Freescale Semiconductor
Application Note

Document Number: AN4856
Rev. 0, 03/2014

MPC574xP SENT/SPC Driver

by: Josef Kramoliš

1 Introduction

This application note describes the SENT/SPC driver for the MPC574xP 32-bit microcontrollers. It also provides information on the Single Edge Nibble Transmission protocol (SENT, SAE J2716), along with its Short PWM Code (SPC) enhancement. Additionally, the driver implementation, API, and functional description are also discussed in this application note.

The information about the SENT protocol has been derived from the SAE J2716 Surface Vehicle Information Report, JAN2010.

The information about the SPC enhancement has been derived from the TLE4998C Data Sheet, Rev 1.0, December 2008.

The MPC574xP SENT/SPC driver itself and the example application are part of the AN4856SW package.

Contents

1 Introduction 1
2 Overview 2
2.1 SENT encoding scheme 2
2.2 SPC protocol 6
2.3 SENT physical layer 7
3 MPC574xP SENT/SPC Driver 9
3.1 Utilized MPC574xP peripherals 9
3.2 SPC bidirectional communication 9
3.3 Driver configuration 11
3.4 Application programming interface 12
3.5 Functional description 21
3.6 Memory allocation 32
3.7 Application example 33
4 Conclusion 34
5 References 35
6 Acronyms and Definitions 36

2 Overview

The Single Edge Nibble Transmission protocol is targeted for use in those applications where high-resolution data is transmitted from a sensor to the ECU. It can be considered as an alternative to conventional sensors providing analog output voltage, and for PWM output sensors. It can also be considered as a low-cost alternative to the LIN or CAN communication standards.

Applications for electric power steering, throttle position sensing, pedal position sensing, airflow mass sensing, liquid level sensing, etc., can be used as examples of target applications for SENT-compatible sensor devices.

2.1 SENT encoding scheme

SENT is a unidirectional communication standard where data from a sensor is transmitted independently without any intervention of the data receiving device (for example, the MCU). A signal transmitted by the sensor consists of a series of pulses, where the distance between consecutive falling edges defines the transmitted 4-bit data nibble representing values from 0 to 15. Total transmission time is dependent on the transmitted data values and on the clock variation of the transmitter (sensor). A consecutive SENT transmission starts immediately after the previous transmission ends (the trailing falling edge of the SENT transmission CRC nibble is also the leading falling edge of the consecutive SENT transmission synchronization/calibration pulse – see Figure 1).

A SENT communication fundamental unit of time (unit time – UT, nominal transmitter clock period) can be in the range of 3–90 µs, according to the SAE J2716 specification. The maximum allowed clock variation is ± 20% from the nominal unit time, which allows the use of low-cost RC oscillators in the sensor device.

NOTE

A 3 µs fundamental unit time will be considered as nominal for unification of further timing descriptions.

The transmission sequence consists of the following pulses:

1. Synchronization/calibration pulse (56 unit times)
2. 4-bit status nibble pulse (12 to 27 unit times)
3. Up to six 4-bit data nibble pulses (12 to 27 unit times each)
4. 4-bit checksum nibble pulse (12 to 27 unit times)

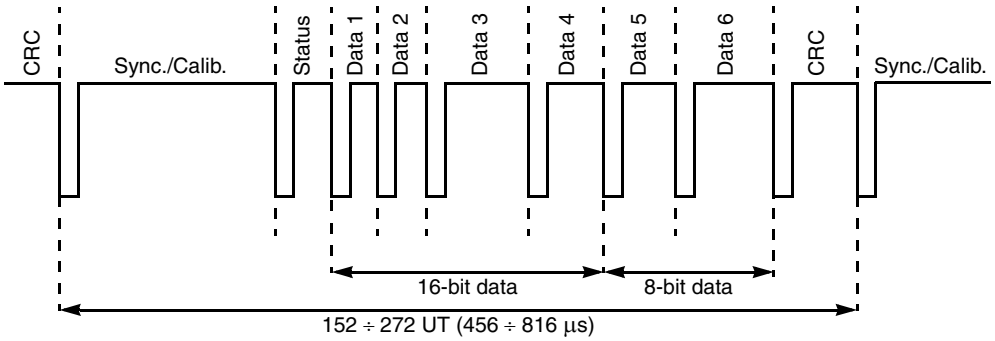

Figure 1. Transmission example of 16-bit and 8-bit signal data

2.1.1 Synchronization/calibration pulse

Since the SAE J2716 specification allows a ± 20% transmitter clock deviation from the nominal unit time, the synchronization/calibration pulse provides information on the actual transmitter (sensor) unit time period. The time between synchronization/calibration pulse falling edges defines the 56 unit time periods. The receiver can calculate the actual unit time period of the sensor from the pulse width, and can thus re-synchronize. The actual sensor data is measured during the synchronization/calibration pulse duration.

The pulse starts with the falling edge and remains low for five or more unit times. The remainder of the pulse width is driven high (see Figure 2).

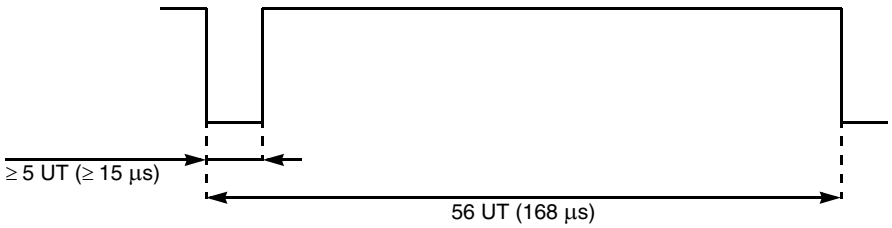

Figure 2. Synchronization/calibration pulse format

2.1.2 Status and communication nibble pulse

The status nibble contains 4-bit status information of the sensor (for example, fault indication and mode of operation). It can also carry a serial message (refer to Section 2.1.2.1, “Short serial message” and Section 2.1.2.2, “Enhanced serial message” for details).

Table 1. Status and communication nibble format

Bit	Description
3	Serial message start bit (=1), otherwise 0 or serial message data bits.
2	Serial message data bits.
1	Sensor specific.
0	Sensor specific.

The width of the status nibble pulse is dependent on the nibble value. The status nibble pulse and data nibble pulse formats are identical. Refer to Section 2.1.3, “Data nibble pulse”.

2.1.2.1 Short serial message

A 16-bit serial message can be transmitted using bit 2 of the status and communication nibble in 16 consecutive frame transmissions. Start of the short serial message is indicated by bit 3 of the status and communication nibble set to 1. Bit 3 has to be cleared in the remaining 15 frame transmissions. If at least one of 16 messages is not successfully received due to receiver diagnostic error, the short serial message is discarded by the receiver. The format of the short serial message is shown in Table 2.

Table 2. Short serial message format

Status and communication nibble received	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Status and communication nibble (bit 3)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Status and communication nibble (bit 2)	4-bit Message ID				8-bit Data Byte								4-bit CRC			

The 4-bit CRC is calculated over the 4-bit message ID and two serial data nibbles the same way as described in Section 2.1.4, “Checksum nibble pulse”.

2.1.2.2 Enhanced serial message

For larger data, an 18-bit enhanced message can be transmitted using bits 2 and 3 of the status and communication nibble in 18 consecutive frame transmissions. Two types of enhanced serial message can be determined by the configuration bit (bit 3 in the eight received status and communication nibble):

- 8-bit message ID, 12-bit data (configuration bit C = 0)
- 4-bit message ID, 16-bit data (configuration bit C = 1)

If at least one of 18 messages is not successfully received due to receiver diagnostic error, the enhanced serial message is discarded by the receiver. The format of both enhanced serial message types is shown in Table 3 and Table 4.

Table 3. Enhanced serial message format with 8-bit ID and 12-bit data (C = 0)

Status and communication nibble received	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Status and communication nibble (bit 3)	1	1	1	1	1	1	0	C=0	8-bit Message ID [7:4]				0	8-bit Message ID [3:0]				0
Status and communication nibble (bit 2)	6-bit CRC						12-bit Data											

Table 4. Enhanced serial message format with 4-bit ID and 16-bit data (C = 1)

Status and communication nibble received	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Status and communication nibble (bit 3)	1	1	1	1	1	1	0	C=1	4-bit Message ID				0	16-bit Data [15:12]				0
Status and communication nibble (bit 2)	6-bit CRC						16-bit Data [11:0]											

An enhanced serial message CRC calculation method description is beyond the scope of this document. For details, refer to SAE J2716 at www.sae.org.

2.1.3 Data nibble pulse

A single data nibble pulse carries 4-bit sensor data. A maximum of six data nibbles can be transmitted in one SENT transmission. The total number of data nibbles depends on the size of the data provided by the sensor, and this is fixed during the sensor operation (see Figure 1 for a combined 16-bit and 8-bit data transmission example). Some sensors provide the possibility of pre-programming the resolution of the measured value using special tools, thus changing the number of data nibbles.

The width of the data nibble pulse is dependent on the nibble value. Figure 3 depicts the format of the data nibble pulse. The pulse starts with the falling edge and remains low for five or more unit times. The remainder of the pulse width is driven high. The next pulse falling edge occurs after twelve unit times from the initial falling edge plus the number of unit times equal to the nibble value. The data pulse width in the number of unit times is defined by Equation 1:

Eqn. 1

$$DataNibblePulseWidth = (12 + NibbleValue)$$

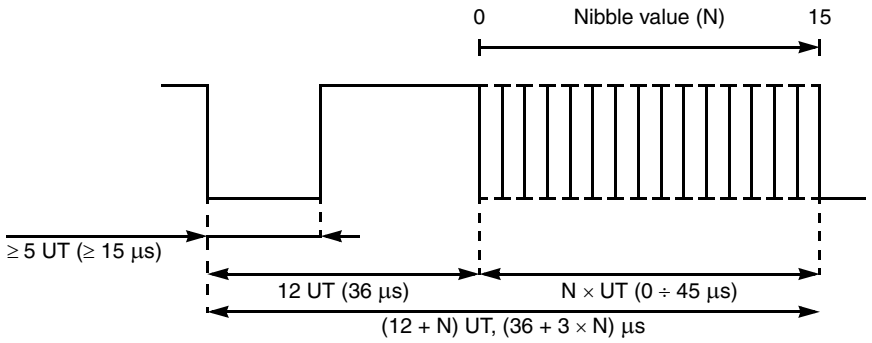

Figure 3. Data nibble pulse format

2.1.4 Checksum nibble pulse

The checksum nibble contains a 4-bit CRC. The checksum is calculated using the $x^4 + x^3 + x^2 + 1$ polynomial with the seed value of 5 (0b0101), and is calculated over all nibbles except for the status and communication nibble (according to SAE J2716).

The CRC allows detection of the following errors:

1. All single bit errors.
2. All odd number of nonconsecutive errors.
3. All single burst errors of length ≤ 4.
4. 87.5% of single burst errors of length = 5.
5. 93.75% of single burst errors of length > 5.

Refer to SAE J2716 (www.sae.org) for more information about the SENT CRC polynomial error detection.

