Freescale Semiconductor
Application Note

Document Number: AN4432
Rev. 0, 01/2012

SENT/SPC Driver for the MPC560xP and MPC564xL Microcontroller Families

by: Josef Kramoliš
Rožnov pod Radhoštěm, Czech Republic

1 Introduction

This application note describes the SENT/SPC software driver for the MPC560xP and MPC564xL 32-bit microcontrollers. The fundamentals of the Single Edge Nibble Transmission protocol (SENT, SAE J2716), along with its Short PWM Code (SPC) enhancement, are discussed in the overview section of the document. The driver implementation, API, state diagrams, and the recommended program flow, along with the application code examples, are shown in the remainder of the document.

Most of the information about the SENT protocol was derived from the SAE-J2716 Surface Vehicle Information Report, FEB2008.

Contents

1 Introduction 1
2 Overview 2
2.1 SENT encoding scheme 2
2.2 SPC protocol 5
2.3 SENT/SPC physical layer 6
3 SENT/SPC software driver for the MPC560xP and MPC564xL 8
3.1 Physical layer topology 8
3.2 Utilized MPC560xP/MPC564xL peripherals 8
3.3 Driver configuration 9
3.4 API 15
3.5 Master trigger pulse generation 18
3.6 SENT data acquisition 18
3.7 API calling sequence 20
3.8 Resource metrics 25
3.9 Application example 27
4 Conclusion 33
5 References 34
6 Acronyms 35

© Freescale Semiconductor, Inc., 2012. All rights reserved.

freescale™ semiconductor

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 electronic 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 transmitted data values and on 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 3–10 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)

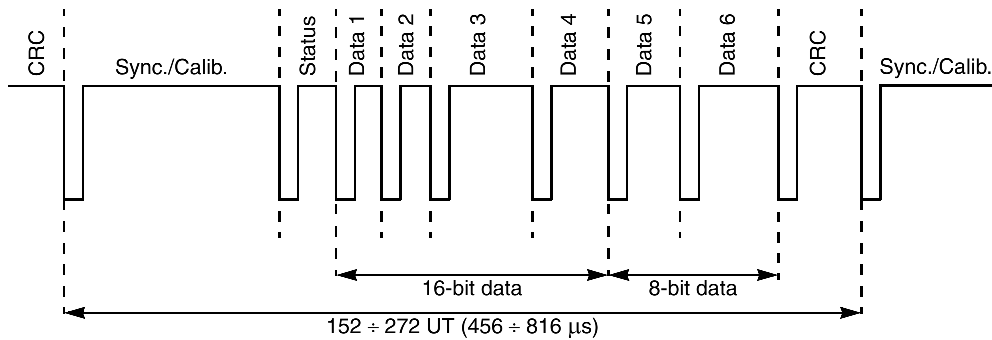

Figure 1. Transmission example of 16-bit and 8-bit signal data

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 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).

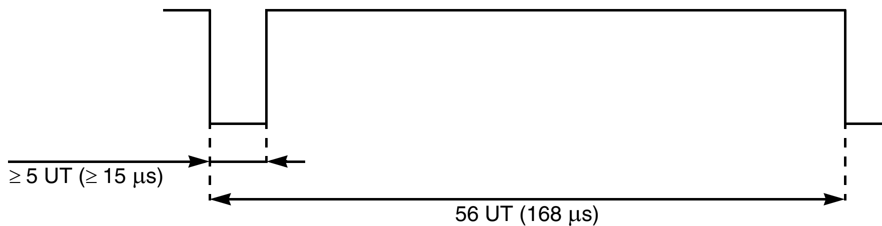

Figure 2. Synchronization/calibration pulse format

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 contain a serial message (one bit as a serial data bit, one bit as a start bit). The complete 16-bit serial message is then transmitted in 16 consecutive SENT transmissions (refer to SAE J2716 at www.sae.org for detailed description).

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”.

2.1.3 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

$$DataNibblePulseWidth = (12 + NibbleValue)$$

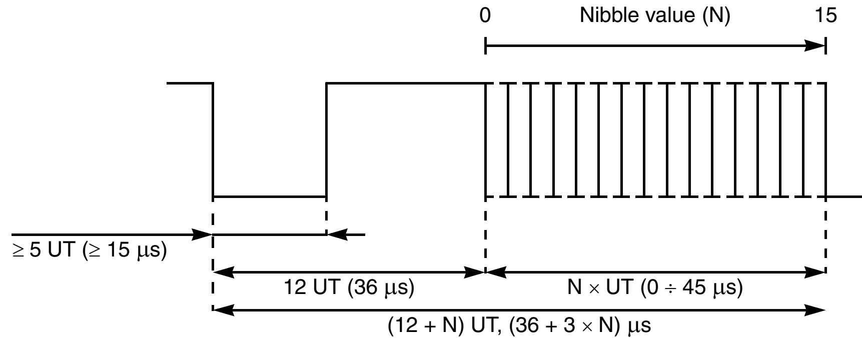

Figure 3. Data nibble pulse format

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 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.

NOTE

The driver CRC calculation also includes the status and communication nibble value as it is primarily intended for use with the Infineon TLE4889C Hall sensor.

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 communication line 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 communication line then remains idle until a new master trigger pulse is generated by the receiver. Figure 4 depicts the SENT/SPC frame format.

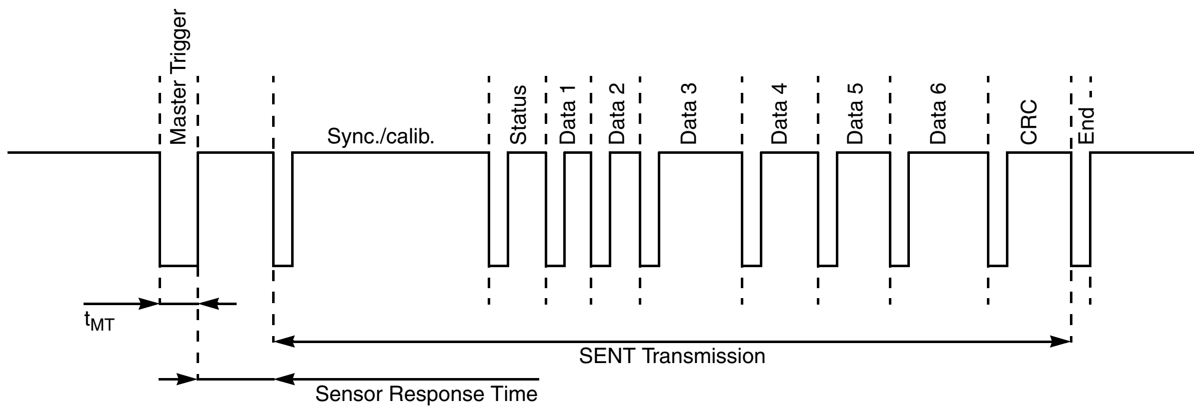

Figure 4. SENT/SPC frame format

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/SPC physical layer

The receiver side (ECU) provides the stabilized 5 V voltage to supply the sensor. The communication line 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 communication line. The open-drain output pin on the MCU pulls down the communication line 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 communication line. The communication line 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 for the falling and rising edge durations are $T_{FALL} = 6.5$ µs and $T_{RISE} = 18$ µs with a maximum allowed 0.1 µs falling edge jitter. An example of a TLE4998C SENT/SPC compatible Hall sensor waveform is shown in Figure 6.

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.

