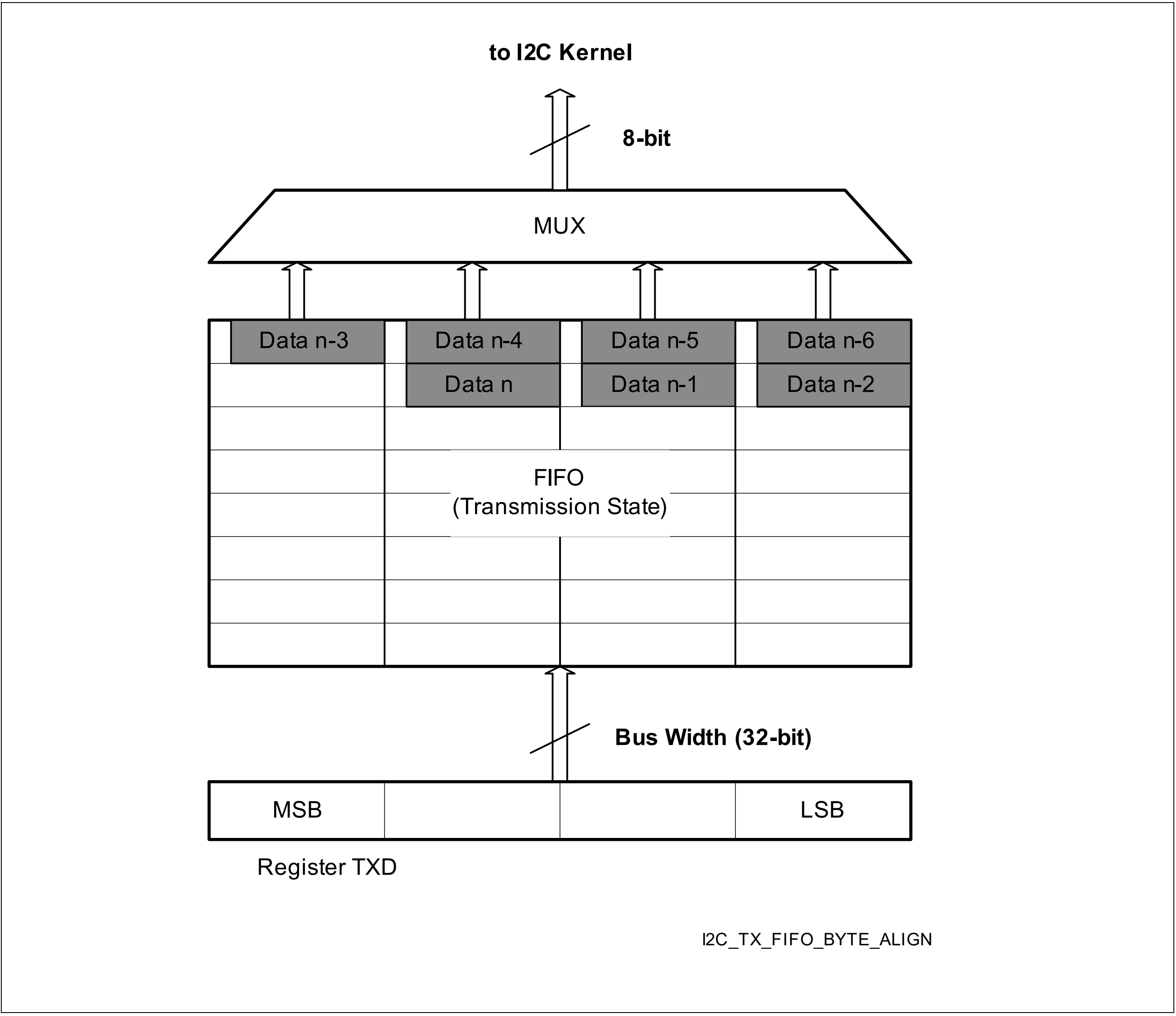

Figure 340 FIFO in Transmission State with Byte Aligned Data

Half Word Aligned

If the TX characters are half word aligned, then the FIFO extracts the half words and shifts them to the I2C kernel via a multiplexer as it is shown in **Figure 341**.

Note: The I2C kernel extracts the right data bits from the half word it receives from the FIFO.

Since the number of characters of the packet to be transmitted is not necessarily a multiple of 2, the last word must be filled up with a dummy half word if necessary. The FIFO transmits only the valid half words of the last word depending on the setting of the bit-field **TPS**.

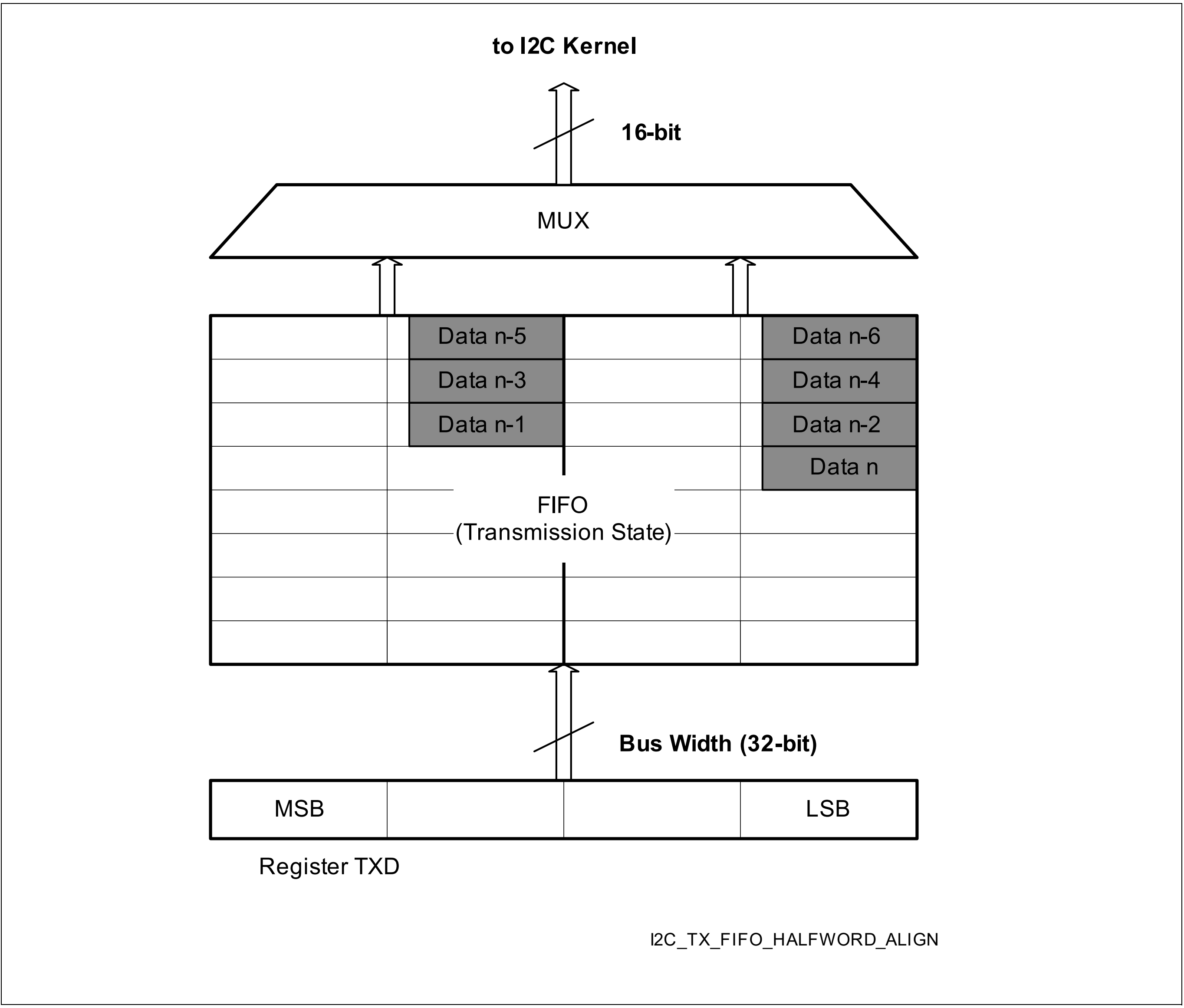

Figure 341 FIFO in Transmission State with Half Word Aligned Data

Word Aligned

If the TX characters are word aligned, then the FIFO shifts them to the I2C kernel as it is shown in **Figure 342**.

Note: The I2C kernel extracts the right data bits from the word it receives from the FIFO.

