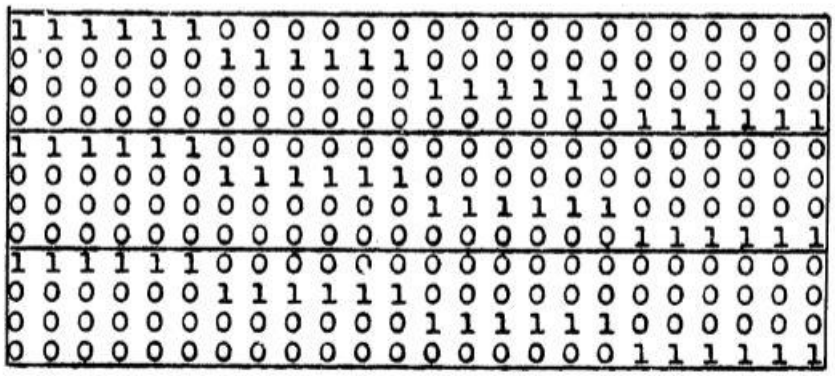
LDPC 介绍与 OAI 模块实现

简单介绍：

LDPC 中文全称为低密度奇偶校验码，是线性分组码的一种。LDPC 的校验矩阵 H 是一个稀疏矩阵，这也是 LDPC 中低密度一说的由来。得益于校验矩阵 H 的稀疏特性，LDPC 的 H 矩阵可以按照稀疏特性存储以减少存储空间。稀疏的校验矩阵 H 使得 LDPC 的译码具有线性的复杂度，即当编码长度增加时译码复杂度会线性的增加。

LDPC 编码由校验矩阵 H 获得 ，这与我们所了解的线性分组码有所不同，例如循环码。校验矩阵 H 是设计 LDPC 编码的关键所在。校验矩阵 H 的设计方式有多种，诸如 QC-LDPC、 Gallager 构造法(下图)等等。LDPC 校验矩阵 H 的性能以及特征关系到 LDPC 编码的效率以及解码的情况。



根据校验矩阵获得 LDPC 码序列同样也有多种方法。传统的高斯消元得到码生成矩阵 G 生成 LDPC 码序列的方式计算量大但非常的便捷，类似的方法还有上三角矩阵法以及准上三角矩阵法。LDPC 还有一种迭代式的编码方式，这里不再赘述。综上，LDPC 的编码可分解为两步：①构造校验矩阵 H；②选择适当的方法生产 LDPC 码序列。

LDPC 的解码是 LDPC 编码形式的灵魂所在。与传统的线性分组码所不同的是，LDPC 不再采用基于校验矩阵与纠错图样的方式来解码，而是使用迭代的译码方式进行译码（这部分将在 LDPC 解码原理部分介绍）。软迭代的方式使得 LDPC 的译码复杂度降低译码速度大大增加且译码准确度极高。

LDPC 编码

3GPP 38.212 中规定 NR 中所使用的 LDPC 为 QC-LDPC。QC-LDPC 中文全称为准循环低密度奇偶校验码。按照前文所述，我们先讨论 QC-LDPC 的校验矩阵生成。QC-LDPC 的校验矩阵由 38.212 中的基本图形规定（Base Graph, BG），并由 BG 扩展成大校验矩阵 H。在 38.212 中定义了许多 BG 矩阵$H_{BG} \in N^{A\times M}$，H_{BG}矩阵中的非稀疏元素均为非负整数。H_{BG2}的部分元素如下图所示：

$$\mathbf{H}_{\mathrm{BG2}} = \begin{bmatrix} 9 & 117 & 204 & 26 & \emptyset & \emptyset & 189 & \emptyset & \emptyset & 205 & 0 & 0 & \emptyset & \emptyset \\ 127 & \emptyset & \emptyset & 166 & 253 & 125 & 226 & 156 & 224 & 252 & \emptyset & 0 & 0 & \emptyset \\ 81 & 114 & \emptyset & 44 & 52 & \emptyset & \emptyset & \emptyset & 240 & \emptyset & 1 & \emptyset & 0 & 0 \end{bmatrix}.$$

