life.augmented

AN6024

Application note

SR5 E1 line–SENT emulation via TIM

Introduction

The single edge nibble transmission (SENT, SAE J2716) protocol is targeted for use in those applications where high-resolution data needs to be transmitted from a sensor to an engine control unit. It is intended as a replacement for the lower resolution methods as conventional sensors providing analog output voltage and PWM and as a simpler low-cost alternative to CAN, LIN or PSI5.

Applications for throttle position sensing, mass airflow sensing, pressure sensing, temperature sensing, and so on, can be used as examples of automotive applications for SENT-compatible sensor devices.

This document applies to the SR5E1x 32-bit Arm® Cortex®-M7 microcontrollers. Although the SENT peripheral is not present on the device, a SENT emulation driver can be implemented via built-in timers (TIMx) peripherals. This application note describes the implementation, APIs, and usage of the SENT emulation driver provided in the StellarE SDK.

1 SENT encoding scheme

The SENT encoding scheme is a unidirectional communication scheme (from the sensor/transmitting device to the controller/receiving device) which does not include a coordination signal from the controller/receiver device. It occurs independently of any action of the receiver module and does not require any synchronization signal from the receiver module. The signal transmitted by the sensor consists of a series of pulses with data encoded as falling to falling edge periods. The datum is transmitted in units (nibbles) of 4 bits for which the interval between two falling edges (single edge) of the modulated signal with a constant amplitude voltage is evaluated and represents values from 0 to 15. The SENT message transmission time depends on the sensor characteristics: tick time (frequency), transmitted data value, presence of a pause pulse.

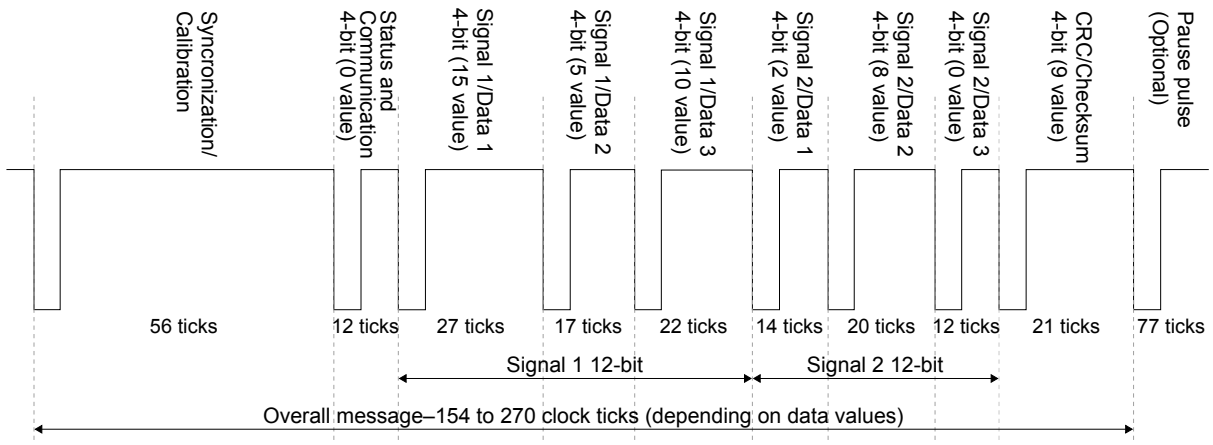
A consecutive SENT transmission starts immediately after the previous transmission ends (the trailing falling edge of the SENT transmission CRC nibble is also the leading falling edge of the consecutive SENT transmission synchronization/calibration pulse).

