Microcontrollers

XC2000
SENT Decoder for XC2000

AP1615410
Application Note, v1.0, SEP. 2008

infineon
Never stop thinking
LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information
For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (www.infineon.com).

Warnings
Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.
# Table of Contents

1. **Scope** ..............................................................................................................................................5
2. **Preconditions** .................................................................................................................................6
3. **SENT Implementation on XC2287** ..................................................................................................7
   3.1 SENT receiver-channel state machine ............................................................................................7
   3.2 Single-channel requirement ............................................................................................................8
   3.3 Resource usage ...............................................................................................................................8
   3.4 Falling-edge detection .....................................................................................................................9
   3.5 Interrupt service routines ...............................................................................................................10
   3.6 Validation of calibration pulse (on falling-edge interrupt, "FIND SYNC" state) .........................11
   3.7 Buffering the nibbles ("GET DATA" state).....................................................................................12
   3.8 Analyze Data (on end of DMA/PEC transfer).............................................................................12
4. **Execution times** .............................................................................................................................14
   4.1.1 SENT Decoder (CPU@80MHz) ....................................................................................................14
5. **Configuration** .................................................................................................................................15
   5.1 Configurable parameters in sent_cfg.h ..........................................................................................15
   5.2 Configurable parameters in DAvE .................................................................................................16
   5.2.1 sent_cfg.c ......................................................................................................................................16
   5.2.2 DMA/PEC Channel Configuration ............................................................................................17
6. **Driver's API** .....................................................................................................................................20
   6.1 Global channels variable ...............................................................................................................20
   6.2 Bit mask of ui_StatusFlags ............................................................................................................21
   6.3 Functions prototypes in sent.h ........................................................................................................21
   6.3.1 Sent_ChannelInit ........................................................................................................................21
   6.3.2 Sent_CopyData .............................................................................................................................22
   6.4 Functions prototypes in CCU60.h .................................................................................................22
7. **Outlook** ...........................................................................................................................................22
1 Scope

Single Edge Nibble Transmission (SENT) is a promising low-cost solution for communication between off-ECU sensors and a microcontroller. This Application Note describes the decoding of a SENT signal using the CAPCOM 6 module.

This document will give you step by step instruction in order to install and operate the receiver, but also described in detail the configuration needed to realize such a function on example of XC2287 using an Easy Kit XC2287.

The following features are supported by the receiver:

- Reception and decoding of SENT compliant frames.
- Configurable SAE SENT or Infineon specific SENT protocol receiver
- Configurable number of data nibbles per frame.
- Serial data decoding.
- Error detection: signal loss, synchronization loss, clock drift, invalid CRC, invalid data nibble, serial data error.
- Full source code and DAvE configuration file available.

Note: Single Edge Nibble Transmission (SENT) refers to the SAE standard J2716. For more information, please visit www.sae.org.

Note: The receiver decodes logical SENT frames. Electrical characteristics as defined by the standard are not covered.

Note: The code delivered with this application note is aimed at development and demonstration purpose only. Neither is its quality nor its robustness guaranteed.
2 Preconditions

The following HW and SW are used:

- XC2287 Easy Kit (evaluation board) with connected sensor with SENT interface (e.g. TLE 4948)
- DAvE r2.1 and DIP file for XC2287
- Tasking toolset v8.7r1
- The SENT receiver source code and executable (AP1615410.exe).

The receiver software delivered with this application note is made of:

- C and Header files for the receiver.
- Executable files (.out and .hex)
- A Tasking project files for the toolchain v8.7r1

To download the executable to the Easy Kit you can use the integrated CrossViewPro debugger or any other debugger supporting XC2287.
3 SENT Implementation on XC2287

3.1 SENT receiver-channel state machine

After initialization, the channel will be in the “FIND SYNC” state. Any falling-edge event at the input pin will trigger an interrupt request “IsrNibble”. If valid synchronization (or calibration) signal has been received, it will switch to “GET DATA” state and turn-on the DMA unit to collect several nibbles (status, data and CRC) from the next incoming pulses. When all nibbles have been collected, the DMA will generate an interrupt request and the data can be analyzed. Note that on XC2287, at the end of PEC transfer interrupt is generated using the same interrupt frame as the normal one, i.e. “IsrNibble”.