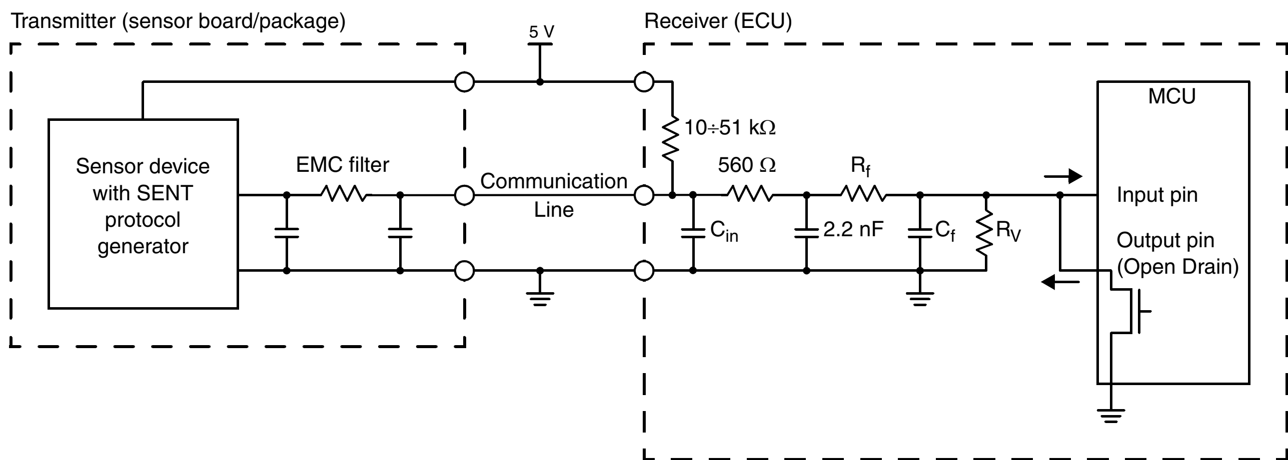

Figure 5. SENT/SPC circuit topology

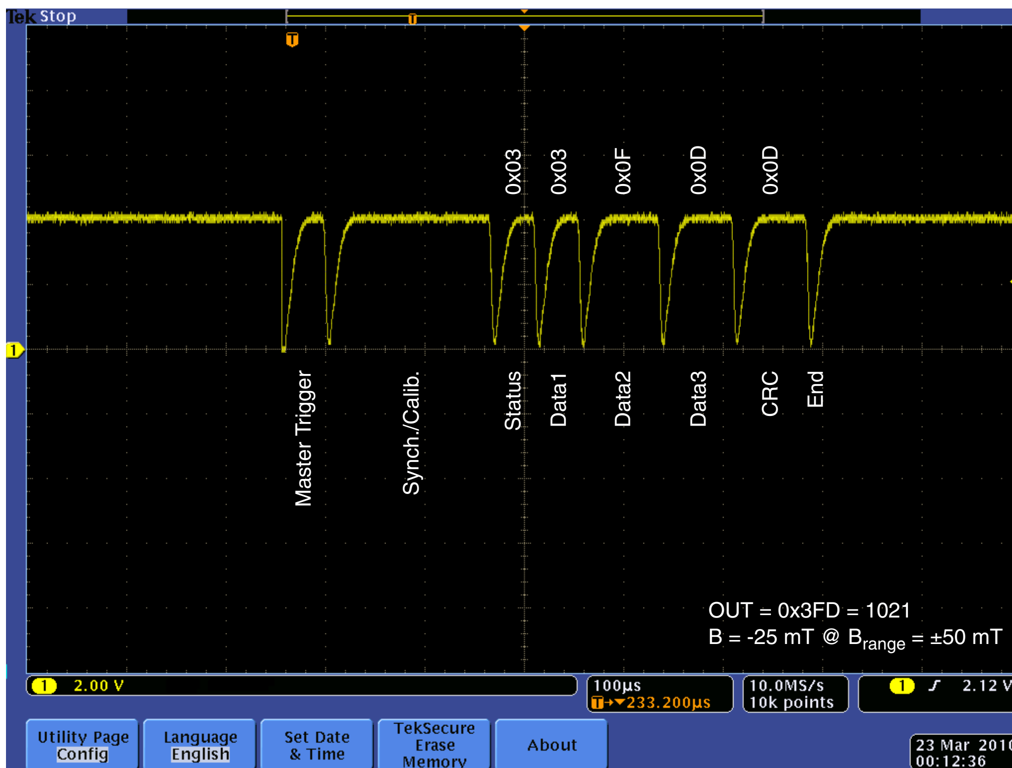

Figure 6. TLE4998C SENT/SPC 12-bit Hall waveform

3 SENT/SPC software driver for the MPC560xP and MPC564xL

The driver is provided as example code only, and in the form of source code optimized for the Green Hills compiler. It is intended for use with all members of the MPC560xP and MPC564xL families and the Infineon TLE4998C programmable linear Hall sensor. The driver supports code execution by both the MPC564xL e200z4 cores, and can be used for handling up to four independent SENT/SPC channels on the MPC560xP and up to six independent SENT/SPC channels on the MPC564xL.

3.1 Physical layer topology

The driver is designed to control an external transistor connected to the output pin (2-pin solution). The output transistor is driven by a pulse of positive polarity, thus pulling the communication line low to generate the master trigger pulse. The output pin driver operates in the push-pull output mode. Figure 7 shows a typical TLE4998C Hall sensor application circuit with an external transistor.

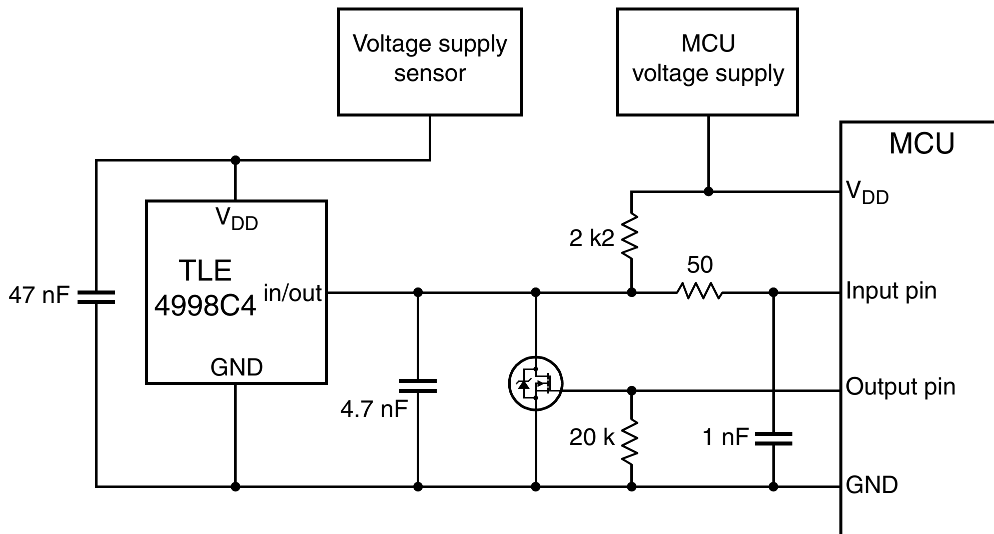

Figure 7. Typical TLE4998C application circuit with external transistor

3.2 Utilized MPC560xP/MPC564xL peripherals

The driver utilizes the following MPC560xP/MPC564xL peripherals:

- System Integration Unit Lite (SIUL) — 2 pins for a single SENT/SPC channel
- Enhanced Motor Control Timer (eTimer)
  - 1 channel for a single SENT/SPC channel
  - 1 eTimer DMA request channel for a single SENT/SPC channel
- Enhanced Direct Memory Address engine (eDMA) — a single channel for a single SENT/SPC channel

3.3 Driver configuration

There are four pre-processor macros (accessible in the SENT_SPC_Driver.h header file) that need to be properly defined before the final application can be finally built. Additionally, a target microcontroller specific macro symbol needs to be specified by the compiler option during compilation. Table 1 lists a description of all macros.

Table 1. Pre-compile time parameters

Macro	Range	Description
SENT_SPC_INTERRUPT	0 or 1	Defines whether the eDMA channel interrupt or an additional eDMA transfer request is generated at the end of the SENT/SPC frame transfer.
SENT_SPC_INTVEC_MODE	0 or 1	Defines the interrupt vector mode type. 0 Interrupt Controller configured in the software vector mode 1 Interrupt Controller configured in the hardware vector mode
SENT_SPC_UT	—	Defines the number of the eTimer module primary input clock ticks per 3 µs. This can be calculated using the formula: $SENT_SPC_UT = MotorControlClockFrequency \cdot 3 \cdot 10^{-6}$ **Eqn. 2**
MPC5604P	—	This symbol needs to be defined by the compiler -D option (-DMPC5604P) to select the MPC560xP as the target device family.
MPC5643L	—	This symbol needs to be defined by the compiler -D option (-DMPC5643L) to select the MPC564xL as the target device family.