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

The transmission sequence consists of the following pulses (all times nominal):

- Synchronization/calibration pulse (56 clock ticks)
- One 4-bit status and serial communication nibble pulse (12 to 27 clock ticks)
- A sequence of one up to six data nibble pulses (12 to 27 clock ticks each) representing the values of the signal(s) to be communicated. The number of nibbles will be fixed for each application of the encoding scheme (that is, throttle position sensors, mass air flow, etc.) but can vary between applications.
- One 4-bit checksum nibble pulse (12 to 27 clock ticks)
- One optional pause pulse

Figure 1. Example encoding scheme for two 12-bit signals



Minimum nibble period = 36 µs at 3 µs clock tick
Nibble encoded period = 36 µs + x*(3 µs) (where x = 0, 1, ..., 15)

1.1 Synchronization/calibration pulse

The transmitter clock is not synchronized to the receiver clock. The standard allows deviation from the nominal clock frequency of ±20%. The data transmission format specifies a synchronization/calibration pulse of 56 ticks able to provide information on the actual transmitter (sensor) clock ticks period: careful measurement of this calibration pulse allows the receiver to reliably normalize and interpret nibble periods as data values.

The pulse starts with the falling edge and remains low for more than 4 clock ticks. The remaining clock ticks are driven high.

1.2 Status and communication nibble pulse

The status nibble contains 4 bits. The least significant bits 0 and 1 are reserved to enable the sensor to transmit miscellaneous information such as part numbers or error code information. Bits 2 and 3 define a transmitter optional serial message channel which can be implemented either as a short or an enhanced serial message format. The complete 16-bit short serial message is then transmitted in 16 consecutive SENT transmissions whereas the complete 36-bit enhanced serial message is then transmitted in 18 consecutive SENT transmissions. The width of the status nibble pulse is dependent on the nibble value. The status nibble pulse and data nibble pulse formats are identical.

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 will be fixed for each application of the encoding scheme. The width of the data nibble pulse is dependent on the nibble value.

1.4 Nibble pulse checksum

The checksum nibble contains a 4-bit CRC. The checksum is calculated using the x4 + x3 + x2 + 1 polynomial with the seed value of 5 (0b0101) and is calculated over all data nibbles. The communication nibble is not included in the CRC calculation whilst the status nibble can be included depending on the sensor characteristics.

The CRC allows detection of the following errors:

- All single bit errors.
- All odd number of errors.
- All single burst errors of length ≤ 4.
- 87.5% of single burst errors of length = 5.
- 93.75% of single burst errors of length > 5.

1.5 Pause pulse

The time for an individual message is a function of the data in the message and the transmitter clock rate only. If the nibble values are small, the message period is short. Conversely, when the values are large, the message period is long. This is a disadvantage for applications that desire constantly spaced data. To create a SENT transmission with a constant number of clock ticks, the SENT protocol includes an optional pause pulse, which acts as a filler between the checksum nibble and the next calibration pulse. If implemented, the pause pulse has the following properties:

- Minimum length 12 ticks (equivalent to a nibble with 0 value)
- Maximum length 768 ticks (3x256)

2 SENT emulation driver

The emulation driver supports up to eight driver instances and each instance supports four SENT channels. The received data can be managed using DMA and interrupts service routines.

2.1 Involved peripherals

The SENT emulation driver involves the following peripherals:

- TIM–Using the input capture mode of TIMs to catch the falling edges.
- DMA–Using DMA to copy the captured values from TIMs to a buffer in RAM.

2.1.1 TIM

According to the SENT protocol, a signal transmitted by the sensor consists of a series of pulses. The distance between consecutive falling edges defines the transmitted 4-bit data nibble, representing values from 0 to 15.

A transmitter specific nominal clock period (tick) is between 3 and 90 µs.

The input capture feature on the Stellar E1 device is supported through several timers: high-resolution timers (HRTIM), advanced-control timers (TIM1/TIM8), general-purpose timers (TIM2/TIM3/TIM4/TIM5, TIM15/TIM16).

Note: Basic timers (TIM6 and TIM7) do not support the input capture feature (only used as generic timers for time-base generation).