H_{BG2}中的每一个元素将由一个方阵替换得到最终的校验矩阵 H。38.212 使用单位矩阵的移位所得到的矩阵对H_{BG2}进行填充。首先选定单位矩阵的宽度Z_c，其中Z_c的值在标准中也有所规定。我们根据如下公式选择Z_c的值：

$$Z_c = \min_{Z \in \mathbf{Z}} \left[Z \geq \frac{B}{M} \right]$$

其中 B 表示输入序列的长度，M 为 BG 的列向量个数。

根据得到的Z_c值即可扩充 BG 矩阵得到校验矩阵 H。首先，我们将 BG 矩阵中的稀疏元素全部用宽为Z_c的全零矩阵替换；其次，我们将 BG 中的所有其他非稀疏元素对Z_c取模得到循环位移值；最后，我们将宽为Z_c的单位矩阵根据上一步得到的位移值进行循环移位然后填入 BG 相应的位置得到最终的校验矩阵 H。

得到校验矩阵后根据 **Hc=0** 计算出 LDPC 码序列。在 OAI 中，LDPC 的编码使用高斯消元法得到的生产矩阵 G 进行计算。需要注意的是 OAI 中目前只支持 BG1 与 BG2 以及部分Z_c的 QC-LDPC 编码。

QC-LDPC 编码模块实现：

LDPC 编码模块文件结构：

```
|  Gen_shift_value.h
|  ldpc_encoder.c
|  ldpc_encoder2.c
|  ldpc_encoder_optim.c
|  ldpc_encoder_optim8seg.c
|  ldpc_encoder_optim8segmulti.c
|  ldpc_encode_parity_check.c
|  ldpc_generate_coefficient.c
|  nrLDPCdecoder_defs.h
|  nrLDPC_defs.h
|  nrLDPC_types.h
|  time_meas.c
|  time_meas.h
|
├──LDPC_BG_ZC
|       ldpc176_byte.c
|       ldpc176_byte_test.c
|       ldpc192_byte.c
|       ldpc208_byte.c
|       ldpc224_byte.c
|       ldpc240_byte.c
|       ldpc256_byte.c
|       ldpc288_byte.c
|       ldpc320_byte.c
|       ldpc352_byte.c
|       ldpc384_byte.c
```

```
|       ldpc_BG2_Zc104_byte.c
|       ldpc_BG2_Zc112_byte.c
|       ldpc_BG2_Zc120_byte.c
|       ldpc_BG2_Zc128_byte.c
|       ldpc_BG2_Zc144_byte.c
|       ldpc_BG2_Zc160_byte.c
|       ldpc_BG2_Zc16_byte.c
|       ldpc_BG2_Zc176_byte.c
|       ldpc_BG2_Zc192_byte.c
|       ldpc_BG2_Zc208_byte.c
|       ldpc_BG2_Zc224_byte.c
|       ldpc_BG2_Zc240_byte.c
|       ldpc_BG2_Zc256_byte.c
|       ldpc_BG2_Zc288_byte.c
|       ldpc_BG2_Zc2_byte.c
|       ldpc_BG2_Zc320_byte.c
|       ldpc_BG2_Zc32_byte.c
|       ldpc_BG2_Zc352_byte.c
|       ldpc_BG2_Zc384_byte.c
|       ldpc_BG2_Zc4_byte.c
|       ldpc_BG2_Zc64_byte.c
|       ldpc_BG2_Zc72_byte.c
|       ldpc_BG2_Zc80_byte.c
|       ldpc_BG2_Zc88_byte.c
|       ldpc_BG2_Zc8_byte.c
|       ldpc_BG2_Zc96_byte.c
|       README.md
|       sse_intrin.h
|       types.h
|
└──PART_TEST
        ldpc_encoder2_test.c
        ldpc_encoder_optim8segmulti_test.c
        ldpc_encoder_optim8seg_test.c
        ldpc_encoder_optim_test.c
        ldpc_encoder_test.c
        ldpc_encode_parity_check_test.c
        ldpc_generate_coefficient_test.c
        README.md
```

