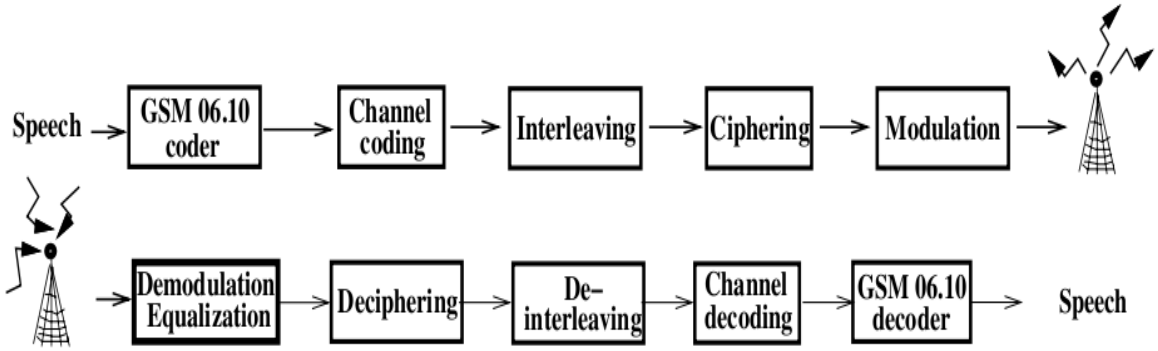
Chapter 9 Xoring Stage

Algorithm A5 is implemented into both the MS and the BSS. The A5 algorithm is implemented for each physical channel (TCH or DCCH). The ciphering takes place before modulation and after interleavin; the deciphering takes place after demodulation symmetrically.the bottom layer of GSM(layer 1) which is known as the pysical layer is used to convert speech into radio waves and vice verse.so,we will to take a closer look onto this layer to be able to imagine the whole sequence.

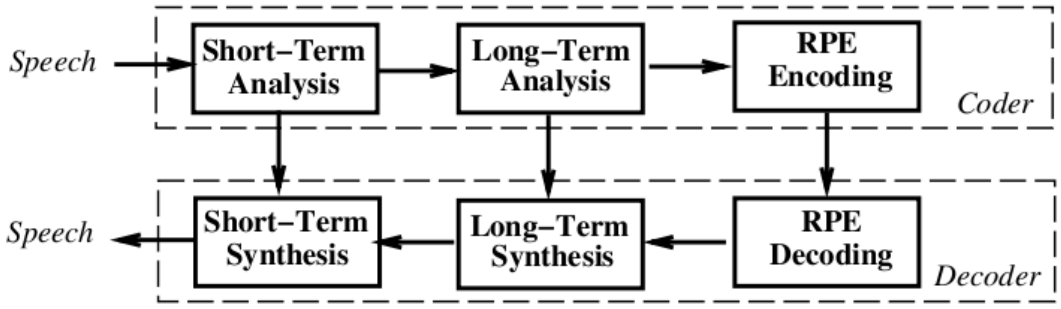
Forward error correction and Radio Modem in L1



so the sequence of operation consists of mainly six important stages:

1-The GSM Speech Coding:

The full rate speech codec in GSM is described as Regular Pulse Excitation with Long Term Prediction GSM 06.10 RPE-LTP.Basically, the encoder divides the speech into short-term predictable parts, long-term predictable part and the remaining residual pulse. Then, it encodes that pulse and parameters for the two predictors. The decoder reconstructs the speech by passing the residual pulse first through the long-term prediction filter, and then through the short-term predictor.