In input capture mode, when a capture event occurs, an interrupt or a DMA request can be invoked. The SENT emulation driver supports eight instances, based on TIM1, TIM2, TIM3, TIM4, TIM5, TIM8, TIM15, TIM16.

Input capture mode

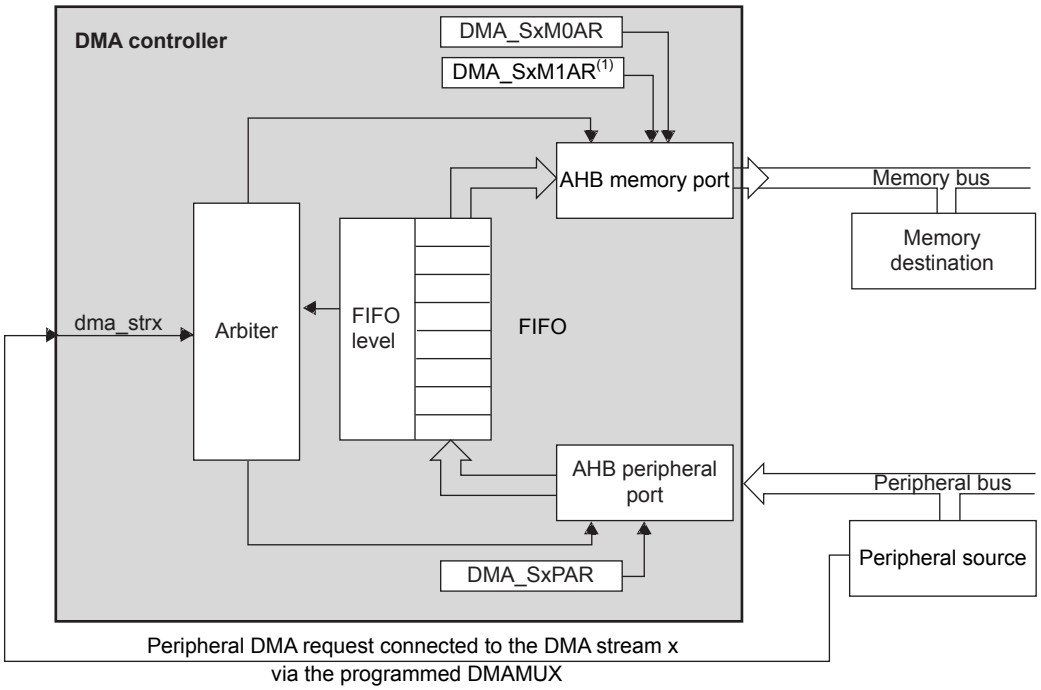
The SENT emulation driver configures the TIMx module in input capture mode. When the input capture occurs:

- The TIMx_CCR1 register gets the value of the counter on the active transition.
- The CC1F flag is set. CC1OF is also set if at least two consecutive captures occurred whereas the flag was not cleared.
- An interrupt is generated depending on the CC1IE bit status.
- A DMA request is generated depending on the CCIDE bit status.

2.1.2 DMA

DMA moves the captured count value from TIMx register TIMx_CCR1 to RAM. Stellar E1 DMA stream supports buffer management and pointer incrementation.