3.3.1 SENT/SPC channel configuration structure

Each SENT/SPC channel has its own configuration structure in the form of a variable of type SENT_SPC_CONTROL_T which needs to be initialized before the driver can be initialized, using the appropriate API function. The driver uses a pointer to the SENT/SPC channel configuration structure as an input parameter to all API functions. Use the steps below to properly initialize the configuration structure.

1. Declare a variable of type SENT_SPC_CONTROL_T.
2. Initialize members of this variable:
   a) Initialize structure member SentSpcEtimer.
   b) Initialize structure member SentSpcEtimerOutput.
   c) Initialize structure member SentSpcEtimerInput.
   d) Initialize structure member SentSpcOutputMux.
   e) Initialize structure member SentSpcInputMux.
   f) Initialize structure member SentSpcEtimerDma.
   g) Initialize structure member SentSpcDma.
   h) Initialize structure member SentSpcFrame.

Consult Table 2 for proper channel configuration structure member values.

Table 2. Mandatory parameters of the SENT/SPC channel configuration structure

Structure member	Range	Description
SentSpcEtimer	0..1 (MPC560xP), 0..2 (MPC564xL)	The eTimer module number used for SENT/SPC channel operation.
SentSpcEtimerOutput	0..5	The eTimer channel number used for driving the external transistor and for data reception.
SentSpcEtimerInput	0..5	The eTimer module input number which will be used for data reception.
SentSpcOutputMux	See Table 3, Table 4, Table 5, Table 6, Table 7	The eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel output pin multiplexing settings.
SentSpcInputMux	See Table 3, Table 4, Table 5, Table 6, Table 7	The eTimer_[SentSpcEtimer] module input pin SentSpcEtimerInput multiplexing settings.
SentSpcEtimerDma	0..1	The eTimer DMA request channel used for SENT/SPC channel operation.
SentSpcDma	0..15	The eDMA channel number used for SENT/SPC channel operation.
SentSpcFrame	SPC_FRAME_6, SPC_FRAME_5, SPC_FRAME_4, SPC_FRAME_3	SENT/SPC frame format of the device connected to the SENT/SPC channel. SPC_FRAME_6 6 data nibbles (16-bit Hall, 8-bit temperature) SPC_FRAME_5 5 data nibbles (12-bit Hall, 8-bit temperature) SPC_FRAME_4 4 data nibbles (16-bit Hall) SPC_FRAME_3 3 data nibbles (12-bit Hall)

Each SENT/SPC channel has to have its own unique eDMA channel, eTimer DMA request channel, and eTimer channel(s) input/output pins assigned in the channel configuration structure variable. The driver, however, provides an internal checking mechanism for duplicated parameter selection.

See Section 3.9, “Application example” for the example of the declaration and initialization of two SENT/SPC channel configuration structure variables.

Refer to Table 3, Table 4, Table 5, Table 6, and Table 7 for input/output pin multiplexing options.

Table 3. MPC560xP eTimer_0 I/O pin multiplexing options

SentSpcEtimerOutput/ SentSpcEtimerInput	SentSpcOuputMux/ SentSpcInputMux		
	0	1	2
0	A[0]	—	—
1	A[1]	—	—
2	A[2]	—	—
3	A[3]	—	—
4	A[4][1]	C[11]	B[14][2]
5	C[12]	B[8][2]	—

[1] Pin A[4] is shared between MPC560xP eTimer_0 and eTimer_1.

[2] Pins B[8] and B[14] can by used only as respective eTimer_0 channel's inputs.

Table 4. MPC560xP eTimer_1 I/O pin multiplexing options

SentSpcEtimerOutput/ SentSpcEtimerInput	SentSpcOuputMux/ SentSpcInputMux			
	0	1	2	3
0	A[4][1]	C[15]	—	—
1	C[13]	D[0]	—	—
2	B[0]	C[14]	D[1]	—
3	B[1]	D[2]	F[12]	—
4	A[14]	C[3]	D[3]	F[13][2]
5	A[5]	A[15]	D[4]	—

[1] Pin A[4] is shared between MPC560xP eTimer_0 and eTimer_1.

[2] Pin F[13] can by used only as the eTimer_1_ETC[4] input.

Table 5. MPC564xL eTimer_0 I/O Pin multiplexing options

SentSpcEtimerOutput/ SentSpcEtimerInput	SentSpcOuputMux/ SentSpcInputMux			
	0	1	2	3
0	A[0]	D[10][1]	—	—
1	A[1]	D[11][1]	—	—
2	A[2]	F[0][1]	—	—
3	A[3]	D[14][1]	—	—
4	A[4][2]	C[11]	B[14][1]	G[3][1]
5	C[12]	E[13]	B[8][1]	G[4][1]

[1] Pins B[8], B[14], D[10], D[11], D[14], F[0], G[3], and G[4] can be used only as the respective eTimer_0 channel's inputs.

[2] Pin A[4] is shared between MPC564xL eTimer_0 and eTimer_1.

Table 6. MPC564xL eTimer_1 I/O pin multiplexing options

SentSpcEtimerOutput/ SentSpcEtimerInput	SentSpcOuputMux/ SentSpcInputMux			
	0	1	2	3
0	A[4][1]	C[15]	—	—
1	C[13]	D[0]	—	—
2	B[0]	C[14]	D[1]	—
3	B[1]	D[2]	F[12]	—
4	A[14]	D[3]	D[8]	F[13]
5	A[5]	A[15]	D[4]	E[14]

[1] Pin A[4] is shared between MPC564xL eTimer_0 and eTimer_1.

Table 7. MPC564xL eTimer_2 I/O pin multiplexing options

SentSpcEtimerOutput/ SentSpcEtimerInput	SentSpcOuputMux/ SentSpcInputMux	
	0	1
0	H[4]	I[0]
1	H[7]	I[1]
2	H[10]	I[2]
3	H[13]	I[3]
4	H[14]	—
5	H[15]	—

Figure 8 illustrates possible utilization of particular MCU modules by two SENT/SPC channels, differentiated by colors (a single eTimer module usage example). It also shows channel control structure parameters assigned to an appropriate hardware module configuration.

Figure 8. Example of MCU peripheral utilization (two SENT/SPC channels)

In eTimer module terminology, the SentSpcEtimerInput parameter refers to the eTimer channel *m* secondary input selection; ergo eTimer input *n* is equal to the SentSpcEtimerInput value, and *m* is equal to the SentSpcEtimerOutput parameter value defining the eTimer channel used for SENT/SPC operation. The output *m* is statically assigned to the eTimer channel *m*. The value of the SentSpcEtimerInput is not limited to the same value as SentSpcEtimerOutput, thus providing more flexibility for SENT/SPC channel input and output pin assignment.

The SentSpcInputMux parameter provides selection of the input pin from the group of pins dedicated to the selected eTimer input *n*. The SentSpcOutputMux parameter provides similar selection of the selected eTimer channel *m* output pin. Since input and output pin selection is made from the single group of I/O pins when SentSpcEtimerInput is equal to SentSpcEtimerOutput (*m* equals *n*, SIUL multiplexers are identical for In *m* and Out *m* signals), the driver provides an internal checking mechanism for duplicate assignment of the same input and output pin.

The SentSpcEtimerDma parameter provides selection of the eTimer module DMA request channel. Further assignment of an eDMA channel is made by the SentSpcDma parameter.

Lastly, the SentSpcEtimer parameter value selects the eTimer module number.

3.4 API

The driver API consists of the following functions:

1. SENT_SPC_Init()
2. SENT_SPC_Request()
3. SENT_SPC_Load()
4. SENT_SPC_Read_Hall()

3.4.1 SENT_SPC_Init

Syntax: `SENT_SPC_STATE_T SENT_SPC_Init(SENT_SPC_CONTROL_T *pParam);`

Reentrancy: Non-reentrant.

Parameters: `*pParam` — pointer to the SENT/SPC channel configuration structure variable.

Return: 16-bit driver status word.

Description: The function initializes all on-chip peripherals which are required for the proper generation of the master trigger pulse, SENT data reception, and processing of the selected SENT/SPC channel data. The function updates the internal SENT/SPC channel 16-bit status word (see Table 11).

NOTE

Initialization of the e200z0 core (MPC560xP) and the e200z4 core(s) (MPC564xL), system clock (PLL_0), motor control clock (PLL_1), on-chip FLASH memory, SRAM, interrupt controller (INTC), and the interrupt vector table is not handled by the driver — it is the responsibility of the user.