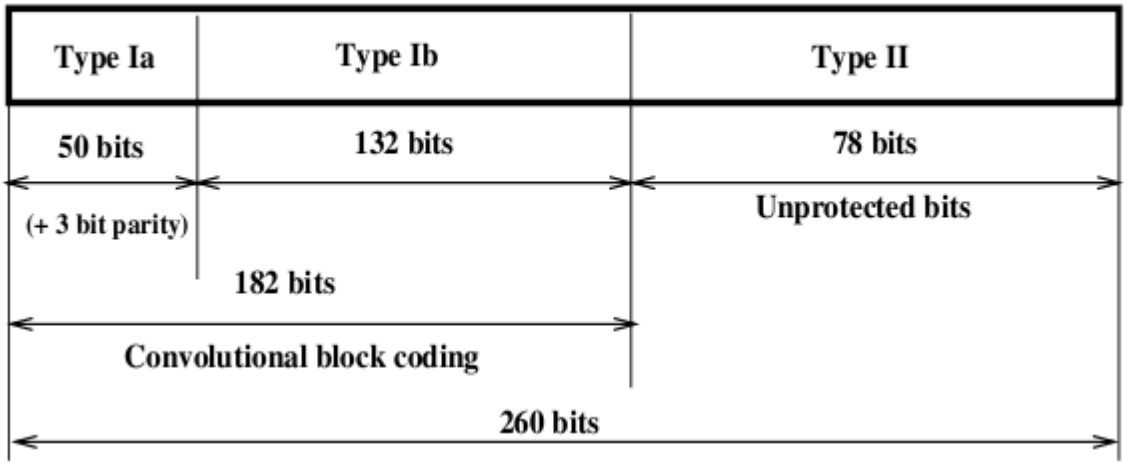


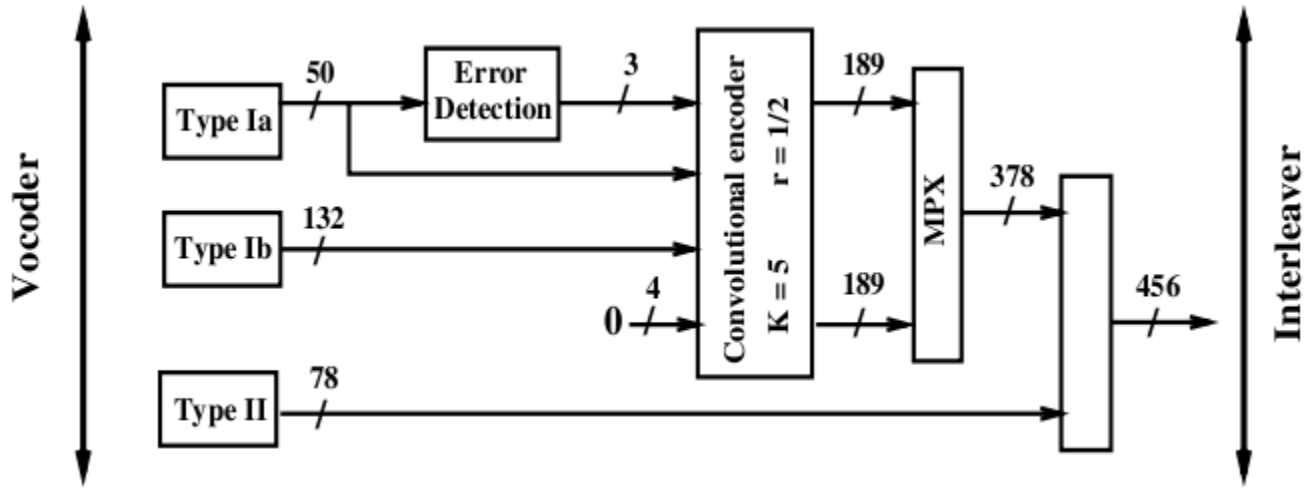
OpenBTS will initially support the “full rate” vocoder of GSM 06.10, relying on Asterisk to run the transcoders for us.Half rate traffic channels are ignored as They are not a priority in a first release.

2-The GSM Channel Coding:

Channel coding introduces redundancy into the data flow in order to allow the detection or even the correction of bit errors introduced during the transmission.The speech coding algorithm produces a speech block of 260 bits every 20 ms i.e. bit rate 13 kbit s. In the decoder, these speech blocks are decoded and converted to 13 bit uniformly coded speech samples. The 260 bits of the speech block

are classified into two groups. The 78 Class II bits are considered of less importance and are unprotected. The 182 Class I bits are split into 50 Class Ia bits and 132 Class Ib bits .Class Ia bits are first protected by 3 parity bits for error detection. Class Ib bits are then added together with 4 tail bits before applying the convolutional code with rate r = 1 and constraint length K = 5. The resulting 378 bits are then added to the 78 unprotected Class II bits resulting in a complete coded speech frame of 456 bits.