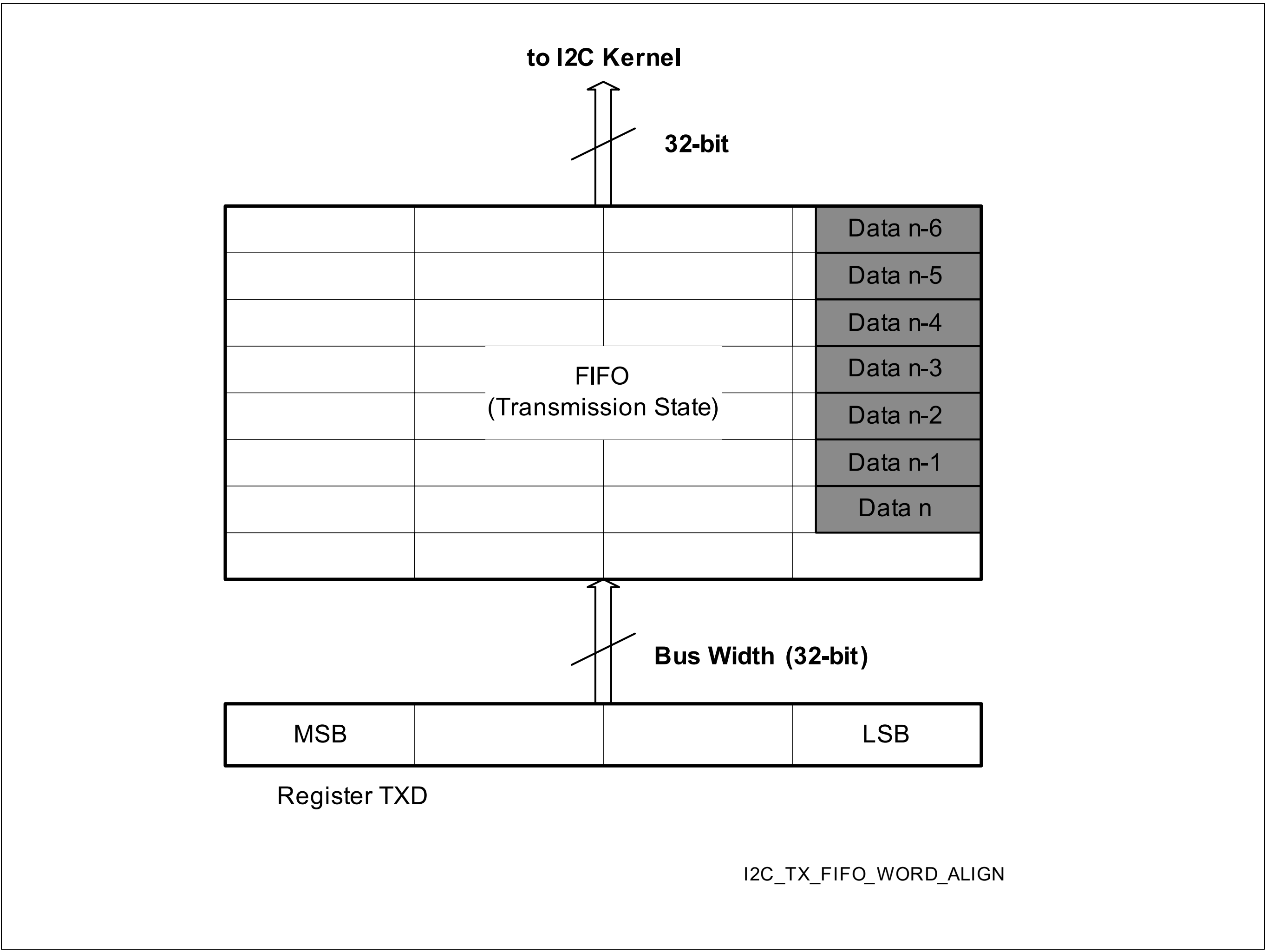

Figure 342 FIFO in Transmission State with Word Aligned Data

34.3.1.6.4 Data Reception

The software can read received data from the FIFO by reading the Reception Data Register **RXD**. Reading data from the FIFO should be done according to the data transfer request signals generated by the FIFO controller. The register **FFSSTAT** indicates the number of filled FIFO stages. The data alignment within the FIFO is described in **Section 34.3.1.6.6**. The I2C kernel shifts data into the FIFO by means of the handshake signals RX_FIFO_rdy and RX_Data_ind.

Before the first reception starts, the I2C kernel has to set the RXS_set line. This causes the assertion of RX_FIFO_rdy to indicate the I2C kernel that one data character can be written into the FIFO.

With the signal FIFO_flush the I2C kernel is able to clear the FIFO content. But if a data transfer request is pending and the FIFO is configured as flow controller, then the FIFO is not cleared until the corresponding signal REQCLR has been cleared.

In bit-field **CRBC** in the register **FIFOCFG**can be configured how to clear the FIFO request.

Error Handling

- If the RX FIFO is completely full (i.e. the software reads out the data too slow), then the I2C kernel generates an RX FIFO overflow interrupt request.
- If an RX FIFO underflow occurs (i.e. the software reads from empty FIFO), then an RX FIFO underflow interrupt request is generated.
- If an error (including the FIFO overflow condition described above) is detected in the I2C kernel and the current reception is stopped, then it also has to generate an EORXP_ind signal, so that the remaining characters in the FIFO can be moved out by means of data transfer requests.

34.3.1.6.5 Receive Request Generation

If the whole data of the current received packet has been moved into the buffer, then the I2C kernel will set the EORXP_ind (end of receive packet indication) signal. The received characters, which are shifted in the FIFO, are counted by anRPS (received packet size) counter. If an EORXP_ind occurs, then the content of the RPS counter is moved to the register **RPSSTAT** and the counter is reset. If data alignment in the FIFO is used, then the **RPSSTAT** value can be used to check for valid characters in the last read word (see **Section 34.3.1.6.6**).

For the following it is assumed that the FIFO is used as flow controller, selected via bit RXFC of the **FIFOCFG** register.

The FIFO unit asserts burst requests BREQ each time the FIFO filling level is greater than the burst size set in bit-field RXBS of register **FIFOCFG**. An EORXP_ind causes the FIFO unit to assert burst requests BREQ until the amount of data still to be transferred is less than or equal to the burst size. At this point, if the remaining data is equal to the burst size, a last burst request LBREQ is issued. Otherwise, single requests SREQ and a last single request LSREQ will be issued.

If a data transfer (BREQ, LSBREQ, SREQ, LSREQ) request is pending, then no more request/interrupt will be done until the corresponding signal REQCLR has been cleared. The CPU (software) or DMA have to get the data from the FIFO and clear the generated interrupt.

When the packet has been shifted out of the FIFO by software, the FIFO controller provides the EORXP_ack (end of receive packet acknowledge) signal to the I2C kernel, which allows the kernel to set the next EORXP_ind.

The next received packet may be shifted into FIFO immediately after the current packet, so that the FIFO contains data of two different packets. But the EORXP_ind of the next packet must not be asserted before the EORXP_ind of the current packet has been acknowledged. The **RPS** counter is counting the characters of the next packet, while in the register **RPSSTAT** the packet size of the current packet remains readable. The first character of the new packet is always shifted in a new FIFO stage as indicated in **Figure 343**. (The FIFO data alignment is described in **Section 34.3.1.6.6**.)

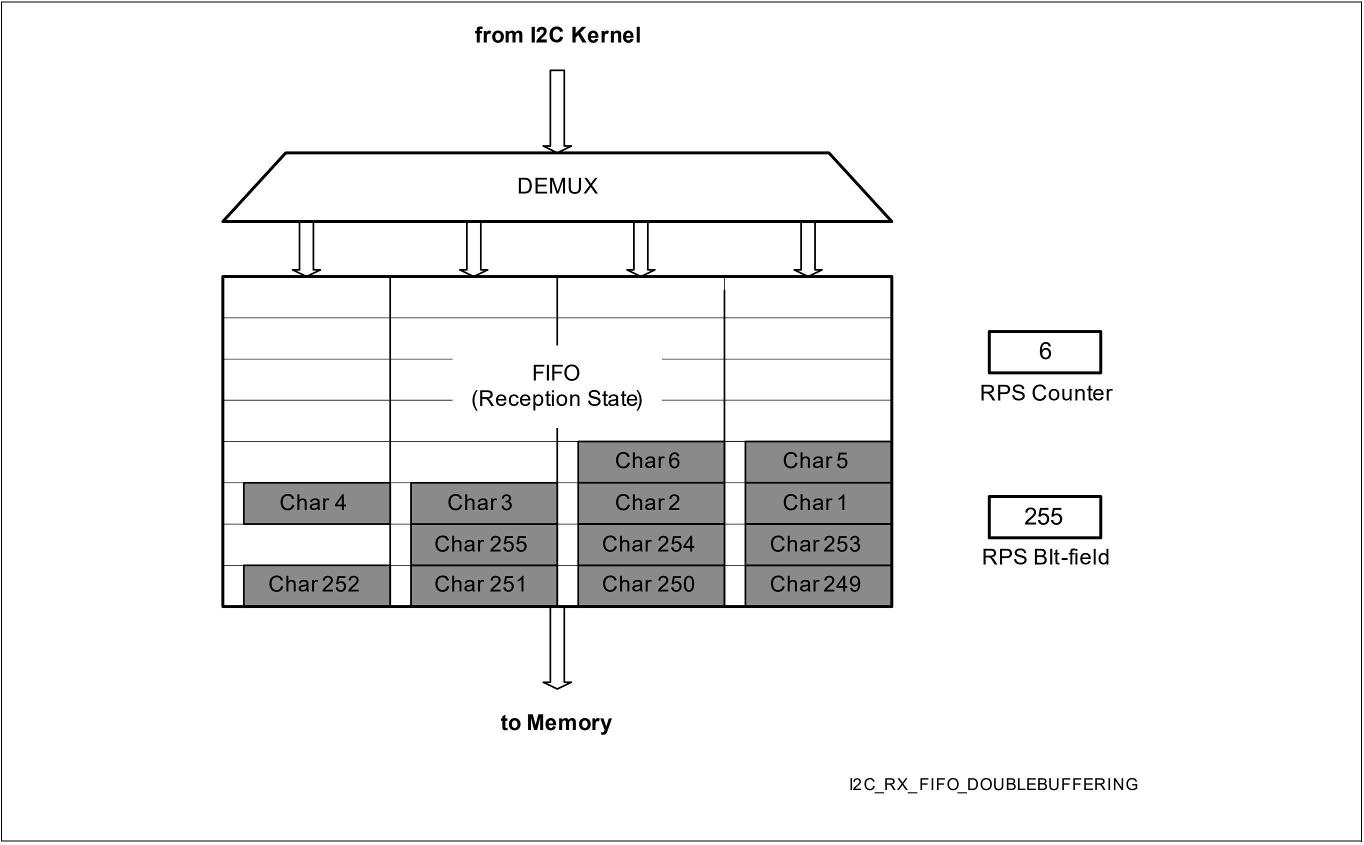

Figure 343 FIFO Containing Data of Two Different Receive Packets

If the number of the received characters of one packet is equal to the maximum received packet size, which is defined by bit-field MRPS in register **MRPSCTRL**, then the FIFO controller gets into the state, where it generates LBREQ or SREQ and LSREQ. If the I2C kernel generates an EORXP_ind in this state, then this indication is valid for the received packet and the FIFO accepts no more characters from the I2C kernel until the current packet has been moved out of the FIFO or the start of a new packet has been detected by asserting RXS_set.

