1. GNU RADIO界面设计
   1. 位置
2. OOT设计

参考链接：

使用python设计OOT：<https://wiki.gnuradio.org/index.php?title=Creating_Python_OOT_with_gr-modtool>

使用C++设计OOT：

<https://wiki.gnuradio.org/index.php?title=Creating_C%2B%2B_OOT_with_gr-modtool>

使用python设计：

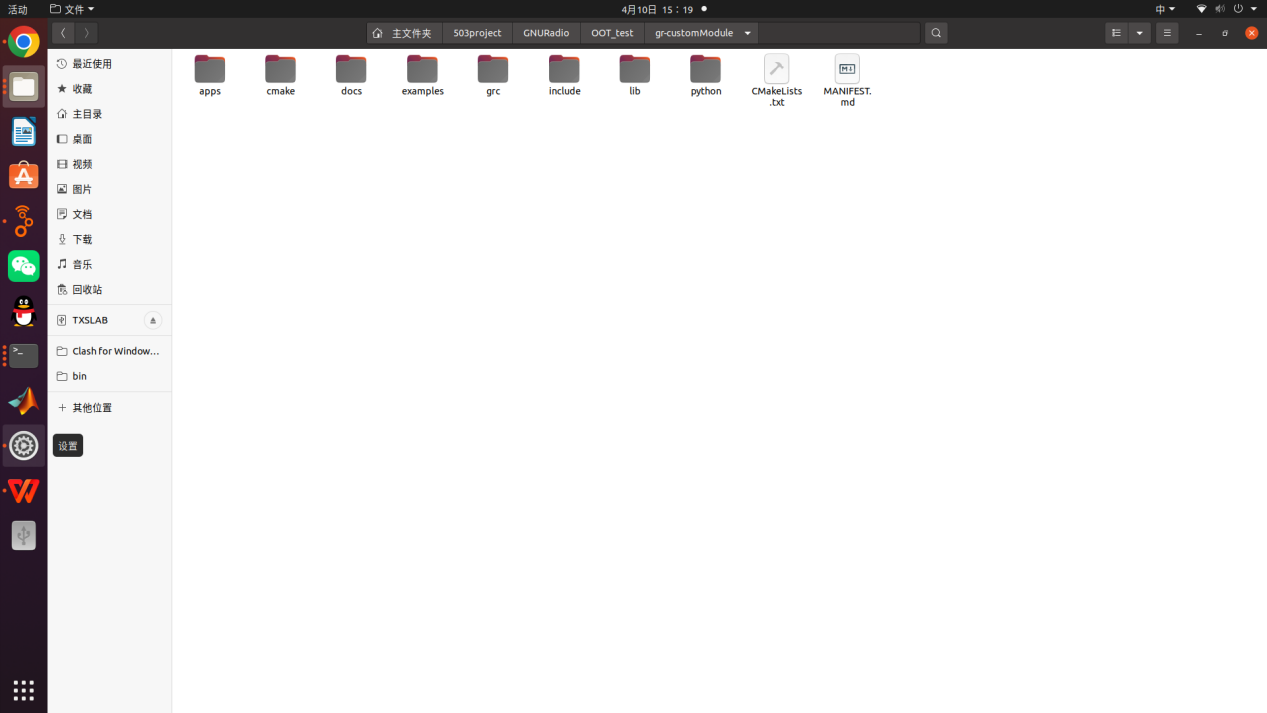
1. terminal里

cd $HOME %HOME是创建OOT的地址

1. GNU Radio附带了gr\_motool，这是一种用于创建OOT Module的软件工具。OOT Module可以被认为是自定义GNU Radio blocks的集合。使用gr\_motool创建一个名为customModule的OOT Module：

gr\_modtool newmod customModule

1. 此时创建了gr-customModule目录，该目录包含OOT Module的所有骨架代码，但它还没有任何block。



1. 移动到gr-customModule目录：

cd gr-customModule

此时ls的内容：

apps/ cmake/ CMakeLists.txt docs/ examples/ grc/ include/ lib/ MANIFEST.md python/