The GSM standard uses a 3-bit error redundancy code to enable assessment of the correctness of the bits which are more sensitive to errors in the speech frame the category Ia 50-bits. If one of these bits are wrong, this may create a loud noise instead of the 20 ms speech slice. Detecting such errors allows the corrupted block to be replaced by something less disturbing such as an extrapolation of the preceding block.The polynomial representing the detection code for category Ia bits is GX = X 3 + X + 1.At the receiving side, the same operation is done and if the remainder differs, an error is detected and the audio frame is eventually discarded.The GSM convolutional code consists in adding 4 bits set to 0" to the initial 185 bit sequence and then applying two different convolutions: polynomials are respectively G1 X = X 4 + X 3 + 1 and G2X = X 4 + X 3 + X + 1. The final result is composed of twice 189 bits sequences.Convolutional

decoding can be performed using a Viterbi algorithm 2 . A Viterbi decoder logically explores in parallel every possible user data in sequence. It encodes and compare each one against the received sequence and picks up the closest match: it is a maximum likelihood decoder. To reduce the complexity the number of possible data sequence double with each additional data bit, the decoder recognizes at each point that certain sequences cannot belong to the maximum likelihood path and it discards them.
The encoder memory is limited to K bits; a Viterbi decoder in steady-state operation keeps only 2K,1 paths. Its complexity increases exponentially with the constraint length K.
in Openbts,All GSM channels use some type of block code. Some channels use simple CRC-type parity check codes. Most channels use a Fire code that can also be used for burst-error correction, described in GSM 05.03 Section 4.1.2.Also Most GSM channels use a rate-1/2 convolutional encoder described in GSM 05.03 Section 4.1.3.

Code Implementation in File :GSML1FEC.cpp, **line: 808**

```
void XCCHL1Encoder::encode()
{
    // Perform the FEC encoding of GSM 05.03 4.1.2 and 4.1.3
    // GSM 05.03 4.1.2
    // Generate the parity bits.
     mBlockCoder.writeParityWord(mD,mP);
    OBJLOG(DEEPDEBUG) << "XCCHL1Encoder u[]=" << mU;
    // GSM 05.03 4.1.3
    // Apply the convolutional encoder.
    mU.encode(mVCoder,mC);
    OBJLOG(DEEPDEBUG) << "XCCHL1Encoder c[]=" << mC;
}
```

3-Interleaving/ De-interleaving:

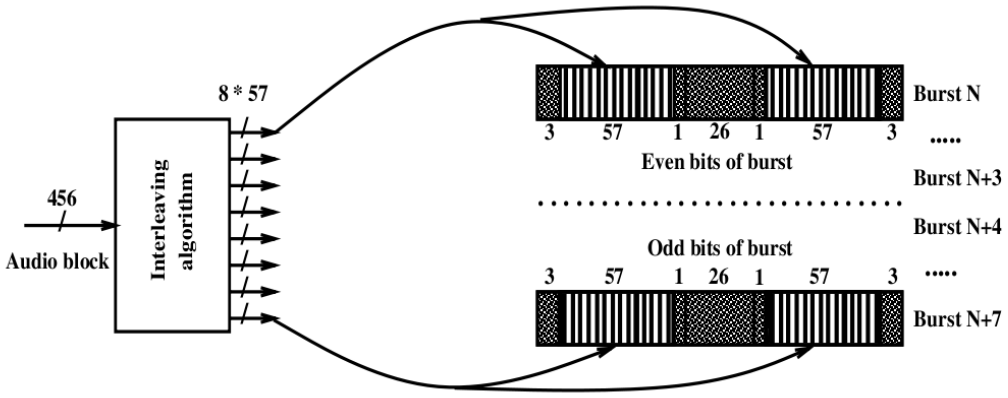
Interleaving is meant to decorrelate the relative positions of the bits respectively in the code words and in the modulated radio bursts. The aim of the interleaving algorithm is to avoid the risk of loosing consecutive data bits. GSM blocks of full rate speech are interleaved on 8 bursts: the 456 bits of one block are split into 8 bursts in sub-blocks of 57 bits each. A sub-block is defined as either the odd- or the even-numbered bits of the coded data within one burst. Each sub-blocks of 57 bit is carried by a di erent burst and in a different TDMA frame. So, a burst contains the contribution of two successive speech blocks A and B. In order to destroy the proximity relations between successive bits, bits of block A use the even positions inside the burst and bits of block B, the odd positions . De-interleaving consists in performing the reverse operation. The major drawback of interleaving is the corresponding delay: transmission time from the first burst to the last one in a block is equal to 8 TDMA frames i.e. about 37 ms.
In most GSM channels, a higher-layer data frame is spread over 4 radio bursts with an interleaving pattern according to the rules of GSM 05.03 Sections 4.1.4 and 4.1.5.Interleaving. some channels(TCH/FACCH) use 8-burst diagonal interleaving as per GSM 05.03 Sections 3.1.3 and 3.1.4.
In some channels (RACH, RACCH and SCH) each protocol element maps to a single radio burst and there is no interleaving.