The bit-field **MRPS** is double buffered, i.e. the software can write the **MRPS** value of the next packet as soon as the current **MRPS** value has been loaded, i.e. reading **MRPS** returns 0. It loads the new MRPS value as soon as it has room in the FIFO. If an **MRPS** overflow occurs (one character before the last character is received), then the MRPS_stop trigger to the I2C kernel is generated. But if the next **MRPS** value is written to **MRPS** before the last data character of current frame has been completely received, then no MRPS_stop is generated. The I2C-bus interface will use the MRPS_stop trigger to stop the reception of the corresponding data packet.

But the **MRPS** bit-field can also be used to initiate a reception, when the peripheral is master-receiver. Therefore writing to MRPS generates a trigger MRPS_start to the I2C kernel. The next MRPS value written to **MRPS** generates only an MRPS_start when an MRPS_stop was generated before.

The MRPS features can only be used if the FIFO is the flow controller.

Notes

1. *When the MRPS control feature of the FIFO is used to limit the packet size of a continuous incoming data stream (MRPS is only written once and left unchanged), characters exceeding MRPS are discarded. The FIFO can not handle more than two packets. When the FIFO is filled with data characters of two packets then a further RXS_set will be overseen by the FIFO. Additionally the FIFO does not set the RX_FIFO_rdy.*
2. *If the FIFO is not used as flow controller, a single request SREQ is always generated as soon as the readable FIFO stage is filled up with characters. Additionally, a burst request BREQ is generated as soon as the number of filled FIFO stages is equal to or greater than the burst size set in bit-field RXBS of register FIFOCFG. When the I2C kernel*

sets EORXP_ind, the FIFO controller provides the EORXP_ack (end of receive packet acknowledge) signal to the I2C kernel, which allows the kernel to set the next EORXP_ind.

34.3.1.6.6 Receive Data Alignment

Depending on bit-field **RXFA** (RX FIFO Alignment) in register **FIFOCFG**, the FIFO can deal with byte aligned, half word aligned or word aligned characters (or even more) as it is described in the following.

Byte Aligned

If the RX data should be byte aligned, then the characters from the I2C kernel are aligned and shifted to the FIFO by a demultiplexer as it is shown in **Figure 344**.

Since the number of characters of the received packet is not necessarily a multiple of 4, the upper bytes of the last word can be invalid. The software has to check for invalid bytes of the last word by means of the bit-field **RPS** (received packet size) in register **RPSSTAT**.

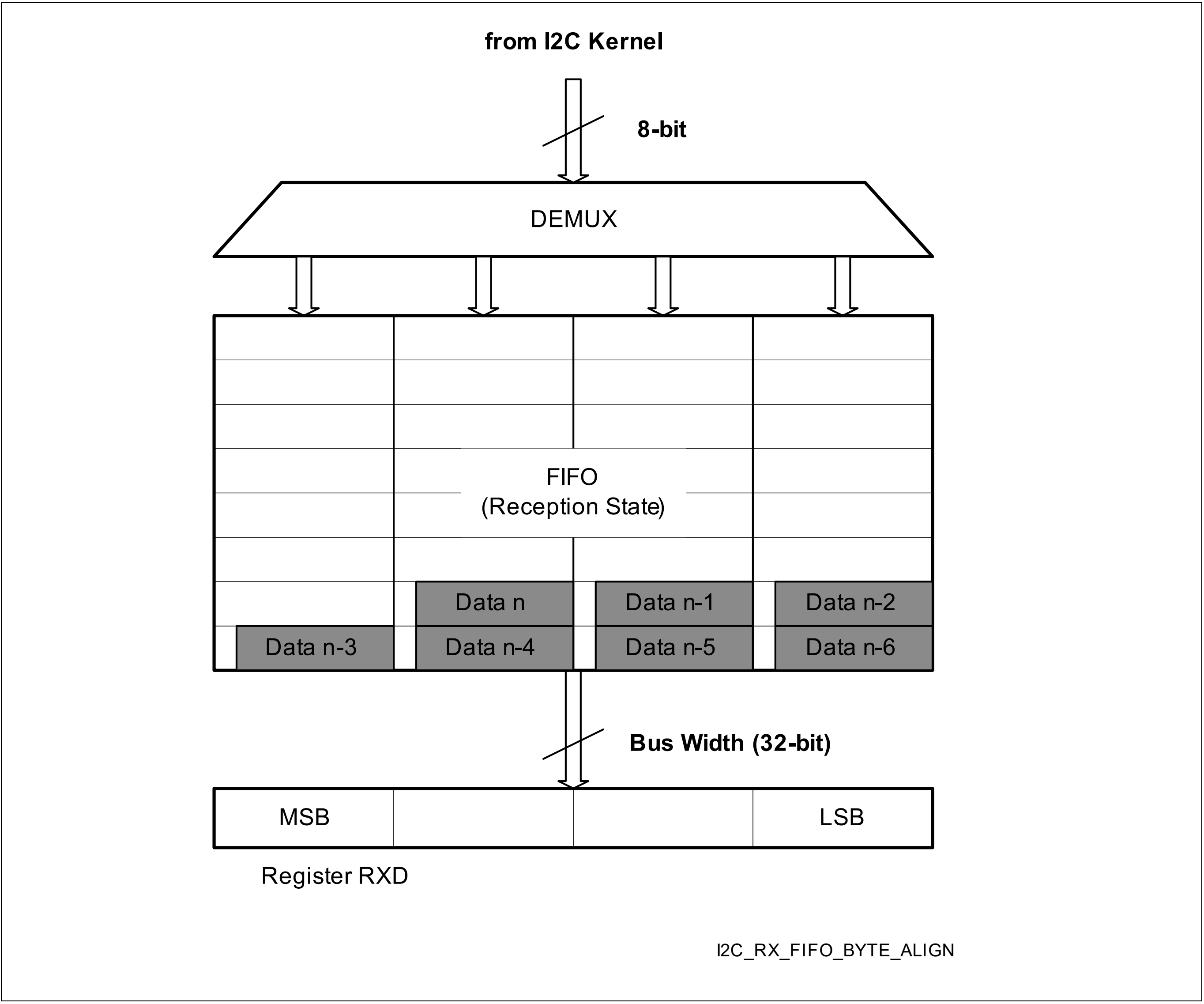

Figure 344 FIFO in Reception State with Byte Aligned Data

Half Word Aligned

If the RX data should be half word aligned, then the characters from the I2C kernel are aligned and shifted to the FIFO by a demultiplexer as it is shown in **Figure 345**.

Note: The I2C kernel fills up the half words with dummy bits.

Since the number of characters of the received packet is not necessarily a multiple of 2, the upper bytes of the last word can be invalid. The software has to check for invalid half words of the last word by means of the bit-field RPS (received packet size) in register **RPSSTAT**.