2.2 SPC protocol

The SPC protocol enhances the SENT protocol defined by the SAE 2716 specification. SPC introduces a half-duplex synchronous communication. The receiver (MCU) generates the master trigger pulse on the sensor interface by pulling it low for a defined amount of time (t_{MT}). The pulse width is measured by the transmitter (sensor) and the SENT transmission is initiated only if the width is within defined limits. The end pulse is generated additionally after the SENT transmission has completed to provide a trailing falling edge for the CRC nibble pulse. The sensor interface then remains idle until a new master trigger pulse is generated by the receiver. Figure 4 depicts the SENT/SPC frame format.

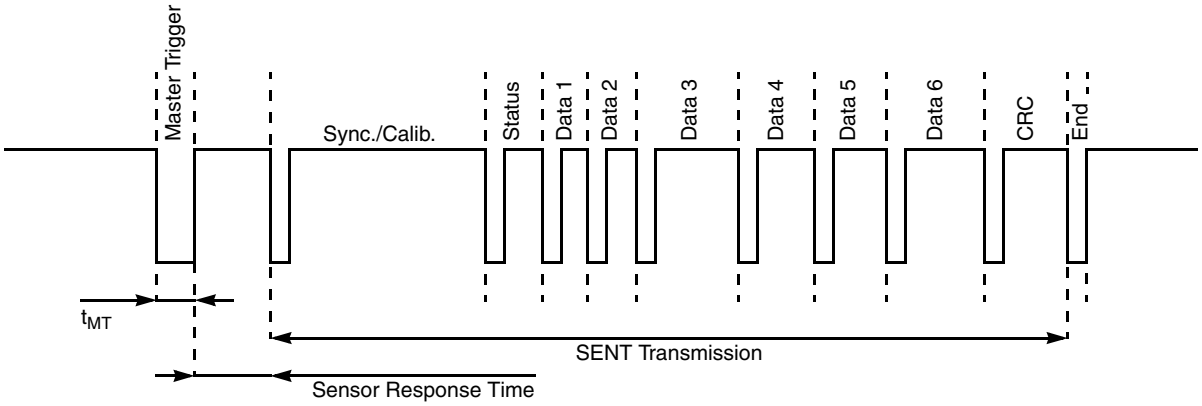

Figure 4. SENT/SPC frame format

The SPC protocol allows choosing between various protocol modes. For example, the TLE4998C Hall sensor can be pre-programmed in one of three protocol modes:

1. Synchronous mode – a single sensor is connected to the MCU; a master trigger pulse width in a defined range triggers the transmission.
2. Synchronous mode with range selection – a single sensor is connected to the MCU; the width of the master trigger pulse defines the magnetic range for the triggered transmission.
3. Synchronous transmission with ID selection – up to four sensors are connected in parallel to the MCU; the width of the master trigger pulse defines which sensor will start the transmission.

2.3 SENT physical layer

The receiver side (ECU) provides the stabilized 5 V voltage to supply the sensor. The sensor interface is pulled up by the 10 ÷ 51 kΩ resistor to the supply voltage. The receiver input is formed by the parasitic capacitance of the input pin and its ESD protection, and the 560 Ω/2.2 nF EMC low-pass filter to suppress RF noise coupled to the sensor interface. The open-drain output pin on the MCU pulls down the sensor interface to generate the master trigger pulse. See Figure 5.

The transmitter provides a bidirectional open-drain I/O pin with an EMC filter to suppress the RF noise coupled to the sensor interface. The sensor interface is pulled down by its output driver to generate the SENT pulse sequence. See Figure 5.

Signal shaping is required to limit the radiated emissions. The maximum limits of the falling and rising edge durations for the 3 µs nominal clock tick are T_{FALL} = 6.5 µs and T_{RISE} = 18 µs with a maximum allowed 0.1 µs falling edge jitter (these values can be increased proportionally for higher clock tick times). An example of a TLE4998S SENT waveform and one of a TLE4998C SENT/SPC compatible Hall sensor waveform are shown in Figure 6 and Figure 7.

The overall resistance of all connectors is limited to 1 Ω, the bus wiring to 0.1 nF/m capacitance, and the maximum cable length to 5 m.

The transmitter-receiver network devices are protected from short-to-ground and short-to-supply conditions. Upon recovery from these faults, normal operation is resumed.

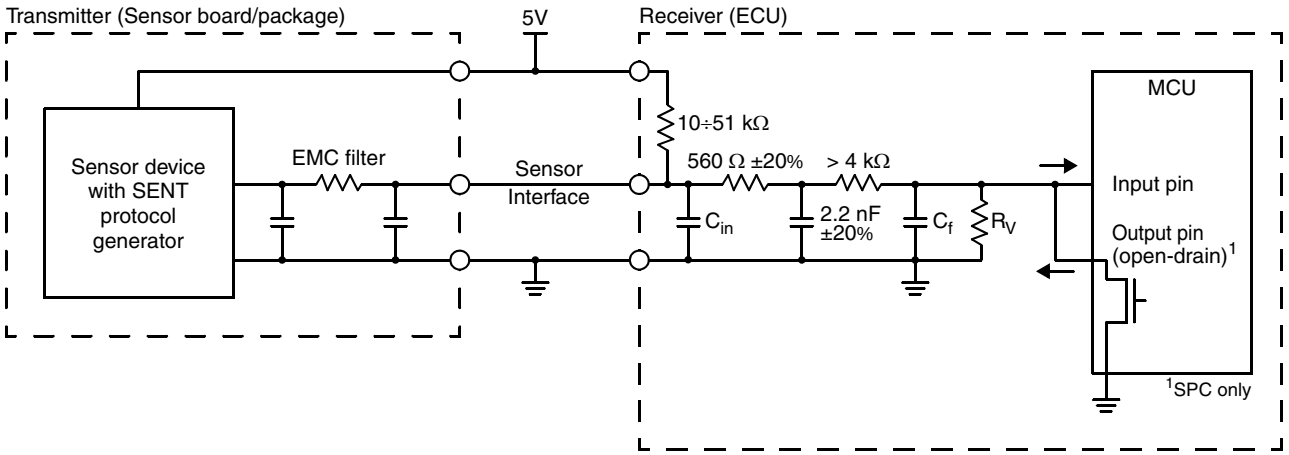

Figure 5. SENT/SPC circuit topology

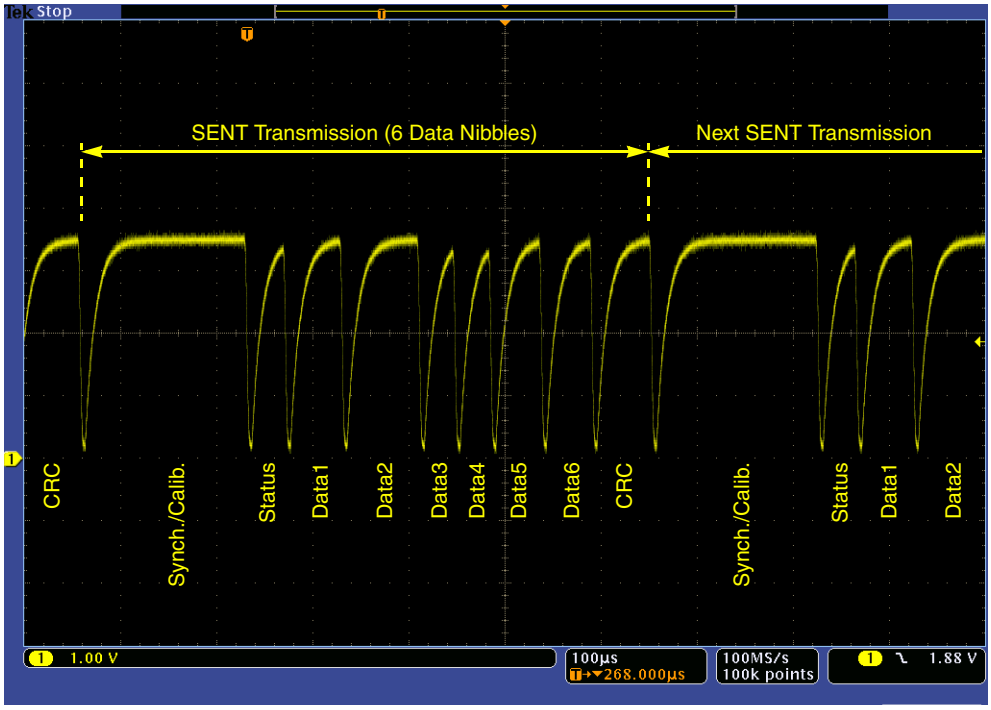

Figure 6. TLE4998S SENT 16-bit Hall, 8-bit temperature waveform

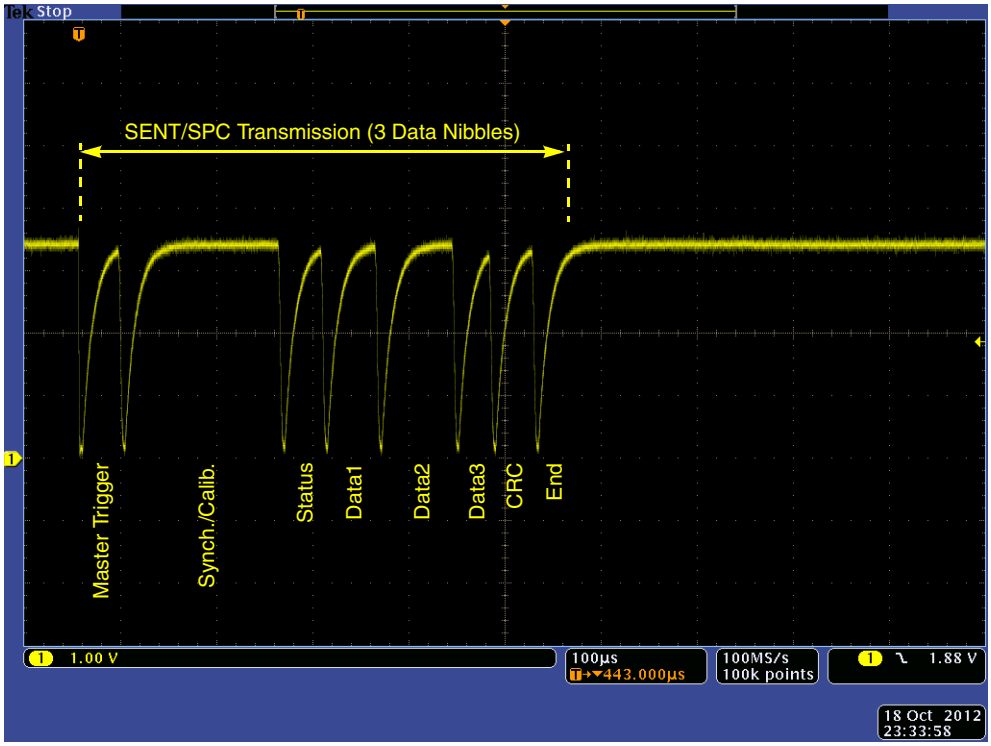

Figure 7. TLE4998C SENT/SPC 12-bit Hall waveform

3 MPC574xP SENT/SPC Driver

The driver is provided as an example code only, and in the form of source code optimized for the Green Hills compiler. It is intended for use with the MPC574xP microcontroller and any SENT or SENT/SPC compatible sensor. The driver supports up to four independent channels and reception of two main message types:

1. Message consisting of the data from a status and communication nibble, and all data nibbles received from a single SENT frame (further referred to as a fast message)
2. Message consisting of the data from status and communication nibble bits 2 and 3 in 16 or 18 consecutively received SENT frames (further referred to as a serial message)

3.1 Utilized MPC574xP peripherals

The driver utilizes the following MPC57xP peripherals:

- System Integration Unit Lite 2 (SIUL2) module
- SENT Receiver (further referred to as SRX) modules – 1 module channel per driver channel
- Enhanced Motor Control Timer (eTimer) modules – 1 module channel per driver channel operating in the SENT/SPC mode
- Enhanced Direct Memory Access (eDMA) module – 1 channel per SRX module with at least one channel with the fast message FIFO enabled in the channel configuration

3.2 SPC bidirectional communication

The driver is designed to support various sensor to MCU connection options (see Figure 8).