Code Implementation in File :GSML1FEC.cpp, **line:** 842

```
void XCCHL1Encoder::interleave()
 {
      // GSM 05.03, 4.1.4.  Verbatim.
       for (int k=0; k<456; k++) {
            int B = k%4;
           int j = 2*((49*k) % 57) + ((k%8)/4);
           mI[B][j] = mC[k];
       }
}
```

4-Ciphering/ Deciphering:

A protection has been introduced in GSM by means of transmission ciphering. The ciphering method does not depend on the type of data to be transmitted speech, user data or signaling but is only applied to normal bursts.Ciphering is achieved by performing an exclusive or" operation between a pseudo-random bit sequence and 114 useful bits of a normal burst i.e. all information bits except the 2 stealing flags. The pseudo-random sequence is derived from the burst number and a key session established previously through signaling means. Deciphering follows exactly the same operation.

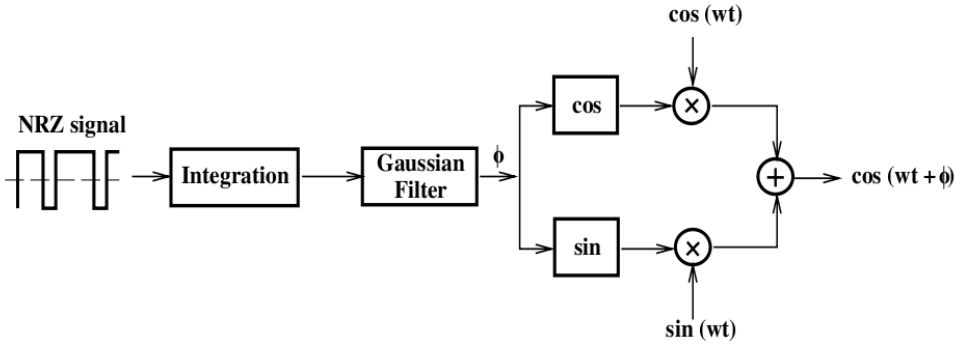


the open source OpenBTS version didn't support encryption but in our project we succeed to integrate this feature which will be discussed later.

5-Modulation /Demodulation:

GSM uses the Gaussian Modulation Shift Keying GMSK with modulation index h =0:5, BT (filter bandwidth times bit period) equal to 0.3 and a modulation rate of 271 (270 5/6)kbauds. The GMSK modulation has been chosen as a compromise between a fairly high spectrum effeciency of the order of 1 bit Hz and a reasonable demodulation complexity. The constant envelope allows the use of simple power amplifiers and the low out-of-band radiation minimizes the e ect of adjacent channel interference. GMSK

differs from Minimum Shift Keying MSK in that a pre-modulation Gaussian filter is used.



Code Implementation in File :sigProcLib.cpp., **line:** 521
signalVector *modulateBurst(const BitVector &wBurst,const signalVector &gsmPulse,int guardPeriodLength, int samplesPerSymbol)
as the ciphering is applied on the normal bursts so we need also to focus on the construction of Normal bursts

Normal burst:

As an indication, due to the TDMA techniques used in the system, the useful data (also called the plain text in the sequel) are organized into blocks of 114 bits. Then, each block is incorporated into a normal burst and transmitted during a time slot.

Bit Number (BN)	Length of field	Contents of field
0 - 2	3	tail bits
3 - 60	58	encrypted bits (e0 . e57)
61 - 86	26	training sequence bits
87 - 144	58	encrypted bits (e58 . e115)
145 - 147	3	tail bits
(148 - 156	8,25	guard period (bits)

-where the "tail bits" are defined as modulating bits with states as follows:
(BN0, BN1, BN2) = (0, 0, 0) and
(BN145, BN146, BN147)= (0, 0, 0)

Code Implementation in File :GSML1FEC.cpp, **line:** 742
mU.zero();

-where the "training sequence bits" are defined as modulating bits with states as given in the following table according to the training sequence code, TSC. For broadcast and common control channels, the TSC must be equal to the BCC. In networks supporting E-OTD Location services (see GSM 03.71 Annex C), the TSC shall be equal to the BCC for all normal bursts on BCCH frequencies.

