Freescale Semiconductor
Application Note

Document Number: AN4219
Rev. 0, 10/2010

SENT/SPC Driver for the MPC5510 Microcontroller Family

by: Josef Kramolis
Roznov pod Radhostem, Czech Republic

1 Introduction

This application note describes the SENT/SPC driver for the MPC5510 32-bit family of 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 example are shown in the further sections.

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

Contents

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 be also considered as a low-cost alternative to the LIN or CAN communication standards.

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

2.1 SENT Encoding Scheme

SENT is an unidirectional communication standard where data from a sensor is transmitted independently without any intervention of the data receiving device (e.g. 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 nibble, see Figure 1).

A SENT communication fundamental unit of time (unit time - UT, nominal transmitter clock period) can be in the range of 3 microseconds to 10 microseconds, 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 three microsecond 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)

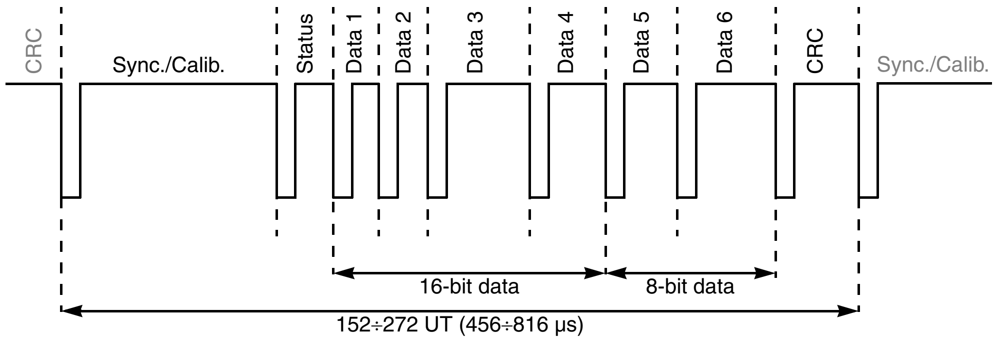

Figure 1. Transmission Example of 16-bit and 8-bit Signal Data

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 5 or more unit times. The remainder of the pulse width is driven high (see Figure 2).

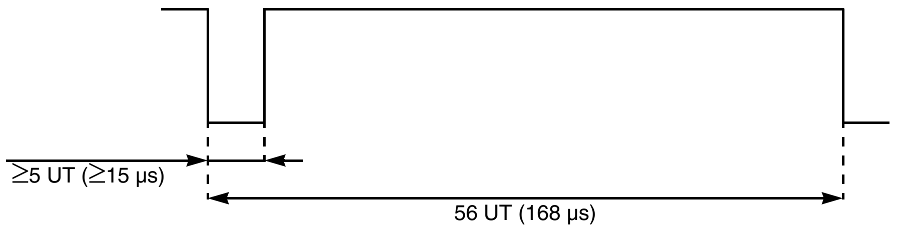

Figure 2. Synchronization/Calibration Pulse Format

2.1.2 Status and Communication Nibble Pulse

The Status nibble contains 4-bit status information of the sensor (e.g. 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 6 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 5 or more unit times. The remainder of the pulse width is driven high. The next pulse falling edge occurs after 12 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

$$DataNibblePulseWidth = (12 + NibbleValue)$$

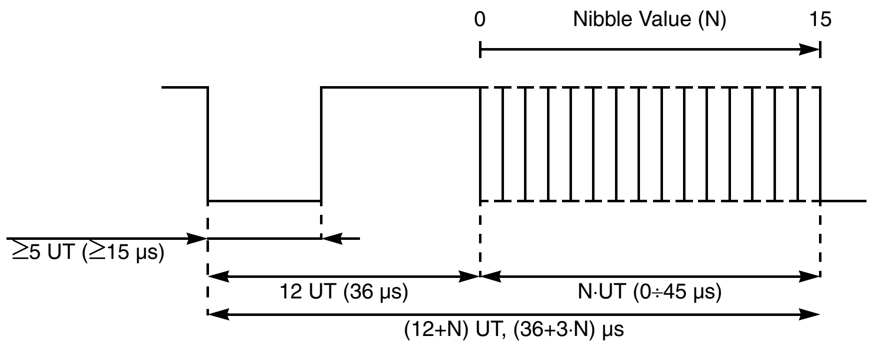

Figure 3. Data Nibble Pulse Format

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 includes also 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.