3.4.2 SENT_SPC_Request

Syntax: `SENT_SPC_STATE_T SENT_SPC_Request(SENT_SPC_CONTROL_T *pParam, uint8_t u8MasterTime);`

Reentrancy: Non-reentrant.

Parameters: `*pParam` — pointer to the SENT/SPC channel configuration structure.
`u8MasterTime` — the width of the external transistor gate driving pulse in µs.

Return: 16-bit driver status word.

Description: The function generates the master trigger pulse on the communication line of the selected SENT/SPC channel via the external transistor. The function updates the internal SENT/SPC channel 16-bit status word (see Table 11).

NOTE

The actual master trigger pulse width is dependent on the communication line resistor/capacitor parameters and the operating temperature, and is always wider than the gate pulse width defined by the *u8MasterTime* input parameter. The user shall ensure (for example by a measurement) that the master trigger pulse width will be always within the proper limits with respect to the sensor edge detection thresholds.

The driver provides predefined macros for the *u8MasterTime* input parameter, which were tested for compliance of the master trigger pulse width according to the TLE4998C data sheet at a 23 °C ambient temperature, and using the typical application circuit shown in Figure 7. Table 8, Table 9, and Table 10 list the provided macros based on the preprogrammed SPC protocol mode of the TLE4998C device(s).

Table 8. Typical master trigger pulse timing macro for TLE4998C Synchronous mode

Macro	Master trigger pulse width [UT]	Gate pulse width [µs]
SPC_SYNCH	2.75	4

Table 9. Typical master trigger pulse timing macros for TLE4998C ID Selection mode

Macro	Sensor ID	Master trigger pulse width [UT]	Gate pulse width [µs]
SPC_ID_0	0	10.5	28
SPC_ID_1	1	21	59
SPC_ID_2	2	38	110
SPC_ID_3	3	64.5	190

Table 10. Typical master trigger pulse timing macros for TLE4998C Dynamic Range mode

Macro	Magnetic field range	Master trigger pulse width [UT]	Gate pulse width [µs]
SPC_RANGE_200	±200 mT	3.25	6
SPC_RANGE_100	±100 mT	12	32
SPC_RANGE_50	±50 mT	31.5	91

3.4.3 SENT_SPC_Load

Syntax: `SENT_SPC_STATE_T SENT_SPC_Load(SENT_SPC_CONTROL_T *pParam);`

Reentrancy: Non-reentrant.

Parameters: `*pParam` — pointer to the SENT/SPC channel configuration structure variable.

Return: 16-bit driver status word.

Description: The function checks the time-out condition and cause of the timeout (no master trigger pulse, or an invalid number of received nibbles with respect to the selected frame format). It decodes and stores the data nibble values into an internal memory array which is part of the SENT/SPC channel configuration structure. It also tests the nibble value range, calculates a CRC checksum, and compares it with the received checksum nibble value. The function updates the internal SENT/SPC channel 16-bit status word (see Table 11).

3.4.4 SENT_SPC_Read_Hall

Syntax:

```
SENT_SPC_STATE_T SENT_SPC_Request(SENT_SPC_CONTROL_T *pParam,
uint16_t *pHall, uint8_t *pStatus);
```

Reentrancy: Non-reentrant.

Parameters:
`*pParam` — pointer to the SENT/SPC channel configuration structure.
`*pHall` — pointer to the user variable where the received sensor Hall value will be stored.
`*pStatus` — pointer to the user variable where the received sensor status will be stored.

Return: None

Description: The function returns the actual Hall value and the status of the sensor. If any SENT/SPC channel error status bit is set, this function does nothing.

3.5 Master trigger pulse generation

The *SENT_SPC_Request()* API function initiates generation of the external transistor gate driving pulse to generate the master trigger pulse on the communication line.

The eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel operates in the output compare mode. The pulse width is defined by the *u8MasterTime* input parameter of the *SENT_SPC_Request()* API function. The eTimer channel compare load control 2 is set up by the *SENT_SPC_Init()* API function in such a way that the channel counter behaves as a modulo counter with a modulo value 0xFFFE (channel counter reset on 0xFFFE compare 2). The gate driving pulse rising edge is generated immediately once the eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel is enabled (compare on 0x0000), since the channel counter is already reset to 0x0000 by the software (a match with reload has already occurred). The gate driving pulse falling edge compare value is then set by the software (compare 1, according to the *u8MasterTime* value), and the compare 1 reload value is set to 0xFFFF. Once the falling edge of the transistor gate driving pulse is generated, the next compare 1 value is automatically reloaded to 0xFFFF (compare load control 1) outside of the counter range. This ensures that the gate driving pulse won't be generated again after a modulo counter overflow.

The channel is disabled and its counter and compare registers are re-initialized each time the *SENT_SPC_Request()* API function is called.

3.6 SENT data acquisition

The eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel detects falling edges on the eTimer_[SentSpcEtimer] module input pin SentSpcEtimerInput. Detection of each falling edge of the SENT/SPC frame captures the actual counter value of the eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel in the CAPT1 register and resets the channel counter to 0x0000. Simultaneously, an eDMA channel transfer request is generated on the selected eTimer_[SentSpcEtimer] DMA request channel (SentSpcEtimerDma). The eDMA engine then transfers the captured value to the driver timestamp buffer. The timestamp of each falling edge is used by the *SENT_SPC_Load()* API function to calculate the actual sensor unit time value and sensor data values.

After all the falling edges (defined by the selected SENT/SPC frame format) of the SENT/SPC frame are detected, the eDMA interrupt is invoked. Its ISR updates the driver status. The eDMA interrupt is invoked only if the *SENT_SPC_INTERRUPT* macro value is set to 1.

NOTE

> On the MPC564xL, the eDMA_0 interrupt can be detected by the INTC_0 and handled by Core_0 only. The SENT/SPC channel control structure must reside in the lower half of the SRAM (0x40000000–4000FFFF) in the Decoupled Parallel Mode (DPM).

An additional eDMA transfer request is generated when the *SENT_SPC_INTERRUPT* is set to 0. This additional eDMA transfer clears the driver status. This interrupt-free approach saves on CPU execution time but increases SRAM memory consumption (see Section 3.8.1, “Memory consumption,” on page 25).

The eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel counter is reset each time the *SENT_SPC_Request()* API function is called, and each time the falling edge is detected. Figure 9 illustrates the data acquisition process.