Code Implementation in File :GSML1FEC.cpp, **line:**739
gTrainingSequence[mTSC].copyToSegment(mBurst,61);

Code Implementation in File :GSMCommon.cpp, **line:**44

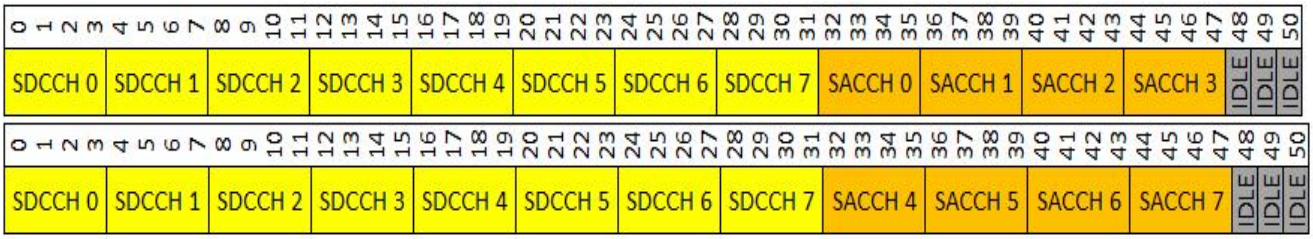
```
const BitVector GSM::gTrainingSequence[] = {
BitVector("00100101110000100010010111"),
BitVector("00101101110111100010110111"),
BitVector("01000011101110100100001110"),
BitVector("01000011101101000100011110"),
BitVector("00011010111001000001101011"),
BitVector("01001110101100000100111010"),
BitVector("10100111110110001010011111"),
BitVector("11101111000100101110111100"),
};
```

So according to GSM 05.03, the useful information bits into a block are numbered e0 to e56 and e59 to e115 (the flag bits e57 and e58 are ignored .

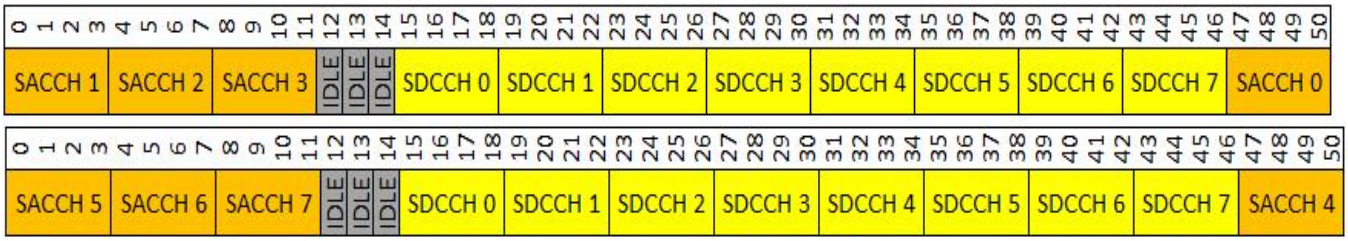
SDCCH channel mapping:

Control Channel are composed of 51 TDMA frames. On a time slot Within the multiframe, the 51 TDMA frames are divided up and allocated to the various logical channels.There are several channel combinations allowed in GSM but we are only concerned with two channels the SDCCH and the TCH as they are the channels where the ciphering will be applied.

SDCCH/8(0 .7) + SACCH/C8(0 . 7)



Uplink



Downlink

the downlink and uplink multiframes do not align with each other. This is done so that if the BTS sends an information request to the MS, it does not have to wait an entire multiframes to receive the needed information. The uplink is transmitted 15 TDMA frames behind the downlink. For example, the BTS might send an authentication request to the MS on SDCCH0 (downlink) which corresponds to TDMA frames 22-25. The MS then has enough time to process the request and reply on SDCCH0 (uplink) which immediately follows it on TDMA frames 37-40.