Figure 2. Peripheral-to-memory mode



1. For double-buffer mode.

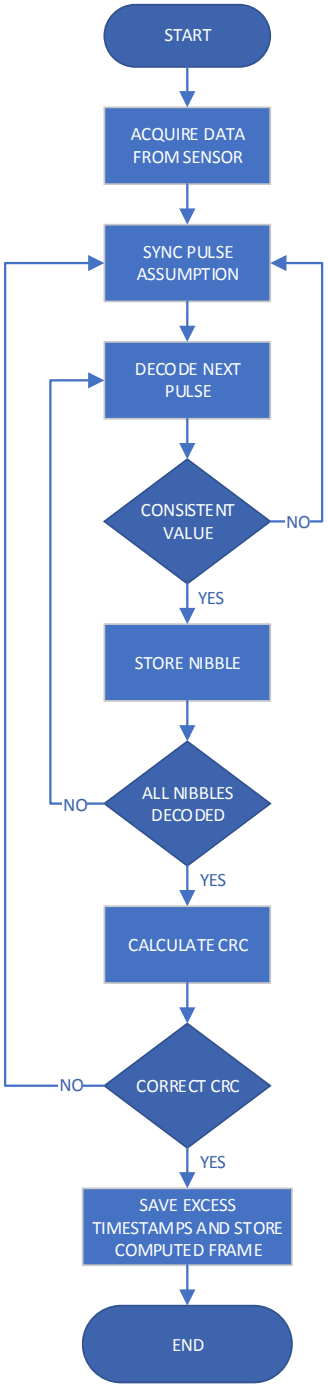
MSv41971V1

2.2 Algorithm description

Using the TIM peripheral in input capture mode a series of timestamps is acquired and transferred into a buffer making use of DMA configured as peripheral to memory with memory increment.

Since a timestamp is saved on each falling-edge of the external SENT sensor connected to the TIM, the difference between two consecutive timestamps constitutes a nibble. Based on this feature, the following algorithm has been implemented to decode the SENT sensor data.

Figure 3. **Flowchart of the algorithm**



Once started, the algorithm acquires the buffer containing 20 timestamps. The algorithm starts with a pulse assumption: the difference between the first two timestamps is assumed to be the sync pulse. Since the sync pulse is always composed of 56 ticks, dividing it by 56 gives the value of a single tick. On the next timestamp deltas, the tick value is used to check if the nibbles are consistent (12 to 27 ticks value). If an inconsistent nibble is found the nibbles are discarded and the sync pulse assumption is done on the following timestamps delta. As soon as the required number of nibbles is acquired, the CRC is checked to ensure the data is valid and the decoded frame is stored, ending the decoding process. If the buffer does not contain another frame the excess of timestamps is stored for the next iteration and the algorithm ends.

2.3 Software architecture

2.3.1 User level APIs

At application level, the driver can be configured calling the following APIs.

Table 1. sent_init()

Name	sent_init
Behavior	Initializes SENT instance.
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	**null**
Usage constraints	The SENT instances are defined in sent_intances.c. The parameter sdp could only be selected from DRV_SENT1~ 5, DRV_SENT8, DRV_SENT15~16.

Table 2. sent_get_tim()

Name	sent_get_tim
Behavior	Returns the TIM driver used by this SENT instance
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	**tim_driver_t** pointer to the TIM structure
Usage constraints	null

Table 3. sent_set_prio()

Name	sent_set_prio
Behavior	Configures SENT interrupt priority
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]uint32_t prio**: interrupt priority to be configured
Return values	Previous interrupt priority
Usage constraints	This configuration becomes effective only after sent_start

Table 4. sent_set_drv_mode()

Name	sent_set_drv_mode()
Behavior	Configures SENT driver mode (synchronous or asynchronous)
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]sent_drv_mode_t drv_mode**: driver mode to be configured. It can be one of the following values: SENT_DRV_MODE_SYNCHRONOUS, SENT_DRV_MODE_ASYNCHRONOUS
Return values	Previous driver mode
Usage constraints	This configuration becomes effective only after sent_start

Table 5. sent_set_freq()

Name	sent_set_freq()
Behavior	Configures frequency of TIM used for the SENT emulation
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]uint32_t freq**: TIM frequency [Hz] to be configured
Return values	Previous frequency
Usage constraints	This configuration becomes effective only after sent_start

Table 6. sent_enable_dma()

Name	sent_enable_dma()
Behavior	This configuration becomes effective after sent_start
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4 **[in]bool enable**: DMA mode enable status ('true' means the DMA is enabled, 'false' otherwise).
Return values	**bool**: previous DMA mode enable status.
Usage constraints	This configuration becomes effective after sent_start