Figure 1 SENT receiver channel state machine

Find Sync Frame and Serial Data Data Analysis
If there is no received pulse after specific timeout (larger than the maximum calibration pulse length\(^1\)), interrupt request will be generated and the channel switches back into “FIND SYNC” state. If it was from “GET DATA” state, DMA transfer will be stopped.

### 3.2 Single-channel requirement

Each SENT receiver channel requires the following resources:

- One self-reset timer per channel, timer directly gives pulse period when captured on falling-edge.
- One capture register per channel for input falling edge event
- One compare register per channel for timeout event
- One DMA/PEC channel for transferring captured timer into a temporary buffer
- Falling-edge event interrupt, or end-of-DMA transfer interrupt
- Timeout event interrupt

### 3.3 Resource usage

<table>
<thead>
<tr>
<th>Resource</th>
<th>Channel 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peripheral</td>
<td>CCU60</td>
</tr>
<tr>
<td>SENT input pin</td>
<td>P9.7 (pull-up)</td>
</tr>
<tr>
<td>Timer register</td>
<td>CCU60_T12R</td>
</tr>
<tr>
<td>Capture register</td>
<td>CCU60_CC60R</td>
</tr>
<tr>
<td>Compare register</td>
<td>CCU60_T12PR</td>
</tr>
<tr>
<td>DMA / PEC Channel(^2)</td>
<td>0</td>
</tr>
</tbody>
</table>

\(^1\) Nominal calibration pulse length is 56 x 3us = 168us. Maximum calibration pulse is nominal +25%, i.e. 210us.

\(^2\) DMA/PEC Channel number to use is configurable, see section 5.2.2 “DMA/PEC Channel Configuration”

Figure 2  SENT receiver channel 0 resource usage
3.4 Falling-edge detection

In order to detect falling-edge input on CCPOS0, CCU6x shall be set to “Hall-sensor mode” and the pattern is:

CURRS = '001'

EXPHS = '000'

Therefore, in file sent_ccu6x.h some registers are configured.

For channel 0, used in $CCU60\_vInit()$

```c
#define VAL_PISELL(ist12hr, ispos2, ispos1, ispos0, istrp, iscc62, iscc61, iscc60) \
   ((iscc60 << 0) | (iscc61 << 2) | (iscc62 << 4) | (istrp << 6) | (ispos0 << 8) | \
   (ispos1 << 10) | (ispos2 << 12) | (ist12hr << 14))

_inline void Sent_CCU6x_Init_0(void)
{
    Sent_PEC_Init_0();
    Sent_TimeoutDisable_0();

    /* Load CCU60 Port Input Selection register to select only CCPOSA(P9.7) 
     * and others to input zero */
    CCU60_PISELL = VAL_PISELL(0, 2, 2, 1, 3, 2, 2, 2);

    CCU60_MCMOUTS = 0x8800; // CURHS = '001', EXPHS = '000'
}
```
3.5 Interrupt service routines

The interrupt service routines API are called from corresponding interrupt vector/frame. These routines mainly evaluate channel state and performs necessary operations.

**Sent_IsrNibble**

```c
void Sent_IsrNibble(TSentChannel* Channel)
{
    if (Channel->ui_Rx_SENT_Status==SENT_STATE_FIND_SYNCH)
    {
        if (Sent_CheckCalibrationPulse(Channel)==TRUE)
        {
            Channel->ui_Rx_SENT_Status = SENT_STATE_GET_DATA;
        }
    }
    else if (Channel->ui_Rx_SENT_Status==SENT_STATE_GET_DATA)
    {
        // future state after exiting this interrupt context
        Channel->ui_Rx_SENT_Status=SENT_STATE_FIND_SYNCH;
        ...
        Sent_AnalyzeNibbles(Channel);
        ...
        if (Channel->cbDataReady)
        {
            Channel->cbDataReady(Channel->ub_ChannelNr);
        }
    }
}
```
Sent_IsrTimeout

void Sent_IsrTimeout(TSentChannel* Channel)
{
    ...
    if (Channel->ui_Rx_SENT_Status==SENT_STATE_GET_DATA)
    
    Sent_StopDMA(Channel);

    Sent.RaiseError(Channel, SENT_ERR_TIMEOUT);
    Channel->ui_Rx_SENT_Status = SENT_STATE_FIND_SYNCH;
    ...
}

3.6 Validation of calibration pulse (on falling-edge interrupt, “FIND SYNC” state)

Validation of calibration pulse is done by using both of these two criteria:

- Pulse-length is within expected nominal length (i.e. 168us) +/- 25%
- Difference (jitter) between successive calibration pulses is not above 1.5625% (1/64 time).