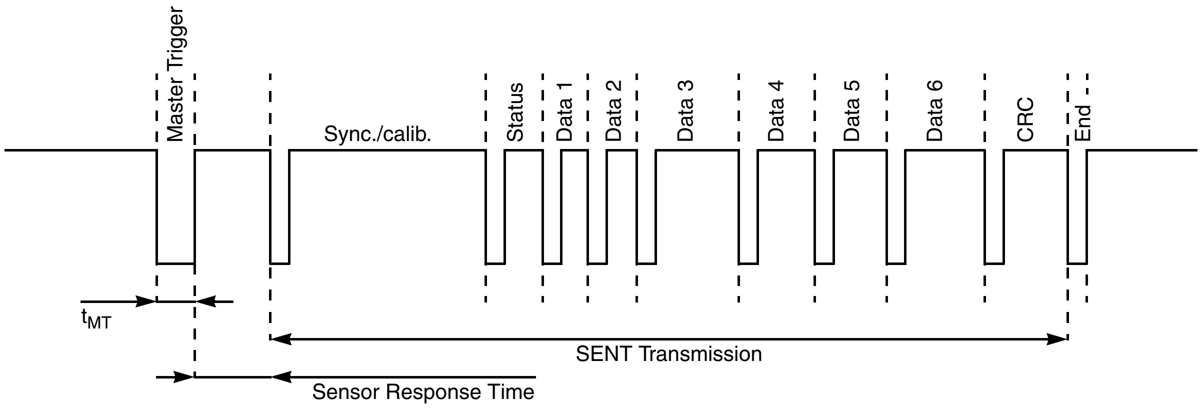

Figure 4. SENT/SPC Frame Format

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 4 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 5V voltage to supply the sensor. The communication line is pulled-up by the 10÷51kΩ resistor to the supply voltage. The receiver input is formed by the parasitic capacitance of the input pin and its ESD protection, and the 560Ω/2.2nF 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 0.1nF/m capacitance and the maximum cable length is limited 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.

