Technical Specification Group Services and System Aspects
Meeting #11, Palm Springs, USA, 19-22 March 2001

TSGS#11(01)0108

Source: **TSG-SA WG4**

Title: **CR to Ts 26.230 on Bug fix in source code of the CTM receiver (Release 4)**

Document for: **Approval**

Agenda Item: **7.4.3**

The following CRs were agreed at the TSG-SA WG4 meetings #15 and are presented to TSG SA #11 for approval.

Spec	CR	Rev	Phase	Subject	Cat	Ver	WG	Meeting	S4 doc
26.230	001		Rel-4	Bug fix in source code of the CTM receiver	F	4.0.0	S4	TSG-SA WG4#15	S4-010060

3GPP TSG-SA4 Meeting #15
Munich, Germany, January 22-26, 2001

Tdoc S4-(01)0060

CR-Form-v3

CHANGE REQUEST

⌘ **26.230** **CR** **001** ⌘ rev **-** ⌘ Current version: **4.0.0** ⌘

*For **HELP** on using this form, see bottom of this page or look at the pop-up text over the ⌘ symbols.*

Proposed change affects: ⌘ (U)SIM ☐ ME/UE ☐ Radio Access Network ☐ Core Network ☐

Title:	⌘	Bug fix in source code of the CTM receiver.
Source:	⌘	TSG-SA WG4
Work item code:	⌘	Global Text Telephony / CTM
Date:	⌘	2001-01-23
Category:	⌘	**F**
Release:	⌘	REL-4

Use one of the following categories:
F *(essential correction)*
A *(corresponds to a correction in an earlier release)*
B *(Addition of feature),*
C *(Functional modification of feature)*
D *(Editorial modification)*
Detailed explanations of the above categories can be found in 3GPP TR 21.900.

Use one of the following releases:

2	*(GSM Phase 2)*
R96	*(Release 1996)*
R97	*(Release 1997)*
R98	*(Release 1998)*
R99	*(Release 1999)*
REL-4	*(Release 4)*
REL-5	*(Release 5)*

Reason for change:	⌘	Bug fix in the source code of the example implementation of the CTM receiver. This bug has been reported during the T1 ballot process of the related Technical Standard for T1 (Letter Ballot 924).
Summary of change:	⌘	Fix of a bug in the synchronization mechanism of the CTM receiver. This bug in the (non-mandatory) example implementation of the CTM receiver prevents the appropriate resynchronization of the CTM receiver, if the previous initial synchronization has been triggered by the resynchronization sequence rather than by the preamble. The change is documented in the following. It refers to the file “ctm_receiver.c” of the source code archive that is attached to specification 26.230.
Consequences if not approved:	⌘	The example implementation of the CTM receiver might fail to track the synchronism after a cell hand-over, if the initial preamble has been missed and the initial synchronization has been triggered by the detection of the resynchronization sequence.