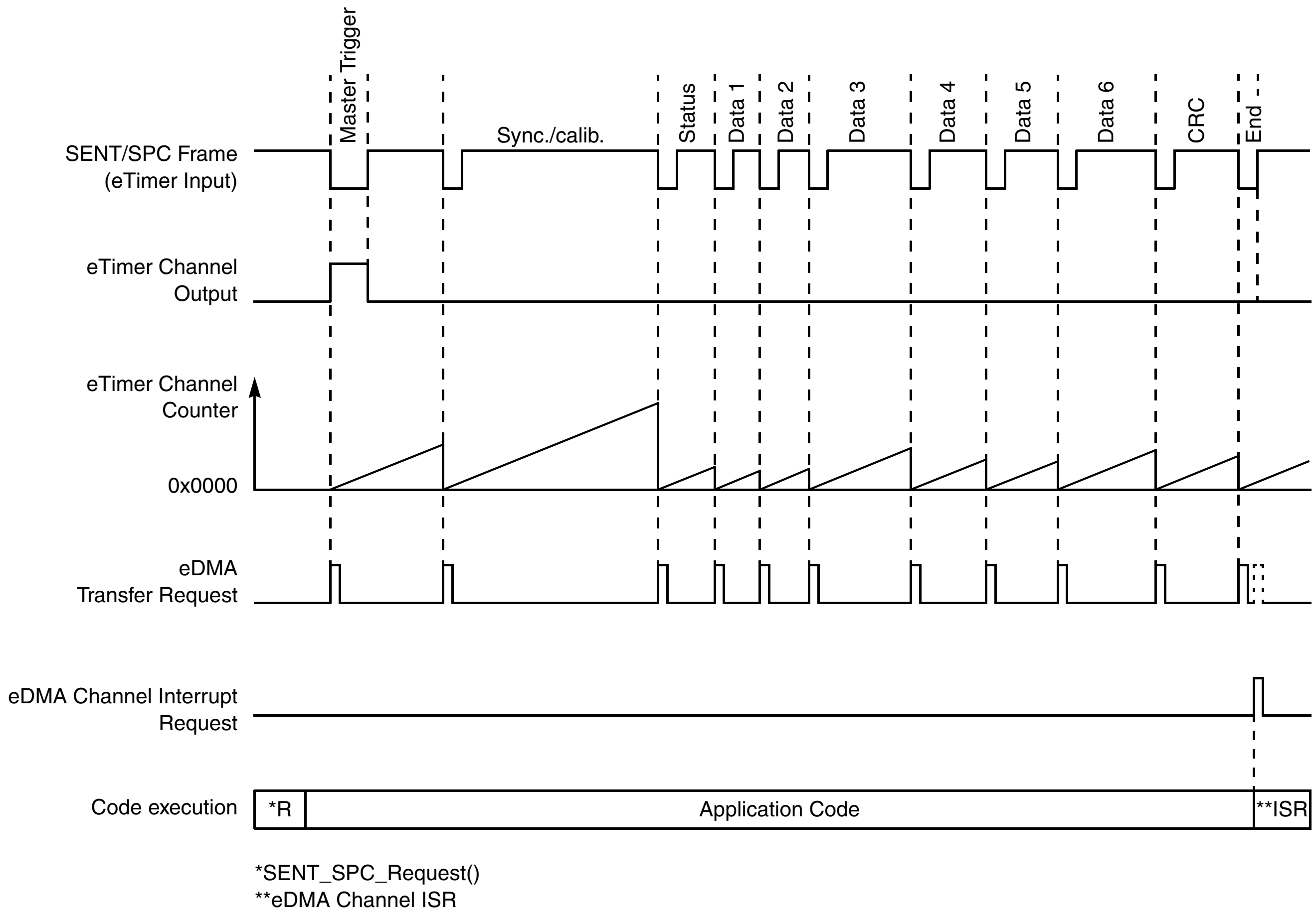

Figure 9. SENT/SPC data acquisition

3.7 API calling sequence

To guarantee the correct behavior of the driver, the following API call sequence is recommended (Figure 10 for illustration):

1. SENT_SPC_Init()
2. SENT_SPC_Request() (after a 1.2 ms modulus timer start — not handled by the driver)
3. SENT_SPC_Load() (after a modulus timer interrupt — not handled by the driver)
4. SENT_SPC_Read_Hall()
5. SENT_SPC_Request()
6. SENT_SPC_Load() (after the following modulus timer interrupt — not handled by the driver)
7. SENT_SPC_Read_Hall()
8. SENT_SPC_Request()
9. ...

3.7.1 Functional description

The driver channel status is internally held in the SENT/SPC channel configuration structure. However, all API functions, except for SENT_SPC_Read_Hall(), update and return the driver status in the form of data type SENT_SPC_STATE_T. Table 11 lists all SENT_SPC_STATUS_T type structure members.

Table 11. SENT_SPC_STATUS_T status word type definition

Structure bit member	Size	Range	Updated by API function(s)	Description
ErrorCRC	1-bit	0 or 1	SENT_SPC_Load	This bit reflects the result of the cyclic redundancy check. 0 CRC correct 1 CRC incorrect
StateInvalidData	1-bit	0 or 1	SENT_SPC_Init, SENT_SPC_Load	This bit indicates if the data is prepared for reading by the *SENT_SPC_Read_Hall()* API function. 0 Data is ready for reading 1 Data isn't ready for reading or is invalid
ErrorMultipleDMA	1-bit	0 or 1	SENT_SPC_Init	This bit indicates the result of eDMA channel initialization. 0 eDMA channel initialization done properly 1 eDMA channel is already used by another SENT/SPC channel or the channel number is out of range
ErrorMultipleEtimer	1-bit	0 or 1	SENT_SPC_Init	This bit indicates the result of eTimer unified channel initialization. 0 eTimer channel initialization done properly 1 eTimer_[SentSpcEtimer]_ETC[SentSpcEtimerOutput] channel and/or eTimer_[SentSpcEtimer] module input SentSpcEtimerInput is already used by another SENT/SPC channel

Table 11. SENT_SPC_STATUS_T status word type definition (continued)

Structure bit member	Size	Range	Updated by API function(s)	Description
StateTransmission	1-bit	0 or 1	SENT_SPC_Request	This bit indicates if the driver is waiting on new data from a sensor. 0 Driver acquired all data according to the selected frame format 1 Driver is waiting on new data
ErrorTimeout	1-bit	0 or 1	SENT_SPC_Load	This bit indicates if all data from a sensor was acquired properly at the time of the *SENT_SPC_Load()* API function call. 0 Data was acquired properly 1 Master trigger pulse was not generated or an incorrect number of data nibbles was received
ErrorNibbleOverflow	1-bit	0 or 1	SENT_SPC_Load	This bit reflects the result of the data nibble value check. 0 Data nibble value is in the proper range (0x00..0x0F) 1 Data nibble overflow (greater than 0x0F or pulse shorter than 12UT)
ErrorNumberOfNibbles	1-bit	0 or 1	SENT_SPC_Load	This bit indicates if the number of received nibbles is correct according to the selected frame format. 0 Correct number of nibbles was received 1 Incorrect number of nibbles was received
ErrorNoMasterPulse	1-bit	0 or 1	SENT_SPC_Load	This bit indicates if the master trigger pulse was properly generated on the communication line. 0 Master trigger pulse properly generated 1 Master trigger pulse not generated (*SENT_SPC_Request()* API function was not called or an external transistor malfunction occurred)
ErrorMultipleETDma	1-bit	0 or 1	SENT_SPC_Init	This bit indicates the result of the eTimer DMA request channel initialization. 0 eTimer_[SentSpcEtimer] DMA request channel initialization done properly 1 eTimer_[SentSpcEtimer] DMA request channel is already used by another SENT/SPC channel

Table 11. SENT_SPC_STATUS_T status word type definition (continued)

Structure bit member	Size	Range	Updated by API function(s)	Description
ErrorPinMuxing	1-bit	0 or 1	SENT_SPC_Init	This bit reflects the result of the eTimer input/output pin multiplexing configuration. 0 eTimer input/output pin multiplexing configuration done properly 1 eTimer input/output pin multiplexing error (SentSpcInputMux and/or SentSpcOutputMux values set incorrectly in the SENT/SPC channel control structure) — one or more of the following conditions occurred: • SentSpcInputMux and/or SentSpcOutputMux value is greater than the number of pin multiplexing options for the selected eTimer_[SentSpcEtimer] module • Invalid multiplexing combination (no pin assigned to a multiplexing option) • Selected input pin is already used as an output by another SENT/SPC channel (or vice versa) • Duplicate usage of pin A[4] between eTimer_0 and eTimer_1
Reserved	5-bit	—	—	Reserved bits.

The driver initialization is done by the *SENT_SPC_Init()* API function. If any of the eTimer channel outputs or inputs (including assigned pins), eTimer DMA request channels, or eDMA channels defined in the SENT/SPC channel configuration structure is already used by another initialized SENT/SPC channel, the *ErrorMultipleEtimer*, *ErrorPinMuxing, ErrorMultipleETDma*, or *ErrorMultipleDMA* status bits are set. These are the development errors. The SENT/SPC channel configuration structure needs to then be re-initialized to proper values.

If the configuration structure is properly initialized, the *StateInvalidData* status bit is set to indicate that the driver is initialized and the data in the internal buffer is invalid.

The *SENT_SPC_Request()* API function needs to be called to request the data from the sensor. The *StateTransmission* status bit is set after the request is processed, indicating that the request was properly processed and the driver is waiting on new data. This bit is then cleared automatically after a successful SENT/SPC frame reception.

The *SENT_SPC_Request()* function call should be done periodically. The minimum possible period of time is defined by the sum of the complete SENT/SPC frame maximal width and the execution time of the *SENT_SPC_Load()*, *SENT_SPC_Read_Hall()* and *SENT_SPC_Request()* API functions (see Table 14). The 1.2 ms time period is considered as a safe value.