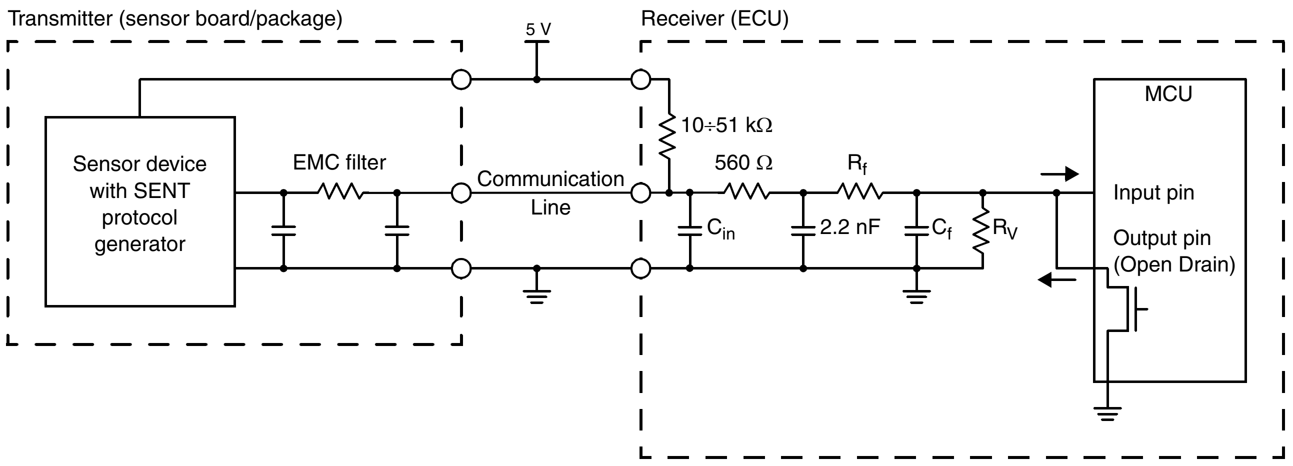

Figure 5. SENT/SPC Circuit Topology

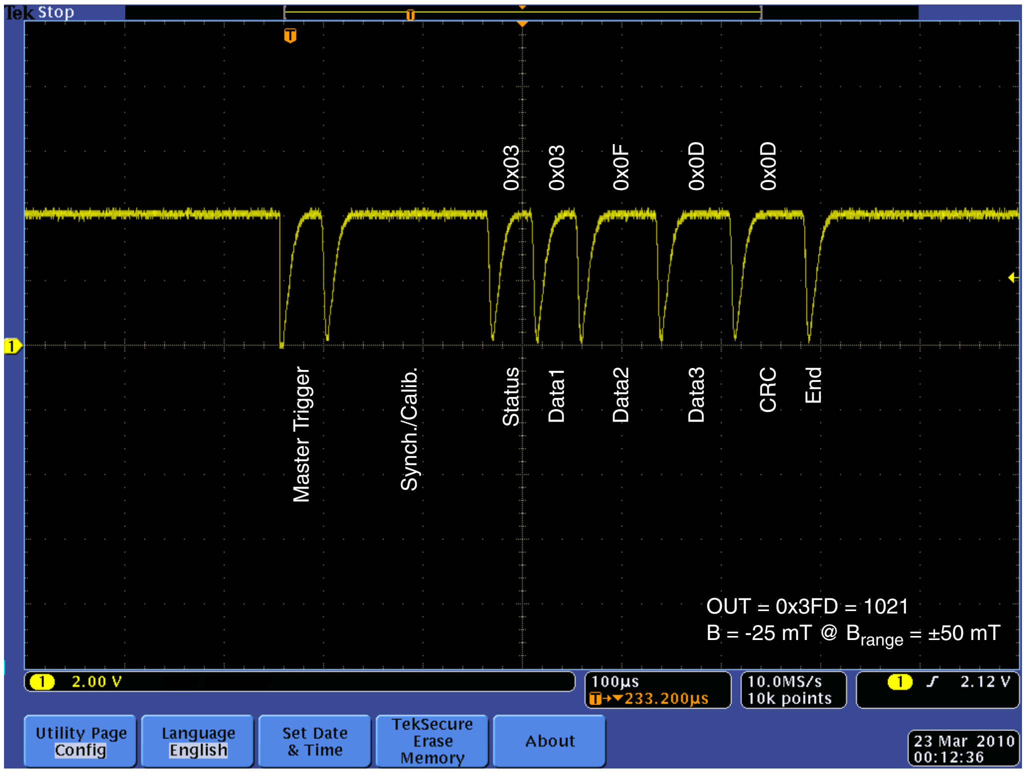

Figure 6. TLE4998C SENT/SPC 12-bit Hall Waveform

3 SENT/SPC Software Driver for the MPC5510

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 all members of the MPC5510 family and the Infineon TLE4998C programmable linear Hall sensor. The driver supports code execution by both the MPC5510 e200z1 core and the e200z0 core (if available on the device), and can be used for handling up to eight independent SENT/SPC channels.

