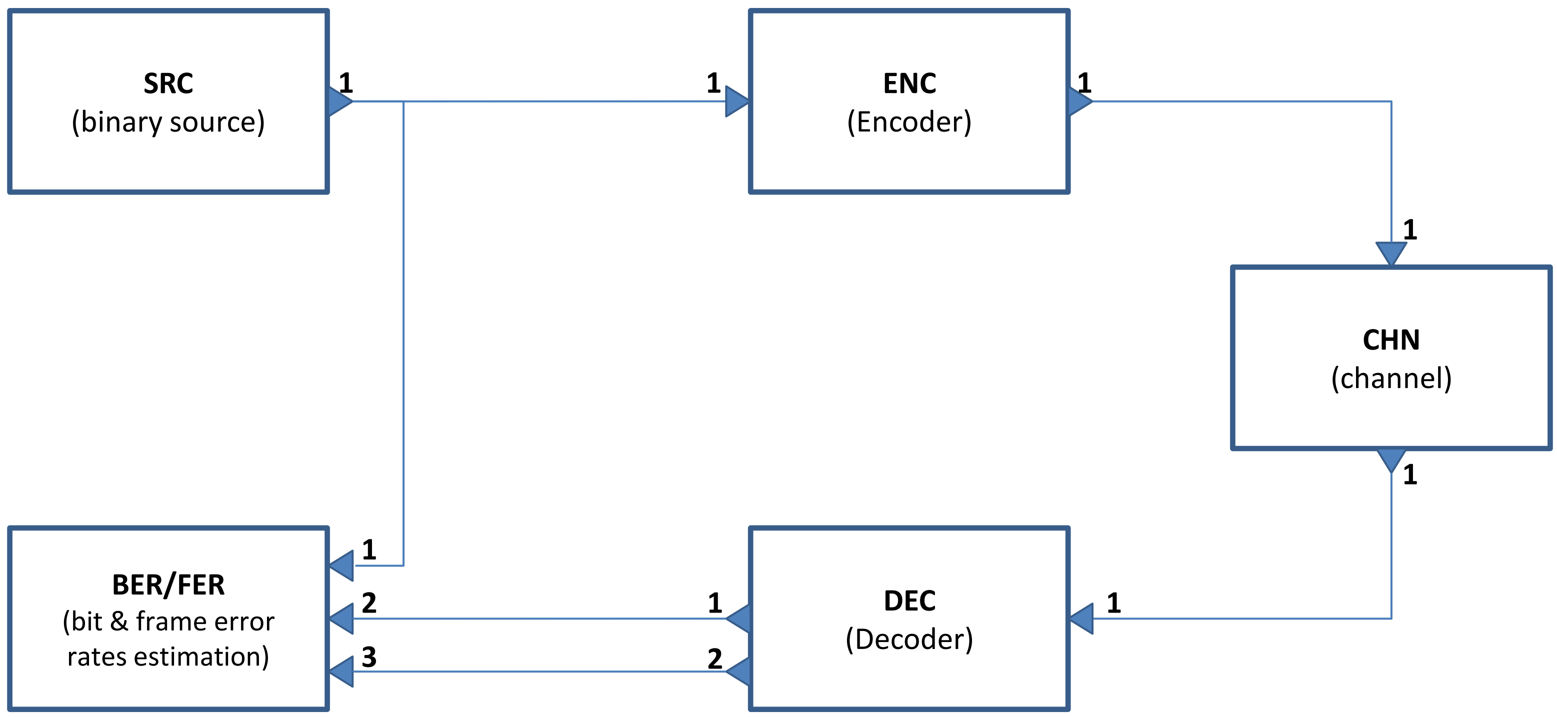
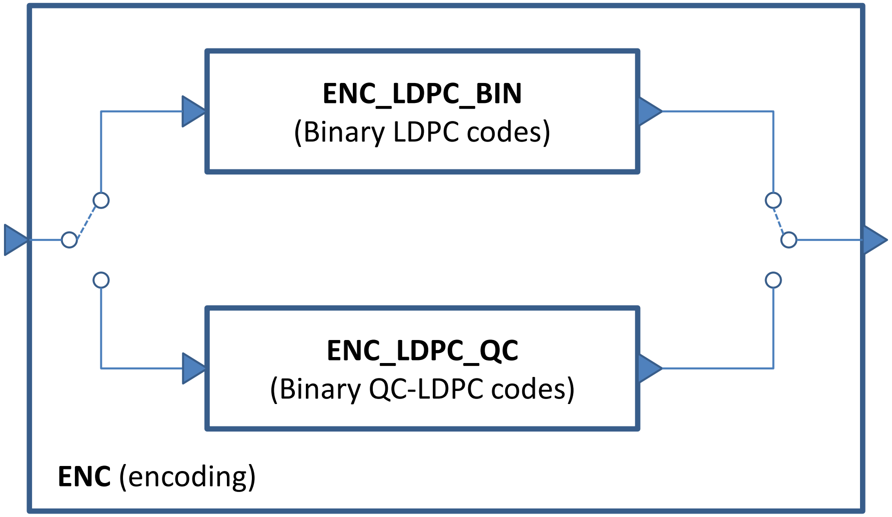
Main modules of the simulator