The 228-bit Keystream is supposed to be generated from the Frame Number and the Kc . According to this and the channel mapping,this shouldn't cause any problems in the TCH as the Frame Number is the same for both the Uplink and the Downlink, the 228 bits are split into 2*114 bits for the same frame number.but for SDCCH the UL and DL are in different frames,so we would only use one of the pair of 114 bits .

The ciphering process:
For ciphering, Algorithm A5 produces, each 4.615 ms, a sequence of 114 encipher/decipher bits (here called BLOCK) which is combined by a bit-wise modulo 2 addition with the 114-bit plain text block. The first encipher/decipher bit produced by A5 is added to e0, the second to e1 and so on. As an indication, the resulting 114-bit block is then applied to the burst builder (see GSM 05.01).
For each slot, deciphering is performed on the MS side with the first block (BLOCK1) of 114 bits produced by A5, and enciphering is performed with the second block (BLOCK2). As a consequence, on the network side BLOCK1 is used for enciphering and BLOCK2 for deciphering. Therefore Algorithm A5 must produce two blocks of 114 bits (i.e. BLOCK1 and BLOCK2) each 4.615 ms.

Ciphering/deciphering Codes Part:

We have two channels that we should encrypt and decrypt in order to make correct ciphering procedures which are the Traffic Channel (TCH) and Stand alone dedicated channel (SDCCH)

Reference of Functions:

copyToSegment: void Vector< T >::copyToSegment (Vector< T > & *other*, size_t *start*, size_t *span*) : Copy part of this Vector to a segment of another Vector.

void encrypt(): performs the Xoring Stage between the 114 bits downlink with the interleaved bit

void decrypt(): performs the Xoring Stage between the 114 bits uplink with the Received bits

virtual void sendFrame(const L2Frame&):sends a single L2 frame.It makes sure there's a downstream data there to take this uncoded bits and start the encoding process.

void transmit(): Formats i[] into timeslots and send them down for transmission. Set stealing flags assuming a control channel. Also updates mWriteTime.

const SoftVector data1():Return a SoftVector alias to the first data field.

const SoftVector data2(): Return a SoftVector alias to the second data field.

Important Definitions:

BitVector: Construct from a string of "0" and "1".

SoftVector: The SoftVector class is used to represent a soft-decision signal. Values 0..1 represent probabilities that a bit is "true".

ChannelType: Describes the Type of the Channel

L1FEC:this class is concerned with the encapsulation of the encoder and the decoder.The L1FEC member functions are over-ridden for different channel types.

Naming Conventions:

"k" and "j" for numbering of bits in data blocks and bursts;
"n" is used for numbering of delivered data blocks
"N" marks a certain data block;
"B" is used for numbering of bursts or blocks
Interleaved data bits: i(B,k)
for k = 0,1,...,Ki-1
B = B0, B0+1,....
Interleaved data symbols: I(B,k)
for k = 0,1,...,KI-1
Bits in one burst :e(B,k)
for k = 0,1,...,114,115

The encryption after the interleaving stage is done by Xoring the 114 bits keystream with the plaintext in order to produce a cipher text, those bits are the first 114 bits generated from the 228 output bits from the A51 Algorithm which takes Frame number and the Kc of the mobile as an input parameters, the same thing happens for the Uplink when we need to decipher the Text in order to get the plain text again by Xoring the second 114 bits with the cipher text.

1-Traffic Channel (TCH) :

In class TCHFACCHL1Encoder:

the changes that we made in GSML1FEC.h concerning this channel was :

1-defining a new variable which contains the encrypted bits in one burst this variable will be a private member for the class TCHFACCHL1Encoder which is inherited from the class XCCHL1Encoder where Private members can be accessed only within methods of the class itself.

Code Implementation in File :GSML1FEC.h, **line:746**

```
BitVector mE[8];
```

and this variable will be initialized GSML1FEC.cpp inside the constructor of TCHFACCHL1Encoder

Code Implementation in File :GSML1FEC.cpp, **line:1439**

```
for(int k = 0; k<8; k++) {
        mE[k] = BitVector(114);
         mI[k] = BitVector(114);
         // Fill with zeros just to make Valgrind happy.
         mI[k].fill(0);
         mE[k].fill(0);
}
```

2-adding a new function to the class TCHFACCHL1Encoder .this is the function where Xoring takes place .