When valid calibration pulse is detected, the channel will store:

- the actual calibration pulse length for future jitter validation
- the ratio between captured and expected tick, example: ratio = CCU60_CC60R / 56

Sent_CheckCalibrationPulse

bool Sent_CheckCalibrationPulse(TSentChannel* Channel)
{
    uword ui_SCapture = (*Channel->CAPREG);
    ...
    uword ui_PulseLength = ui_SCapture;
    uword ActualTime = 0;
    ...

    if (Sent_IsValidCalibrationPulse(ui_PulseLength) == FALSE)
    {
        Sent.RaiseError(Channel, SENT_ERR_INVALID_CALIBRATION_PULSE);
        return FALSE;
    }
    else if ((Channel->Synchronized==TRUE)
        && (Sent_IsValidCalibrationJitter(Channel, ui_PulseLength)==FALSE))
    {
        Sent.RaiseError(Channel, SENT_ERR_INVALID_CALIBRATION_PULSE);
    }
return FALSE;
}
else
{
    Sent_StartDMA(Channel);
    ...
    Channel->MaxCalibrationJitter = Sent_CalcCalibrationPulseJitter(ui_PulseLength);
    Channel->Synchronized = TRUE;
    ...
    Channel->ui_TickRatio = SENT_TICK_RATIO_CALC(ui_PulseLength);
    Channel->LastCalibrationPulse = (sword)ui_PulseLength;
    return TRUE;
}
}

3.7 Buffering the nibbles (“GET DATA” state)

In order to offload the CPU that decodes the SENT frame, the nibbles are buffered by the PEC engine until a complete frame has been received. Then the PEC raise an interrupt to the processor using the same interrupt frame as the synchronization/calibration pulse interrupt (for example, see interrupt frame CCU60_viNodeI0 which then calls Sent_IsrNibble).

The PEC transfers the pulse length (in timer tick) from the capture register (e.g. CCU60_CC60R) into array-variable ui_Capture of TSentChannel structure.

typedef struct {
    ...
    ubyte ub_NumNibbles; // number of data nibbles of this channel
    ...
    uword ui_Capture[SENT_NIBBLES_MAX]; // nibble values, calculated
    ...
} TSentChannel;

The Sent_StartDMA and Sent_StopDMA code are used to start and stop accordingly the required DMA/PEC channel for this purpose.

3.8 Analyze Data (on end of DMA/PEC transfer)

At the end of PEC transfer interrupt request is generated using the same interrupt frame as the normal one, i.e. “IsrNibble”. Therefore, inside Sent_IsrNibble the state variable is checked, then if it was in “GET DATA” state:

- Every pulse length representing value of status, all data, and CRC are decoded and validated
- CRC is calculated and compared with the received CRC
- Channel’s status flag is updated, in case of error or new data.

Pulse decoding is done by using 16-bit fixed-point math operation.
#define ROUND_MASK (0x1 << (SENT_CONVERSION_Q-SENT_TICK_RATIO_Q-1))
ubyte Sent_DecodePulse(TSentChannel* Channel, uword ui_PulseLength)
{
    uword Value = (ui_PulseLength << SENT_CONVERSION_Q) / Channel->ui_TickRatio;
    #if SENT_CONVERSION_Q != 0
    Value += ((Value & ROUND_MASK) << 1);  // round-up if needed
    #endif
    Value = Value >> (SENT_CONVERSION_Q-SENT_TICK_RATIO_Q);
    if (!(Value >= 12 && Value <= 27))
        Sent_RaiseError(Channel, SENT_ERR_INVALID_NIBBLE);

    return (ubyte)(Value - 12); /* return nibble value represented by pulse length */
}