OUTPUT	INPUT	DESCRIPTION
SRC-OUT-1	ENC-IN-1 BER-IN-1	Vector of source bits (size = ‘info_size’)
ENC-OUT-1	CHN-IN-1	Vector of coded bits (size = ‘coded_size’)
CHN-OUT-1	DEC-IN-1	Vector of soft bits (size = ‘coded_bits’). Remark: LLRs are computed in the channel module!!!
DEC-OUT-1	BER-IN-2	Vector of estimated source bits (size = ‘info_size’)
DEC-OUT-2	BER-IN-3	Decoding statistics (error detection, number of iterations, etc.)

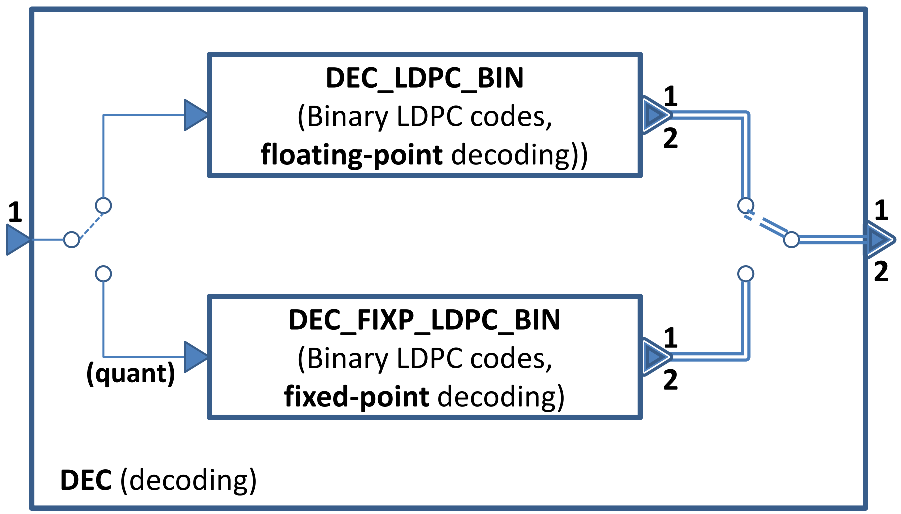
The encoding (ENC) module



QC = Quasi-Cyclic

OUTPUT	INPUT	DESCRIPTION
--	ENC_LDPC_BIN-IN ENC_LDPC_QC-IN	Vector of source bits
ENC_LDPC_BIN-OUT ENC_LDPC_QC-OUT	--	Vector of coded bits

The decoding (DEC) module



OUTPUT	INPUT	DESCRIPTION
--	DEC_LDPC_BIN-IN DEC_FIXP_LDPC_BIN-IN	Binary LLRs (binary LLRs are quantified before the fixed-point decoding!)
DEC_[…]-OUT-1	--	Vector of estimated (decoded) source bits
DEC_[…]-OUT-2	--	Decoding statistics (error detection, number of iterations, etc.)