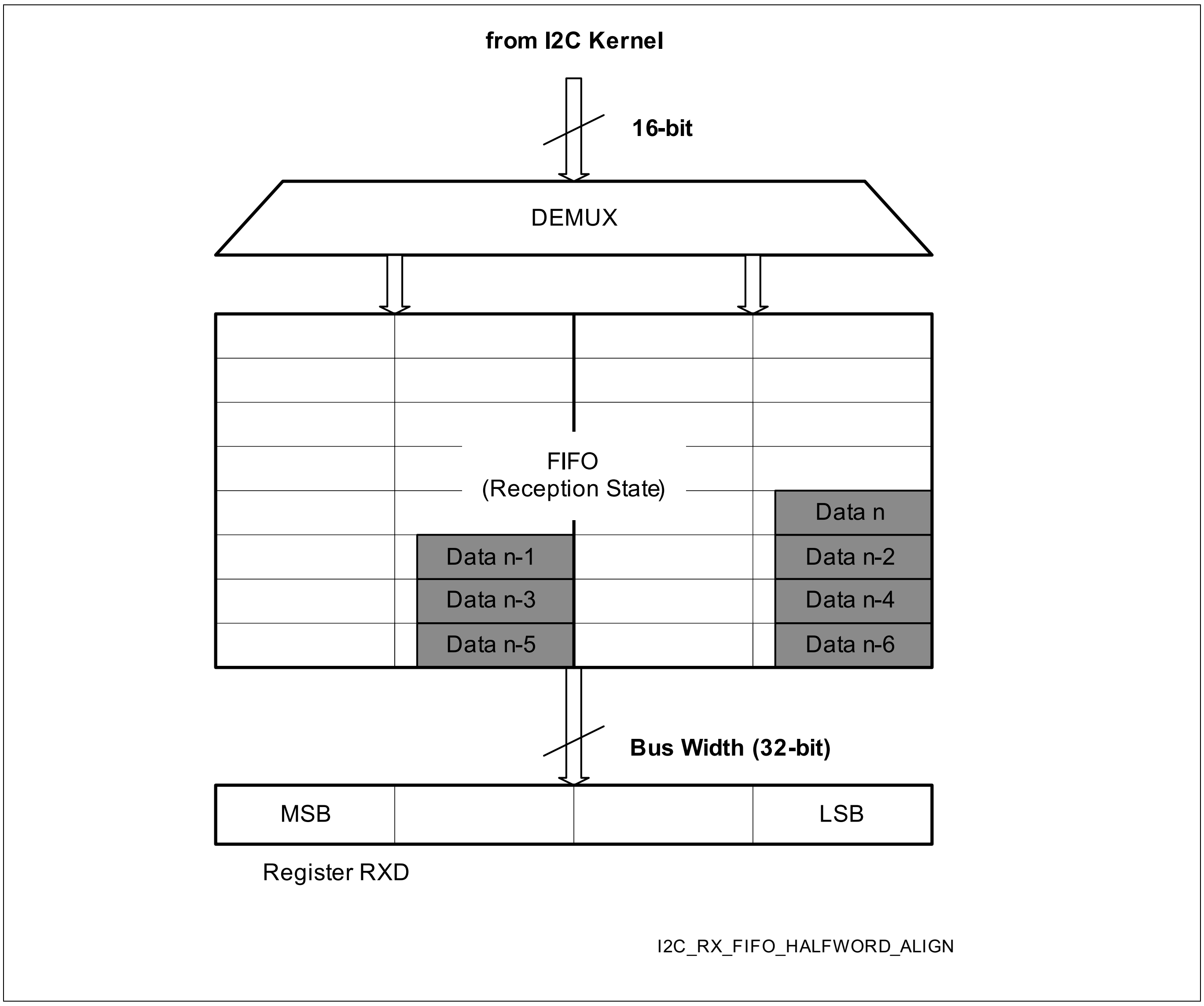

Figure 345 FIFO in Reception State with Half Word Aligned Data

Word Aligned

If the RX data should be word aligned, then the characters from the I2C kernel are shifted to the FIFO as it is shown in **Figure 346**.

Note: The I2C kernel fills up the words with dummy bits.

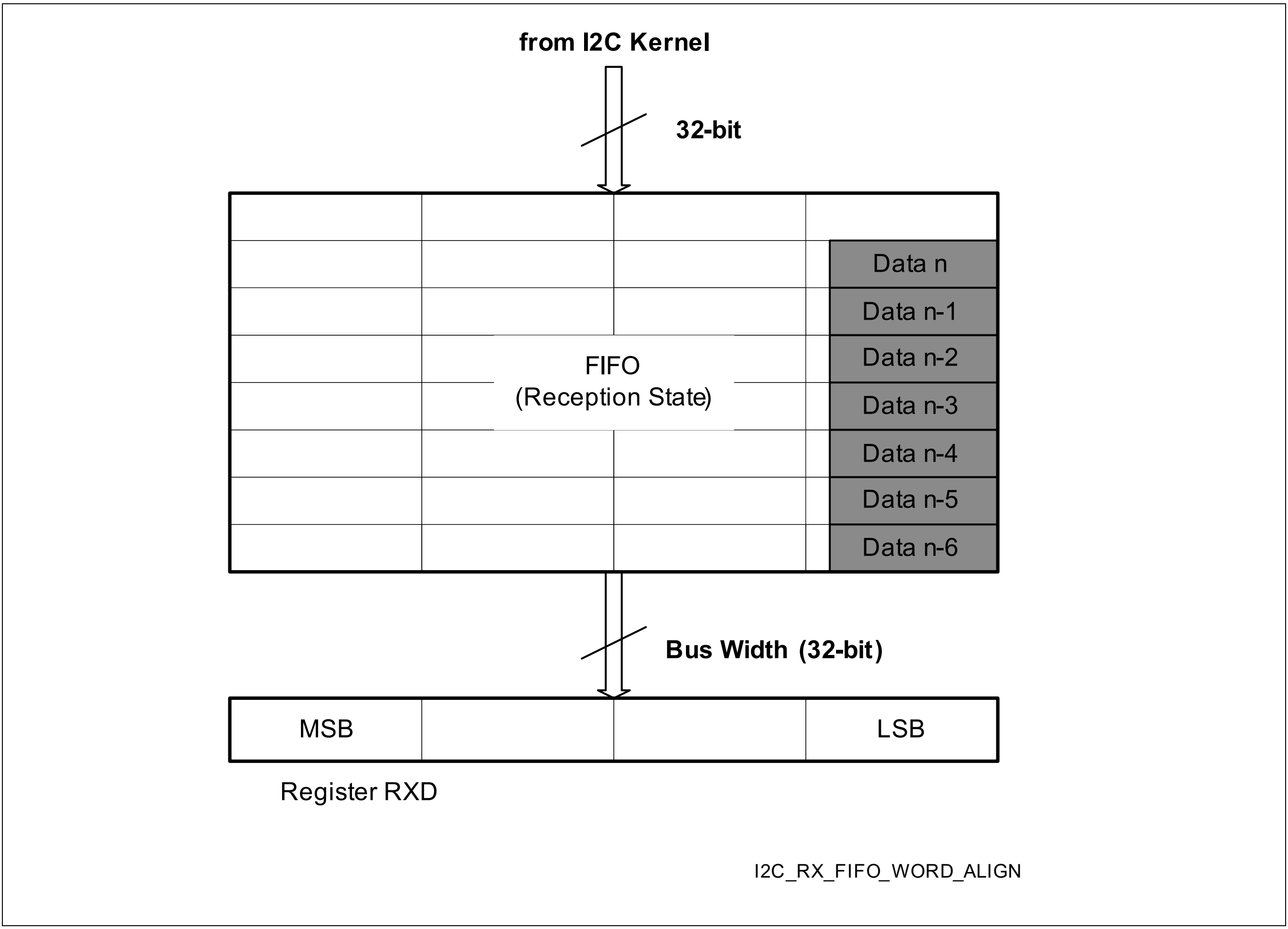

Figure 346 FIFO in Reception State with Word Aligned Receive Data

34.3.1.6.7 Switching between Transmission and Reception

Initially, the FIFO is in the TX state, so that the software can initiate a TX transfer simply by writing data to the bit-field **TXD**. If the I2C kernel wants the FIFO to switch to the RX state in the meantime, then it can set the RXS_set line and even if transmission is ongoing the TX transfer is aborted. This causes the FIFO to flush its content, switch to the RX state and reset the received packet size counter. If the software wants to write to the register **TXD** when the FIFO is in RX state, then this causes a bus error. As soon as the FIFO has received an EORXP_ind signal from the kernel and the software has moved all RX characters out of the FIFO via the register **RXD**, the FIFO automatically switches back into the TX state.

If the software writes to the register **TPSCTRL** when the FIFO is in the RX state, the TPS value is pending until the FIFO returns to the TX state. Then the transmission is initiated, if the FIFO is the flow controller.

If the FIFO is flow controller and the software writes to the register **MRPSCTRL**, then an MRPS_start is generated independent of the state of the FIFO.

With the signal FIFO_flush the I2C kernel is able to clear the FIFO content. But if a data transfer request (xREQ) is pending, then the FIFO is not cleared until the corresponding signal REQCLR has been cleared.

If the FIFO has been cleared in the RX state, then it switches back to the TX state afterwards.

34.3.1.6.8 Switching between Reception and Transmision

When the I2C kernel is changing from RX state to TX state the FIFO is flushed. To avoid loosing data when the FIFO is flushed the software should proceed in the right sequence. In a scenario where the device is addressed as slave and is asked to return data, this new data must be entered in the FIFO only after detection of the address and TX_END interrupt requests and then can transfer the data to the FIFO or can trigger the DMA that fills the FIFO for the TX transfer.

34.3.1.7 Service Request Block Operation

The SRB (Service Request Block) of the I2C module is used to prepare the interrupt and data transfer requests for the interrupt controller.

34.3.1.7.1 Overview of Service Requests

The I2C service requests are partially combined in the SRB (see **Section 34.4.4** and **Section 34.4.5**). **Figure 347** shows an overview of the service requests. **Table 301** provides a list of all service requests of the I2C module.