1. Single-pin (open-drain) – the sensor interface is connected to the MCU via a single open-drain input/output pin shared between the SRX and eTimer modules.
2. Two-pin (open-drain) – the sensor interface is connected to the MCU via two pins connected together, the SRX input pin and eTimer open-drain output pin.
3. Two-pin (push-pull) – the sensor interface is connected to the MCU via a single SRX input pin, the eTimer push-pull output pin drives an external transistor connected to the sensor interface.

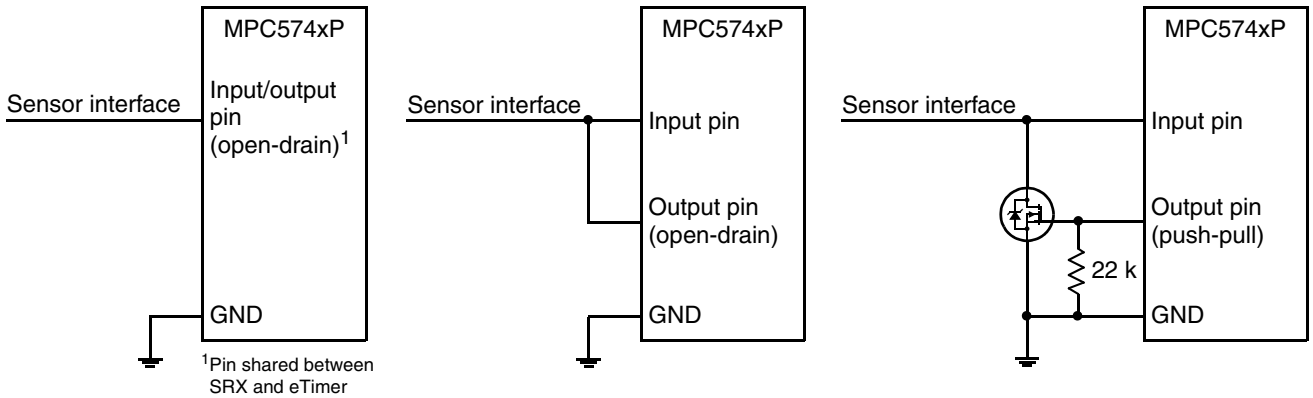

Figure 8. Sensor to MCU connection options

Generally, SENT frame data is received on the input pin, while the master trigger pulse is generated either by the eTimer open-drain output directly connected to the sensor interface or by the eTimer push-pull output via an external transistor connected to the sensor interface. Open-drain output can be enabled by the SentSpcEtimerOdEnable configuration parameter (see Section 3.3, “Driver configuration”).

Figure 9 shows an example of the TLE4998C programmable Hall sensor application circuit with an external MOSFET transistor.

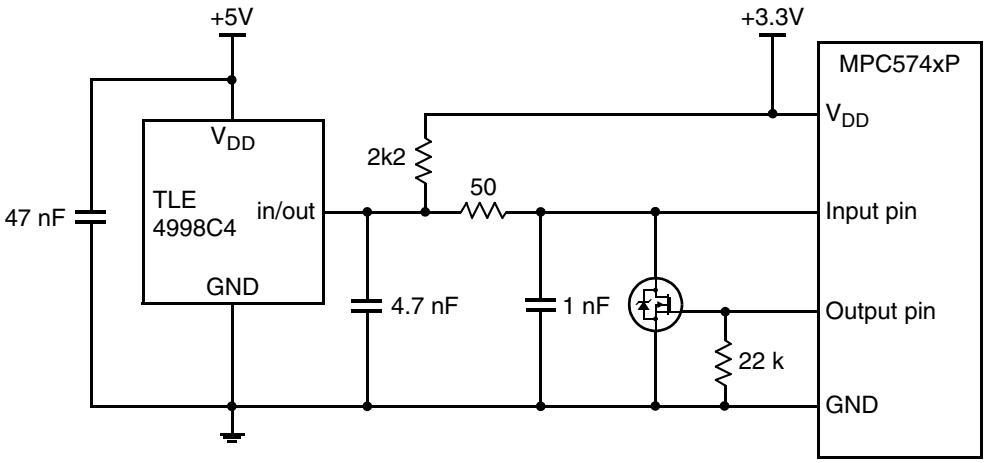

Figure 9. Typical TLE4998C application circuit with external MOSFET transistor

3.3 Driver configuration

The SENT/SPC Driver is configurable via the MPC574xP SENT/SPC Driver Configurator application. The application is supplied in the form of an Excel macro-enabled workbook SentSpc_Configurator.xlsm. Microsoft Excel® 2007 or newer, with the Visual Basic for Applications feature installed, is required to run the application.

Driver configuration parameters are stored in files SentSpc_Driver_Cfg.h and SentSpc_Driver_Cfg.c generated by the configurator application in the form of a configuration structure definition.

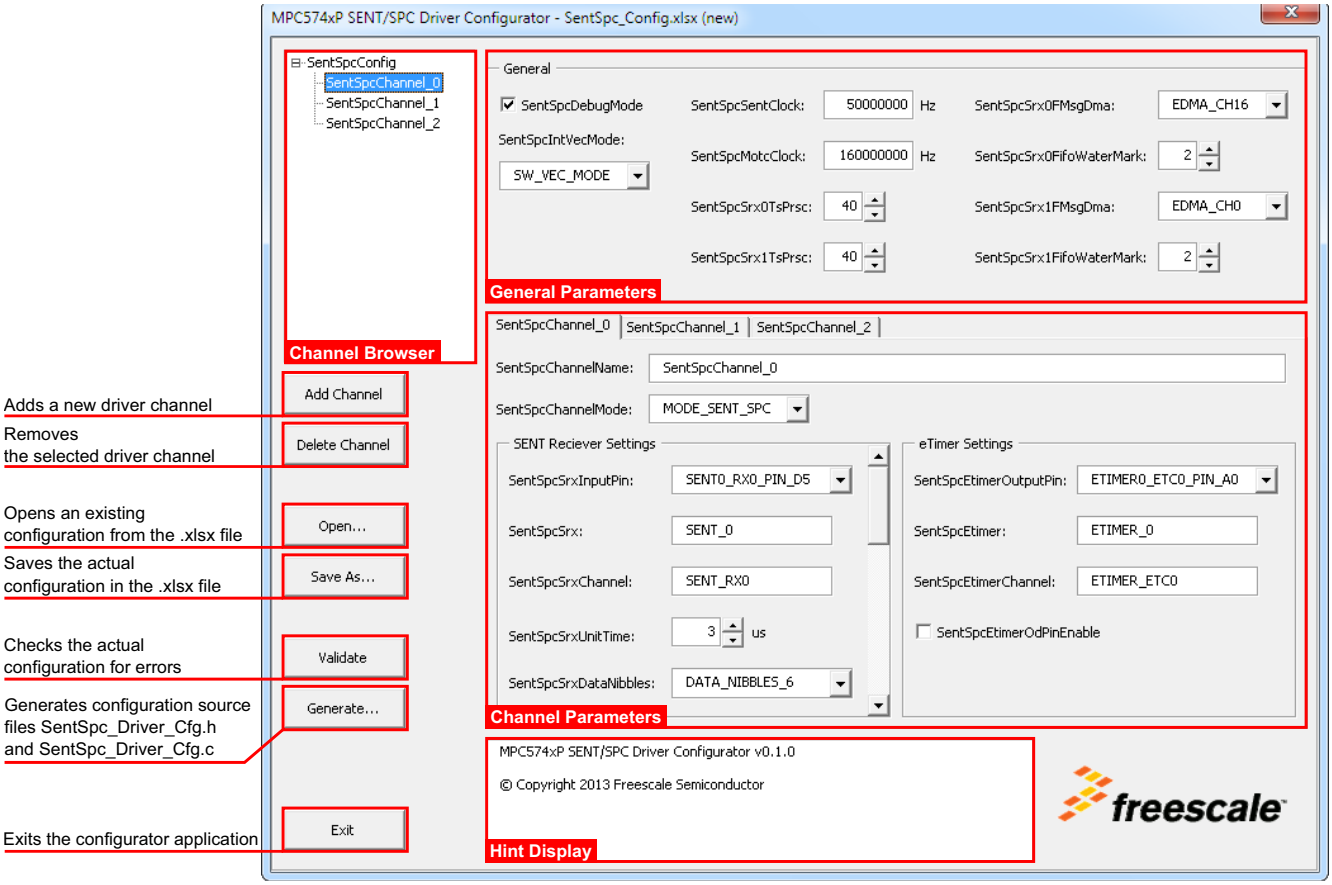

Figure 10. MPC5744P SENT/SPC driver configurator application window

For a detailed list of the configuration parameters, refer to the driver user manual document (SENTSPCMPC574XPUM.pdf) which is part of the AN4856SW.

3.4 Application programming interface

This section provides the description of the API return type and the list of driver API functions.

3.4.1 API return type

The driver API functions return values based on the enumeration type SentSpc_ReturnType. See Table 5.

Table 5. Enumeration SentSpc_ReturnType

Value	Description
SENT_SPC_OK	Function executed successfully.
SENT_SPC_E_NOT_OK	Function executed with an error.
SENT_SPC_E_ALREADY_INIT[1]	SentSpc_Init() has been called while the SENT/SPC Driver is already initialized.
SENT_SPC_E_UNINIT[1]	SentSpc_Init() has not been called prior to the function call.
SENT_SPC_E_PARAM_CONFIG[1]	Function called with an incorrect configuration parameter (configuration pointer is NULL_PTR).
SENT_SPC_E_PARAM_CHANNEL[1]	Invalid channel ID passed as a function argument.
SENT_SPC_E_PARAM_MODULE[1]	SentSpc_GetFastMsgFifoData() function called with an invalid SRX module ID or the requested SRX module has no channel configured with fast message FIFO enabled.
SENT_SPC_E_PARAM_MODE[1]	SentSpc_Request() function called on a channel configured in SENT mode.

[1] These development errors are reported only if the driver is configured in the debug mode (SentSpcDebugMode = true).

3.4.2 API functions

3.4.2.1 SentSpc_Init()

The function initializes the SRX, eTimer, SIUL2, and eDMA hardware modules and the driver structures.

NOTE

The function has to be called prior to any other driver function call.

Prototype: `SentSpc_ReturnType SentSpc_Init(const SentSpc_ConfigType *ConfigPtr);`

Table 6. SentSpc_Init() arguments

Type	Name	Direction	Description
const SentSpc_ConfigType *	ConfigPtr	Input	Pointer to the configuration.

3.4.2.2 SentSpc_Request()

The function requests the master trigger pulse generation on the sensor interface of a selected channel (see Section 3.2, “SPC bidirectional communication” for sensor to MCU option).