For example in sent_ccu6x.h:
#define SENT_CONVERSION_Q  3
#define SENT_TICK_RATIO_Q  0
when ui_PulseLength = 1141 (decimal) and pre-calculated Channel->ui_TickRatio = 60,

<table>
<thead>
<tr>
<th>initial value</th>
<th>0000010001110101</th>
<th>(binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>left shift (Q):</td>
<td>0010001101010000</td>
<td>(binary)</td>
</tr>
<tr>
<td>divide</td>
<td>0000000010011000</td>
<td>(binary)</td>
</tr>
<tr>
<td>(bold bit is the masked bit for rounding):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add:</td>
<td>0000000010011000</td>
<td>(binary)</td>
</tr>
<tr>
<td>right-shift (Q - TICK_RATIO_Q) :</td>
<td>0000000000010011</td>
<td>(binary)</td>
</tr>
<tr>
<td>end value</td>
<td>19</td>
<td>(decimal)</td>
</tr>
<tr>
<td>decoded value (subtract by 12)</td>
<td>9</td>
<td>(decimal)</td>
</tr>
</tbody>
</table>

In comparison to floating point operation, 1141 / 60 = 19.02 => 19 (rounded).
4 Execution times

On normal operation and serial frame decoding is disabled, for each channel the receiver gives following figures:

4.1.1 SENT Decoder (CPU@80MHz)

<table>
<thead>
<tr>
<th>Frame time (us)</th>
<th>Execution time without callback function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find sync</td>
<td>2.4us</td>
</tr>
<tr>
<td>Analyze/decoding data</td>
<td>9.8us</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td><strong>12.2us</strong></td>
</tr>
</tbody>
</table>

- **Average case**
  - 642.0 us
  - 1.9%

- **Worst case**
  - 364.8 us
  - 3.3%

- **Best case**
  - 993.6 us
  - 1.3%

---

3 Having a nominal frame tick of 3us, average frame length 214 ticks
4 Having a nominal frame tick of 3us - 20% = 2.4us, min frame length 152 ticks
5 Having a nominal frame tick of 3us + 20% = 3.6us, max frame length 276 ticks
5 Configuration

5.1 Configurable parameters in sent_cfg.h

The settings that can be modified by the user are “defines” located in the file “sent_cfg.h”. After a parameter has been modified, the program should be recompiled.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENT_IFX_CRC</td>
<td>1: use IFX CRC</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0: use original SENT CRC</td>
<td></td>
</tr>
<tr>
<td>SENT_SERIAL_DATA_ENABLED</td>
<td>1: enable serial data decoding</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0: disable serial data decoding</td>
<td></td>
</tr>
<tr>
<td>SENT_CHKSUM_STATUS</td>
<td>1: enable code to include status nibble in CRC calculation</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0: disable feature</td>
<td></td>
</tr>
<tr>
<td>SENT_SERIAL_FRAME_LENGTH</td>
<td>Number of bit in the serial frame ( ).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Only used if SENT_SERIAL_DATA_ENABLED != 0</td>
<td>16</td>
</tr>
<tr>
<td>SENT_DEVELOPMENT_DEBUG</td>
<td>If defined, debug information is given</td>
<td>(defined)</td>
</tr>
<tr>
<td>OPTIMIZE_INLINE_SENT</td>
<td>If set to _inline the driver is optimized using inline function else should be defined empty</td>
<td>_inline</td>
</tr>
<tr>
<td>SENT_CALLBACK_ENABLED</td>
<td>1: enable callback function</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0: disable callback function</td>
<td></td>
</tr>
<tr>
<td>SENT_PERIPHERAL</td>
<td>In this implementation, always USE_CCU6</td>
<td>USE_CCU6</td>
</tr>
<tr>
<td>SENT_CHANNELS_MAX</td>
<td>1 or 2 channels receiver will be implemented.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 channels implementation only uses CCU60.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 channels implementation uses CCU60 and CCU62</td>
<td>2</td>
</tr>
<tr>
<td>SENT_NIBBLES_MAX</td>
<td>Maximum nibbles to handle: status, data, and CRC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>equiv.: number of data nibbles + two</td>
<td>8</td>
</tr>
<tr>
<td>SENT_SPC_PERIOD_SYNCH</td>
<td>Only valid for SENT_PROTOCOL_SPC != 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Period of synchronization signal in microsecond.</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>Minimum period is FRAME_TICK x 3us x 125%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>where:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRAME_TICK =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56 // tick of calibration pulse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ (27 x SENT_NIBBLES_MAX) // status, data, crc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ 57 // idle nibble</td>
<td></td>
</tr>
<tr>
<td></td>
<td>with SENT_NIBBLES_MAX = 8, minimum is 1326us</td>
<td></td>
</tr>
</tbody>
</table>
### Configuration