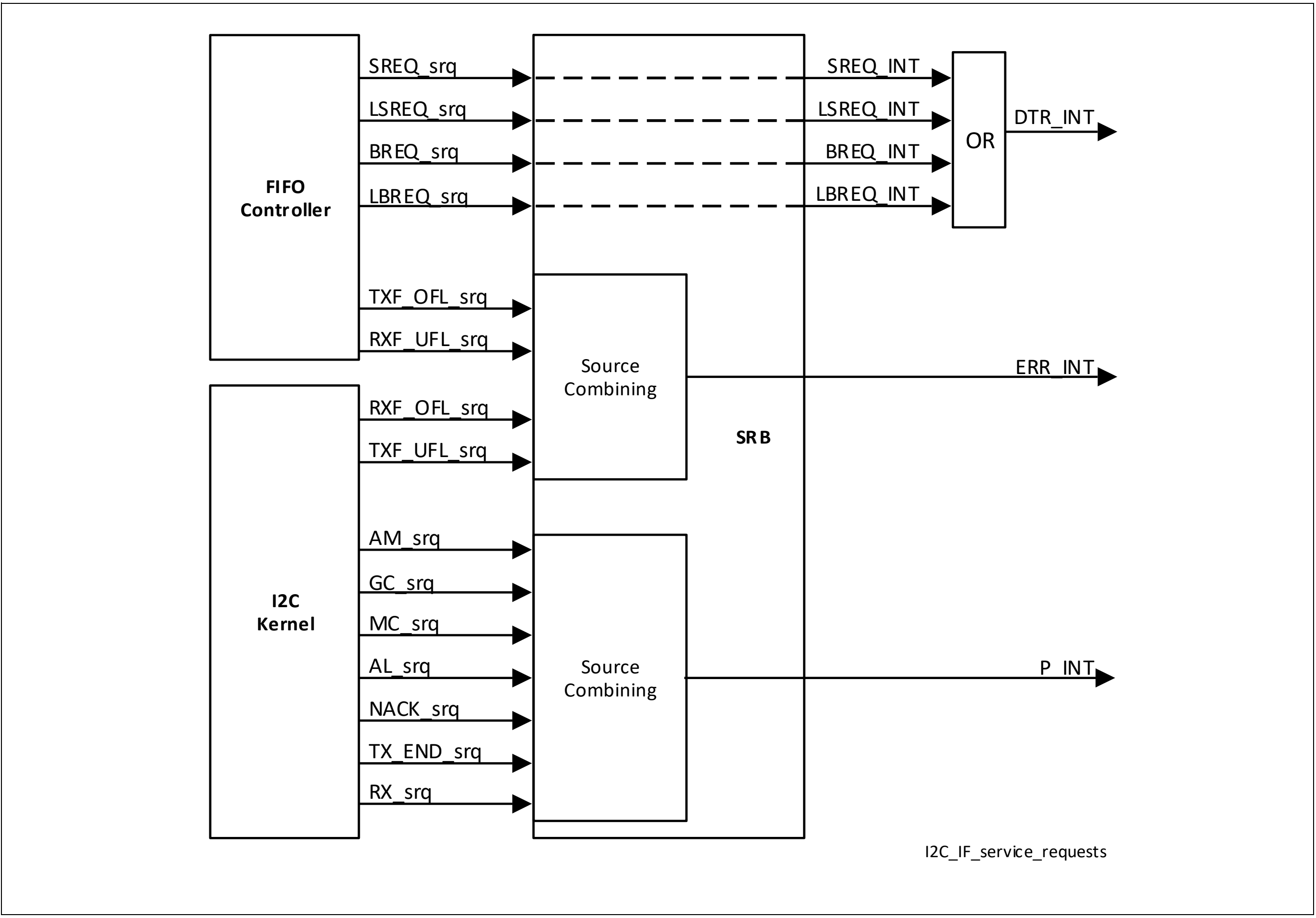

Figure 347 Overview of I2C Module Service Requests

Table 301 I2C Module Service Requests

Service Request	Interrupt Request	Description
BREQ_srq	DTR_INT	**Burst Data Transfer Request** FIFO requests a transfer of a programmed burst number of words from/to the memory.
LBREQ_srq	DTR_INT	**Last Burst Data Transfer Request** FIFO requests a last burst transfer from/to the memory.
SREQ_srq	DTR_INT	**Single Data Transfer Request** FIFO requests a single transfer of a word from/to the memory.
LSREQ_srq	DTR_INT	**Last Single Data Transfer Request** FIFO requests a last single transfer from/to the memory.
TXF_OFL_srq	ERR_INT	**TX FIFO Overflow Request** FIFO has detected a TX FIFO overflow.
TXF_UFL_srq	ERR_INT	**TX FIFO Underflow Request** I2C kernel has detected a TX FIFO underflow. The transmission is finished after the current byte. A stop condition is generated if the kernel works as master. The kernel changes to listening state.
RXF_OFL_srq	ERR_INT	**RX FIFO Overflow Request** I2C kernel has detected an RX FIFO overflow and the incoming character is discarded. The kernel puts a not-acknowledge on the bus and changes to listening state. A stop condition is generated if the kernel works as master.
RXF_UFL_srq	ERR_INT	**RX FIFO Underflow Request** FIFO has detected an RX FIFO underflow.
AM_srq	P_INT	**Address Match Request** Device (master/slave) has been addressed by a remote master (also indicated in bit-field BS in register **BUSSTAT**).
GC_srq	P_INT	**General Call Request** When the general call matching process is activated this interrupt indicates that another master has put a general call on the I2C-bus.
MC_srq	P_INT	**Master Code Request** When the master code matching process is activated this interrupt indicates the appearing of a master code on the I2C-bus issued by a remote master. The request is generated after a not-acknowledge and the clock is released to high again.
AL_srq	P_INT	**Arbitration Lost Request** Arbitration is lost after the device has tried to start a transmission on the I2C_bus.
NACK_srq	P_INT	**Not-acknowledge Received Request** Not-acknowledge received when working as transmitter (i.e. RnW bit is 0).

Table 301 I2C Module Service Requests (cont'd)

Service Request	Interrupt Request	Description
TX_END_srq	P_INT	**Transmission End Request** In master mode this event is produced by the I2C kernel to indicate that the transmission of the current packet has ended properly after the stop condition has been put on the I2C-bus or MASTER RESTART state has been entered. (At this point, a restart condition can be generated or the connection can be finished by generating a stop condition). This request is created in master mode at any stop condition to indicate that the bus is free again and it could be obtained. In slave mode this event is produced by the I2C kernel if it was addressed by a master and the current transmission was terminated by a stop or restart condition.
RX_srq	P_INT	**Receive Mode** Request I2C kernel indicates to the FIFO switching from transmit to receive mode. When FIFO is operating in non flow controller mode, this service request can be used to distinguish between transmit and receive FIFO data requests.

The generation of these events is visualized in the state machine **Figure 334**, **Figure 335** and **Figure 336**. The timing is shown in **Figure 337**.

34.3.1.7.2 Interrupt Service Request Structure

The I2C module provides level based service request lines, which can be processed by an interrupt controller. The service requests must be cleared in the interrupt service routine. For test purposes, all service requests can also be set by SW via a register bit. All service requests can be masked within the peripheral. Furthermore, requests that are not necessarily mutually exclusive are combined in order to reduce the number of request lines.

Flow of Data Transfer Requests

Figure 348 shows the flow of a data transfer request, which comes from the FIFO controller.

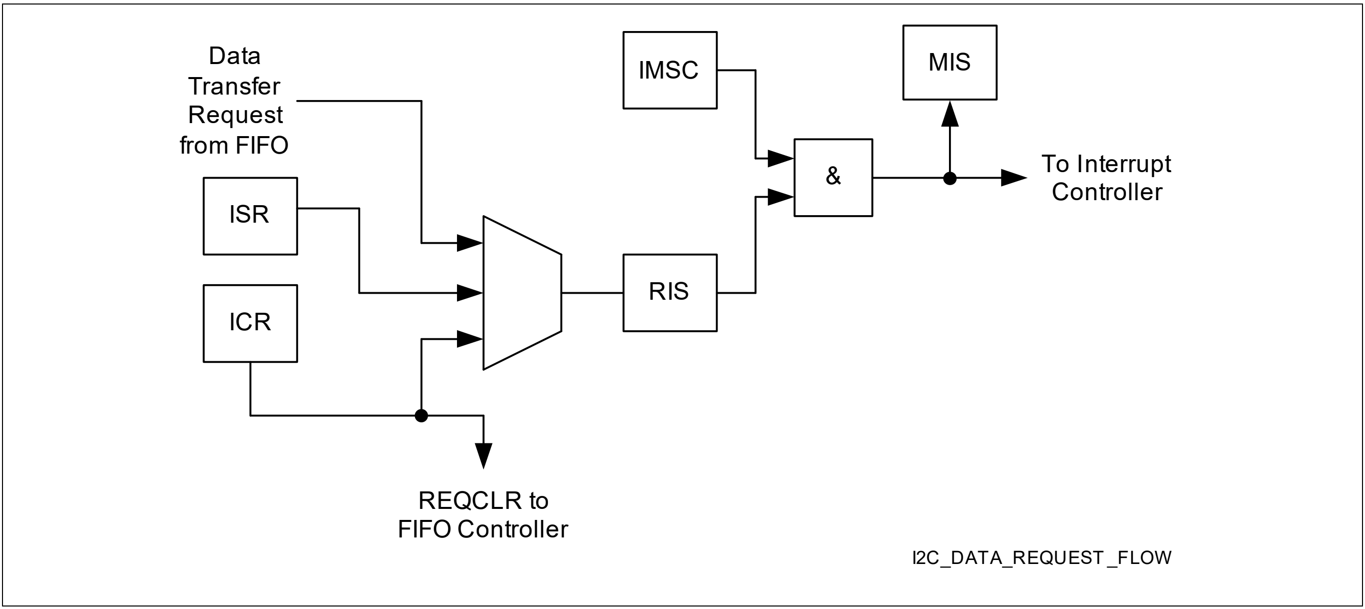

Figure 348 Data Transfer Request Flow

A data transfer request sets the corresponding status bit in the Raw Interrupt Status Register **RIS**. The status bit can also be set by writing 1 to the corresponding bit in the Interrupt Set Register **ISR**. It will be cleared by writing 1 to the corresponding bit in the Interrupt Clear Register **ICR**.

The Interrupt Mask Control Register **IMSC** enables or disables the requests to the interrupt controller. The Masked Interrupt Status Register **MIS** contains the current masked values of the requests.

If a request is disabled via register **IMSC** while the request is active, the request will be removed but only until the **IMSC** bit is set again, unless the request has been cleared in the meantime.

If a request is disabled via register **IMSC** and the source becomes active, then the request bit will only be set in the **RIS** register. If the corresponding **IMSC** bit is later enabled, the request will consequently become active in the **MIS** register.

Before enabling a request, it is good practice to always first clear the corresponding bit in the **RIS** register via **ICR**.

Multiple Source Interrupt Requests

For error and protocol interrupt requests which have multiple sources, the interrupt structure is extended. Additional register sets are implemented within the SRB.

Figure 349 gives an overview of the combining of several request sources to a single request.