Table 7. sent_set_dma_conf()

Name	sent_set_dma_conf()
Behavior	Configures channel DMA
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4, **[in]sent_dma_conf_t *dma_conf**: pointer to DMA configuration to be configured
Return values	**null**
Usage constraints	This configuration becomes effective only after sent_start

Table 8. sent_set_nibbles()

Name	sent_set_nibbles()
Behavior	Configures number of nibbles in received frame
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4, **[in]sent_nibbles_t nibbles**: number of nibbles into the frame. It can be one of the following values: • SENT_NIBBLES_6 • SENT_NIBBLES_8
Return values	**sent_nibbles_t**: previous number of nibbles. It can be one of the following values: • SENT_NIBBLES_6 • SENT_NIBBLES_8
Usage constraints	This configuration becomes effective only after *sent_start*.

Table 9. sent_set_order()

Name:	sent_set_order()
Behavior	Configures order of nibbles in a received frame
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4 **[in]ent_order_t order**: nibbles order into a frame. It can be one of the following values: • SENT_ORDER_LSB • SENT_ORDER_MSB
Return values	**sent_order_t**: previous order into a frame. It can be one of the following values: • SENT_ORDER_LSB • SENT_ORDER_MSB
Usage constraints	This configuration becomes effective only after *sent_start*

Table 10. sent_set_crc()

Name	sent_set_crc()
Behavior	Configures the CRC algorithm
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4 **[in]sent_crc_t crc_type**: CRC algorithm to use. It can be one of the following values: • SENT_CRC_DATA • SENT_CRC_STATUS_DATA
Return values	**sent_crc_t**: previous CRC algorithm. It can be one of the following values: • SENT_CRC_DATA • SENT_CRC_STATUS_DATA
Usage constraints	This configuration becomes effective only after *sent_start*.

Table 11. sent_set_cb()

Name	sent_set_cb()
Behavior	Configures pointers to SENT callback functions. This callback is invoked when a new SENT frame is received
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]sent_channel_t ch**: channel to be configured. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4, **[in]sent_cb_t callback**: pointer to SENT callback function
Return values	**sent_cb_t**: pointer to previous SENT callback function.
Usage constraints	This configuration becomes effective only after *sent_start*.

Table 12. sent_start()

Name	sent_start()
Behavior	Starts an SENT instance
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	**null**
Usage constraints	null

Table 13. sent_set_private()

Name	sent_set_private()
Behavior	Sets application private data pointer
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]void *priv**: pointer to application private data
Return values	**null**
Usage constraints	null

Table 14. sent_get_private()

Name	sent_get_private()
Behavior	Gets application private data pointer
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	Pointer to application private data
Usage constraints	null

Table 15. sent_start_channel()

Name	sent_start_channel()
Behavior	Starts acquisition on specified SENT channel
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel to be started. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4
Return values	**null**
Usage constraints	This function is the last called function before starting the SENT receiving frame. It will start TIM used for SENT emulation. User should not call this function before init and configuring the SENT

Table 16. sent_is_frame_available()

Name	sent_is_frame_available()
Behavior	Returns a flag to check if there are frames avaiable
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure. **[in]sent_channel_t ch**: channel from which to receive. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4
Return values	**bool**: availability flag
Usage constraints	null

Table 17. sent_receive()

Name	sent_receive()
Behavior	Receives a frame
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]sent_channel_t ch**: channel from which to receive. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4, **[out]sent_frame_t *frame**: received frame
Return values	**null**
Usage constraints	null

Table 18. sent_stop_channel()

Name	sent_stop_channel()
Behavior	Stops a SENT channel.
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure, **[in]sent_channel_t ch**: channel to be stopped. It can be one of the following values: SENT_CHANNEL_1~SENT_CHANNEL_4
Return values	**null**
Usage constraints	null

Table 19. **sent_stop()**

Name	sent_stop()
Behavior	Stops SENT instance
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	**null**
Usage constraints	null

Table 20. **sent_deinit()**

Name	sent_deinit()
Behavior	Deinitializes SENT instance
Source file	sent.c
Parameters	**[in]sent_driver_t *sdp**: pointer to a sent_driver_t structure
Return values	**null**
Usage constraints	null

2.3.2 Driver instance configuration

To use a SENT instance, the driver must be configured according to the following steps.