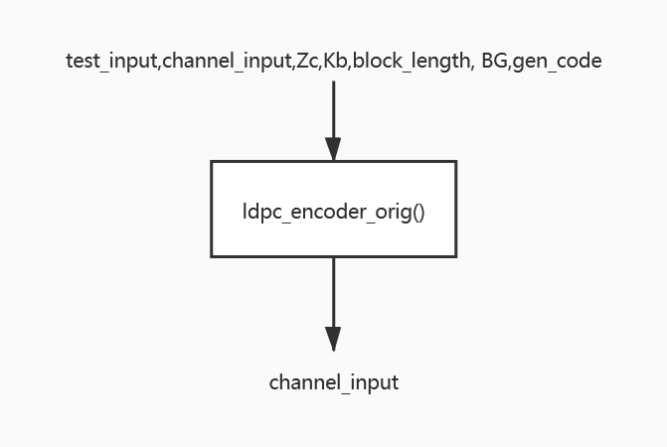
在主文件夹下，Gen_shift_value.h 中存有 BG1 与 BG2 的多种准循环位移值矩阵。其余的 nrLDPC 头文件均为一些 LDPC 编码的宏定义信息。time_meas 依赖为 LDPC 编解码中统计时间信息的函数结构体与宏。ldpc_generate_coefficient.c 与 ldpc_encode_parity_check.c 均为 LDPC 奇偶

校验部分编码的链接文件,所不同的是前者是单纯的LDPC编码而后者加入了CPU指令集优化。同样的是，ldpc_encoder.c、ldpc_encoder2.c、ldpc_encoder_optim.c、ldpc_encoder_optim8seg.c与ldpc_encoder_optim8segmulti.c均为LDPC的编码文件，所不同的地方在于是否使用优化或者优化方式是否相同。

在LDPC_BG_ZC文件夹下，所有的源文件均为使用SSE指令集优化的编码文件，对应不同的BG以及不同的Z_c值。PART_TEST文件夹下为各个编码模块的测试源文件。需要注意的是测试的源文件与主文件夹下的编码文件有些许改动，适应单独的编码模块编译。

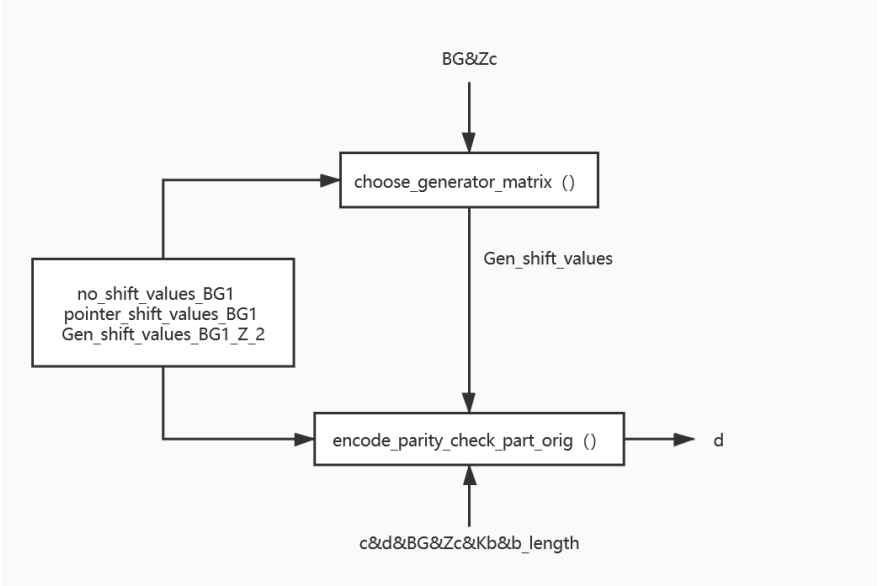
根据上述我们可以将整个LDPC_ENCODER下文件分为三个部分：①types.h等头文件依赖；②Gen_shift_value.h、ldpc_generate_coefficient.c以及ldpc_encoder.c所组成的纯LDPC编码模块③剩下文件组成的优化后的LDPC编码模块。在ldpc_encoder2.c中同时提供了ldpc_encoder_optim.c、ldpc_encoder_optim8seg.c与ldpc_encoder_optim8segmulti.c中的三种优化后的编码以供选择，因此ldpc_encoder2.c是其余三个文件的综合。