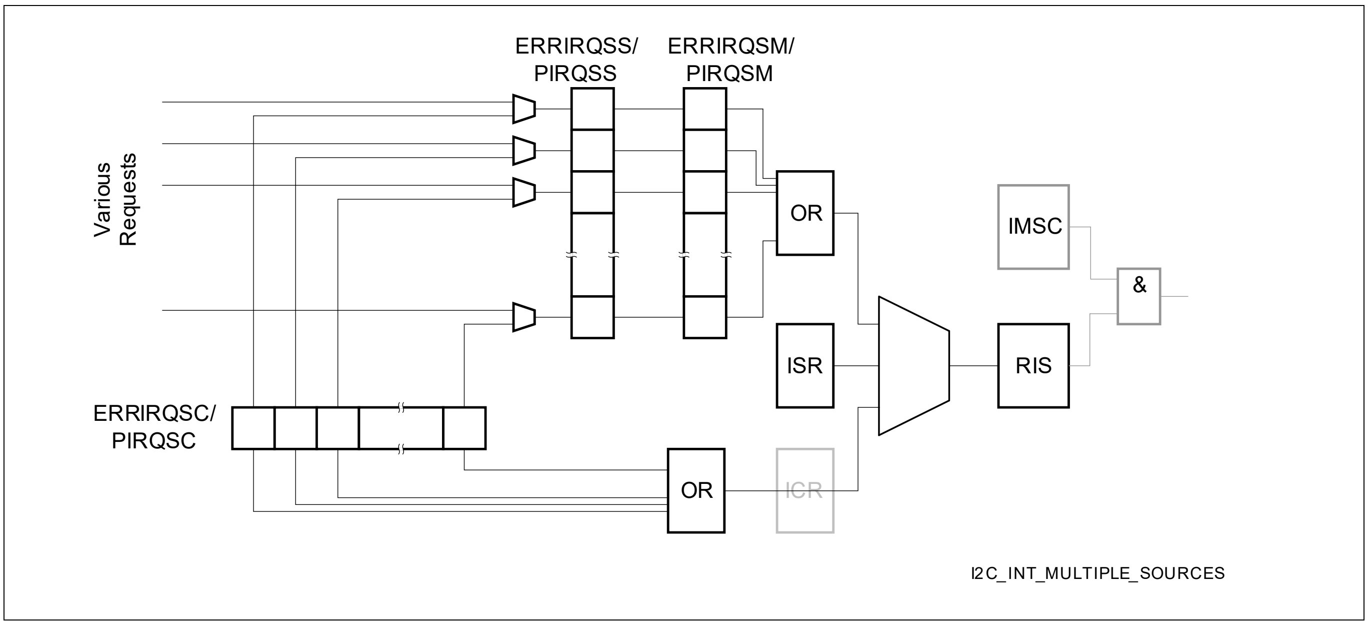

Figure 349 Interrupt Request with Multiple Sources

An interrupt request sets the corresponding raw status bit in the Interrupt Request Source Status Register **ERRIRQSS** (for error) or **PIRQSS** (for protocol).

The status bits can be cleared via the Interrupt Request Source Clear Register **ERRIRQSC** (for error) or **PIRQSC** (for protocol). If no further error/protocol request sources are active, the whole error/protocol interrupt request is cleared. This register replaces the functionality of the bit I2C_ERR_INT (for error) or bit I2C_P_INT (for protocol) in the Interrupt Clear Register **ICR**.

The Interrupt Request Source Mask Register **ERRIRQSM** (for error) or **PIRQSM** (for protocol) enables or disables the interrupt requests. The request lines from the enabled sources are combined (OR) to a single level sensitive request line, which acts as input for the corresponding bit I2C_ERR_INT (for error) or I2Cm_bit I2C_P_INT (for protocol) in the Raw Interrupt Status Register **RIS**.

34.3.2 I2C Module Implementation

This section describes I2C module interfaces with the clock control, port control and interrupt control.

34.3.2.1 Interfaces of the I2C Module(s)

Figure 350 shows the specific implementation details and interconnections of the I2C module(s):

- A clock control block generates the clock signals required for the module.
- For the I2C-bus lines SCL and SDA, different I/O lines can be selected.
- The interrupt outputs of the module are connected to the interrupt control unit.

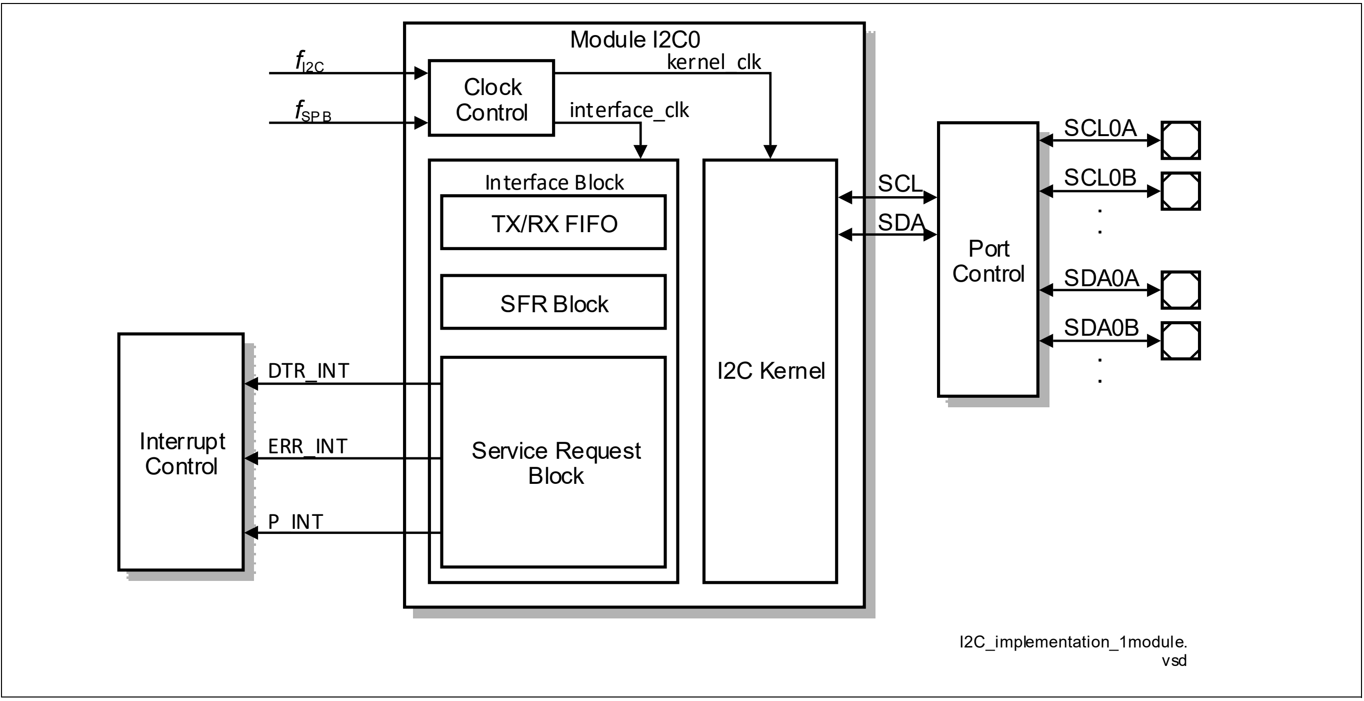

Figure 350 I2C Module Implementation and Interconnections

34.3.2.2 Module Clock Control

The clock control allows the programmer to adapt the peripherals functionality and power consumption to the application's requirement. By programming the kernel's operating frequency, an optimal ratio between power consumption, EMC behavior and functionality can be achieved. Furthermore, for power saving reasons the peripheral can be disabled as a whole by switching off the peripheral's clocks via clock gating cells.

A simplified description of the clock control is shown in **Figure 351**. See register **CLC1** for a description of the clock control parameters.

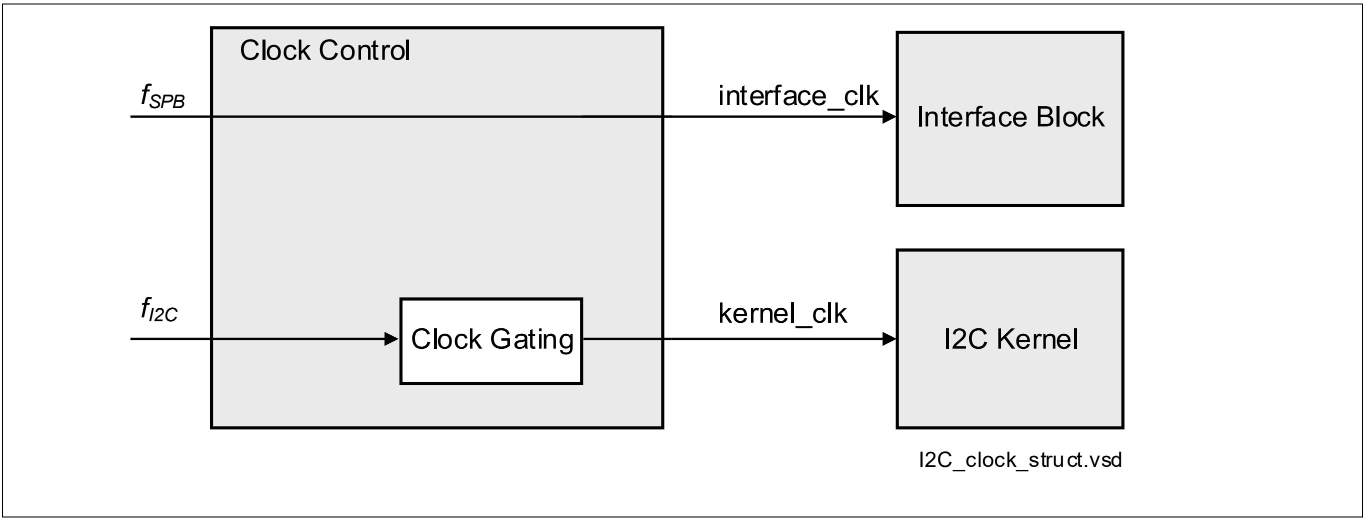

Figure 351 Module Clock Control

The following clocking modes are supported for the peripheral:

- Disabled mode
- OCDS suspend mode

Disabled Mode

In Disabled Mode, the clocks are stopped and register write accesses are not possible (except access to **CLC1** register itself).