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.

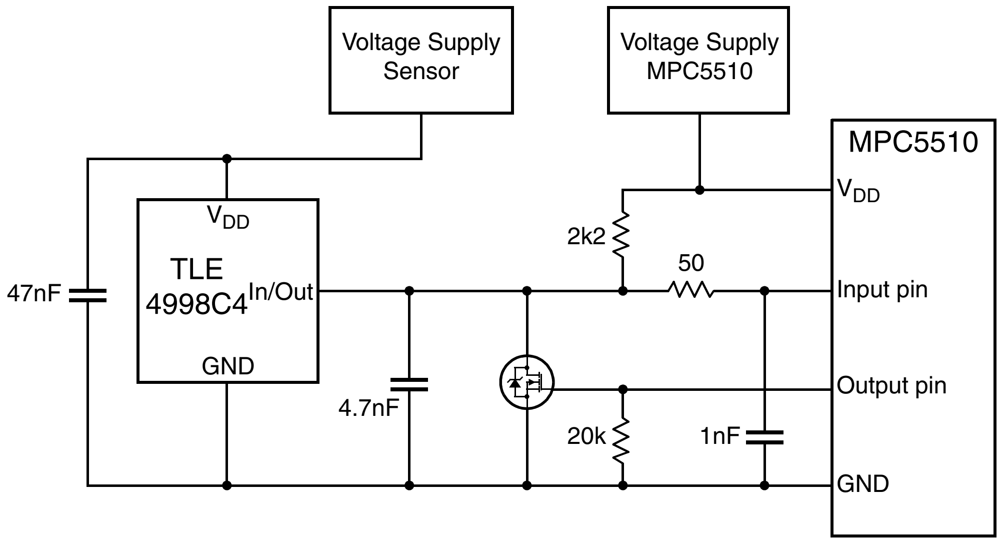

Figure 7. Typical TLE4998C Application Circuit with External Transistor

3.2 Utilized MPC5510 Peripherals

The driver utilizes the following MPC5510 peripherals:

- System Integration Unit (SIU) - 2 pins for a single SENT/SPC channel.
- Enhanced Input/Output Subsystem (eMIOS) - 2 unified channels for a single SENT/SPC channel.
- Enhanced Direct Memory Address engine (eDMA) - a single channel for a single SENT/SPC channel.