Prototype: `SentSpc_ReturnType SentSpc_Request(uint8_t ui8ChannelIndex, uint8_t ui8MasterTime);`

Table 7. SentSpc_Request() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
uint8_t	ui8MasterTime	Input	External transistor gate/open-drain output pin driving pulse width in microseconds.

NOTE

The master trigger pulse width is dependent on the sensor interface resistor/capacitor parameters and the operating temperature, and it is always wider than the gate pulse width defined by the SentSpc_Request() function input parameter (see Figure 11). The user shall ensure (e.g. by a measurement) that the master trigger pulse width will be always within proper limits (consult the target sensor data sheet for the limit values).

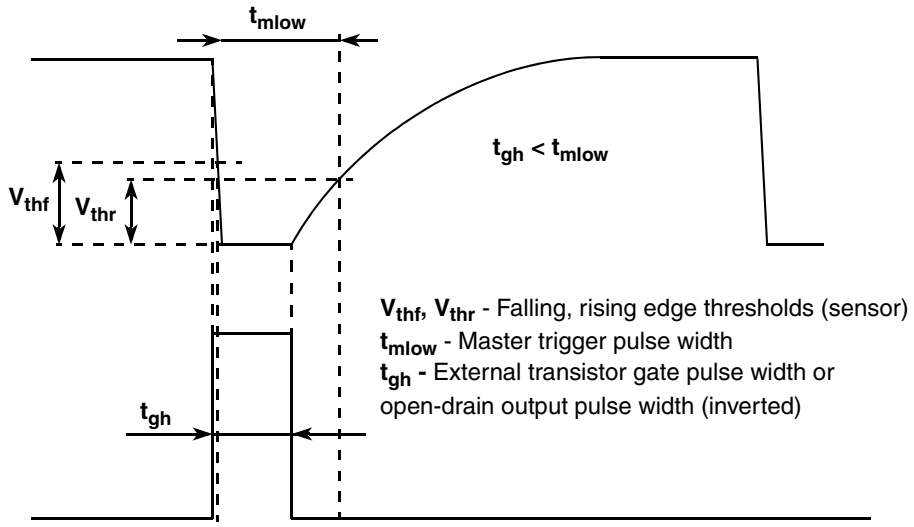

Figure 11. Master trigger pulse timing

External transistor gate pulse width values shown in the Table 8, Table 9, and Table 10 correspond with the typical master trigger pulse widths as defined in the TLE4998C Data Sheet, V 1.0, December 2008. Equivalent gate pulse widths were obtained by measurement at 23°C ambient temperature using a typical application circuit with a BSS138 N-channel MOSFET transistor.

Table 8. Typical master trigger pulse timing for Synchronization Mode of the TLE4998C sensor

Define	Notes	Master trigger pulse width [UT]	Pulse width [µs]
SPC_SYNCH	–	2.75	4

Table 9. Typical master trigger pulse timing for ID Selection Mode of the TLE4998C sensor

Define	Notes	Master trigger pulse width [UT]	Pulse width [µs]
SPC_ID_0	ID = 0	10.5	28
SPC_ID_1	ID = 1	21	59
SPC_ID_2	ID = 2	38	110
SPC_ID_3	ID = 3	64.5	190

Table 10. Typical master trigger pulse timing for Dynamic Range Mode of the TLE4998C sensor

Define	Notes	Master trigger pulse width [UT]	Pulse width [µs]
SPC_RANGE_200	Range = 200 mT	3.25	6
SPC_RANGE_100	Range = 100 mT	12	32
SPC_RANGE_50	Range = 50 mT	31.5	91

3.4.2.3 SentSpc_GetFastMsgData()

The function reads and decodes received raw fast message data from the channel's internal buffer.

A SW based CRC calculation is performed on the data from the channels with the SW based CRC calculation method selected in the configuration.

A configured number of data nibble values, concatenated into 16-bit right-aligned format (data nibble 1 as most significant), the status nibble value, and the 32-bit transmission time stamp are then stored in the destination variable of data type SentSpc_ChannelDataType.

NOTE

If the SW based calculated CRC value is not identical to the received CRC nibble value, data from the entire fast message are discarded. The error notification function is then called by the driver if the error notification mechanism for that particular channel is enabled using the SentSpc_EnableErrNotification() function.

Prototype:
```
SentSpc_ReturnType SentSpc_GetFastMsgData(uint8_t ui8ChannelIndex,
SentSpc_ChannelDataType *DestDataPtr);
```

Table 11. SentSpc_GetFastMsgData() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_ChannelDataType *	DestDataPtr	Output	Pointer to where to store the data.

3.4.2.4 SentSpc_GetFastMsgFifoData()

The function reads and decodes raw fast message data from the SRX module's fast message FIFO (module selected by the ui8SrxModuleIndex input parameter). It loops through the internal FIFO buffer entries starting with the last message in the buffer based on the FIFO water mark level value configured. If multiple fast messages from the same channel are found in the buffer, only the most recent message data is stored in the destination variable.

A SW based CRC calculation is performed on the data from the channels with the SW based CRC calculation method selected in the configuration.

A configured number of data nibble values, concatenated into 16-bit right-aligned format (data nibble 1 as most significant), the status nibble value, and the 32-bit transmission time stamp from each channel found in the FIFO are stored into the destination array of data type SentSpc_ChannelDataType at the index position equal to the respective channel's numeric ID.

NOTE

If the SW based calculated CRC value is not identical to the received CRC nibble value, data from the entire fast message are discarded. The error notification function is then called by the driver if the error notification mechanism for that particular channel is enabled using the SentSpc_EnableErrNotification() function.

Prototype: `SentSpc_ReturnType SentSpc_GetFastMsgFifoData(uint8_t ui8SrxModuleIndex, SentSpc_ChannelDataType DestDataArray[]);`

Table 12. SentSpc_GetFastMsgFifoData() arguments

Type	Name	Direction	Description
uint8_t	ui8SrxModuleIndex	Input	Numeric ID of the SRX module.
SentSpc_ChannelDataType	DestDataArray[]	Output	Array where to store the data.

3.4.2.5 SentSpc_GetSerialMsgData()

The function reads and decodes received raw serial message data from the channel's internal buffer. The message ID, data, and the 32-bit time stamp are then stored in the destination variable of data type SentSpc_ChannelDataType.

Prototype: `SentSpc_ReturnType SentSpc_GetSerialMsgData(uint8_t ui8ChannelIndex, SentSpc_ChannelDataType *DestDataPtr);`

Table 13. SentSpc_GetSerialMsgData() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_ChannelDataType *	DestDataPtr	Output	Pointer to where to store the data.

3.4.2.6 SentSpc_EnableRxNotification()

The function enables the fast or slow message reception notification mechanism for the selected driver channel.

Prototype: `SentSpc_ReturnType SentSpc_EnableRxNotification(uint8_t ui8ChannelIndex, SentSpc_MessageType ui8TypeOfMessage);`

Table 14. SentSpc_EnableRxNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_MessageType	ui8TypeOfMessage	Input	SENT_SPC_FAST_MSG, SENT_SPC_SERIAL_MSG

3.4.2.7 SentSpc_DisableRxNotification()

The function disables the fast or slow message reception notification mechanism for the selected driver channel.

Prototype: `SentSpc_ReturnType SentSpc_DisableRxNotification(uint8_t ui8ChannelIndex, SentSpc_MessageType ui8TypeOfMessage);`

Table 15. SentSpc_DisableRxNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_MessageType	ui8TypeOfMessage	Input	SENT_SPC_FAST_MSG, SENT_SPC_SERIAL_MSG

3.4.2.8 SentSpc_EnableErrNotification()

The function enables the error notification mechanism for the selected driver channel.

Prototype: `SentSpc_ReturnType SentSpc_EnableErrNotification(uint8_t ui8ChannelIndex);`

Table 16. SentSpc_EnableErrNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.

3.4.2.9 SentSpc_DisableErrNotification()

The function disables the error notification mechanism for the selected driver channel.

Prototype: `SentSpc_ReturnType SentSpc_DisableErrNotification(uint8_t ui8ChannelIndex);`

Table 17. SentSpc_DisableErrNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.

3.4.2.10 SentSpc_GetChannelStatus()

The function retrieves the actual channel status information.

Prototype: `SentSpc_ReturnType SentSpc_GetChannelStatus(uint8_t ui8ChannelIndex, SentSpc_ChannelStatusType *DestPtr);`

Table 18. SentSpc_GetChannelStatus() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_ChannelStatusType *	DestPtr	Output	Pointer to where to store the status.

3.4.3 Notification functions

Notification functions notify the application about asynchronous events such as the reception of a fast or serial message, reaching the water mark level value of messages in the SRX fast message FIFO, and a channel diagnostic error.

3.4.3.1 SentSpc_FastMsgNotification()

The function notifies the application that the channel fast message data is ready to be read by the SentSpc_GetFastMsgData() function. The notification function is called by the driver only if the fast message notification mechanism for a particular channel is enabled using the SentSpc_EnableRxNotification() function.

Prototype: `void SentSpc_FastMsgRxNotification(uint8_t ui8ChannelIndex);`

NOTE

Function SentSpc_FastMsgNotification() has to be manually defined in the application. The body of the function is user-defined.

Table 19. SentSpc_FastMsgNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.

3.4.3.2 SentSpc_FastMsgFifoNotification()

The function notifies the application that the fast message FIFO of a particular SRX module has reached the configured water mark level and the FIFO data is available to be read from the internal buffer using the SentSpc_GetFastMsgFifoData() function.

Prototype: `void SentSpc_FastMsgFifoNotification(uint8_t ui8SrxModuleIndex);`

NOTE

Function SentSpc_FastMsgFifoNotification() has to be manually defined in the application. The body of the function is user-defined.

Table 20. SentSpc_FastMsgFifoNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8SrxModuleIndex	Input	Numeric ID of the SRX module.

3.4.3.3 SentSpc_SerialMsgNotification()

The function notifies the application that the channel serial message data is ready to be read by the SentSpc_GetSerialMsgData() function. The notification function is called by the driver only if the serial message notification mechanism for a particular channel is enabled using the SentSpc_EnableRxNotification() function.

Prototype: `void SentSpc_FastMsgRxNotification(uint8_t ui8ChannelIndex);`

NOTE

Function SentSpc_SerialMsgNotification() has to be manually defined in the application. The body of the function is user-defined.

Table 21. SentSpc_SerialMsgNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.

3.4.3.4 SentSpc_ErrorNotification()

The function notifies the application that a particular channel error was detected.

Prototype: `void SentSpc_ErrorNotification(uint8_t ui8ChannelIndex, SentSpc_ChannelErrorType ChannelError);`

NOTE

Function SentSpc_ErrorNotification() has to be manually defined in the application. The body of the function is user-defined.

Table 22. SentSpc_ErrorNotification() arguments

Type	Name	Direction	Description
uint8_t	ui8ChannelIndex	Input	Numeric ID of the driver channel.
SentSpc_ChannelErrorType	ChannelError	Output	Channel error status.

3.4.4 Interrupt service routines

This section lists the driver interrupt service routines. Interrupt service routines are responsible for:

1. Raw fast message data transfers between SRX data registers and the driver channel buffer (of those channels not configured to use the SRX fast message FIFO), and fast message data ready notification.
2. Raw serial message data transfers between SRX data registers and the driver channel buffer, and for serial message data ready notification.
3. Channel internal status updates (if a channel error occurs) and error notification.
4. Fast message FIFO data ready notification.

