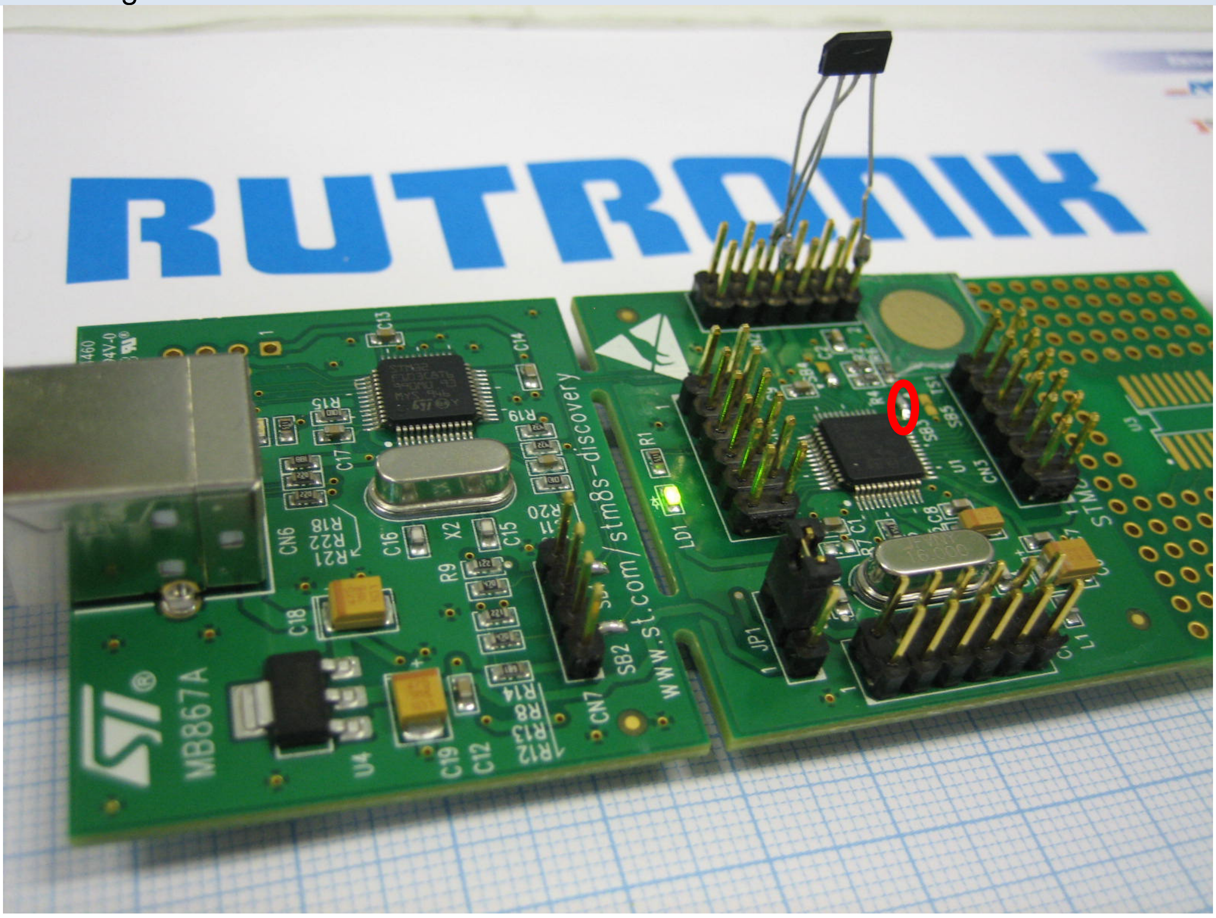

Image 1: The TLE4998S4 soldered to the STM8S-Discovery Board. Solder bridge **SB3** has to be closed = soldered.

SENT: Single Edge Nibble Transmission

The unidirectional data transmission SENT, Single Edge Nibble Transmission, has been developed to connect high-resolution sensors to electronic control units in the automotive environment. The connecting cable carries three wires: power supply, ground and data. The data is digitally coded in the duration between two adjacent negative edges of the data signal. Data is coded in nibbles i.e. 4 bits.

The benefits of SENT-coding are:

- Digitizing of the signal is located near the point of measurement in the sensor. This increases noise immunity.
- Simple galvanical isolation possible with optocoupler.
- CRC generated in the sensor to verify error free data transmission.