Only 16 out of a total 24 eMIOS unified channels available on MPC5510 devices are suitable for the driver operations, thus only 8 SENT/SPC channels can be handled by a single MPC5510 device.

3.3 Driver Configuration

There are three 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 build (see Table 1).

Table 1. Pre-compile Time Parameters.

Macro	Range	Description
SENT_SPC_CORE_SELECT	0 or 1	Defines the MPC5510 core which will handle the execution of the interrupt service routine (if enabled by SENT_SPC_INTERRUPT, see the table row below) to update the driver status at the end of the SENT/SPC frame transfer. 0 e200z1 core 1 e200z0 core
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_UT	—	Defines the number of the eMIOS unified channel ticks per 3 µs. This can be calculated using the formula: $SENT\_SPC\_UT = \frac{SystemClockFrequency \cdot 3 \cdot 10^{-6}}{eMIOSPeripheralDivider \cdot GlobalEMIOSPrescaler} - 1$ **Eqn. 2**

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. Follow 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 SentSpcDma
   b) Initialize structure member SentSpcEmiosInput
   c) Initialize structure member SentSpcEmiosOuput
   d) Initialize structure member SentSpcOutputPin
   e) 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
SentSpcDma	0..15	The eDMA channel number used for channel operation.
SentSpcEmiosInput	0..15	The eMIOS unified channel number used for data reception.
SentSpcEmiosOutput	0..15	The eMIOS unified channel number used for driving the external transistor (Master Trigger pulse generation).
SentSpcOutputPin	SENT_SPC_PIN_INOUT, SENT_SPC_PIN_OUT	Type of the eMIOS unified channel output pin. SENT_SPC_PIN_INOUT Input/output pin type (PC[15..0]) SENT_SPC_PIN_OUT Output pin type (PE[5..0], PD[15..12], PD[2..7])
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)

NOTE

Each SENT/SPC channel has to have its own unique eDMA channel and eMIOS unified channels for input/output assigned in the channel configuration structure variable. The driver, however, provides an internal checking mechanism for duplicated channels.

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

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)`
Re-entrancy	Non re-entrant
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 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 6)

NOTE

Initialization of the e200z1 core and the e200z0 core (if available), system clock (PLL), on-chip flash memory, SRAM, interrupt controller (INTC) and the interrupt vector table is not handled by the driver and 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)`
Re-entrancy	Non re-entrant
Parameters	`*pParam` — pointer to the SENT/SPC channel configuration structure. `u8MasterTime` — the width of the external transistor gate driving pulse in microseconds
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 6)

NOTE

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

The driver provides predefined macros for *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 3, Table 4, Table 5 list the provided macros based on the pre-programmed SPC protocol mode of the TLE4998C device(s).

Table 3. Typical Master Pulse Timing Macro for TLE4998C Synchronous Mode

Macro	Master Pulse Width [UT]	Gate Pulse Width [μs]
SPC_SYNCH	2.75	4

Table 4. Typical Master Pulse Timing Macros for TLE4998C ID Selection Mode

Macro	Sensor ID	Master 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 5. Typical Master Pulse Timing Macros for TLE4998C Dynamic Range Mode

Macro	Magnetic Field Range	Master 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)`
Re-entrancy	Non re-entrant
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 time-out (no master 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 6)

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)`
Re-entrancy	Non re-entrant
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 output dedicated eMIOS channel operates in the Double Action Output Compare mode. The pulse width is defined by the *u8MasterTime* input parameter of the *SENT_SPC_Request()* API function.

3.6 SENT Data Acquisition

The MPC5510 eMIOS unified channel dedicated to the input pin operates in Single Action Input Capture mode. Detection of each falling edge of the SENT/SPC frame captures the actual counter value of the input eMIOS unified channel in the internal eMIOS register. Simultaneously, an eDMA channel transfer request is generated by the eMIOS channel. The eDMA engine then transfers the captured value to the driver timestamp buffer. Timestamps of each falling edge are used by the *SENT_SPC_Load()* API function to calculate 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.