<table>
<thead>
<tr>
<th>SENT_SPC_LOW_TIME</th>
<th>Only valid for SENT_PROTOCOL_SPC != 0</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time in microsecond when the SPC line will be driven low.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>By default, as per SENT protocol, 5 ticks =&gt; 15 us</td>
<td></td>
</tr>
<tr>
<td></td>
<td>By default, as per SPC protocol, it can be lower, i.e. 3 ticks =&gt; 9us</td>
<td></td>
</tr>
</tbody>
</table>

Other settings can be done during the driver initialization with the Sent_ChannelInit() function.

#### 5.2 Configurable parameters in DAvE

The DAvE will regenerate CCU60.c. Depend on how far the changes in the DAvE, sent_cfg.c will need to be modified too.

**5.2.1 sent_cfg.c**

By default, this file implements `TSentChannelConfig SentChannelsConfig[SENT_CHANNELS_MAX]`

*in sent.h*

```c
typedef struct {
    void (*cbDataReady)(ubyte Channel); // callback function upon data ready
    uword ub_NumNibbles;               // number of data nibbles, upto SENT_NIBLES_MAX-2
    ubyte ub_PEC_Nr;                   // DMA/PEC channel Nr.
    uword* CAPREG_Ptr;                // address of signal capture register
} TSentChannelConfig;
```

Example configuration:

```c
extern void cbSent0_DataReady(ubyte Channel); // in MAIN.c

TSentChannelConfig SentChannelsConfig[SENT_CHANNELS_MAX] =
{
    /* cbDataReady  ub_NumDataNibbles  ub_PEC_Nr  CAPREG_Ptr */
    {cbSent0_DataReady, 6, 0, (uword*)&CCU60_CC60R},
    {0, 6, 1, (uword*)&CCU62_CC60R},
};
```

The configuration of PEC channel number might be changed as described in the section 5.2.2 “DMA/PEC Channel Configuration”.

---

**AP1615410**

**SENT Decoder for XC2000**
5.2.2 DMA/PEC Channel Configuration

By default, CCU6x interrupts are organized as shown in the following picture. Interrupt nodes CCU60_I0_INT is handled by PEC channel 0 during the "GET DATA" state.

**Interrupt configuration:**

![Interrupt Controller (INT)](image)

<table>
<thead>
<tr>
<th>Level 15</th>
<th>Level 14</th>
<th>Level 13</th>
<th>Level 12</th>
<th>Level 11</th>
<th>Level 10</th>
<th>Level 9</th>
<th>Level 8 CCU60_I0_INT</th>
<th>CCU62_I0_INT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: To change the level and the group of an interrupt source, click on it, drag it to its new position and drop it. To set an interrupt source to the non interrupting level (Level 0) click on it, drag it to the Level 0 list and drop it.
PEC channel assignment:

<table>
<thead>
<tr>
<th>Configure PEC channel 0</th>
<th>Interrupt source for Group = 0 and Level = 14/12/10/8</th>
<th>CCU60 ID INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure PEC channel 1</td>
<td>Interrupt source for Group = 1 and Level = 14/12/10/8</td>
<td>CCU62 ID INT</td>
</tr>
<tr>
<td>Configure PEC channel 2</td>
<td>Interrupt source for Group = 2 and Level = 14/12/10/8</td>
<td>none</td>
</tr>
<tr>
<td>Configure PEC channel 3</td>
<td>Interrupt source for Group = 3 and Level = 14/12/10/8</td>
<td>none</td>
</tr>
<tr>
<td>Configure PEC channel 4</td>
<td>Interrupt source for Group = 0 and Level = 15/13/11/9</td>
<td>none</td>
</tr>
<tr>
<td>Configure PEC channel 5</td>
<td>Interrupt source for Group = 1 and Level = 15/13/11/9</td>
<td>none</td>
</tr>
<tr>
<td>Configure PEC channel 6</td>
<td>Interrupt source for Group = 2 and Level = 15/13/11/9</td>
<td>none</td>
</tr>
<tr>
<td>Configure PEC channel 7</td>
<td>Interrupt source for Group = 3 and Level = 15/13/11/9</td>
<td>none</td>
</tr>
</tbody>
</table>
Single PEC channel X configuration:

It is important and necessary to keep to these settings:

- “Decremented COUNT”
- “Increment destination pointer”
- “Transfer a word”
- SRCPx Segment = 0x00

Other settings will be filled by the driver:

- SRCPx Offset
- DSTPx Segment and Offset
6 Driver’s API

For the user (upper-layer), the driver provides following API:

6.1 Global channels variable

Definition in sent.h

typedef struct {
    /* processing results */
    volatile ubyte ui_Result[SENT_NIBBLES_MAX]; // received result nibbles:
    // [0]=>Status and Comm, [1..NumNibbles-1]=>Data,
    // [NumNibles]=>CRC
    volatile uword RSerialData; // received serial data
    uword RDataCounter; // received data counter

    /* information */
    volatile uword ui_StatusFlags; // driver status (sticky flag)
    uword ui_Error; // error counter
    ...
} TSentChannel;

extern TSentChannel SentChannels[SENT_CHANNELS_MAX]; // implemented in sent.c
6.2 Bit mask of ui_StatusFlags

These are defined as follows in sent.h, which can be used as mask to test the bit field.

```c
#define SENT_ERR_UNEXPECTED_CALIBRATION_PULSE     ( 0x0001)
#define SENT_ERR_INVALID_CALIBRATION_PULSE        ( 0x0002)
#define SENT_ERR_FRAME_TIMEOUT                    ( 0x0004)
#define SENT_ERR_INVALID_NIBBLE                   ( 0x0008)
#define SENT_ERR_DATA_CRC                         ( 0x0010)
#define SENT_ERR_UNEXPECTEDED_START_BIT           ( 0x0020)
#define SENT_ERR_SERIAL_DATA_CRC                  ( 0x0040)
#define SENT_ERR_TIMEOUT                           (0x0080)
#define SENT_NEW_DATA                              (0x0100)
#define SENT_ERR_FRAME_MISSING_CALIBRATION_PULSE   ( 0x0200)
#define SENT_NEW_SERIAL_DATA                       (0x0400)
#define SENT_NEW_ERROR                             (0x0800)
#define SENT_RAISE_INTERRUPT                      (0x1000)
#define SENT_ERR_MISSING_START_BIT                (0x2000)
```

6.3 Functions prototypes in sent.h

6.3.1 Sent_ChannelInit

**Note:**
This function shall be called before initialization of CCU6x (calling to vCCU60_Init()).

**Purpose:**
Initialization of channel variables.

Copying values from driver's global configuration SentChannelConfig.

**Prototype:**

```c
void Sent_ChannelInit(ubyte ChannelNr);
```

**Parameters:**

- ChannelNr Channel number to initialize
### 6.3.2 Sent_CopyData

**Purpose:**
To copy all the latest data nibbles. If in between the copy process there’s a new data, the copy data will be repeated.

**Prototype:**
```c
void Sent_CopyData(ubyte ChannelNr, ubyte* DestinationPtr);
```

**Parameters:**
- **ChannelNr**: Channel number
- **DestinationPtr**: Location the data nibbles will be copied into

### 6.4 Functions prototypes in CCU60.h

**Purpose:**
Initialization of the CCU6x peripherals

**Prototype:**
```c
void CCU60_vInit(void);
```

### 7 Outlook

This AppNote describes an implementation with one channel. In many applications two or more SENT interfaces are used. By using another CCU peripheral as well more channels can be implemented in parallel. We recommend to use CCU62 for the second channel.