Initialize the driver instance:

```
sent_init(sdp);
```

Set the DMA configuration:

```
dma_conf.dma_stream_id = DMA1_STREAM1_ID;
dma_conf.dma_stream_bus_prio = DMA_PRIORITY_MAX;
dma_conf.dma_stream_irq_prio = IRQ_PRIORITY_10;
sent_set_dma_conf(sdp, SENT_CHANNEL_1, &dma_conf);
(void)sent_enable_dma(sdp, SENT_CHANNEL_1, true);
```

Set the number of valid nibbles in received frame:

```
(void)sent_set_nibbles(sdp, SENT_CHANNEL_1, SENT_NIBBLES_6);
```

Set crc algorithm:

```
(void)sent_set_crc(sdp, SENT_CHANNEL_1, SENT_CRC_DATA);
```

Set LSB/MSB order of valid nibbles in received frame:

```
(void)sent_set_order(sdp, SENT_CHANNEL_1, SENT_ORDER_MSB);
```

Initialize TIM driver instance:

```
tim_init(sent_get_tim(sdp));
```

Set the callback invoked when new frames are received:

```
(void)sent_set_cb(sdp, SENT_CHANNEL_1, rx_callback);
```

Apply the configuration:

```
sent_start(&SENTD);
```

Start the channel:

```
sent_start_channel(&SENTD, SENT_CHANNEL_1);
```

Note: the parameters may change according to the SENT sensor used or the user defined configuration.

2.3.3 Algorithm implementation

To get a better understanding of the algorithm it is useful to look at the actual implementation.

As described earlier, the decoding process starts when a timestamps buffer has been sampled. At this point, the decode function is invoked.

It is important to note that the *sent_decode_frame* function is placed in ITCM along with the *sent_crc_check* function to obtain the maximum performance in terms of execution time on the decoding phase.

```
static void __attribute__((section(".itcm")))
                    sent_decode_frame(sent_driver_t *sdp, sent_channel_t ch)

    bool frame_avail = false;
    uint8_t nibble_counter, nibble_data, ts_num, bit_num, i, j, len;
    uint8_t end_idx = 0U;
    uint16_t tick_time, tick_num;
    uint32_t tmp1, tmp2;
    sent_frame_t frame;

    /* Status + N nibbles + CRC => N + 3 timestamps.*/
    ts_num = (uint8_t)((uint8_t)sdp->nibbles[ch] + 3U);
    /* Number of data bits.*/
    bit_num = (uint8_t)((uint8_t)sdp->nibbles[ch] * 4U);
```

First of all, the new timestamps are chained with the remaining timestamps of the previous iteration.

```
/* Add the new data in the total buffer containing the excess of last
      iteration.*/
    for(i = 0U; i < SENT_TS_BUFF_SIZE; i++) {
        sdp->ts_tot_buff[ch][i + sdp->ts_excess_num[ch]] = sdp->ts_buff[ch][i];
    }

    /* Save buffer length.*/
    len = SENT_TS_BUFF_SIZE + sdp->ts_excess_num[ch];
    i = 0U;
```

To optimize the number of iterations, the algorithm stops if the sync pulse isn't followed by *ts_num* timestamps, which is the minimum number of timestamps required to compose a frame.

```
/* To have a complete frame the sync must be present
      before BUFFER SIZE - ts_num timestamps.*/
    while(i < (SENT_TS_BUFF_SIZE + sdp->ts_excess_num[ch] - ts_num)) {

        /* Initialize the variable where the values of the nibbles of each
           correct frame will be stored.*/
        frame.data = 0U;

        /* Initialize the nibble counter.*/
        nibble_counter = 0U;

        /* Initialize the variable where the value of a single nibble will be
           stored.*/
        nibble_data = 0U;
```