3.4.4.1 Fast message interrupt service routines

Fast message ISR functions listed in Table 23 are used by the driver.

Table 23. Fast message interrupt service routines

ISR Name	Interrupt vector	Source signal
SentSpc_Srx0_Rx0_FastMsg_Isr()	570	Valid fast message received on SENT0_RX0
SentSpc_Srx0_Rx1_FastMsg_Isr()	573	Valid fast message received on SENT0_RX1
SentSpc_Srx1_Rx0_FastMsg_Isr()	582	Valid fast message received on SENT1_RX0
SentSpc_Srx1_Rx1_FastMsg_Isr()	585	Valid fast message received on SENT1_RX1

The fast message ISR is executed once a valid fast message has been received by the SRX module and combines the processing of both fast and serial message types. See Section 3.5.2, “Message reading via interrupts” for more details.

NOTE

The fast message ISR function of a particular SRX channel is available only if its related driver channel is configured with fast message FIFO disabled (SentSpcSrxFastMsgFifoEnable = false in configuration).

3.4.4.2 Serial message interrupt service routines

The serial message ISR functions listed in Table 24 are used by the driver.

Table 24. Serial message interrupt service routines

ISR Name	Interrupt vector	Source signal
SentSpc_Srx0_Rx0_SerialMsg_Isr()	571	Valid serial message received on SENT0_RX0
SentSpc_Srx0_Rx1_SerialMsg_Isr()	574	Valid serial message received on SENT0_RX1
SentSpc_Srx1_Rx0_SerialMsg_Isr()	583	Valid serial message received on SENT1_RX0
SentSpc_Srx1_Rx1_SerialMsg_Isr()	586	Valid serial message received on SENT1_RX1

Serial message ISR is executed once a valid serial message has been received by the SRX module.

The ISR calls the processing function which copies data from the SRX channel serial message data registers into the internal buffer of the related driver channel. If the channel serial message reception notification is enabled, serial message notification function SentSpc_SerialMsgNotification() is executed as part of the ISR.

NOTE

The serial message ISR function of a particular SRX channel is available only if its related driver channel is configured with fast message FIFO enabled (SentSpcSrxFastMsgFifoEnable = true in configuration).

3.4.4.3 End of eDMA Transfer interrupt service routines

End of eDMA Transfer ISR functions listed in Table 25 are used by the driver.

Table 25. End of eDMA Transfer interrupt service routines

ISR Name	Interrupt vector	Source signal	Source module
SentSpc_Srx0_Dma_Isr()	69–84	eDMA Channel [16–31]	eDMA
SentSpc_Srx1_Dma_Isr()	53–68	eDMA Channel [0–15]	eDMA

The End of eDMA Transfer ISR function is executed once a number of entries, equal to the configured FIFO water mark level value of the related SRX module, have been transferred by the eDMA engine from the FIFO into the internal buffer of the driver. The fast message FIFO notification function SentSpc_FastMsgFifoNotification() is executed as part of the ISR.

NOTE

The SentSpc_Srx*N*_DMA_Isr() (where *N* is the index of the SRX module) ISR is available only if at least one driver channel is configured to utilize SRX_*N* with fast message FIFO enabled. The interrupt vector number of the ISR is defined by the value of configuration parameter SentSpcSrx*N*FMsgDma + 53.

3.4.4.4 Channel error interrupt service routines

Channel error ISR functions listed in Table 26 are used by the driver.

Table 26. Channel error interrupt service routines

ISR Name	Interrupt vector	Source signal
SentSpc_Srx0_Rx0_Err_Isr()	572	SENT0_RX0 receive error interrupt
SentSpc_Srx0_Rx1_Err_Isr()	575	SENT0_RX1 receive error interrupt
SentSpc_Srx1_Rx0_Err_Isr()	584	SENT1_RX0 receive error interrupt
SentSpc_Srx1_Rx1_Err_Isr()	587	SENT1_RX1 receive error interrupt

The error interrupt ISR is executed if a channel error is detected during a message reception. If channel error notification is enabled, the notification function SentSpc_ErrorNotification() is executed as part of the ISR.

3.5 Functional description

This section provides a functional description of the MPC574xP SENT/SPC driver.

3.5.1 Driver initialization and SENT data acquisition

The SENT data acquisition is completely handled by the SRX module linked to the driver channel. The SentSpc_Init() API function initializes the SRX module based on the channel configuration parameter values. The SRX module's channel properly receives only messages with the format specified by the configuration parameters.

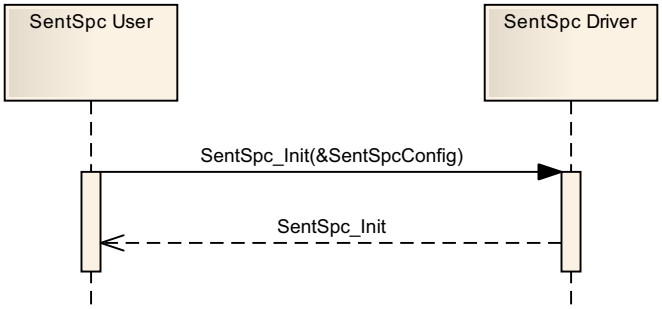

Figure 12. Driver initialization

3.5.2 Message reading via interrupts

The driver channel is configured for the interrupt operation if the SentSpcSrxFastMsgFifoEnable configuration parameter is set to false.

Reception of each valid fast message invokes the fast message interrupt related to the SRX module's channel linked to the driver channel (see Section 3.4.4.1, "Fast message interrupt service routines").

The interrupt service routine calls a processing function which copies data from the SRX channel fast message data registers into the internal buffer of the related driver channel.

To avoid the stacking/unstacking delay caused by an additional possibly pending serial message interrupt request, the processing function also checks if this message was the last message required to construct the serial message. If so, the processing function copies data from the SRX channel serial message data registers into the internal buffer of the related driver channel.

Additionaly, the proccessing function loops through all the SRX module channels to check if there was another fast message received on a different channel during its execution. If so, it repeats the proccess described above. Figure 13 illustrates fast message reception with enabled reception notification.

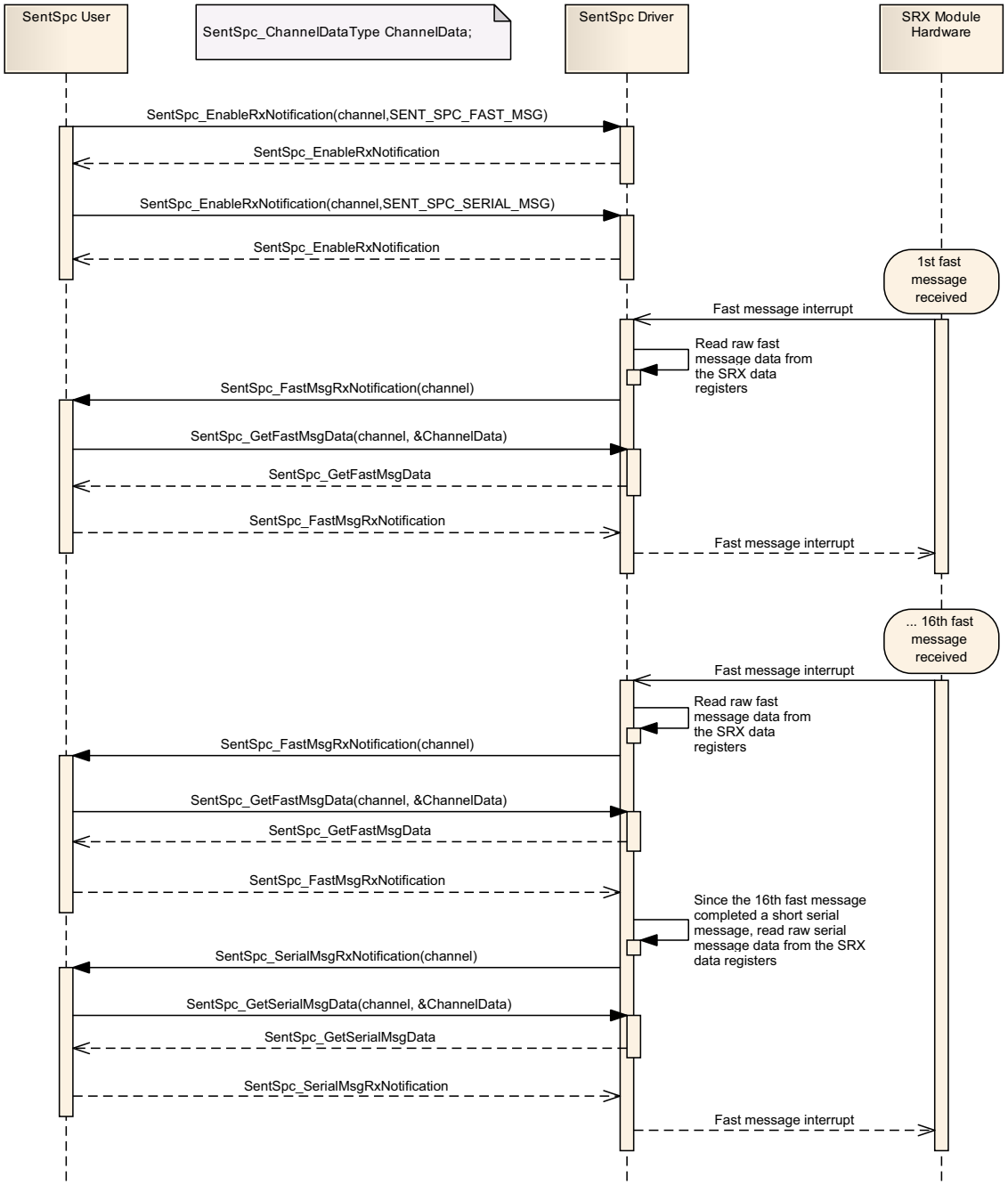

Figure 13. Message reading via interrupts (notification enabled)

3.5.2.1 Message reception notification

If the channel fast message reception notification is enabled, the fast message notification function SentSpc_FastMsgNotification() is executed as part of the fast message interrupt service routine. Similarly, if the channel serial message reception notification is enabled and the received fast message has completed the serial message, the serial message notification function SentSpc_SerialMsgNotification() is executed as part of the fast message interrupt service function.

Figure 14 illustrates driver channel behavior with fast message notification enabled/disabled. Serial message notification follows the same behavior as depicted in Figure 14.