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 20).

The input eMIOS unified channel counter is reset each time the *SENT_SPC_Request()* API function is called. Figure 8 illustrates the data acquisition process.

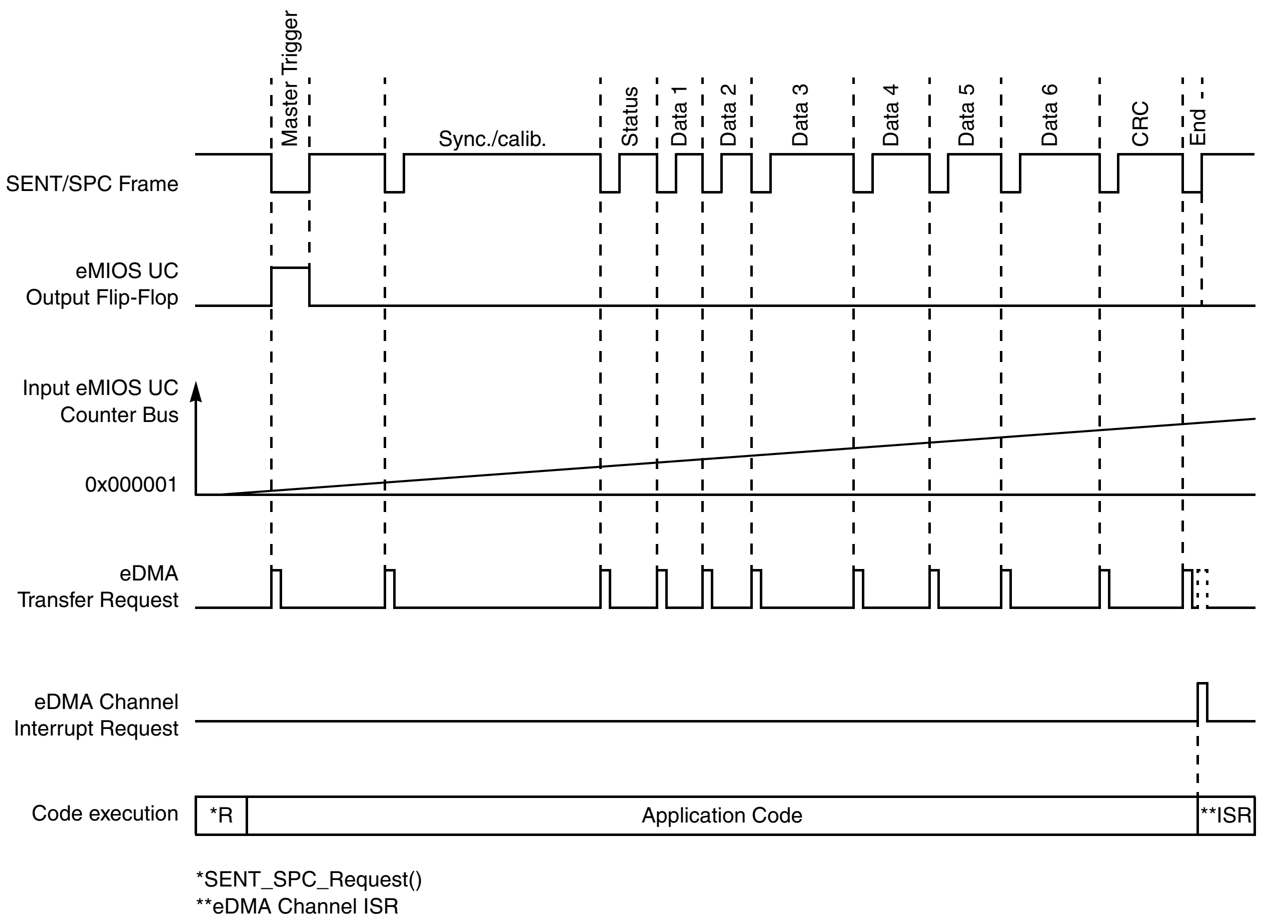

Figure 8. SENT/SPC Data Acquisition

3.7 API Calling Sequence

To guarantee the correct behavior of the driver, the following API call sequence is recommended (Figure 9 for illustration):

1. SENT_SPC_Init()
2. SENT_SPC_Request() (after a 1.2ms 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 6 lists all SENT_SPC_STATUS_T type structure members.