Clauses affected:	⌘	This Change Request refers to the source code file “ctm_receiver.c” in the zip archive “ctm_src.zip”, which is attached to specification 26.230. The modifications are located in the header of the file as well as in the function ctm_receiver(), which are both listed in the following. Additionally, the updated file “ctm_receiver.c” is provided as an attachment to this Change Request.
Other specs affected:	⌘	☐ Other core specifications ⌘ ☐ Test specifications ☐ O&M Specifications
Other comments:	⌘	

```
*****************************************************************************
*
*      File             : ctm_receiver.c
*      Author           : EEDN/RV Matthias Doerbecker
*      Tested Platforms : Sun Solaris, MS Windows NT 4.0
*      Description      : Complete CTM Receiver including Demodulator,
*                         Synchronisation, Deinterleaver, and Error Correction
*
*      Changes since October 13, 2000:
*      - added reset function reset_ctm_receiver()
*
*      Changes since December 07, 2000:
*      - Bug fix within the code for initial synchronization
*        based on the detection of the resync sequence
*
*      $Log: $
*
*****************************************************************************
```

```
/*****************************************************************************/
/* ctm_receiver()                                                            */
/* **************                                                            */
/* Runs the Cellular Text Telephone Modem Receiver for a block of            */
/* (nominally) 160 samples. Due to the internal synchronization, the         */
/* number of processed samples might vary between 156 and 164 samples.       */
/* The input of the samples and the output of the decoded characters         */
/* is handled via fifo buffers, which have to be initialized                 */
/* externally before using this function (see fifo.h for details).           */
/*                                                                           */
/* input/output variables:                                                   */
/* *ptr_signal_fifo_state      fifo state for the input samples              */
/* *ptr_output_char_fifo_state fifo state for the output characters          */
/* *ptr_early_muting_required  returns whether the original audio signal     */
/*                             must not be forwarded. This is to guarantee   */
/*                             that the preamble or resync sequence is       */
/*                             detected only by the first CTM device, if     */
/*                             several CTM devices are cascaded              */
/*                             subsequently.                                 */
/* *rx_state                   pointer to the variable containing the        */
/*                             receiver states                               */
/*****************************************************************************/

void ctm_receiver(fifo_state_t*  ptr_signal_fifo_state,
                  fifo_state_t*  ptr_output_char_fifo_state,
                  Bool*          ptr_early_muting_required,
                  rx_state_t*    rx_state)
{
  Shortint  toneVec[SYMB_LEN+1];
  Shortint  bitsDemod[2];
  Shortint  cnt;
  Shortint  numValidBits;
  Bool      actual_sync_found;
  Bool      octetAvailable;
  Shortint  fecNetBit;
  UShortint utfOctet = 0;
  Shortint  ShortintValueTmp;
  Shortint  syncOffset;
  Shortint  resyncDetected;
  Shortint  wait_interval;
  Shortint  numViterbiOutBits;
  UShortint ucsCode  = 0;

  static Shortint  ucsBits[BITS_PER_SYMB];
  static Shortint  fecGrossBitsIn[CHC_RATE];

#ifdef DEBUG_OUTPUT
  static Bool      firsttime = true;
  static FILE      *rx_bits_file;
  if (firsttime)
```

```
    {
      firsttime = false;
      if ((rx_bits_file=fopen("rx_bits.srt", "wb"))==NULL)
        {
          fprintf(stderr, "Error while opening rx_bits.srt\n\n") ;
          exit(1);
        }
    }
#endif

  while (Shortint_fifo_check(ptr_signal_fifo_state)>SYMB_LEN)
    {
      /* Pop SYMB_LEN-1, SYMB_LEN, or SYMB_LEN+1 samples from fifo, */
      /* depending on the state of samplingCorrection.              */
      Shortint_fifo_pop(ptr_signal_fifo_state, toneVec,
                        SYMB_LEN+rx_state->samplingCorrection);

      /* Run the tone demodulator */
      tonedemod(bitsDemod, toneVec,
                (Shortint)(SYMB_LEN+rx_state->samplingCorrection),
                &(rx_state->samplingCorrection),
                &(rx_state->tonedemod_state));

#ifdef DEBUG_OUTPUT
      if (fwrite(bitsDemod, sizeof(Shortint), 2, rx_bits_file) == 0)
        {
          fprintf(stderr, "Error while writing to file\n\n") ;
          exit(1);
        }
#endif

      /* Find the synchronization sequence and run the */
      /* deinterleaver on the synchronized bitstream   */