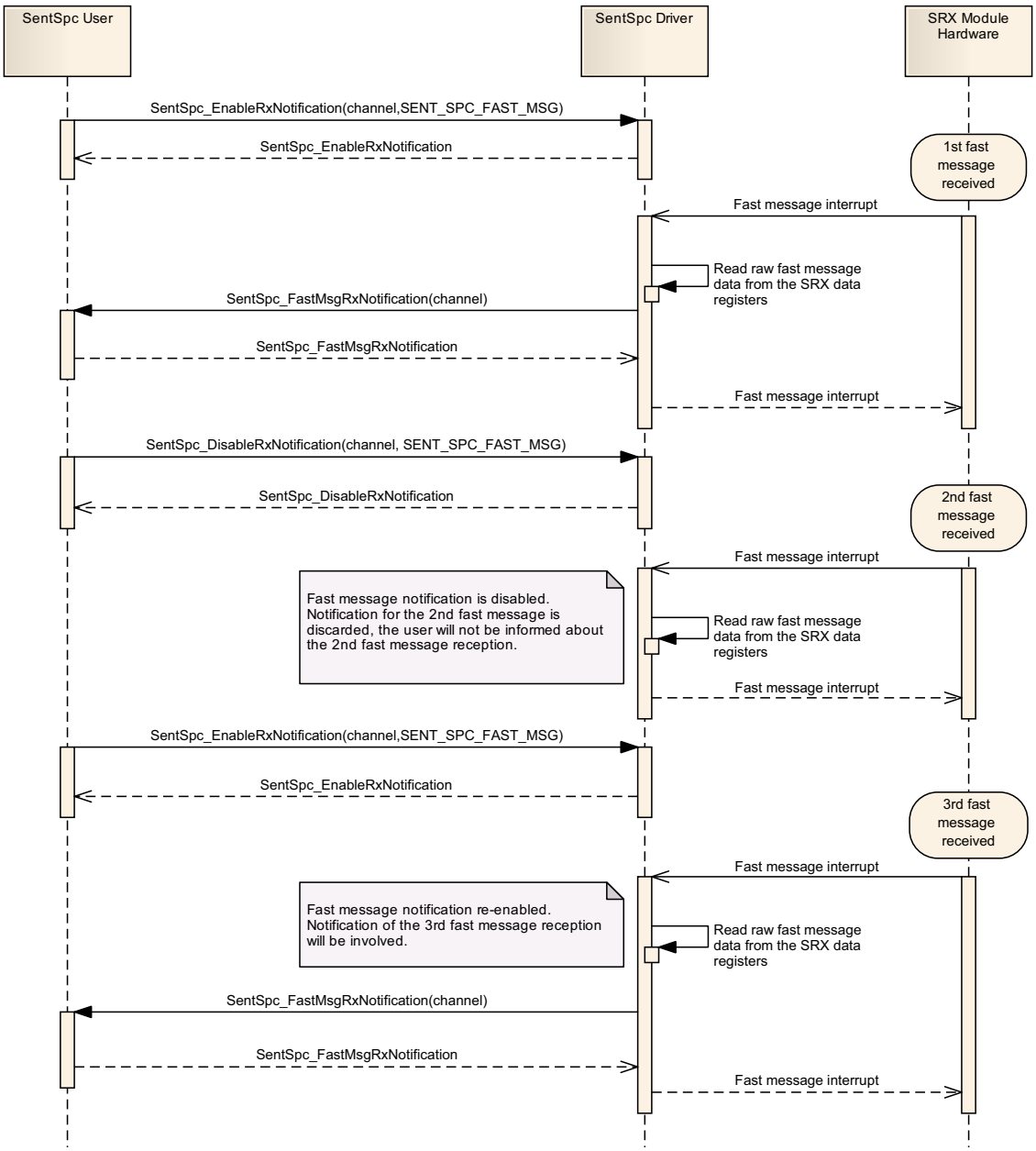

Figure 14. Disable/re-enable fast message notification

3.5.2.2 Message reception polling

Fast message and serial message are used to notify the application about a message reception. However, these functions are executed as part of interrupt service routines, so prolonging interrupt duration. To minimize interrupt service routine durations, the driver can be used in the polling mode with disabled notifications.

The driver channel status can be polled by the SentSpc_GetChannelStatus() API function. The actual channel status is returned in the form of the data type SentSpcChannelStatusType (see Table 27).

Table 27. Channel status structure (SentSpc_ChannelStatusType)

Member	Description
SentSpc_FastMsgRxNotifEnabled	Fast Message Reception Notification Enable. 0 Fast message reception notification disabled 1 Fast message reception notification enabled
SentSpc_SerialMsgRxNotifEnabled	Serial Message Reception Notification Enable. 0 Serial message reception notification disabled 1 Serial message reception notification enabled
SentSpc_ErrorNotifEnabled	Reception Error Notification Enable. 0 Reception error notification disabled 1 Reception error notification enabled
SentSpc_Reserved	This bit is reserved.
SentSpc_FastMsgDataReady	Fast Message Data Ready. 0 New fast message data is not ready in the channel buffer 1 New fast message data is ready in the channel buffer for reading by the SentSpc_GetFastMsgData() function.
SentSpc_SerialMsgDataReady	Serial Message Data Ready. 0 New serial message data is not ready in the channel buffer 1 New serial message data is ready in the channel buffer for reading by the SentSpc_GetSerialMsgData() function.
SentSpc_FastMsgDataOverflow	Fast Message Data Overflow. This bit is set when data stored in the internal buffer is overwritten by the fast message interrupt service routine before it was read by the SentSpc_GetFastMsgData() function. 0 No fast message overflow 1 Fast message overflow.
SentSpc_SerialMsgDataOverflow	Serial Message Data Overflow. This bit is set when data stored in the internal buffer is overwritten by the serial message interrupt service routine before it was read by the SentSpc_GetSerialMsgData() function. 0 No serial message overflow. 1 Serial message overflow.
SentSpc_ErrorStatus	Channel error status byte (see Table 28).

Figure 15 shows the sequence diagram illustrating fast message reception polling. Serial message reception polling uses the same mechanism.

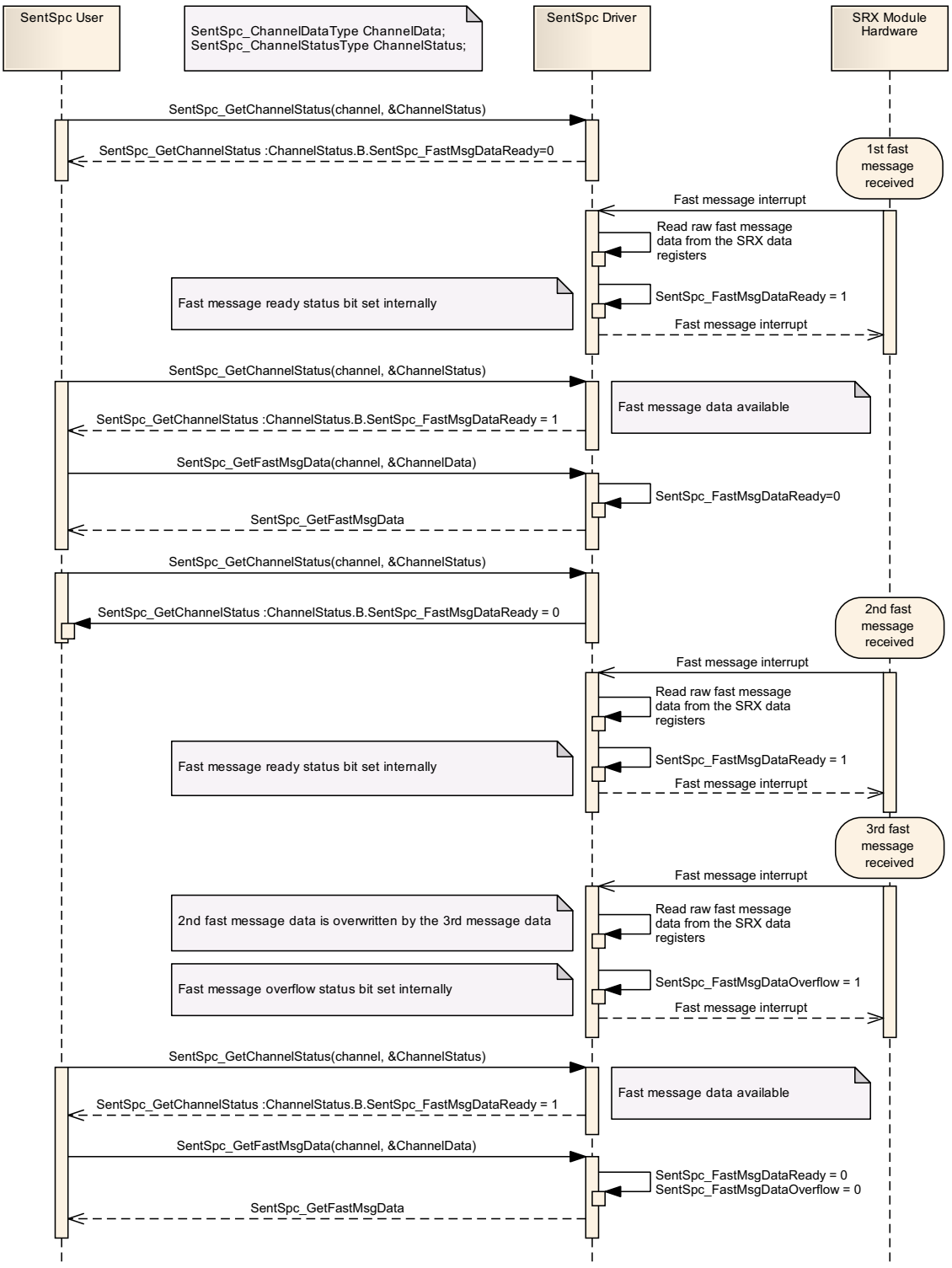

Figure 15. Fast message reception (with and without overflow) polling

3.5.3 Fast message reading using FIFO

The driver channel is configured for the FIFO operation if its related SentSpcSrxFastMsgFifoEnable configuration parameter is set to true.

Each successfully received raw fast message data is stored in the fast message FIFO. The FIFO water mark level is configurable by the SentSpcSrx*N*FifoWaterMark (where *N* is the index of the SRX module) configuration parameter value on a per SRX module basis. That means, if at least one driver channel is configured with fast message FIFO enabled, then the SentSpcSrx*N*FifoWaterMark configuration parameter for the related SRX module is made configurable in the configuration tool.

Once the configured number of received messages in the FIFO reaches the water mark level, a DMA transfer request is asserted. The eDMA engine transfers a number of fast messages, equal to the FIFO water mark level, from the fast message FIFO into the internal buffer of the driver. Once the transfer is complete, the End of eDMA Transfer interrupt is invoked (see Section 3.4.4.3, “End of eDMA Transfer interrupt service routines”). The SentSpc_FastMsgFifoNotification() notification function is then executed as part of the ISR.

NOTE

The SentSpc_FastMsgFifoNotification() is always executed once the FIFO data has transferred into the driver internal buffer. The user is advised to read the data using the SentSpc_GetFastMsgFifoData() API function within the fast message FIFO notification function. This approach eliminates possible collision with the following eDMA FIFO data transfer while reading the driver internal buffer.

As the FIFO can be used only for fast messages, processing of the received serial messages is performed using the separate serial message interrupt (see Section 3.4.4.2, “Serial message interrupt service routines”).

Figure 16 illustrates driver channel behavior with an enabled fast message FIFO.