There are several ways possible to bring the module into Disabled mode:

- Software disable
  - Set **CLC1**.DISR = 1 (software disable request)
- Hardware disable
  - Must be enabled via **CLC1**.EDIS = 0
  - Drive input signal sleep_n_i = 0(active low: 0= disable; 1= enable)

In all cases, current bus accesses are finished and a kernel handshake is issued to make sure that the state machine is in a safe state (same as **CLC1**.FSOE = 0 in OCDS Suspend Mode; see description below).

Module is in Disabled Mode after reset and must be enabled by setting **CLC1**.DISR = 0 and **CLC1**.RMC > 0-.

OCDS Suspend Mode

In this mode the clocks are disabled. If bit SPEN in register **CLC1** is set, the mode can be entered by activating the input module_ocds.

In OCDS suspend mode all registers of the suspended module can be read. Note that each read during disabled clock can not affect hardware. For example multiple reads to FIFO port registers always return the same value as the FIFO control is not working.

No write access to the registers is supported during suspend mode, except writes to the **CLC1** register. For writing other registers the suspend mode for the module has to be removed first by clearing bit SPEN. After the write access is executed, bit SPEN should be set again.

OCDS suspend mode supports two different disable features, selectable by bit FSOE in register **CLC1**:

*Note: To write to SPEN or FSOE, bit SBWE in register **CLC1** must also be set in the same write access.*

- **Secure Clock Switch Off**

Disabling the clock via secure clock switch off additionally activates the handshake mechanism with the peripheral. The peripheral is switched to a secure state before disabling the clock.

Note: This feature is only available in slave mode. In master mode, there is no difference and I2C will always perform “fast clock switch off”.

- **Fast Clock Switch Off**

Selecting the fast clock switch off stops the clock immediately after all pending read and write accesses are finished (including synchronization mechanisms with the I2C kernel).

34.3.2.3 Bus Peripheral Interface Registers

Clock Control 1 Register

This register controls the clock gating and dividing circuitry of the peripheral.

CLC1

Clock Control 1 Register **(00000_H)** **Application Reset Value: 0000 0003_H**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0								**0**							
r								r							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RMC								**0**		**FSOE**	**SBWE**	**EDIS**	**SPEN**	**DISS**	**DISR**
rwh								r		rw	w	rw	rw	rh	rw

Field	Bits	Type	Description
DISR	0	rw	**Module Disable Request Bit** Used for enable/disable control of the module. 0_B Module disable not requested 1_B Module disable requested
DISS	1	rh	**Module Disable Status Bit** Bit indicates the current status of the module. 0_B Module enabled 1_B Module disabled
SPEN	2	rw	**Module Suspend Enable Bit for OCDS** 0_B Module suspend disabled 1_B Module suspend enabled
EDIS	3	rw	**External Request Disable** *Note: This bit is not used in AURIX™ TC3xx Platform and should always be written with 0.* 0_B External clock disable request is enabled 1_B External clock disable request is disabled
SBWE	4	w	**Module Suspend Bit Write Enable for OCDS** This bit is always read as 0. 0_B Bits SPEN and FSOE are write protected 1_B Bits SPEN and FSOE are overwritten by respective value of SPEN or FSOE
FSOE	5	rw	**Fast Switch Off Enable** 0_B FSOE Clock switch off in OCDS suspend mode via Disable Control Feature (Secure Clock Switch Off) 1_B Fast clock switch off in OCDS suspend mode

Field	Bits	Type	Description
RMC	15:8	rwh	**Clock Divider for Standard Run Mode** Max. 8-bit divider value If RMC is set to 0 the module is disabled. *Note: As long as the new divider value RMC is not valid, reading register returns 0000 00XX$_H$*
0	7:6, 23:16, 31:24	r	**Reserved** Read as 0; should be written with 0.

34.3.2.4 Port and I/O Line Control

Not only the interconnections between the I2C module and the port I/O lines have to be configured, but also the function and the characteristics of the port pins. The following control operations must be executed for the used port pins:

- Selection of I2C module via output port multiplexer
- Configuration of output port function with open drain
- Configuration of pad driver characteristics
- Selection of I2C input lines

34.3.2.5 Interrupt Control

The interrupt functionality is described in the chapter for the interrupt router (IR).

The signal names used in the interrupt router description are explained in **Table 302**.

Table 302 Interrupt Router Signal Names for I2Cm Module (m = 0/1)

Signal Name	**Name in Module**	**Explanation**
SRC_I2CmDTR	DTR_INT	**Data Transfer**
SRC_I2CmERR	ERR_INT	**Error**
SRC_I2CmP	P_INT	**Protocol**

34.3.3 Module Integration

This section describes the topics regarding the integration of the module on chip.

34.3.3.1 Integration Overview

The I2C module consists of a delivery providing an AHB bus interface integrated into the FPI bus system by using an FPI2AHB adapter.

The adapter contains a slave bus interface providing translation between the FPI and the AHB bus protocols. The module supports additionally the following features:

- power saving (sleep mode)
- debug suspend
- local module reset
- PISEL - port input selection regarding the serial data input

The power saving and debug suspend features are supported by the AHB based I2C module itself, and the local module reset and PISEL features are supported by the FPI2AHB adapter, see **Figure 352**.

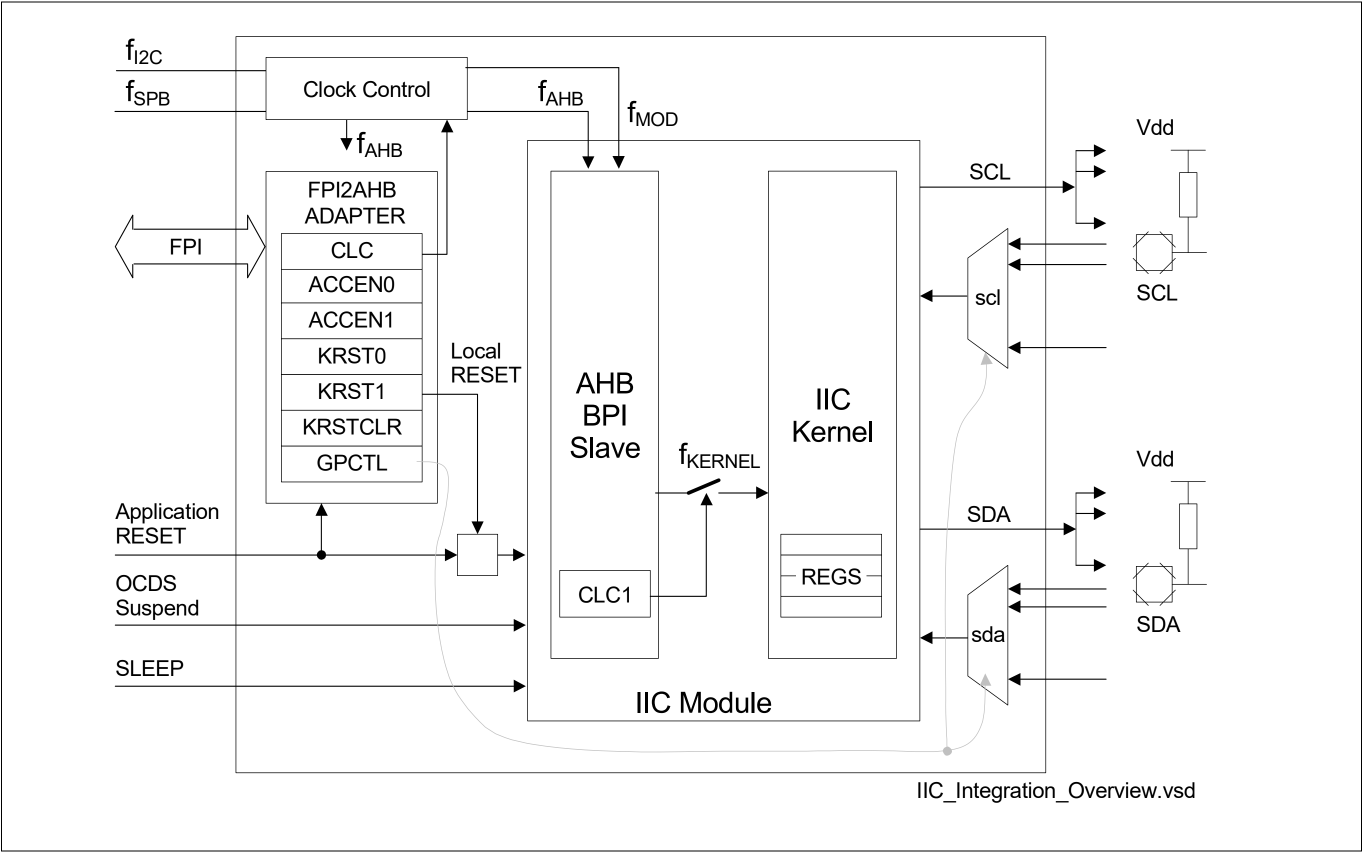

Figure 352 Integration Overview

34.3.3.2 BPI_SPB Module Registers Overview

Figure 353 shows all registers associated with the BPI_FPI module, configured for one kernel. See also **Table 303** for the address mapping.