To reconstruct the measurement value it is necessary to measure the time between the negative edges. This is a typical task for a capture timer.

Every data frame starts with a syncfield of known duration. This text considers Infineon's Hall-sensor TLE4998 whose syncfield is 168µs long. A microcontroller with a compare timer can be used to measure this interval. The measured value is consequently taken as a timing reference when the durations of the following data nibbles are evaluated.

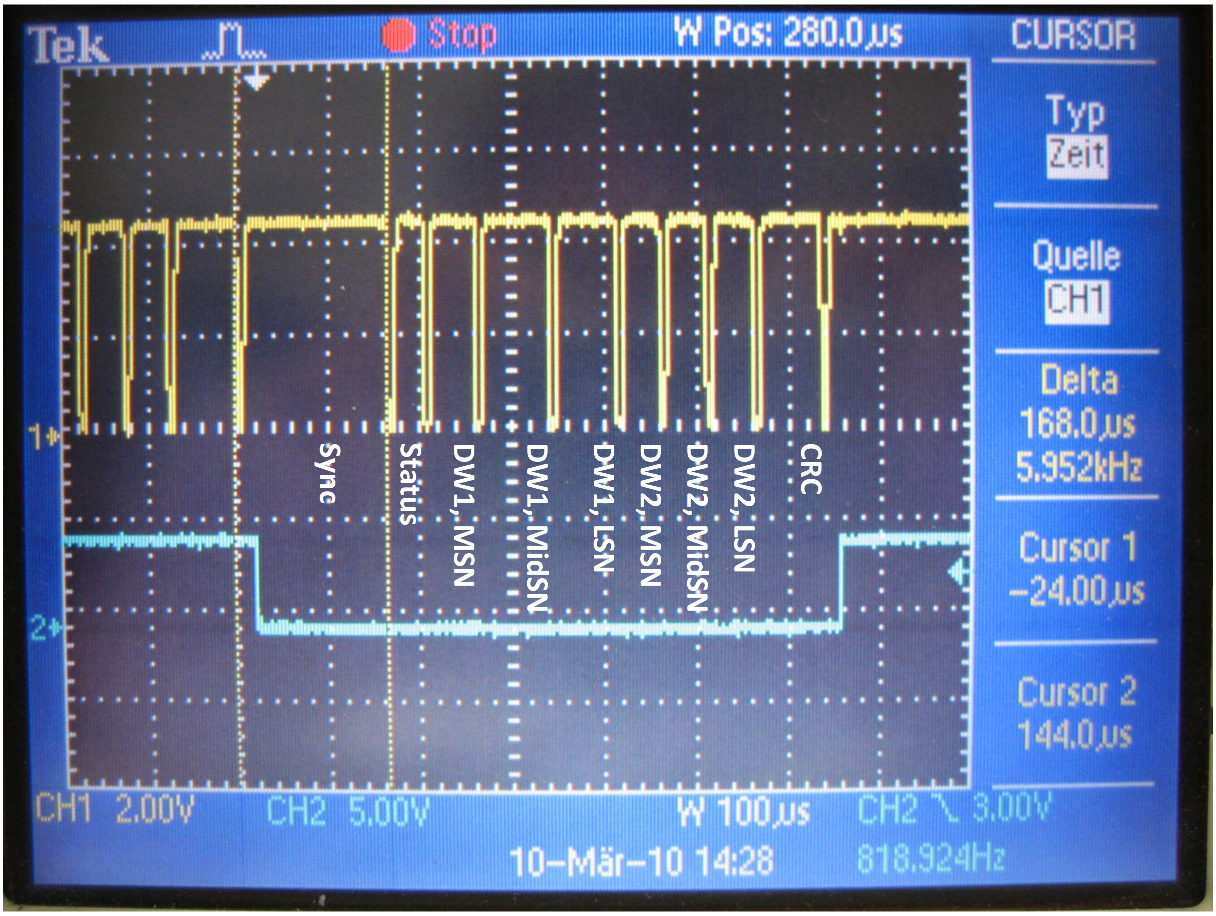

Image2: A dataframe on the screen.

The intervals between two negative edges are as follows:
The factory setting of TLE4998S for one LSB-time is 3µs. The interval for one nibble consists of an offset time of 36µs + the value of nibble x 3µs.