      actual_sync_found =
        wait_for_sync(rx_state->waitSyncOut, bitsDemod, 2,
                      rx_state->cntIdleSymbols, &numValidBits,
                      &wait_interval, &resyncDetected,
                      ptr_early_muting_required,
                      &(rx_state->wait_state));

      if (actual_sync_found)
        {
#ifdef DEBUG_OUTPUT
          if (wait_interval==0)
            fprintf(stderr, ">>>Initial sync found<<<");
          else
            fprintf(stderr, ">>>Initial sync found (based on resync sequence)<<<");
#endif
          resyncDetected = -1;
          reinit_deinterleaver(&(rx_state->deintl_state));
          viterbi_reinit(&(rx_state->viterbi_state));
          rx_state->cntIdleSymbols = 0;
          rx_state->numDeintlBits = 0;
          rx_state->cntRXBits = 0;
          rx_state->syncCorrect = 0;

          /* Pop all remaining bits from fifos and forget the bits */
          Shortint_fifo_reset(&(rx_state->net_bits_fifo_state));
          Shortint_fifo_reset(&(rx_state->rx_bits_fifo_state));

          /* Calculate the deinterleaver's delay of                     */
          /* intlvB*deintSyncLns+intlvB*(intlvB-1)*intlvD elements      */
          /* plus an additional delay of RESYMC_SEQ_LENGTH, if the      */
          /* synchronization was triggered by detecting resync sequence */
          /* and not by detecting the preamble.                         */
          rx_state->intl_delay
            = intlvB*deintSyncLns+intlvB*(intlvB-1)*intlvD + wait_interval;
        }

      /*************************************************************/
      /*       The following lines are for resynchronisation       */
      /*         (also the deinterleaver is executed here)         */
      /*************************************************************/

      if (resyncDetected>=0)
        {
          syncOffset
```

```
            = (rx_state->numDeintlBits + resyncDetected + 1
               - rx_state->intl_delay~~(intlvB*deintSyncLns+intlvB*(intlvB-1)*intlvD)~~)
              % (NUM_BITS_BETWEEN_RESYNC+RESYNC_SEQ_LENGTH);

            if (syncOffset
                > (NUM_BITS_BETWEEN_RESYNC+RESYNC_SEQ_LENGTH)/2)
              syncOffset = syncOffset
                - (NUM_BITS_BETWEEN_RESYNC+RESYNC_SEQ_LENGTH);

            if ((syncOffset>-16) && (syncOffset<16))
              rx_state->syncCorrect = syncOffset;
            else
              rx_state->syncCorrect = 0;

#ifdef DEBUG_OUTPUT
            if (rx_state->syncCorrect!=0)
              fprintf(stderr, ">>resync detected: %d<<", syncOffset);
#endif
          }

        if (rx_state->syncCorrect==0)
          {
            /* The synchronisation is still ok. --> no adaptation */
            diag_deinterleaver(rx_state->deintlOut, rx_state->waitSyncOut,
                               numValidBits, &(rx_state->deintl_state));
          }
        else if (rx_state->syncCorrect>0)
          {
            /* The ResyncSecquence was too late */
            /* --> some bits have to be dropped */
            if (rx_state->syncCorrect>=numValidBits)
              {
                /* The incoming bits are only inserted into the deinterleaver, */
                /* no bits are handled to the following functions              */
                shift_deinterleaver((Shortint)(-numValidBits),
                                    rx_state->waitSyncOut,
                                    &(rx_state->deintl_state));
                rx_state->syncCorrect -= numValidBits;
                numValidBits=0;
              }
            else
              {
                shift_deinterleaver((Shortint)(-rx_state->syncCorrect),
                                    rx_state->waitSyncOut,
                                    &(rx_state->deintl_state));

                numValidBits -= rx_state->syncCorrect;
                diag_deinterleaver(rx_state->deintlOut,
                                   &(rx_state->waitSyncOut[rx_state->syncCorrect]),
                                   numValidBits, &(rx_state->deintl_state));
                rx_state->syncCorrect=0;
              }
          }
        else
          {
            /* The ResyncSequence was too early        */
            /* --> additional Bits have to be inserted */
            ShortintValueTmp=0;
            for (cnt=0; cnt<-(rx_state->syncCorrect); cnt++)
              {
                diag_deinterleaver(&(rx_state->deintlOut[cnt]),
                                   &ShortintValueTmp,
                                   1, &(rx_state->deintl_state));
                shift_deinterleaver(1, &ShortintValueTmp,
                                    &(rx_state->deintl_state));
              }
            diag_deinterleaver(&(rx_state->deintlOut[-(rx_state->syncCorrect)]),
                               rx_state->waitSyncOut,
                               numValidBits, &(rx_state->deintl_state));