BPI_FPI Registers Overview

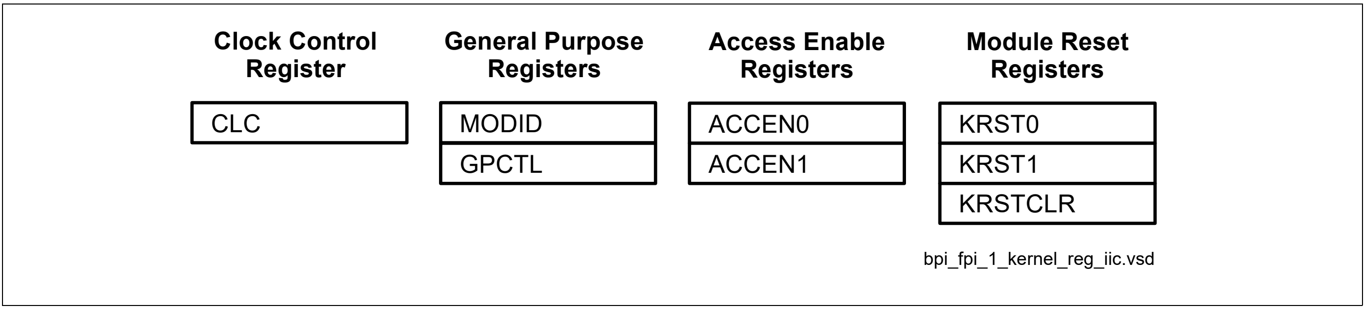

Figure 353 BPI_FPI Registers

This section describes the registers implemented in the slave (FPI2AHB) adapter component.

34.3.3.2.1 BPI_SPB Module Registers

Kernel Reset

If a kernel reset is requested, the adapter will synchronously assert the internal kernel reset output and error any ongoing accesses on the FPI bus not addressed to the adapter's internal SFRs.

The method of using the kernel reset output from the adapter to initialise the associated AHB module is module specific and outside the scope of this specification.

Any AHB accesses in progress will be synchronously terminated.

GPCTL

A single, adapter specific, SFR is implemented in the adaptor.

The SFR contains general purpose read/write control bits without ENDINIT protection. All the bits are connected to output ports on the FPI2AHB entity. The SFR will implement no safety requirements and is not be suitable for controlling hardware related to safety functions. The SFR supports, byte, half word and word transactions only. All other transactions are rejected with an FPI error termination.

Note: In the I2C module, this register implements the PISEL functionality. The SDA and SCL input multiplexors are controlled in parallel with each PISEL setting. Since 3 bits are implemented for PISEL, up to 8 sets of SDA and SCL pins can be defined.

Principle of Operation

When the controlling state machine detects an FPI access, it compares the address to the address of the SFR. In the case that the address matches the SFR address, the normal state machine transitions are interrupted and the access is not passed to the AHB bus.

If the access opcode is not SDTB, SDTH or SDTW, then the access is terminated with an FPI error. If the opcode is supported, the SFR data will be returned on the FPI bus if the transaction is a read or the appropriate bits of the register will be updated if the access is a write.

Clock Control Register

The Clock Control Register, CLC, acts globally and allows the complete AHB module to be disabled to reduce power consumption when the module is not required. When the module is disabled (DISS=1_B), only register accesses to the adapter register address space are permitted. All other accesses to the module address space will be errored.

CLC
Clock Control Register **(10000_H)** **Application Reset Value: $0000\ 0003_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0														**DISS**	**DISR**
r														rh	rw

Field	Bits	Type	Description
DISR	0	rw	**Module Disable Request Bit** Used for enable/disable control of the module.
DISS	1	rh	**Module Disable Status Bit** Bit indicates the current status of the module.
0	31:2	r	**Reserved** Read as 0; should be written with 0.

Module Identification Register

The identification register allows the programmer version-tracking of the module.

MODID
Module Identification Register **(10004_H)** **Application Reset Value: $00C2\ C0XX_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MOD_NUMBER															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOD_TYPE								**MOD_REV**							
r								r							

Field	Bits	Type	Description
MOD_REV	7:0	r	**Module Revision Number** This bit field defines the module revision number. The value of a module revision starts with 01H (first revision).
MOD_TYPE	15:8	r	**Module Type** The bit field is set to C0H which defines the module as a 32-bit module.
MOD_NUMBER	31:16	r	**Module Number Value** This bit field defines a module identification number.

General Purpose Control Register

GPCTL

General Purpose Control Register **(10008$_H$)** **Application Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													**PISEL**		
r													rw		

Field	Bits	Type	Description
PISEL	2:0	rw	**Port Input Select** Used to select the input pins providing the serial data and clock input signals, in the range of 0 to 7. *Note: In AURIX™ TC3xx Platform, not all PISEL options are available. See Data Sheet.* 000$_B$ SDA0A and SCL0A are selected. 001$_B$ SDA0B and SCL0B are selected. 010$_B$ SDA0C and SCL0C are selected. 011$_B$ SDA0D and SCL0D are selected. 100$_B$ SDA0E and SCL0E are selected. 101$_B$ SDA0F and SCL0F are selected. 110$_B$ SDA0G and SCL0G are selected. 111$_B$ SDA0H and SCL0H are selected.
0	31:3	r	**reserved** Reads 0. Should be written with 0.

Access Enable Register 0

The Access Enable Register 0 controls write access[1] for transactions to registers with the on chip bus master TAG ID 000000B to 011111B (see On Chip Bus chapter for the products TAG ID <-> master peripheral mapping). The adapter is prepared for an 6 bit TAG ID. The registers ACCEN0 / ACCEN1 are providing one enable bit for each possible 6 bit TAG ID encoding.

Mapping of TAG IDs to ACCEN0.ENx: EN0 -> TAG ID 000000B, EN1 -> TAG ID 000001B,...,EN31 -> TAG ID 011111B.

1) The BPI_FPI Access Enable functionality controls only write transactions to the CLC, KRSTx and the kernel registers. Read transactions are not influenced. SW has to take care for destructive/modifying read functionality in kernel registers.

ACCEN0

Access Enable Register 0 **($1000C_H$)** **Application Reset Value: FFFF $FFFF_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
EN31	**EN30**	**EN29**	**EN28**	**EN27**	**EN26**	**EN25**	**EN24**	**EN23**	**EN22**	**EN21**	**EN20**	**EN19**	**EN18**	**EN17**	**EN16**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN15	**EN14**	**EN13**	**EN12**	**EN11**	**EN10**	**EN9**	**EN8**	**EN7**	**EN6**	**EN5**	**EN4**	**EN3**	**EN2**	**EN1**	**EN0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
ENj (j=0-31)	j	rw	**Access Enable for Master TAG ID j** This bit enables write access to the module register addresses for transactions with the Master TAG ID j 0_B Write access will not be executed. Read accesses will be executed. 1_B Write and read accesses will be executed

Access Enable Register 1

The Access Enable Register 1 controls write access for transactions with the on chip bus master TAG ID 100000B to 111111B (see On Chip Bus chapter for the products TAG ID <-> master peripheral mapping). These tags are not used in this system and so no programmable bits are provided.

ACCEN1

Access Enable Register 1 **(10010_H)** **Application Reset Value: 0000 0000_H**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES															
r															

Field	Bits	Type	Description
RES	31:0	r	**Reserved** Read as 0; should be written with 0.

Kernel Reset Register 0

The Kernel Reset function is used to synchronously reset the AHB module. To activate the kernel reset, it is necessary to set the RST bits by writing with 1_B in both Kernel Reset Registers.The RST bit will be re-set by the adapter with the end of the adapter kernel reset sequence.

Kernel Reset Register 0 includes a kernel reset status bit that is set to 1_B in the same clock cycle the RST bit is reset. This bit can be used to detect that a kernel reset was processed. The bit can be re-set to ´0´ by writing ´1´ to the KRSTCLE.CLR register bit.

During the execution of the kernel reset until RSTSTAT is set, write accesses to the module registers will result in an error acknowledge. Adapter registers can still be accessed.

KRST0

Kernel Reset Register 0 **(10014_H)** **Application Reset Value: 0000 0000_H**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0														**RSTSTAT**	**RST**
r														rh	rwh

Field	Bits	Type	Description
RST	0	rwh	**Kernel Reset** This reset bit can be used to request for a kernel reset. The kernel reset will be executed if the reset bits of both kernel registers are set. The RST bit will be cleared (reset to 0_B) b after the kernel reset was executed. 0_B No kernel reset was requested 1_B A kernel reset was requested
RSTSTAT	1	rh	**Kernel Reset Status** This bit indicates whether a kernel reset was executed or not. This bit is set after the execution of a kernel reset in the same clock cycle both reset bits. This bit can be cleared by writing with ´1´ to the CLR bit in the related KRSTCLR register. 0_B No kernel reset was executed 1_B Kernel reset was executed
0	31:2	r	**Reserved** Read as 0; should be written with 0.

Kernel Reset Register 1

The Kernel Reset Register 1 is used to reset the related module kernel. Kernel registers related to the Debug Reset (Class 1) are not influenced. To reset a module kernel it is necessary to set the RST bits by writing with ´1´ in both Kernel Reset registers. The RST bit will be re-set by the BPI with the end of the BPI kernel reset sequence.

KRST1

Kernel Reset Register 1 **(10018_H)** **Application Reset Value: 0000 0000_H**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							0								
							r								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
							0								**RST**
							r								rwh

Field	Bits	Type	Description
RST	0	rwh	**Kernel Reset** This reset bit can be used to request for a kernel reset. The kernel reset will be executed if the reset bits of both kernel reset registers is set. The RST bit will be cleared (re-set to 0_B) after the kernel reset was executed. 0_B No kernel reset was requested 1_B A kernel reset was requested
0	31:1	r	**Reserved** Read as 0_B; should be written with 0_B.

Kernel Reset Status Clear Register

The Kernel Reset Status Clear register is used to clear the Kernel Reset Status bit (KRST0.RSTSTAT).

KRSTCLR

Kernel Reset Status Clear Register **($1001C_H$)** **Application Reset Value: 0000 0000_H**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
							0								
							r								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
							0								**CLR**
							r								w

Field	Bits	Type	Description
CLR	0	w	**Kernel Reset Status Clear** Read always as 0_B. 0_B No action 1_B Clear Kernel Reset Status KRST0.RSTSTAT
0	31:1	r	**Reserved** Read as 0_B; should be written with 0_B.