The time durations for one nibble range from 36µs for data value 0 up to 36µs + 3µs x 15 = 81µs for data value 15.

The syncfield is followed by a total of nibbles: a status nibble, 6 data nibbles and a CRC nibble.

The 6 data nibbles are assembled to two 12-bit wide data words, DW1 and DW2. The transmission uses Most Significant Nibble first.
TLE4998S uses the first 4 data nibbles to code the value of the magnetic field. The last two data nibbles are used for the temperature value.

Timer Selection

One can see from the above that 1LSB of a data nibble is equivalent to a duration of 3µs. The longest interval that we can expect between two negative edges is168µs x 125% = 210µs. This already includes a tolerance margin of +25%.

What we need to measure this is a simple free running 8-bit timer with capture functionality and sensitivity on negative edges.

For our lab experiment we use an 8-bit microcontroller from STMicroelectronics. With the TIM1 the STM8 features a 16-bit timer with capture register whose input pin is located comfortably near the power supply pins of header CN2 of the STM8S-Discovery board. All other timers TIM2, 3 or 5 are suited equally well.

To demonstrate that SENT can be decoded with 8-bit wide timers we only use the lower byte of the capture register of the 16-bit timer TIM1.

We set the prescaler of TIM1 in a way that the clock period is one third of the LSB time, that is ⅓ x 3µs = 1µs. The input clock frequency of the timer is now 1MHz.

When a pulse width is measured with a timer whose clock is asynchronous to the pulse then a systematic error of +/- 1fclk has to be considered. With the timerclock fclk = 1MHz we can expect the following intervals for the values of nibbles:

		Result Timer Capture:	
		clocks min	clocks max
Value Nibble	**Interval in µs**		
0	36	35	37
1	39	38	40
2	42	41	43
3	45	44	46
4	48	47	49
5	51	50	52
6	54	53	55
7	57	56	58
8	60	59	61
9	63	62	64
10	66	65	67
11	69	68	70
12	72	71	73
13	75	74	76
14	78	77	79
15	81	80	82

Timing Reference

Similar to LIN the data bytes are preceeded by a syncfield. With the TLE4998S the syncfield is 168µs long. The duration of this syncfield is measured and used as a reference for the bit-timing of the following data nibbles.

Calculating data Nibbles

The syncfield is followed by 8 data nibbles. The measured duration of a data field is corrected by the reference time of the sync field and then used as an index to a table with the nibble values.

```
dauer = ((u16)(168*dauer + (dauer_sync/2)))/((u16)dauer_sync) - 35;
```

In an 8-bit microcontroller divisions can take quite some processing time. To avoid one division we spread the lookup table by repeating the nibble values three times.

```
uc8 TLookup[] = {0,0,0, 1,1,1, 2,2,2, 3,3,3, 4,4,4, 5,5,5, 6,6,6, 7,7,7,
            8,8,8, 9,9,9, 10,10,10, 11,11,11, 12,12,12, 13,13,13, 14,14,14,
15,15,15};
```

Now the variable „dauer" can be used as an index to the TLookup table with 48 entries. The value of a nibble computes to

```
nibble[nibble_counter++] = *(TLookup+dauer);.
```

Cyclic Redundancy Check

After reception oft he data nibbles a CRC is calculated and compared to the expected value. We used the original algorithm from the TLE4998S manual. In case of a CRC error the LED on the STM8S-Discovery is lit as fault indication.

Received Data

The received data is stored in the 8 byte long array nibble[] and then transfered tot he global variables Datenwort1 and Datenwort2.
Damit Datenkonsistenz gewährleistet ist muss ein Hauptprogramm Interrupts für die Dauer des Lesens dieser beiden Werte sperren.

Conclusion

Decoding SENT is feasible with an 8-bit microcontroller. Essential peripheral is a free running 8-bit timer with input capture feature.