After an initialization phase it is assumed that the first timestamps delta is the sync pulse and the time of a single tick is calculated based on this assumption:

```
/* Sync pulse assumption.*/
        tmp1 = sdp->ts_tot_buff[ch][i];
        tmp2 = sdp->ts_tot_buff[ch][i + 1U];

        /* Increment index to take into account the timestamps taken.*/
        i++;

        /* Calculate single tick time + TIM overflow handling.*/
        tick_time = (uint16_t)((tmp2 > tmp1) ? (tmp2 - tmp1) / 56U :
                        ((tmp2 + SENT_TIM_INTERVAL) - tmp1) / 56U);
```

Exploring the following nibbles if the value is valid, it is stored updating the frame, otherwise the whole frame is discarded:

```
/* Explore the remaining ts_num timestamps.*/
        for(j = 1U; j < ts_num; j++) {

            tmp1 = tmp2;
            tmp2 = sdp->ts_tot_buff[ch][i + j];

            /* Calculate number of nibble ticks + TIM overflow handling.*/
            tick_num = (uint16_t)((tmp2 > tmp1) ? (tmp2 - tmp1) / tick_time :
                           ((tmp2 + SENT_TIM_INTERVAL) - tmp1) / tick_time);

            /* If the value is between 12 and 27, the nibble is validated.*/
            if ((tick_num >= 12U) && (tick_num <= 27U)) {
                /* Get the nibble data.*/
                nibble_data = (uint8_t)(tick_num - 12U);

                /* Update the frame data with the new nibble.*/
                if ( j == 1U) {
                    /* First nibble is status.*/
                    frame.attrib[SENT_FRAME_STATUS_IDX] = nibble_data;
                } else if(j == ts_num - 1U) {
                    /* Last nibble is crc.*/
                    frame.attrib[SENT_FRAME_CRC_IDX] = nibble_data;
                } else {
                    /* Other nibbles are data.*/
                    frame.data = (sdp->order[ch] == SENT_ORDER_MSB) ?
                            ((frame.data << 4U) | nibble_data) :
                            (frame.data | (((uint32_t)nibble_data) << 4U));
                }
                /* Updates the nibble counter.*/
                nibble_counter++;
            } else {
                /* Invalid nibble, wrong synch pulse assumpion.*/
                break;
            }
```

Once the frame is completed, the CRC check is called on it to assert that the transferred data is correct. In that case, it is added in the valid data FIFO:

```
/* Check if the frame has been completed.*/
            if (nibble_counter == (ts_num - 1U)) {
                /* End of used elements index.*/
                end_idx = i + j;
                i = end_idx;
                /* Invoke CRC function on the frame. If the CRC corresponds,
                   the frame will be validated.*/
                if (sent_crc_check(sdp, ch, frame, bit_num) == true) {
                    /* Save the frame (data, status, crc).*/
                    frame.attrib[SENT_FRAME_VALID_IDX] = SENT_FRAME_VALID;
                    frame.attrib[SENT_FRAME_CHANNEL_IDX] = 1U;
                    frame.timestamp = sdp->ts_tot_buff[ch][i];

                    /* Decoding end time.*/
                    GET_END_TIME(frame);
                    sdp->frames[ch][sdp->wr_ptr[ch]] = frame;
                    /* Increment write pointer on frame fifo.*/
                    sdp->wr_ptr[ch] = sent_inc_fifo_ptr(sdp->wr_ptr[ch]);
                    if (sdp->wr_ptr[ch] == sdp->rd_ptr[ch]) {
                        sdp->rd_ptr[ch] = sent_inc_fifo_ptr(sdp->rd_ptr[ch]);
                    }
                    /* A frame is available.*/
                    frame_avail = true;
                    /* Reset frame data.*/
                    frame.data = 0U;
                }
            }
        }
    }
```