            numValidBits += -(rx_state->syncCorrect);
            rx_state->syncCorrect=0;
          }

        /*************************************************************/
        /*              End of resynchronisation                     */
        /*************************************************************/
```

```
    /* Consider the deinterleaver's delay                          */
    /* and push the demodulated bits into the fifo buffer          */
    for (cnt=0; cnt<numValidBits; cnt++)
      {
        if (rx_state->numDeintlBits >= rx_state->intl_delay)
          {
            /* Ignore all bits that are for muting or resync */
            if (!(mutingRequired((Shortint)(rx_state->cntRXBits),
                                 rx_state->mutePositions,
                                 NUM_MUTE_ROWS*intlvB))
                && (rx_state->cntRXBits<NUM_BITS_BETWEEN_RESYNC))
              {
                Shortint_fifo_push(&(rx_state->rx_bits_fifo_state),
                                   &(rx_state->deintlOut[cnt]), 1);
              }

            rx_state->cntRXBits++;
            if(rx_state->cntRXBits
               ==NUM_BITS_BETWEEN_RESYNC+RESYNC_SEQ_LENGTH)
              rx_state->cntRXBits = 0;
          }
        rx_state->numDeintlBits++;

        /* Avoid Overflows of numDeintlBits */
        if (rx_state->numDeintlBits > 10000)
          rx_state->numDeintlBits
            -= (NUM_BITS_BETWEEN_RESYNC+RESYNC_SEQ_LENGTH);
      }
  }

/* As long as there are gross bits in the fifo: pop them, run  */
/* the channel decoder and pop the net bits into the next fifo */
while (Shortint_fifo_check(&(rx_state->rx_bits_fifo_state)) > CHC_RATE)
  {
    Shortint_fifo_pop(&(rx_state->rx_bits_fifo_state),
                      fecGrossBitsIn, CHC_RATE);

    /* Count gross bits with low reliability (i.e. bits with too low */
    /* magnitute or with their LSB not set).                         */
    for (cnt=0;cnt<CHC_RATE; cnt++)
      if ((abs(fecGrossBitsIn[cnt])<THRESHOLD_RELIABILITY_FOR_GOING_OFFLINE)
          || ((fecGrossBitsIn[cnt] & 0x0001) == 0))
        {
          if (rx_state->cntUnreliableGrossBits < maxShortint)
            rx_state->cntUnreliableGrossBits++;
        }
      else
        rx_state->cntUnreliableGrossBits=0;

    /* Channel decoder */
    viterbi_exec(fecGrossBitsIn, 1*CHC_RATE,
                 &fecNetBit, &numViterbiOutBits,
                 &(rx_state->viterbi_state));
    if (numViterbiOutBits > 0)
      {
        Shortint_fifo_push(&(rx_state->net_bits_fifo_state), &fecNetBit, 1);
      }
  }

octetAvailable = false;

/* As long as there are bits on the fifo: pop them and decode them into */
/* octets until a valid octet (i.e. not an idle symbol) is received     */
while (Shortint_fifo_check(&(rx_state->net_bits_fifo_state))>=BITS_PER_SYMB)
  {
    Shortint_fifo_pop(&(rx_state->net_bits_fifo_state),
                      ucsBits, BITS_PER_SYMB);

    utfOctet = 0;
    octetAvailable = true;
    for (cnt=0; cnt<BITS_PER_SYMB; cnt++)
      {
        if (ucsBits[cnt]>0)
          utfOctet += (1<<cnt);
      }
    // fprintf(stderr, " ((%d)) ", utfOctet);
```

```
      /* Decide, whether received octet is an idle symbol */
      if (utfOctet==IDLE_SYMB)
        {
          octetAvailable = false;
          rx_state->cntIdleSymbols++;
        }

      /* If more than MAX_IDLE_SYMB have been received or if more than */
      /* MAX_NUM_UNRELIABLE_GROSS_BITS gross bits with low reliability */
      /* have been received, assume that synchronization is lost and   */
      /* reset the wait_for_sync function.                             */
      if ((rx_state->cntIdleSymbols>= MAX_IDLE_SYMB) ||
          (rx_state->cntUnreliableGrossBits>MAX_NUM_UNRELIABLE_GROSS_BITS))
        {
          reinit_wait_for_sync(&(rx_state->wait_state));
          reinit_deinterleaver(&(rx_state->deintl_state));
          viterbi_reinit(&(rx_state->viterbi_state));
          rx_state->cntIdleSymbols = 0;
          rx_state->numDeintlBits = 0;

          /* pop all remaining bits from fifos and forget the bits */
          Shortint_fifo_reset(&(rx_state->net_bits_fifo_state));
          Shortint_fifo_reset(&(rx_state->rx_bits_fifo_state));

          octetAvailable = false;
#ifdef DEBUG_OUTPUT
          fprintf(stderr, ">>receiver offline<<");
#endif
        }

      /* If octet available -> push it into octet fifo buffer */
      if (octetAvailable)
        {
          Shortint_fifo_push(&(rx_state->octet_fifo_state), &utfOctet, 1);
          rx_state->cntIdleSymbols=0; /* reset counter for idle symbols */
        }

      /* Try to convert octets from buffer into UCS code.        */
      /* If successful, push decoded UCS code into output buffer */
      if (transformUTF2UCS(&ucsCode, &(rx_state->octet_fifo_state)))
        Shortint_fifo_push(ptr_output_char_fifo_state, &ucsCode, 1);
    }
}
```