Code Implementation in File :GSML1FEC.h, **line:780**

```
virtual void encrypt();//function prototype
```

Code Implementation in File :GSML1FEC.cpp, **line:1611**

```
void TCHFACCHL1Encoder::encrypt()//function definition
{
for (int B=0; B<8; B++)  // the loop till 8 cause the number of blocks in the TCH are 8
        {
                mE[B]=mI[B]^keyStreamDL;
                LOG(INFO) <<"TCHFACCHL1Encoder encrypt";
        }
}
```

and this function will be called after the interleaving and before the coping into segment and transmission.

Code Implementation in File :GSML1FEC.cpp, **line:1575**

```
LOG(INFO)<<"TCH START INTERLEAVE";
  interleave(mOffset);
  LOG(INFO)<<"TCH START ENCRYPT";
  encrypt();
  LOG(INFO)<<"TCH START ENCRYPTed";
```

and instead of filling the segment with interleaved bits it will be filled with encrypted

```
mE[B].segment(0,57).copyToSegment(mBurst,3);
mE[B].segment(57,57).copyToSegment(mBurst,88);
```

In class TCHFACCHL1Decoder:

the changes that we made in GSML1FEC.h concerning this channel was :
1-defining a new variable which contains the encrypted bits in one burst this variable will be a protected member for the class TCHFACCHL1Decoder which is inherited from the class XCCHL1Decoder where Protected is the middle point between the private and puplic .this variable can be accessed by the class itself , Functions/classes marked as friend and Objects that inherit from that class.

Code Implementation in File :GSML1FEC.h, **line:811**

```
SoftVector mE[8];          ///< decrypting history, 8 blocks instead of 4
```

and this variable will be initialized GSML1FEC.cpp inside the constructor of TCHFACCHL1Decoder

Code Implementation in File :GSML1FEC.cpp, **line:1186**

```
for (int i=0; i<8; i++) {
          mE[i] = SoftVector(114);
           mI[i] = SoftVector(114);
           // Fill with zeros just to make Valgrind happy.
           mI[i].fill(.0);
           mE[i].fill(.0);
   }
```

2-adding a new function to the class TCHFACCHL1Decoder .this is the function where Xoring takes place but here we are doing this on the upstream data.

Code Implementation in File :GSML1FEC.h, **line:846**

```
 void decrypt();//function prototype
```

Code Implementation in File :GSML1FEC.CPP, **line:1296**

```
void TCHFACCHL1Decoder::decrypt()
{
for (int B=0; B<8; B++)
          {
          mI[B]=mE[B]^keyStreamUL;
          LOG(INFO) <<"TCHFACCHL1Decoder decrypt";
          }
}
```

and this function will be called before the deinterleaving and after copying the received data into the segment.

```
  decrypt();
   if (B==3)
   {
   LOG(INFO) <<"TCH PROCESS BURST ";
          LOG(INFO) <<"TCH PROCESS BURST DECRYPTED";
   deinterleave(4);
          LOG(INFO) <<"TCH PROCESS BURST INTERLEAVED";
   }
```

```
else
{
        LOG(INFO) <<"TCH PROCESS BURST 1";
deinterleave(0);
LOG(INFO) <<"TCH PROCESS BURST 2";
}
```

2-Stand alone dedicated control channel (SDCCH) :

In class SDCCHL1Encoder:

we have to implement a new class because we need to encrypt the SDCCH only not the whole channels so the new function will be a member of this class.

Code Implementation in File :GSML1FEC.h, **line:681**

```
class SDCCHL1Encoder : public XCCHL1Encoder {

   private:

   /**@name FEC signal processing state.  */
   //@{
   BitVector mE[4];                ///< e[][], as per GSM 05.03 2.2

   //@}

   public:

   SDCCHL1Encoder(
         unsigned wTN,
         const TDMAMapping& wMapping,
         L1FEC* wParent);

    void encrypt();

   /** Extend open() to set up semaphores. */
//    void open();

   protected:

   /** Send a single L2 frame.  */
   virtual void sendFrame(const L2Frame&);

   /**
     Format i[] into timeslots and send them down for transmission.
     Set stealing flags assuming a control channel.
     Also updates mWriteTime.
```

```
     GSM 05.03 4.1.5, 05.02 5.2.3.
   */
    void transmit();

   /** Will start the dispatch thread. */
//    void start();
};
```

the changes that we made in GSML1FEC.h concerning this channel was :
1-defining a new variable which contains the encrypted bits in one burst this variable will be a private member for the class SDCCHL1Encoder which is inherited from the class XCCHL1Encoder where Private members can be accessed only within methods of the class itself.