Literature

TLE4998S3 TLE4998S4 Datasheet V1.0, July 2008
Infineon Applikationsschrift AP1615410, SENT Decoder for XC2000
http://www.micronas.com/en/automotive_and_industrial_products/by_function/hal_2830/product_information/index.html

Anex

Listing of Initialisation and Timer Capture Interrupt Service Routine

```
#include <main.h>
#include <SENT.h>
#include <stm8s_TIM1.h>
#include <stm8s_gpio.h>

#define warten_auf_sync 0
#define nibble_capture 1

// public variables
volatile u16 Datenwort1, Datenwort2;      // die empfangenden Datenworte
u8 feld[128];

// private variables
u8 nibble[8];                    // status, 2 x nibbles 1..3, crc
u8 nibble_counter;
```

```
u8 CheckSum, i;

uc8 CRCLookup[16] = {0, 13, 7, 10, 14, 3, 9, 4, 1, 12, 6, 11, 15, 2, 8, 5};
uc8 TLookup[] = {0,0,0, 1,1,1, 2,2,2, 3,3,3, 4,4,4, 5,5,5, 6,6,6, 7,7,7,
               8,8,8, 9,9,9, 10,10,10, 11,11,11, 12,12,12, 13,13,13, 14,14,14, 15,15,15};


void SENT_init(void)
{
  u8 i;

  nibble_counter = 0;

  // initialisiere Datenfeld zu Null
  for (i=0; i<8; i++)
  {
    nibble[i] = 0;
  }

  GPIO_Init(GPIOC, GPIO_PIN_1, GPIO_MODE_IN_PU_NO_IT); // Port C1 ist inp capt.

  TIM1->PSCRL = 15;          // 16 / 16 MHz = 1MHz Timer Takt
  TIM1->CCMR1 = 0x11;   // filtering and TI1FP1 as trigger input

  TIM1->CCER1 = 0x02;   // falling edge, CC1 disable
  TIM1->CCER1 = 0x03;   // falling edge, CC1 enable

  TIM1->IER   = 0x02;   // IRQ durch CC1
  TIM1->CR1   = 0x01;   // Counter Enable
}

// Die Capture Interrupt Service Routine
@far @interrupt void SENT_Timer_CC1(void)       // Negative Edge Detect
{
  u8 dauer;                  // dauer des aktuellen Intervalls
  static u8 dauer_sync;      // dauer des Sync Intervalls
  static u8 vorheriger = 0; // vorheriger Capture Wert
  static u8 sent_status = warten_auf_sync;
  static u8 index = 0;

  // Zeitdifferenz zwischen zwei negativen Flanken errechnenn
  dauer = TIM1->CCR1L - vorheriger;    // GetCapture CC1 Low Byte, clears Interrupt flag
  vorheriger = TIM1->CCR1L;

  switch (sent_status)
  {
    case warten_auf_sync:
    {
      // Syncpuls empfangen ? Auf Syncpulslänge 168µs testen
      if ((dauer > (u8)(0.75*168)) && (dauer < (u8)(1.25*168)))  // Sync length in µs +/- 25%
      // Ja, Syncpuls empfangen
      {
        // Datenzähler und Fehlerspeicher zurücksetzen
        dauer_sync = dauer;
        nibble_counter = 0;
        sent_status = nibble_capture;
      }
    }
    break;

    // Nein, kein Syncpuls sondern Datennibble
    case nibble_capture:
    {
      // gemessenen Wert skalieren und Versatz abziehen
      dauer = ((u16)(168*dauer + (dauer_sync/2)))/((u16)dauer_sync) - 35; // [dauer] = 1µs

      if (dauer <= 47)
      {
        nibble[nibble_counter++] = *(TLookup+dauer);
      }
      else
      {
        sent_status = warten_auf_sync;
        GPIO_WriteLow(GPIOD, LED1_PIN);             // LED an
        break;
      }
```

```
        // CRC Calculation and Check, Datenexport
        if (nibble_counter >= 8)
        {
          CheckSum = 5;
          nibble_counter = 0;
          sent_status = warten_auf_sync;

          for (i=0; i<7; i++)
          {
            CheckSum = CheckSum ^ nibble[i];
            CheckSum = CRCLookup[CheckSum];
          }

          if (CheckSum == nibble[7]) // errechnete Checksumme = übertragene Checksumme ?
          {
            Datenwort1 = ((u16)nibble[1])<<8 | (u16)(nibble[2]<<4 | nibble[3]);
            Datenwort2 = ((u16)nibble[4])<<8 | (u16)(nibble[5]<<4 | nibble[6]);
            GPIO_WriteHigh(GPIOD, LED1_PIN);  // LED aus
          }
          else
          {
            GPIO_WriteLow(GPIOD, LED1_PIN); // LED an, Fehlermeldung
          }
         }
      } // of case (nibbel_capture)
    } // of switch (sent_status)
}
```