先讨论基本LDPC编码即ldpc_encoder.c的实现：



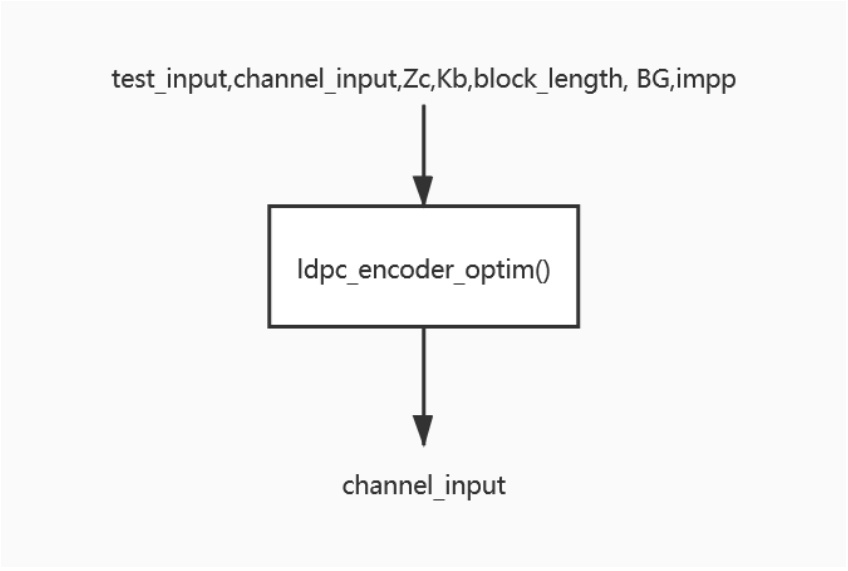
文件中使用ldpc_encoder_orig函数进行LDPC的编码。该函数需输入测试序列指针、信道输出序列（LDPC码序列）指针、宽度Zc、编码块的长度（该长度影响实际LDPC过程中是否在每个码块后加入CRC校验）、BG样号以及编码函数中用于选择的编码指示代码（均与图中的输入对应）。编码函数最终将信道编码的码序列指针返回完成编码过程。

与38.212中相同的是，OAI同样将LDPC码序列的生产过程分为两步：①块序列c（对比38.212）的移位填入；②奇偶检验比特序列d的生成并与c合并。在这两个过程中我们重点介绍第二步中检验比特序列d的产生。

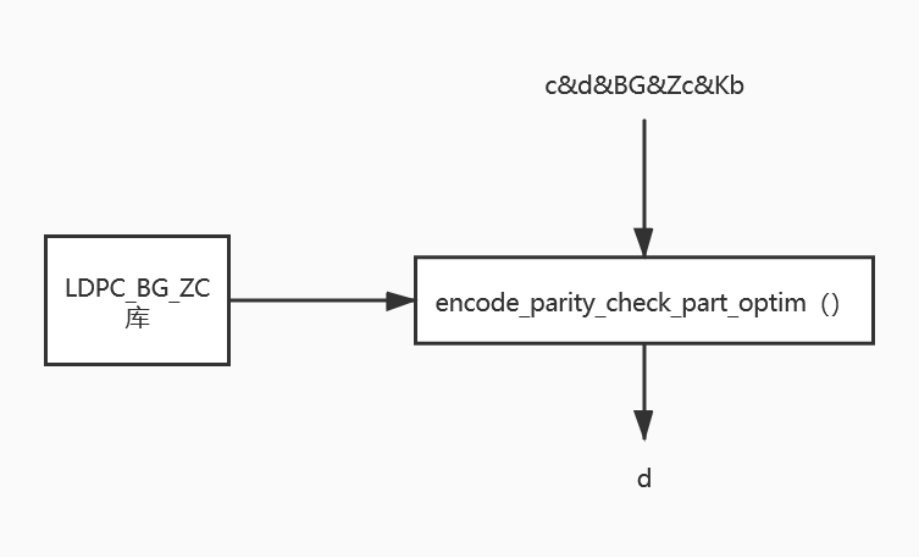


在 ldpc_encoder_orig 函数中，函数调用 choose_generator_matrix 函数用来选择偏移值矩阵并将矩阵指针传给 encode_parity_check_part_orig 函数。最后 encode_parity_check_part_orig 函数根据输入参数返回序列 d。

接下来我们将介绍优化后编码的实现。在 ldpc_encoder2.c 源文件中提供了基于 SSE 与 AVX2 以及多向量运算优化的编码函数。这些函数大同小异，因此我们主要介绍 ldpc_encoder_optim 编码函数的实现流程。



Ldpc_encoder_optim 函数的输入输出与 ldpc_encoder_orig 函数相同，所不同的定在与优化函数输入中由编码样式 impp 结构体（impp 中包含 gen_code 代码）。我们重点介绍优化部分，即 d 序列产生部分。