Code Implementation in File :GSML1FEC.h, **line:687**

```
BitVector mE[4];
```

and this variable will be initialized GSML1FEC.cpp inside the constructor of SDCCHL1Encoder

Code Implementation in File :GSML1FEC.cpp, **line:926**

```
for(int k = 0; k<4; k++) {
          mE[k] = BitVector(114);
           mI[k] = BitVector(114);
           // Fill with zeros just to make Valgrind happy.
           mI[k].fill(0);
           mE[k].fill(0);
   }
```

2-adding a new function to the class XCCHL1Encoder .this is the function where Xoring takes place .

Code Implementation in File :GSML1FEC.h, **line:699**

```
void encrypt();;//function prototype
```

Code Implementation in File :GSML1FEC.cpp, **line:966**

```
void SDCCHL1Encoder::encrypt()
{
LOG(INFO)<< "SDCCH ENCRYPT frmae number" << ((mCount.T1()<<11)|(mCount.T3()<<5)|mCount.T2());
for (int B=0; B<4; B++)
   {
           mE[B]=mI[B]^keyStreamDl;
          LOG(INFO) <<"SDCCHL1Encoder  ENCRYPT";
   }
}
```

and this function will be called after the interleaving and before transmission.

Code Implementation in File :GSML1FEC.cpp, **line:997**

```
interleave();       // Interleave c[] to i[][], GSM 05.03 4.1.4.
  LOG(INFO) << "SDCCHL1ENCODER INTERLEAVED";
  encrypt();                    // Encrypt i[][] to e[].
```

```
  LOG(INFO) << "SDCCHL1ENCODER ENCRYPTED";
 transmit();                    // Send the bursts to the radio, GSM 05.03 4.1.5.
 LOG(INFO) << "SDCCHL1ENCODER TRANSMITTED";
```
and instead of filling the segment with interleaved bits it will be filled with encrypted

In class SDCCHL1Decoder:

the changes that we made in GSML1FEC.h concerning this channel was :
1-defining a new variable which contains the encrypted bits in one burst this variable will be a protected member for the class XCCHL1Decoder which is inherited from the class L1Decoder where Protected is the middle point between the private and puplic .this variable can be accessed by the class itself , Functions/classes marked as friend and Objects that inherit from that class.

Code Implementation in File :GSML1FEC.h, **line:465**

```
SoftVector mE[4];          ///< decrypting history
```
and this variable will be initialized GSML1FEC.cpp inside the constructor of XCCHL1Decoder

Code Implementation in File :GSML1FEC.cpp, **line:1186**

```
for (int K=0; K<4; i++) {
          mE[K] = BitVector(114);
           mI[K] = BitVector(114);
           // Fill with zeros just to make Valgrind happy.
           mI[K].fill(0);
           mE[K].fill(0);
   }
```
2-adding a new function to the class XCCHL1Decoder .this is the function where Xoring takes place but here we are doing this on the upstream data.

Code Implementation in File :GSML1FEC.h, **line:505**

```
 virtual void decrypt();//function prototype
```

Code Implementation in File :GSML1FEC.CPP, **line:1307**

```
void XCCHL1Decoder::decrypt()
{

for (int B=0; B<4; B++)
   {
   if(channelType()==SDCCHType)
   {

mI[B]=mE[B]^keyStreamUL;

          LOG(INFO) <<"XCCHL1Decoder Decrypt";
   }
   else
   {
```

```
   mI[B]=mE[B];
   LOG(INFO) <<"XCCHL1Decoder ELSE";
   }
   }

}
```

and this function will be called before the deinterleaving

Code Implementation in File :GSML1FEC.cpp, **line:559**

```
if (!processBurst(inBurst)) return;
   decrypt();
   deinterleave();
   if (decode()) {
            countGoodFrame();
            mD.LSB8MSB();
            handleGoodFrame();
   } else {
            countBadFrame();
   }
```

Sections in this Chapter:

1- Xoring in TCH

2- Xoring in SDCCH

3- Relation between Frame Numbers in SDCCH

References :1) GSM 05.02