“Module” definition

Each “**module**” is a C++ class, with at least:

- 2 public methods: “Activate()” and “Main()”
- 2 or more public variables: “ptFifoIn” (“ptFifoIn2”, …) and “ptFifoOut” (“ptFifoOut2”,…)

The **public variables** play the role of the **input** and **output ports** of the module.

The “**Activate()**” method is used to:

- Get the values of the configuration parameters of the module
- Allocate the memory necessary to implement the module’s algorithm
- Allocate the memory for the output port(s) (“ptFifoOut”)

The “**Main()**” method implements the specific algorithm.

The use of each "module" is rather intuitive:

- “Plug” the input port (i.e. “ptFifoIn” must point to some already allocated buffer!)
- Call the “Activate()” method – in particular, this will also allocate memory for the output port (“ptFifoOut”) that can be used as the input of the following module
- Call the “Main()” method to execute the algorithm implemented by the module

“Module” definition

```
class MODULE
{
public:
      // ------------------------------------------
      // DECLARATION OF THE MODULE PUBLIC METHODS
      // ------------------------------------------
      MODULE();             // constructor
      ~MODULE();            // destructor
      int  Activate();      // class activation
      void Main();          // method which supports the thread process

      // ----------------------------------------
      // DECLARATION OF THE MODULE PUBLIC FIFOs
      // ----------------------------------------
      int  ptFifoIn;       // input port
      int  ptFifoOut;      // output port1

private:
      // -------------------------------------------
      // DECLARATION OF THE MODULE PRIVATE METHODS
      // -------------------------------------------
      void GetParam();     // get configuration parameters of the module
      void Reset();        // reset ressources
      void Free();         // resources release
      int  Mem();          // memory allocation
      int  Init();         // initialization

      // ---------------------------------------------
      // DECLARATION OF THE MODULE PRIVATE VARIABLES
      // ---------------------------------------------
      double ErrorProba;     // error probability
      int    NumberOfBits;   // number of bits

};
```

Module's activation

```
int MODULE::Activate()
{
      // ---------------------------------------------
      // init the error flag (returned value) in error
      // ---------------------------------------------
      int ErrorFlag = IN_ERROR;

      if ( ptFifoIn == 0 )
      {
            fprintf(stderr, "\nERROR: input fifo must be set before module activation!\n");
            goto end;
      }

      GetParam();                                 // Get configuration parameters
      if ( Mem()   == IN_ERROR ) goto end;   // Allocation of memory
      if ( Init()  == IN_ERROR ) goto end;   // Initialization actions

      if ( ptFifoOut == 0 )
      {
            fprintf(stderr, "\nERROR: Activation failure (out fifo = 0!)\n");
            goto end;
      }

      // ---------------------
      // successful activation
      // ---------------------
      ErrorFlag = NO_ERROR;

end:
      return ErrorFlag;
}
```

Modules's GetParam, Mem, and Init methods

```
void MODULE::GetParam()
{
      // ------------------------------------------
      // get configuration parameters of the module
      // ------------------------------------------
      ErrorProba    = CONFIG::ErrorProba;     // error probability
      NumberOfBits = CONFIG::NumberOfBits;   // number of bits

      return;
}
```

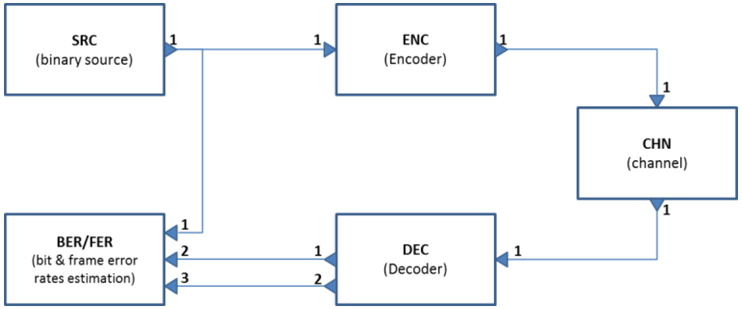
```
int MODULE::Mem()
{
      // --------------------
      // allocation of memory
      // --------------------
      if ( AllocArray(ptFifoOut, "MODULE::ptFifoOut", NumberOfBits) == IN_ERROR )
      {
            return IN_ERROR;
      }

      return NO_ERROR;
}
```

```
int MODULE::Init()
{
      // ----------------------
      // initialization actions
      // -----------------------
      // no needed in this example

      return NO_ERROR;
}
```