Table 6. 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 is not 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
ErrorMultipleEMIOS	1-bit	0 or 1	SENT_SPC_Init	This bit indicates the result of eMIOS unified channel initialization. 0 Both eMIOS unified channels are initialized properly 1 Either one or both eMIOS unified channels are already used by another SENT/SPC channel or the channel number is out of range

Table 6. 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)
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 pulse properly generated 1 Master pulse not generated (*SENT_SPC_Request()* API function was not called or an external transistor malfunction occurred)
Reserved	7-bit	—	—	Reserved bits

The driver initialization is done by the *SENT_SPC_Init()* API function. If any of the eMIOS unified channels or eDMA channels defined in the SENT/SPC channel configuration structure is already used by another initialized SENT/SPC channel, the *ErrorMultipleEMIOS* or *ErrorMultipleDMA* status bits are set. These are the development errors. The SENT/SPC channel configuration structure needs to be then re-initialized to proper channel 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 9). 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 1. 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). If the calculated CRC value is not equal to the received Checksum nibble value, the *ErrorCRC* status bit is set.

The *StateInvalidData* status bit is 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 9 shows the API calling sequence, possible state transitions and error reporting. The figure shows also all possible transitions, differentiated by colors.

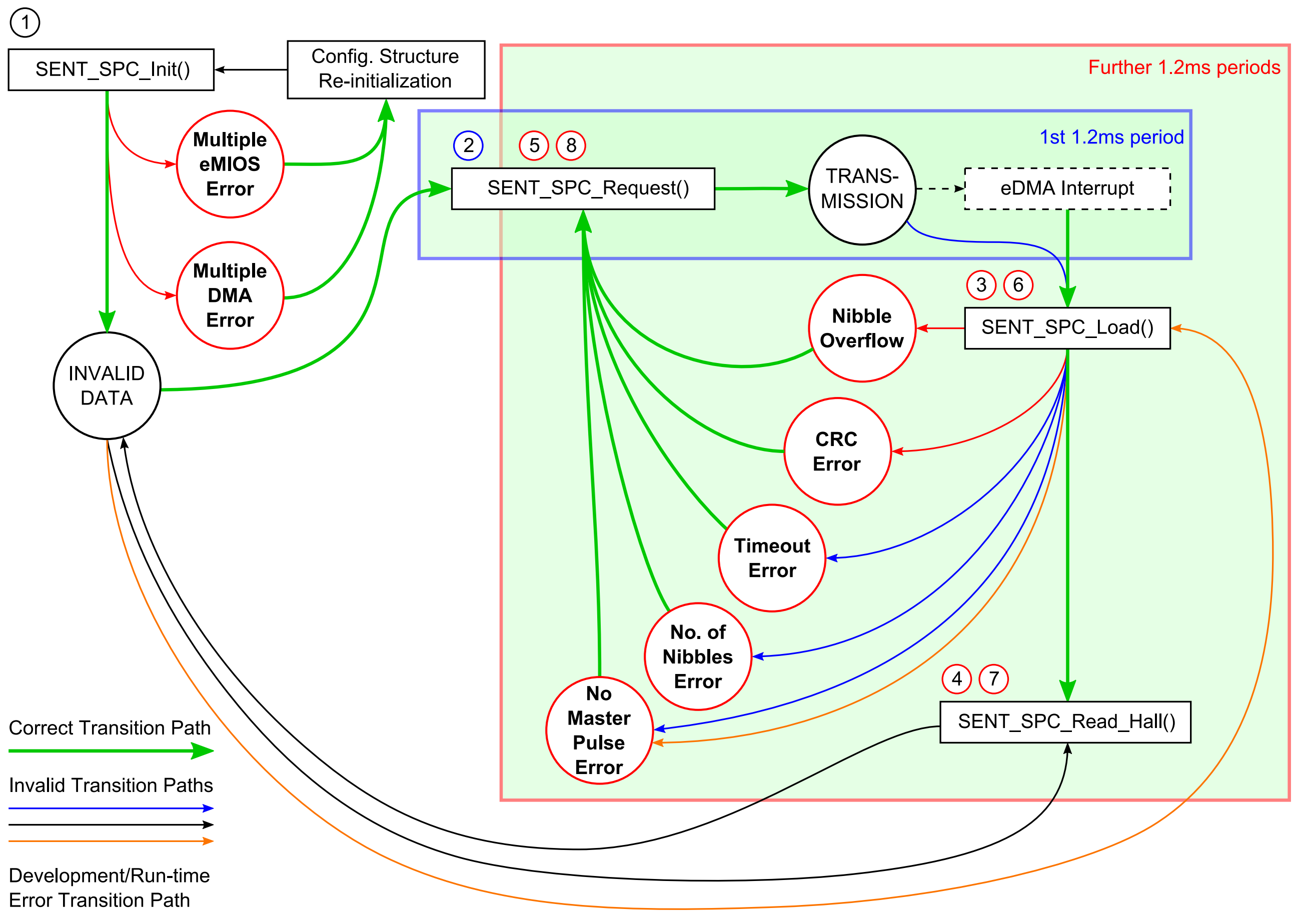

Figure 9. API Calling Sequence and Status

3.8 Resource Metrics

This chapter provides information about the memory consumption and execution times of the driver API and interrupt. The driver is compiled using the Green Hills compiler options listed in Table 7 without any optimization.

Table 7. Compiler Options

Compiler Option	Description
-bsp generic	Generic target board.
-cpu=ppc5516	Target processor.
-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 start-up.
-pnone	Disables call count profiling.
-vle	Enables VLE code generation and linkage with VLE libraries.
-c	Produces an object file for each source file.

3.8.1 Memory Consumption

Table 8 lists the memory consumption of the driver API functions, static functions, static variables and constants.

Table 8. Driver Memory Consumption

API Function / Internal function / ISR / Variable / Constant	Memory Section	Memory Type	Size [Bytes]	
			SENT_SPC_INTERRUPT	
			0	1
SENT_SPC_Init()	.vletext	Flash	934	700
SENT_SPC_Request()	.vletext	Flash	592	534
SENT_SPC_Load()	.vletext	Flash	572	550
SENT_SPC_Read_Hall()	.vletext	Flash	94	90
SENT_SPC_DMA_Process_Interrupt()	.vletext	Flash	—	102
SENT_SPC_Interrupt[15..0]()	.vletext	Flash	—	1822
Single SENT/SPC channel configuration structure variable	.bss	SRAM	128	40
Internal constants	.rodata	Flash	18	16
Internal initialized variables	.data	SRAM	128	128

3.8.2 Execution Time Consumption

The number of cycles listed in Table 9 were measured on the e200z1 core at a 75 MHz system clock frequency using optimal flash read/write wait state control and address pipelining control settings (see the *initSysclk()* function in the Section 3.9, "Application Example"). A 3 data nibble frame format (12-bit Hall) was used for the measurement.

Table 9. Execution Time

API Function / ISR	**Number of Cycles**	
	SENT_SPC_INTERRUPT	
	0	**1**
SENT_SPC_Init	693	537
SENT_SPC_Request	276	219
SENT_SPC_Load	726	726
SENT_SPC_Read_Hall	206	211
SENT_SPC_DMA_Interrupt_Ch[N]	—	202[1]