The eDMA channel interrupt is invoked after all the SENT/SPC frame pulses are properly detected. The respective ISR (*SENT_SPC_DMA_Interrupt_Ch[15..0]*) then clears the StateTransmission status bit to indicate a complete frame reception. The eDMA interrupt is invoked only if the *SENT_SPC_INTERRUPT* macro is equal to one. Otherwise, the additional eDMA transfer request to clear the status is generated. See Table 1 for the *SENT_SPC_INTERRUPT* macro description.

To process the captured timing values, the *SENT_SPC_Load()* API function needs to be called at the beginning of the next 1.2 ms period. If all the SENT/SPC frame pulses are not properly detected by the driver at the time of the *SENT_SPC_Load()* API function call (the *StateTransmission* bit is still set to one), the *ErrorTimeOut* status bit is set. To extend the information value, the *ErrorNoMasterPulse* status bit is then set, even if the master trigger pulse was not detected, or the *ErrorNumberOfNibbles* status bit is set indicating an invalid number of received pulses with respect to the selected SENT/SPC channel frame format.

If all the SENT/SPC frame nibble pulses were properly detected, the *ErrorNibbleOverflow* status bit is set if one or more data nibble pulse contains a data value greater than 15 (0x0F) or the width of the pulse is shorter than 12 UT. If the calculated CRC value is not equal to the received checksum nibble value, the *ErrorCRC* status bit is set.

The *StateInvalidData* status bit remains set during the data processing by the *SENT_SPC_Load()* API function.

NOTE

If the *SENT_SPC_Load()* API function returns any errors, the user is advised to request new data by the *SENT_SPC_Request()* function. The status is then updated by the subsequent *SENT_SPC_Load()* function call at the beginning of the consecutive 1.2 ms periods. If these errors remain set, the SENT/SPC channel frame format might be set incorrectly, the sensor is providing erroneous data, an external transistor malfunction has occurred, or the API sequence was not executed in the proper order.

The actual Hall value is extracted from the received data by the *SENT_SPC_Read_Hall()* API function based on the selected frame format. If any SENT/SPC channel error status bit is set, this function does nothing.

Figure 10 shows the API calling sequence, possible state transitions, and error reporting. The figure shows also all possible transitions, differentiated by colors.

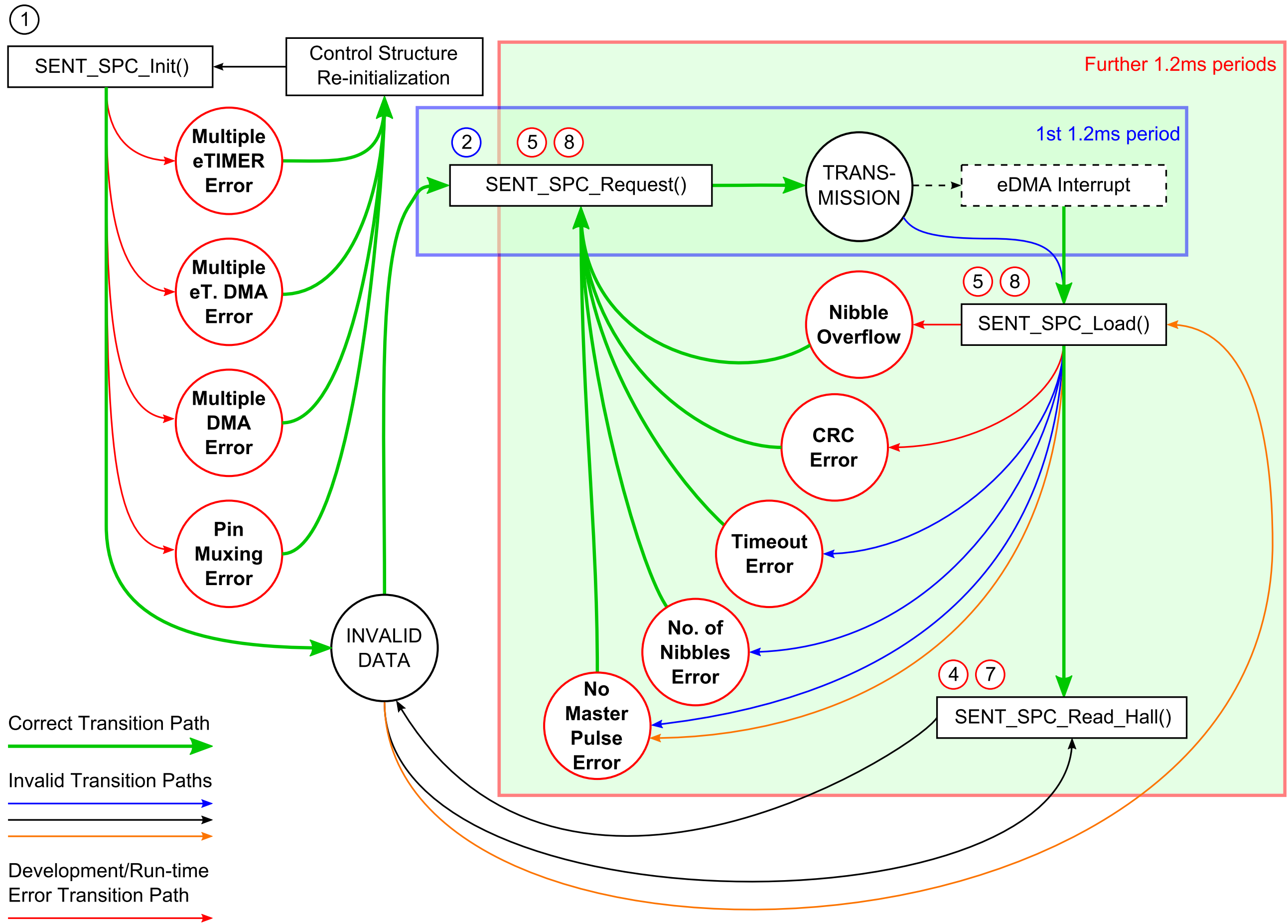

Figure 10. API calling sequence and status

3.8 Resource metrics

This chapter provides information about the memory consumption and execution times of the driver API and interrupt. The driver was compiled using the Green Hills compiler options listed in Table 12 without any optimization.

Table 12. Compiler options

Compiler option	Description
-bsp generic	Generic target board.
-cpu=ppc560xp or -cpu=ppc564xl	Target processor (MPC560xP or MPC564xL).
-G	Generates Green Hills MULTI debugging information.
-dual_debug	Enables generation of DWARF, COFF, or BSD debugging information in the object file, according to the convention of the target.
--no_commons	Allocates uninitialized global variables to a section and initializes them to zero at program startup.
-pnone	Disables call count profiling.
-vle	Enables VLE code generation and linkage with VLE libraries.
-c	Produces an object file for each source file.
-noSPE	Disables the use of the Signal Processing Engine and vector floating point instructions by the compiler.

3.8.1 Memory consumption

Table 13 lists the memory consumption of the driver API functions, static functions, static variables, and constants.

Table 13. Driver memory consumption

API function / internal function / ISR / variable / constant	Memory section	Memory type	Size [bytes] (MPC5604P)		Size [bytes] (MPC5643L)	
			SENT_SPC_INTERRUPT			
			0	1	0	1
SENT_SPC_Init()	.vletext	Flash	2828	2662	3434	3268
SENT_SPC_Request()	.vletext	Flash	710	702	710	702
SENT_SPC_Load()	.vletext	Flash	458	440	458	440
SENT_SPC_Read_Hall()	.vletext	Flash	94	94	94	94
SENT_SPC_DMA_Process_Interrupt()	.vletext	Flash	—	44	—	44
SENT_SPC_Interrupt[15..0]()	.vletext	Flash	—	42	—	114
Single SENT/SPC channel configuration structure variable	.bss	SRAM	128	42	128	42
Internal constants	.rodata	Flash	102	100	138	136
Internal initialized variables	.data	SRAM	128	128	160	160

3.8.2 Execution time consumption

The numbers of cycles listed in Table 14 were measured on the e200z0 core at a 64 MHz system clock frequency, and on the e200z4 core at an 80 MHz system clock frequency using optimal flash read wait state control (see Section 3.9, "Application example") and with an enabled instruction cache (MPC5643L only). A 3–data-nibble frame format (12-bit Hall) was used for the measurement.