Module’s main method

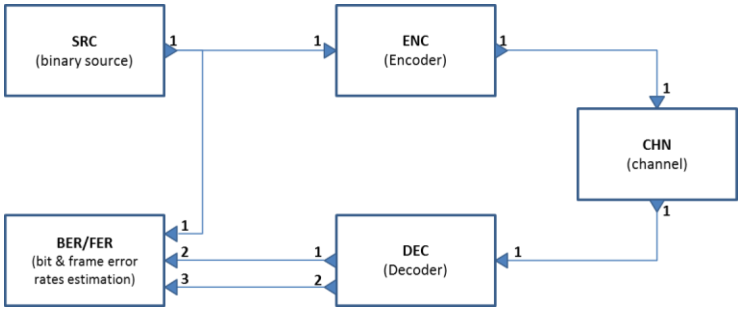
```
void MODULE::Main()
{
      // ---------------------------------
      // implements the specific algorithm
      // (a BSC channel in this example)
      // ---------------------------------
      int Indx;  // local index

      for ( Indx = 0 ; Indx < NumberOfBits ; Indx++ )
      {
            if ( (double) rand()/RAND_MAX < ErrorProba )
            {
                  ptFifoOut[Indx] = ptFifoIn[Indx]^1;  // switch bit’s value
            }
            else
            {
                  ptFifoOut[Indx] = ptFifoIn[Indx];    // not in error
            }
      }

      return;
}
```

LDPC simulator chain (1/2)




```
int main(int argc, char * argv[])
{
      .........

      // ======================================================================
      // modules activation
      // the in fifo(s) of each module must be set before the module activation
      // ======================================================================

      // Binary source module (no in fifo)
      if (ptSrc->Activate() == IN_ERROR) goto end;

      // Encoding module
      ptEnc->ptFifoIn = ptSrc->ptFifoOut;          // buffer of 'InfoSize' source bits
      if (ptEnc->Activate() == IN_ERROR) goto end; // class activation

      // Channel module
      ptChn->ptFifoIn = ptEnc->ptFifoOut;          // buffer of 'CodedSize' coded bits
      if (ptChn->Activate() == IN_ERROR) goto end; // class activation

      // Decoding module
      ptDec->ptFifoIn = ptChn->ptFifoOut;          // buffer of 'CodedSize' soft bits (LLR values)
      if (ptDec->Activate() == IN_ERROR) goto end; // class activation

      // BER/FER module
      ptBer->ptFifoIn1 = ptSrc->ptFifoOut;         // buffer of 'InfoSize' source bits
      ptBer->ptFifoIn2 = ptDec->ptFifoOut1;        // estimated (decoded) source bits
      ptBer->ptFifoIn3 = ptDec->ptFifoOut2;        // decoding statistics
      if (ptBer->Activate() == IN_ERROR) goto end; // class activation

      .........
```

LDPC simulator chain (2/2)




```
.........
// ------------------------------------------------------------
// run the simulator: call the main methods of simulator's modules
// ------------------------------------------------------------
while ( StopSimulation == 0 )
{
    // ---------------------------------------------------------------------
    // simulation loop: binary source -> encoding -> channel -> decoding -> ber/fer update
    // ---------------------------------------------------------------------
    ptSrc->Main();                 // Binary source module
    ptEnc->Main();                 // Encoding module
    ptChn->Main();                 // Channel module
    ptDec->Main();                 // Decoding module
    ptBer->Main(StopSimulation);   // BER/FER update

    // ---------------------------------------------------------------------
    // exit the 'while' loop if the simulation for the current channel param value must be stopped
    // ---------------------------------------------------------------------
    if ( StopSimulation > 0 ) break;

} // end of the 'while ( !StopSimulation )' loop


.........
}
```