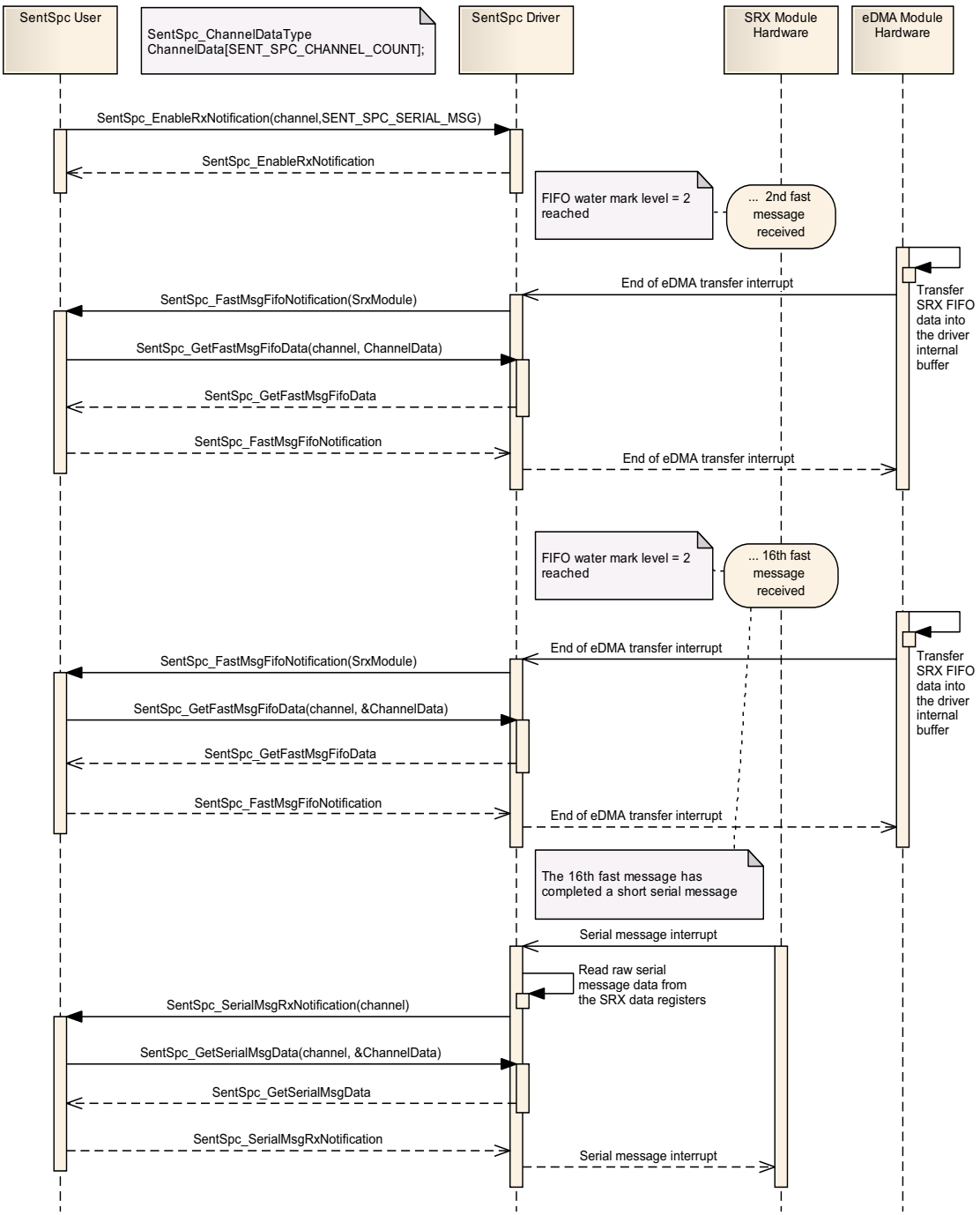

Figure 16. Fast message reading using FIFO

3.5.4 Master trigger pulse generation

The SentSpc_Request() API function initiates generation of the master trigger pulse on the sensor interface. Figure 17 illustrates a 4 µs master trigger driving pulse generation using the eTimer channel operating in the output compare mode with the channel counter running at 160 MHz.

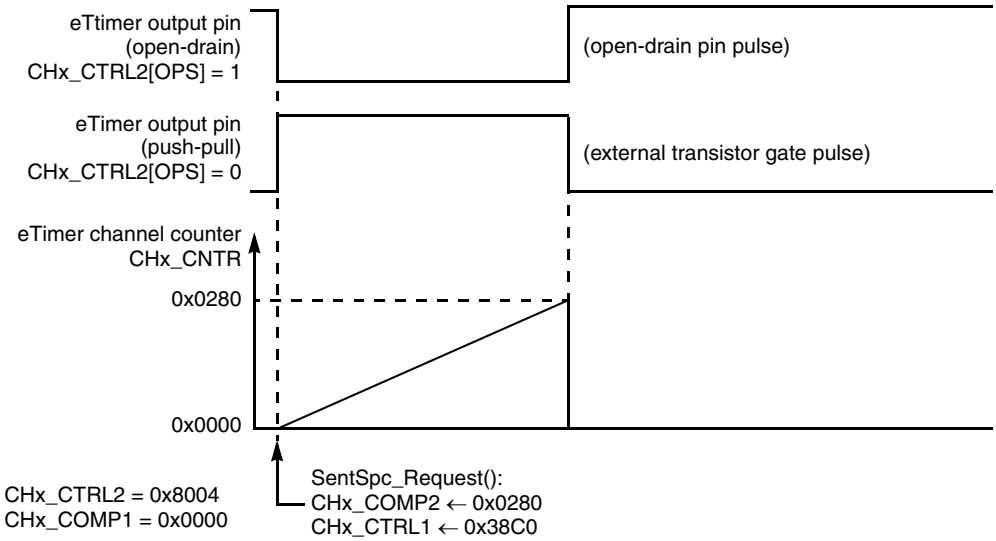

Figure 17. Master trigger pulse generation example

The driving pulse width is defined by the *u8MasterTime* input parameter of the SentSpc_Request() API function. The width of the driving pulse in the number of eTimer channel counter ticks is defined by the Compare Register 2 (COMP2) value, while the Compare Register 1 (COMP1) register is set to 0x0000 by the API functions. The eTimer Channel Control Register 2 (CTRL2) is set up in such a way to toggle the eTimer channel OFLAG output signal using the alternating compare registers COMP1 and COMP2. The polarity of the leading edge (generated on COMP1 compare) is based on the driver channel configuration parameter SentSpcEtimerOdEnable value which corresponds with the CTRL2[OPS] bit value (SentSpcEtimerOdEnable = false → OPS = 1, SentSpcEtimerOdEnable = true → OPS = 0). The eTimer channel Control Register 1 (CTRL1) is written by the SentSpc_Request() API function to start the channel counter (counting from 0x0000). The counter stops counting and resets to 0x0000 on COMP2 compare (trailing edge generated).

Figure 18 illustrates the procedure to request SENT data from the SENT/SPC compatible sensor.
A Periodic Interrupt Timer (PIT) interrupt service routine is used to trigger the transmission periodically in 1.2 ms intervals. Message reception is polled after a 1.2 ms time out.

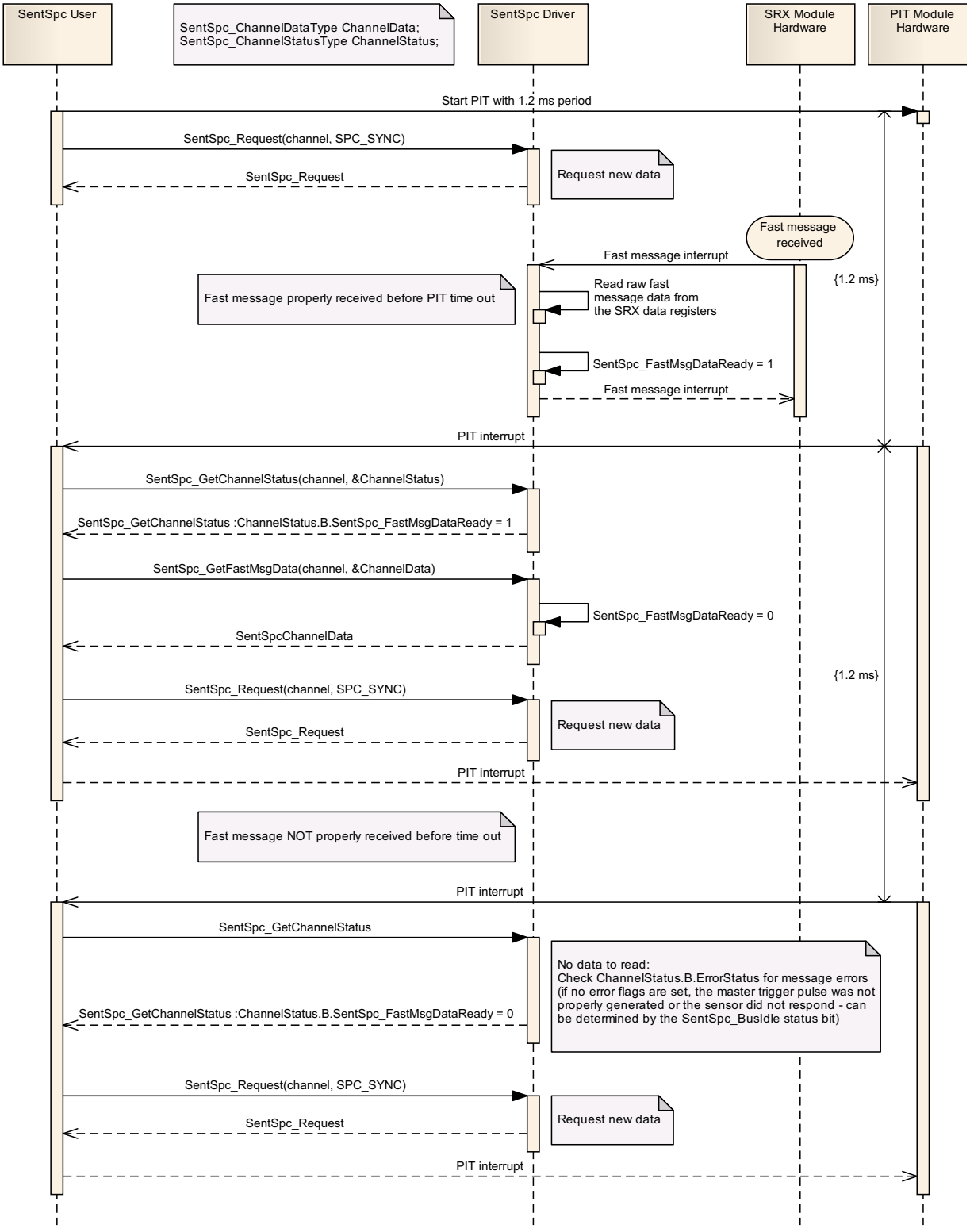

Figure 18. SENT message triggering using SPC master trigger pulse (FIFO disabled)

3.5.5 Receiver diagnostics

The SRX module provides sensor interface and message reception diagnostics defined by the SAE J2716 specification. The actual driver channel error status can be obtained by the SentSpc_GetChannelStatus() API function call. If the channel error notification is enabled by the SentSpc_EnableErrNotification(), the occurrence of the channel error is signalized by the SentSpc_ErrorNotification() within the channel error interrupt service routine, and within the SentSpc_GetFastMsgData() and SentSpc_GetFastMsgFifoData() functions to signalize the fast message CRC error (SW based CRC type only).

For a complete list of supported types of sensor interface and message diagnostics, see the description of the SentSpcChannelErrorType data type in Table 28.