Table 14. Execution time

API Function / ISR	Number of cycles			
	MPC5604P (e200z0) @64 MHz		MPC5643L (e200z4) @80 MHz	
	SENT_SPC_INTERRUPT			
	0	1	0	1
SENT_SPC_Init	885	829	1058	967
SENT_SPC_Request	233	227	273	274
SENT_SPC_Load	449	455	354	368
SENT_SPC_Read_Hall	111	107	102	96
SENT_SPC_DMA_Interrupt_Ch[N]	—	80[1]	—	91[1]

[1] Includes prolog and epilog of the ISR (INTC in hardware vector mode).

3.9 Application example

Example 3-1 MPC5604P SENT/SPC application example

```
#include "mpc5604p.h"              /* The register and bit field definitions for MPC5604P   */

#include "SENT_SPC_Driver.h"

__interrupt void Periodically(void);
void initClock(void);
void initINTC(void);

static uint16_t ui16Nibble_hall_ch0, ui16Nibble_hall_ch1;
static uint8_t ui8Nibble_status_ch0,ui8Nibble_status_ch1;
SENT_SPC_STATE_T ui16Error_ch0,ui16Error_ch1;

#if(SENT_SPC_INTERRUPT == 0)
#pragma alignvar(32)
#endif
static SENT_SPC_CONTROL_T ch0, ch1;

void initClock(void)
{
      CGM.CMU_0_CSR.R = 0x00000006;       /* Avoid CMU reset when fXOSC<fIRC                */

      /* Configure PLL_0, PLL_1 */
      CGM.FMPLL[0].CR.R = 0x12400000;     /* 64MHz PLL_0 @ 40MHz XOSC                       */
      CGM.FMPLL[1].CR.R = 0x113C0000;     /* 120MHz PLL_1 @ 40MHz XOSC                      */

      ME.RUN[0].R = 0x001F00F4;           /* XOSC sys. clk; Main vol. reg. ON               */
                                          /* PLL0/1 ON, XOSC ON, 16MHz_IRC ON               */
                                          /* DFLASH/CFLASH ON */
      ME.RUNPC[0].R = 0x00000010;         /* Peripherals run in RUN0 only                   */

      ME.MCTL.R = 0x40005AF0;             /* Enter RUN0 & key                               */
      ME.MCTL.R = 0x4000A50F;             /* Enter RUN0 & inverted key                      */

      /* Wait for mode transition */
      while(!(ME.IS.R & 0x00000001))
      {
      }

      ME.IS.R &= 0x00000001;      /* Clear I_MTC flag                                       */

      while(ME.GS.B.S_PLL0==0);   /* Check PLL0 lock status                                 */
      while(ME.GS.B.S_PLL1==0);   /* Check PLL1 lock status                                 */

      CGM.AC0SC.R = 0x05000000;   /* PLL_1 output as AUX clk. 0                             */
      CGM.AC0DC.R = 0x80000000;   /* Divided by 1, prescaler enabled                        */
}
```

```
void initINTC(void)
{
      INTC.MCR.R = 1;                 /* Enable HW vector mode                              */

      INTC.PSR[11].R = 1;             /* Set eDMA channel 0 priority higher than 0          */
      INTC.PSR[14].R = 1;             /* Set eDMA channel 3 priority higher than 0          */
      INTC.PSR[59].R = 2;             /* Set PIT_0 interrupt priority                       */

      INTC.CPR.R = 0;                 /* Set current priority for z0 to 0                   */
      asm("wrteei 1");                /* Enable z0 core external interrupts                 */
}

void PIT_0_Init(uint32_t tlval)
{
      PIT.PITMCR.R = 0x00000001;      /* Timer stopped in debug mode                        */
      PIT.CH[0].LDVAL.R = tlval;      /* Period                                             */
      PIT.CH[0].TFLG.R = 0x00000001;  /* Clear RTIF flag                                    */
      PIT.CH[0].TCTRL.R = 0x00000003; /* Enable interrupts and timer                        */
}

void main(void)
{
      CFLASH.PFCR0.R = 0x10C580ED;    /* Read Wait State Control & Address Pipelining       */
                                      /* Control for 64 MHz - Two additional wait states    */
                                      /* added, other settings in default                   */
      initINTC();                     /* Initialize interrupt controller                    */
      initClock();                    /* Set sysclk = 64 MHz running from PLL_0,            */
                                      /* Motor control clock = 120 MHz running from PLL_1   */

      /* e9414PS - Possible false DMA request from eTimer workaround                        */
      ETIMER_0.ENBL.R = 0x0000;
      ETIMER_1.ENBL.R = 0x0000;
      ETIMER_0.DREQ[0].R = 0x001F;
      ETIMER_0.DREQ[1].R = 0x001F;
      ETIMER_1.DREQ[0].R = 0x001F;
      ETIMER_1.DREQ[1].R = 0x001F;

      ch0.SentSpcEtimer = 0;
      ch0.SentSpcEtimerOutput = 0;
      ch0.SentSpcEtimerInput = 1;
      ch0.SentSpcOutputMux = 0;       /* pin A[0] used as output                            */
      ch0.SentSpcInputMux = 0;        /* pin A[1] used as input                             */
      ch0.SentSpcEtimerDma = 0;
      ch0.SentSpcDma = 0;
      ch0.SentSpcFrame = SPC_FRAME_3;

      ch1.SentSpcEtimer = 1;
      ch1.SentSpcEtimerOutput = 3;
      ch1.SentSpcEtimerInput = 3;
      ch1.SentSpcOutputMux = 2;       /* pin F[12] used as output                           */
      ch1.SentSpcInputMux = 0;        /* pin B[1] used as input                             */
      ch1.SentSpcEtimerDma = 1;
      ch1.SentSpcDma = 3;
      ch1.SentSpcFrame = SPC_FRAME_3;
```

```
    ui16Error_ch0 = SENT_SPC_Init(&ch0);
    ui16Error_ch1 = SENT_SPC_Init(&ch1);

    PIT_0_Init(0x12C00);            /* Enable PIT_0 with 1.2 ms period                   */

    ui16Error_ch0 = SENT_SPC_Request(&ch0,SPC_SYNCH);
    ui16Error_ch1 = SENT_SPC_Request(&ch1,SPC_SYNCH);

    while (1)
    {
    }        /* Wait forever */
}

__interrupt void Periodically(void)
{
    PIT.CH[0].TFLG.R = 0x00000001;

    ui16Error_ch0 = SENT_SPC_Load(&ch0);
    SENT_SPC_Read_Hall(&ch0,&ui16Nibble_hall_ch0,&ui8Nibble_status_ch0);
    ui16Error_ch0 = SENT_SPC_Request(&ch0,SPC_SYNCH);

    ui16Error_ch1 = SENT_SPC_Load(&ch1);
    SENT_SPC_Read_Hall(&ch1,&ui16Nibble_hall_ch1,&ui8Nibble_status_ch1);
    ui16Error_ch1 = SENT_SPC_Request(&ch1,SPC_SYNCH);

    INTC.EOIR.R = 0x0;              /* Exit Interrupt (End-of-Interrupt Register)       */
}
```