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

3.9 Application Example

```
#include "typedefs.h"       /* ITU types defined here                                  */
#include "mpc5510.h"        /* The register and bit field definitions for MPC5510      */

#include "SENT_SPC_Driver.h"

__interrupt void Periodically(void);
void initSysclk(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 initSysclk(void)
{
      CRP.CLKSRC.B.XOSCEN = 1;       /* Enable external oscillator                         */
      FMPLL.ESYNCR2.R = 0x00000005;  /* Set ERFD to initial value of 5                     */
      FMPLL.ESYNCR1.R = 0xF001003B;  /* Set CLKCFG=PLL, EPREDIV=0, EMFD=0x20               */
      while (FMPLL.SYNSR.B.LOCK != 1)
      {
      }                              /* Wait for PLL to LOCK                               */
      FMPLL.ESYNCR2.R = 0x00000003;  /* Set ERFD to final value for 75MHz sysclk           */
      SIU.SYSCLK.B.SYSCLKSEL = 2;    /* Select PLL for sysclk                              */
      FLASH.PFCRP0.B.RWSC = 0x2;   /* Read Wait State Control for 75 MHz Two additional     */
                                   /* wait-states are added                                 */
      FLASH.PFCRP0.B.WWSC = 0x1;   /* Write Wait State Control for 75 MHz One additional    */
                                   /* wait-state is added                                   */
      FLASH.PFCRP0.B.APC = 0x2;    /* Address Pipelining Control for 75 MHz Two additional */
                                   /* hold cycles are added                                 */
      FLASH.PFCRP1.B.RWSC = 0x2;   /* Read Wait State Control for 75 MHz Two additional     */
                                   /* wait-states are added                                 */
      FLASH.PFCRP1.B.WWSC = 0x1;   /* Write Wait State Control for 75 MHz One additional    */
                                   /* wait-state is added                                   */
      FLASH.PFCRP1.B.APC = 0x2;    /* Address Pipelining Control for 75 MHz Two additional */
                                   /* hold cycles are added                                 */
}

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[81].R = 2;          /* Set eMIOS channel 23 interrupt priority               */

      INTC.CPR_PRC0.R = 0;         /* Set current priority for z1 to 0                      */
      asm("wrteei 1");             /* Enable z1 core external interrupts                    */
}

void main(void)
{
      initINTC();                   /* Initialize interrupt controller                      */
      initSysclk();                /* Set sysclk = 75 MHz running from PLL                  */

      ch0.SentSpcDma = 0;
      ch0.SentSpcEmiosInput = 0;
      ch0.SentSpcEmiosOutput = 1;
      ch0.SentSpcOutputPin = SENT_SPC_PIN_INOUT;  /* Pad PC1 will be used for eMIOS[1]      */
      ch0.SentSpcFrame = SPC_FRAME_3;

      ch1.SentSpcDma = 3;
      ch1.SentSpcEmiosInput = 9;
      ch1.SentSpcEmiosOutput = 12;
      ch1.SentSpcOutputPin = SENT_SPC_PIN_OUT;    /* Pad PD4 will be used for eMIOS[12]     */
      ch1.SentSpcFrame = SPC_FRAME_3;


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

      // eMIOS Global initialization
      EMIOS.MCR.B.GPRE = 2;             /* eMIOS global clock prescaler divide ratio 3      */
      EMIOS.MCR.B.GPREN = 1;            /* Enable eMIOS clock                               */
      EMIOS.MCR.B.GTBE = 1;             /* Enable global time base                          */
      EMIOS.MCR.B.FRZ = 1;              /* Enable stopping channels when in debug mode      */

      EMIOS.CH[23].CADR.R = 29999;      /* Period will be 29999+1 = 30000 clocks (1,2 msec) */
      EMIOS.CH[23].CCR.B.MODE = 0x50;   /* MPC5510: Modulus Counter Buffered (MCB)          */
      EMIOS.CH[23].CCR.B.BSL = 0x3;     /* Use internal counter                             */
      EMIOS.CH[23].CCR.B.UCPRE = 0;     /* Set channel prescaler to divide by 1             */
```

```
        EMIOS.CH[23].CCR.B.FREN = 1;        /* Freeze channel counting when in debug mode       */
        EMIOS.CH[23].CCR.B.UCPREN = 1;      /* Enable prescaler; uses default divide by 1       */
        EMIOS.CH[23].CCR.B.DMA = 0;         /* Interrupt                                        */
        EMIOS.CH[23].CCR.B.FEN = 1;         /* Enable FLAG flag to generate interrupt           */

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

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

__interrupt void Periodically(void)
{
        EMIOS.CH[23].CSR.B.FLAG = 1;

        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_PRC0.R = 0x0;             /* Exit Interrupt (End-of-Interrupt Register)       */
}
```

4 Conclusion

The Application note AN4219 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 MPC5510 family 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 MPC5510 on-chip hardware peripherals such as the eMIOS and eDMA provides low e200z1/e200z0 core load. The driver consumes approximately 1.34% of the e200z1 execution time, with disabled interrupts, and 1.51% of the execution time with enabled interrupts. The percentages are related to the 1.2 ms transmission triggering loop period at a 75 MHz system clock frequency.

5 References

1. SAE J2716 (R) SENT - Single Edge Nibble Transmission for Automotive Applications, FEB2008
2. MPC5510 Microcontroller Family Reference Manual, Rev. 1, 06/2008
3. MPC5510 Microcontroller Family Data Sheet, Rev. 3, 3/2009
4. TLE4998C Target Data Sheet, V 0.3, July 2008

6 Acronyms

A/D	Analog to Digital
API	Application Programming Interface
CAN	Controller Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
eDMA	Enhanced Direct Memory Access
EMC	Electromagnetic Compatibility
eMIOS	Enhanced Input/Output Subsystem
ESD	Electrostatic Discharge
I/O	Input/Output
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
SIU	System Integration Unit
SPC	Short PWM Code
SRAM	Static Random Access Memory
UT	Unit Time

THIS PAGE IS INTENTIONALLY BLANK

Document Number: AN4219
Rev. 0
10/2010