When it is not possible to decode frames anymore, the excess timestamps are stored to be chained with the next iteration of the algorithm:

```
/* Save the number of excess elements.*/
    sdp->ts_excess_num[ch] = SENT_TS_BUFF_SIZE +
                             sdp->ts_excess_num[ch] -
                             end_idx;

    /* Max shift handling.*/
    if(sdp->ts_excess_num[ch] > (SENT_TOT_TS_BUFF_SIZE - SENT_TS_BUFF_SIZE)){
        sdp->ts_excess_num[ch] = (uint8_t)(SENT_TOT_TS_BUFF_SIZE -
                                           SENT_TS_BUFF_SIZE);
    }

    /* Shift the excess elements.*/
    for(i = 0U; i < sdp->ts_excess_num[ch]; i++){
        sdp->ts_tot_buff[ch][i] =
                        sdp->ts_tot_buff[ch][len + i - sdp->ts_excess_num[ch]];
    }

    SENT_RX_DONE(sdp);

If defined, the associated callback is invoked.

    /* Decoding process finished successfully. Invoke callback.*/
    if (((bool)frame_avail == true) && (sdp->callback[ch] != NULL)) {
        sdp->callback[ch](sdp, ch);
    }
}
```

Note: Due to the algorithm optimizations it is possible to achieve the best performances compiling the code with -O3 optimization.

Appendix A Reference documents

Table 21. **Reference documents**

Document name	Document title
RM0483	SR5E1x 32-bit Arm® Cortex®-M7 architecture microcontroller for electrical vehicle applications
TN1404	SR5E1E3, SR5E1E5, SR5E1E7 IO definition (signal description and input multiplexing tables) and device identification registers

Revision history

Table 22. **Document revision history**

Date	Revision	Changes
06-May-2024	1	Initial release.

Contents

1 **SENT encoding scheme** 2
1.1 Synchronization/calibration pulse 2
1.2 Status and communication nibble pulse 3
1.3 Data nibble pulse 3
1.4 Nibble pulse checksum 3
1.5 Pause pulse 3
2 **SENT emulation driver** 4
2.1 Involved peripherals 4
2.1.1 TIM 4
2.1.2 DMA 4
2.2 Algorithm description 5
2.3 Software architecture 7
2.3.1 User level APIs 7
2.3.2 Driver instance configuration 12
2.3.3 Algorithm implementation 13
Appendix A Reference documents 16
Revision history 17

List of tables

Table 1. sent_init() . . . 7
Table 2. sent_get_tim() . . . 7
Table 3. sent_set_prio() . . . 7
Table 4. sent_set_drv_mode() . . . 7
Table 5. sent_set_freq() . . . 8
Table 6. sent_enable_dma() . . . 8
Table 7. sent_set_dma_conf() . . . 8
Table 8. sent_set_nibbles() . . . 9
Table 9. sent_set_order() . . . 9
Table 10. sent_set_crc() . . . 9
Table 11. sent_set_cb() . . . 10
Table 12. sent_start() . . . 10
Table 13. sent_set_private() . . . 10
Table 14. sent_get_private() . . . 10
Table 15. sent_start_channel() . . . 11
Table 16. sent_is_frame_available() . . . 11
Table 17. sent_receive() . . . 11
Table 18. sent_stop_channel() . . . 11
Table 19. sent_stop() . . . 12
Table 20. sent_deinit() . . . 12
Table 21. Reference documents . . . 16
Table 22. Document revision history . . . 17

List of figures

Figure 1. Example encoding scheme for two 12-bit signals 2
Figure 2. Peripheral-to-memory mode 5
Figure 3. Flowchart of the algorithm 6