Example 3-2 MPC5643L SENT/SPC application example

```
#include "mpc5643l.h"              /* The register and bit field definitions for MPC5643L   */

#include "SENT_SPC_Driver.h"

__interrupt void Periodically(void);
void initClock(void);
void initINTC(void);

static uint16_t ui16Nibble_hall_ch0, ui16Nibble_hall_ch1;
static uint8_t ui8Nibble_status_ch0,ui8Nibble_status_ch1;
SENT_SPC_STATE_T ui16Error_ch0,ui16Error_ch1;

#if(SENT_SPC_INTERRUPT == 0)
#pragma alignvar(32)
#endif
static SENT_SPC_CONTROL_T ch0, ch1;

void initClock(void)
{
      CGM.AC3SC.R = 0x01000000;           /* Clock source for FMPLL_0 to XOSC               */
      CGM.AC4SC.R = 0x01000000;           /* Clock source for FMPLL_1 to XOSC               */

      /* Configure PLL_0, PLL_1 */
      CGM.FMPLL[0].CR.R = 0x1D400000;     /* 80MHz PLL_0 @ 40MHz XOSC                       */
      CGM.FMPLL[1].CR.R = 0x113C0000;     /* 120MHz PLL_1 @ 40MHz XOSC                      */

      ME.RUN[0].R = 0x001F00F4;           /* XOSC sys. clk; Main vol. reg. ON               */
                                          /* PLL0/1 ON, XOSC ON, 16MHz_IRC ON               */
                                          /* DFLASH/CFLASH ON */
      ME.RUNPC[0].R = 0x00000010;         /* Peripherals run in RUN0 only                   */

      ME.MCTL.R = 0x40005AF0;             /* Enter RUN0 & key                               */
      ME.MCTL.R = 0x4000A50F;             /* Enter RUN0 & inverted key                      */

      /* Wait for mode transition */
      while(!(ME.IS.R & 0x00000001))
      {
      }

      ME.IS.R &= 0x00000001;      /* Clear I_MTC flag                                       */

      while(ME.GS.B.S_PLL0==0);   /* Check PLL0 lock status                                 */
      while(ME.GS.B.S_PLL1==0);   /* Check PLL1 lock status                                 */

      CGM.AC0SC.R = 0x05000000;   /* PLL_1 output as AUX clk. 0                             */
      CGM.AC0DC.R = 0x80000000;   /* Divided by 1, prescaler enabled                        */
}
```

```
void initINTC(void)
{
      INTC.MCR.R = 1;                /* Enable HW vector mode                          */

      INTC.PSR[11].R = 1;            /* Set eDMA channel 0 priority higher than 0      */
      INTC.PSR[14].R = 1;            /* Set eDMA channel 3 priority higher than 0      */
      INTC.PSR[59].R = 2;            /* Set PIT_0 interrupt priority                   */

      INTC.CPR.R = 0;                /* Set current priority for z4 Core_0 to 0        */
      asm("wrteei 1");               /* Enable z4 Core_0 external interrupts           */
}

void PIT_0_Init(uint32_t tlval)
{
      PIT.PITMCR.R = 0x00000001;     /* Timer stopped in debug mode                    */
      PIT.CH[0].LDVAL.R = tlval;     /* Period                                         */
      PIT.CH[0].TFLG.R = 0x00000001; /* Clear RTIF flag                                */
      PIT.CH[0].TCTRL.R = 0x00000003; /* Enable interrupts and timer                   */
}

void main(void)
{
      PFLASH2P_LCA.PFCR0.R = 0x10C5EDED;    /* Read Wait State Control & Address Pipelining */
                                            /* Control for 80 MHz - Two additional wait     */
                                            /* states added, other settings in default      */
      initINTC();                    /* Initialize interrupt controller                */
      initClock();                   /* Set sysclk = 80 MHz running from PLL_0,        */
                                     /* Motor control clock = 120 MHz running from PLL_1 */

      PBRIDGE.MPROT0_7.R |= 0x00700000; /* Configure eDMA as a trusted master          */

      /* e9414PS - Possible false DMA request from eTimer workaround                   */
      mcTIMER0.ENBL.R = 0x0000;
      mcTIMER1.ENBL.R = 0x0000;
      mcTIMER2.ENBL.R = 0x0000;
      mcTIMER0.DREQ[0].R = 0x001F;
      mcTIMER0.DREQ[1].R = 0x001F;
      mcTIMER1.DREQ[0].R = 0x001F;
      mcTIMER1.DREQ[1].R = 0x001F;
      mcTIMER2.DREQ[0].R = 0x001F;
      mcTIMER2.DREQ[1].R = 0x001F;

      ch0.SentSpcEtimer = 0;
      ch0.SentSpcEtimerOutput = 0;
      ch0.SentSpcEtimerInput = 1;
      ch0.SentSpcOutputMux = 0;      /* pin A[0] used as output                        */
      ch0.SentSpcInputMux = 1;       /* pin D[11] used as input                        */
      ch0.SentSpcEtimerDma = 0;
      ch0.SentSpcDma = 0;
      ch0.SentSpcFrame = SPC_FRAME_3;
```

```
	ch1.SentSpcEtimer = 2;
	ch1.SentSpcEtimerOutput = 2;
	ch1.SentSpcEtimerInput = 2;
	ch1.SentSpcOutputMux = 1;	/* pin I[2] used as output	*/
	ch1.SentSpcInputMux = 0;	/* pin H[10] used as input	*/
	ch1.SentSpcEtimerDma = 1;
	ch1.SentSpcDma = 3;
	ch1.SentSpcFrame = SPC_FRAME_3;

	ui16Error_ch0 = SENT_SPC_Init(&ch0);
	ui16Error_ch1 = SENT_SPC_Init(&ch1);

	PIT_0_Init(0x12C00);		/* Enable PIT_0 with 1.2 ms period	*/

	ui16Error_ch0 = SENT_SPC_Request(&ch0,SPC_SYNCH);
	ui16Error_ch1 = SENT_SPC_Request(&ch1,SPC_SYNCH);

	while (1)
	{
	}	/* Wait forever */
}

__interrupt void Periodically(void)
{
	PIT.CH[0].TFLG.R = 0x00000001;

	ui16Error_ch0 = SENT_SPC_Load(&ch0);
	SENT_SPC_Read_Hall(&ch0,&ui16Nibble_hall_ch0,&ui8Nibble_status_ch0);
	ui16Error_ch0 = SENT_SPC_Request(&ch0,SPC_SYNCH);

	ui16Error_ch1 = SENT_SPC_Load(&ch1);
	SENT_SPC_Read_Hall(&ch1,&ui16Nibble_hall_ch1,&ui8Nibble_status_ch1);
	ui16Error_ch1 = SENT_SPC_Request(&ch1,SPC_SYNCH);

	INTC.EOIR.R = 0x0;		/* Exit Interrupt (End-of-Interrupt Register)	*/
}
```

4 Conclusion

The application note describes the SENT protocol basics along with its SPC enhancement.
The requirements for external components, a list of utilized peripherals, configuration description, application programming interface description, data acquisition description, the API calling sequence, and a functional description of the SENT/SPC driver for the MPC560xP and MPC564xL families of microcontrollers are provided in the text.

The software driver provides full communication with the Infineon TLE4998C programmable linear Hall sensor. It is fully compatible with all TLE4998C supported SPC modes and SENT/SPC frame formats.

The usage of MPC560xP/MPC564xL on-chip hardware peripherals, such as the eTimer and eDMA, provides a low e200z0/e200z4 core load. The driver consumes approximately 1.03% of the e200z0 execution time without interrupts, and 1.13% of the execution time with interrupts. The e200z4 consumes 0.76% without interrupts, and 0.86% with interrupts. These percentages are related to a 1.2 ms transmission triggering loop period at a 64 MHz (MPC560xP) and 80 MHz (MPC564xL) system clock frequency, and a single SENT/SPC channel operation.

5 References

1. SAE J2716 (R) SENT – Single Edge Nibble Transmission for Automotive Applications, FEB2008
2. *MPC5604P Microcontroller Reference Manual,* Rev. 4, 15 Apr 2011
3. *MPC5604P Microcontroller Data Sheet,* Rev. 7, 04/2011
4. *MPC5643L Microcontroller Reference Manual,* Rev. 8, 09 May 2011
5. *MPC5643L Microcontroller Data Sheet,* Rev. 7, 3/2011
6. TLE4998C Target Data Sheet, V 0.3, July 2008

6 Acronyms

API	Application Programming Interface
CAN	Controller Area Network
COFF	Common Object File Format
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DMA_MUX	eDMA Channel Multiplexer
ECU	Electronic Control Unit
eDMA	Enhanced Direct Memory Access
EMC	Electromagnetic Compatibility
ESD	Electrostatic Discharge
ETC	eTimer Channel
eTimer	Enhanced Motor Control Timer
I/O	Input/Output
ID	Identification
INTC	Interrupt Controller
ISR	Interrupt Service Routine
LIN	Local Interconnect Network
MCU	Microcontroller Unit
PLL	Phase-Locked Loop
PWM	Pulse Width Modulation
RF	Radio Frequency
SAE	Society of Automotive Engineers
SENT	Single Edge Nibble Transmission
SIUL	System Integration Unit Lite
SPC	Short PWM Code
SRAM	Static Random Access Memory
UT	Unit Time
VLE	Variable Length Encoding

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4432
Rev. 0
01/2012

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org

© Freescale Semiconductor, Inc. 2012. All rights reserved.

freescale™
semiconductor