Table 28. Channel error structure (SentSpc_ChannelErrorType)

Member	Description
SentSpc_BusIdle[1]	Bus Idle Status. This status bit indicates that the sensor interface has been idle for more than the period defined in the channel configuration. 0 Bus is not idle 1 Channel has been idle for more than the allowed value
SentSpc_PausePulseDiagError	This error status bit indicates that the ratio of the calibration pulse length to overall message length is greater than ±1.5625% between two messages. 0 Error check has passed 1 Error check has failed
SentSpc_CalibLengthError[2]	This error status bit indicates that the calibration pulse length is greater than 56 ticks ±25% or ±20%, depending on the configuration. 0 Error check has passed 1 Error check has failed
SentSpc_CalibDiagError	This error status bit indicates that the successive calibration pulses differ by more than ±1.5625%. 0 Error check has passed 1 Error check has failed
SentSpc_NibbleValueError	This error status bit indicates that any nibble data value is <0 or >15. 0 Error check has passed 1 Error check has failed
SentSpc_SerialMsgCRCError	This error status bit indicates a checksum error in the slow serial message. 0 Error check has passed 1 Error check has failed
SentSpc_FastMsgCRCError	This error status bit indicates a checksum error in the fast message. 0 Error check has passed 1 Error check has failed
SentSpc_ErrorNumberOfEdges[3]	This error status bit indicates that not the expected number of falling edges was detected between calibration pulses. 0 Error check has passed 1 Error check has failed

1 Bus idle state is not signalized by the SentSpc_ErrorNotification() notification function. It needs to be polled using the SentSpc_GetChannelStatus() API function.

2 If the channel operates in the SENT/SPC mode, the SentSpc_CalibLengthError bit is always zero due to the SRX module HW limitations.

3 If the channel operates in the SENT/SPC mode or Option 2 is selected for the successive calibration pulse check in the configuration, the SentSpc_ErrorNumberOfEdges bit is always zero.

Figure 19 illustrates error reporting with enabled/disabled channel error notification.

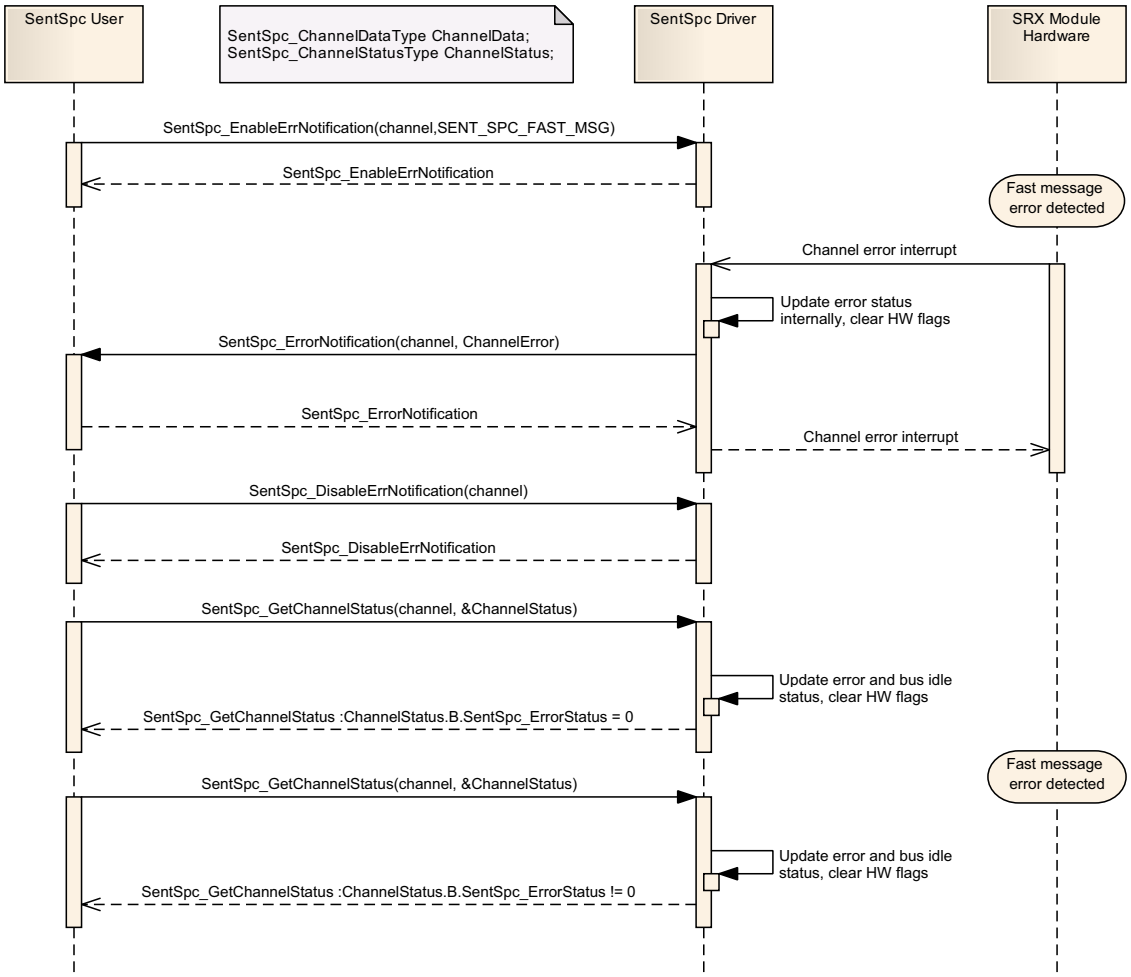

Figure 19. Error reporting with enabled/disabled channel error notification

3.6 Memory allocation

To ensure data coherency when the core's data cache is enabled, certain variables that are write accessible by the eDMA module need to be placed in the non-cacheable section of the SRAM memory. The driver requires up to 96 bytes of the non-cacheable memory. Impacted variables are explicitly placed in the program section ".sentspc_bss" by the compiler pragma in the source code. In order to place impacted variables in the non-cacheable SRAM memory region, do the following:

1. Create a new memory region covering the desired portion of the non-cacheable SRAM memory in the linker directives file of the project. The memory region must be 32-bytes aligned with a size equal to a multiple of 32 bytes.
2. Place the section ".sentspc_bss" into the section map of the memory region created in the previous step, in the linker directives file of the project.
3. Specify a non-cacheable SRAM region by initializing the System Memory Protection Unit (SMPU) module region descriptor, with the region start address in SMPU_WORD0, region end address (last byte address of the section) in SMPU_WORD1, bus master permissions in SMPU_WORD2, and the SMPU_WORD3[CI] Cache Inhibit bit.
4. Mark the SMPU region descriptor as valid by writing 1 to the WORD3[VLD] bit.
5. Enable the SMPU by writing 1 to the CESR0[GVLD] bit.

NOTE

All memory regions accessed by the application must be specified by the SMPU region descriptors before enabling the SMPU.

For more information about the SMPU, consult the MPC5744P Reference Manual, MPC5744PRM or refer to Example 1 and Example 2.

Example 1. Memory region and section map definition in the linker directives file

```
MEMORY
{
      /* ... */
      /* Cache-inhibit memory region */
      int_sram_ci   : ORIGIN = 0x4005F000, LENGTH = 0x1000
}

SECTIONS
{
      /* ... */
      /* SENT/SPC driver section */
      .sentspc_bss  : > int_sram_ci
}
```

Example 2. SMPU initialization code

```
void SMPU_Init(void)
{
      /* SRAM */
      SMPU_0.RGD[0].WORD0.R = 0x40000000; /* Start address */
      SMPU_0.RGD[0].WORD1.R = 0x4005EFFF; /* End address */
      SMPU_0.RGD[0].WORD2.R = 0xCFF0C000; /* Master 0,2,3,4,5,8, RW access enabled */
      SMPU_0.RGD[0].WORD3.R = 0x00000001; /* Valid */

      /* SRAM (cache inhibit memory range) */
      SMPU_0.RGD[1].WORD0.R = 0x4005F000; /* Start address */
      SMPU_0.RGD[1].WORD1.R = 0x4005FFFF; /* End address */
      SMPU_0.RGD[1].WORD2.R = 0xCFF0C000; /* Master 0,2,3,4,5,8, RW access enabled */
      SMPU_0.RGD[1].WORD3.R = 0x00000003; /* Valid, Cache Inhibit */

      /* Flash */
      SMPU_0.RGD[2].WORD0.R = 0x01000000; /* Start address */
      SMPU_0.RGD[2].WORD1.R = 0x011FFFFF; /* End address */
      SMPU_0.RGD[2].WORD2.R = 0xCFF0C000; /* Master 0,2,3,4,5,8, RW access enabled */
      SMPU_0.RGD[2].WORD3.R = 0x00000001; /* Valid */

      /* DMEM */
      SMPU_0.RGD[3].WORD0.R = 0x50800000; /* Start address */
      SMPU_0.RGD[3].WORD1.R = 0x5080FFFF; /* End address */
      SMPU_0.RGD[3].WORD2.R = 0xCFF0C000; /* Master 0,2,3,4,5,8, RW access enabled */
      SMPU_0.RGD[3].WORD3.R = 0x00000003; /* Valid, Cache Inhibit */

      /* Peripheral space */
      SMPU_0.RGD[4].WORD0.R = 0xF8000000; /* Start address */
      SMPU_0.RGD[4].WORD1.R = 0xFFFFFFFF; /* End address */
      SMPU_0.RGD[4].WORD2.R = 0xCFF0C000; /* Master 0,2,3,4,5,8, RW access enabled */
      SMPU_0.RGD[4].WORD3.R = 0x00000003; /* Valid, Cache Inhibit */

      SMPU_0.CESR0.R = 0x00000001; /* Enable SMPU */
}
```

3.7 Application example

Please refer to the sample application which is part of the AN4856SW package.

4 Conclusion

The application note describes the SENT protocol basics along with its SPC enhancement. A list of utilized peripherals, the application programming interface description, and a functional description of the MPC574xP SENT/SPC driver including the API calling sequences are provided in the text.

The driver provides support for SENT compatible sensor data acquisition and full communication with the SENT/SPC compatible Infineon TLE4998C programmable linear Hall sensor, including all its supported SPC modes.

5 References

1. SAE J2716 (R) SENT – Single Edge Nibble Transmission for Automotive Applications, JAN2010
2. *MPC5744P Reference Manual*, Rev. 2, 06/2013
3. *MPC5744P Data Sheet*, Rev. 0.4, 04/2013
4. TLE4998C Target Data Sheet, V 1.0, December 2008

6 Acronyms and Definitions

Table 29. Acronyms and definitions

Term	Definition
API	Application Programming Interface
CAN	Controller Areal Network
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
eDMA	Enhanced Direct Memory Access
EMC	Electromagnetic Compatibility
ESD	Electrostatic Discharge
eTimer	Enhanced Motor Control Timer
FIFO	First In First Out
HW	Hardware
ID	Identification
ISR	Interrupt Service Routine
LIN	Local Interconnect Network
MCU	Microcontroller Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
PIT	Periodic Interrupt Timer
PWM	Pulse Width Modulation
SAE	Society of Automotive Engineers
SENT	Single Edge Nibble Transfer Protocol
SIUL2	System Integration Unit Lite 2
SPC	Short PWM Code
SRAM	Static Random Access Memory
SRX	SENT Receiver
SW	Software
UT	Unit Time

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4856
Rev. 0
03/2014