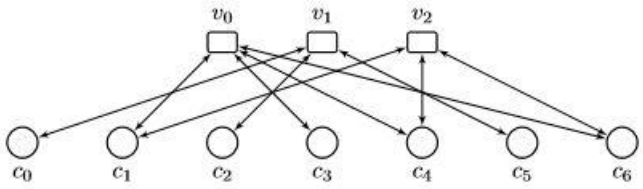


LDPC_BG_ZC 库中定义了部分 BG1 与 BG2 部分 Zc 的奇偶校验码生成函数。这些函数输入块序列 c 以及检验比特序列 d 的指针，返回计算后的 d 指针。函数中用到了 SSE 与 AVX2 的向量优化运算，可以提高编码速度。当 encode_parity_check_part_optim 函数接收到参数时会相应的从 LDPC_BG_ZC 库中调用相应的函数完成 d 序列输出。

LDPC 解码

LDPC 解码原理：

LDPC 目前有比特翻转硬解码、软迭代和积算法以及最小和积算法等解码方式。LDPC 最大的优势在软解码上，接下来简单介绍软解码原理（具体公式请参照 Gallager’s paper）。无论是和积算法还是最小和积算法均采用置信传播的软迭代方式。首先根据 Tanner 图，我们把码节点分为比特节点与检验节点两方进行迭代。在每次比特节点与码节点迭代时会产生两个似然概率，这两个似然概率之间会不断的迭代更新以最终适应检验节点的校验。



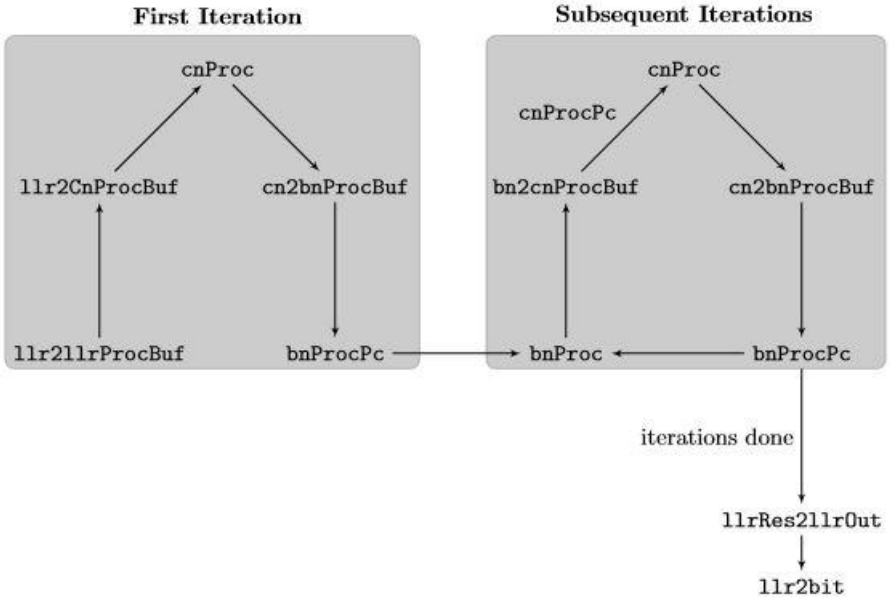
LDPC 解码模块实现

OAI 中使用的时对数域的最小和积算法进行解码。解码模块的文件结构简洁，如下所示：

nrLDPCdecoder_defs.h

nrLDPC_bnProc.h
nrLDPC_cnProc.h
nrLDPC_decoder.c
nrLDPC_init.h
nrLDPC_init_mem.h
nrLDPC_lut.h
nrLDPC_mPass.h
nrLDPC_types.h
README.md
time_meas.c
time_meas.h

该解码实现实现流程如下：



各种 Buf 用作缓冲空间防止迭代时的数据错误也可以用来优化程序。主要的处理在 cnProc（检验位处理计算）与 bnProc（比特位处理计算）。检验位处理主要用于计算最小积概率以得到比特位处理所计算的和概率。而和概率则在不满足检验位时继续送往检验位用于计算新的和概率并循环此过程。

最小积概率：

$$r_{ij} = \prod_{j' \in B_i \backslash j} sgn(q_{ii'}) \min_{j' \in B_i \backslash j} |q_{ij'}|$$

和概率：

$$q_{ij} = \Lambda_j + \sum_{i' \in C_j \backslash i} r_{ji'}$$