(4)现在需要在gr-customModule中创建一个block。例：自定义block将根据输入参数进行加法或减法运算，因此该块被命名为addSubSelect：

gr\_modtool add addSubSelect %addSubSelect block的名字

(5)该命令将启动一个关于如何定义block的调查表：什么block类型、语言和参数：

GNU Radio module name identified: customModule

('sink', 'source', 'sync', 'decimator', 'interpolator', 'general', 'tagged\_stream', 'hier', 'noblock')

Enter block type: sync 每一个输出一个输入

% general:对输入输出关系没有任何限制。

Language (python/cpp): python

Language: Python

Block/code identifier: addSubSelect %block的名字

Please specify the copyright holder: YourName %创作者的名字

Enter valid argument list, including default arguments: %输入参数列表，就像直接编写Python代码一样。在这种情况下，selector将确定block是否执行加法或减法。给出了True的默认参数：

selector=True

Add Python QA code? [Y/n] n %不用检查

Adding file 'python/customModule/addSubSelect.py'...

Adding file 'grc/customModule\_addSubSelect.block.yml'...

Editing grc/CMakeLists.txt...

创建了两个新文件，addSubSelect.py定义了block的操作，customModule\_addSubSelect.block.yml 定义了GNU Radio Companion（GRC）块的接口。CMakeLists.txt文件已被修改，因此在编译和安装模块时将安装这两个文件。

(6)修改 Python .py 文件

\_\_init\_\_（）函数和work（）函数都需要修改。函数\_\_init\_\_（）被修改为定义输入类型。addSubSelect block将接受两个complex输入并产生一个complex输出，因此需要更改in\_sig和out\_sig参数。选择器参数还需要保存为成员变量（member variable）：

全部代码如下：

import numpy as np

from gnuradio import gr

class ott1(gr.sync\_block):

"""

docstring for block ott1

"""

def \_\_init\_\_(self):

gr.sync\_block.\_\_init\_\_(self,

name="ott1",

in\_sig=[np.complex64,np.complex64],

out\_sig=[np.complex64])

self.selector = selector

def work(self, input\_items, output\_items):

in0 = input\_items[0]

in1 = input\_items[1]

if (self.selector):

output\_items[0][:] = in0 + in1

else:

output\_items[0][:] = in0 - in1

return len(output\_items[0])

(7)下一步，需要更新YAML文件以匹配刚刚修改的addSubSelector.py文件。文件位置：grc/customModule\_addSubSelect.block.yml

只有一个参数，选择器。根据以下内容输入参数值：

**parameters**:- **id**: selector

**label**: Add (True) or Subtract (False) Selector

**dtype**: bool

**default**: True‎

两个输入和一个输出：

inputs:

- label: in0

domain: stream

dtype: complex

- label: in1

domain: stream

dtype: complex

outputs:

- label: out0

domain: stream

dtype: complex

vlen: ...

optional: ...

(8)完成后进入gr\_customModule目录，建立build目录

mkdir build

进入build文件夹

cd build

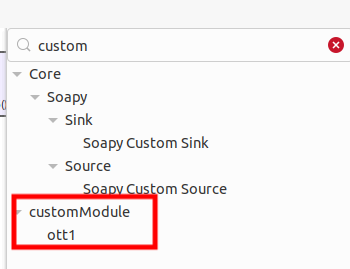
cmake ..

makeDD

sudo make install

sudo ldconfig

(9)完成后进入gnuradio，可以看到customModule里有刚建立的ott1 block



使用C++设计：

（1）

cd your-path/gr-customModule

gr\_modtool add multDivSelect

第（5）步里Language (python/cpp): cpp

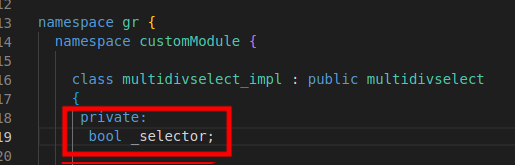
Add Python QA code? [Y/n] n

Add C++ QA code? [Y/n] n

（2）需要修改*multDivSelect\_impl.h* and *multDivSelect\_impl.cc*

lib/multDivSelect\_impl.h

创建一个布尔private member selector，which will hold the value of the selector parameter:



（3）修改.cc文件

lib/multDivSelect\_impl.cc

1. 修改yml文件

grc/customModule\_multDivSelect.block.yml

（6）编译：

cd /gr-customModule

rm -rf build/ %这是强制删除文件夹build的意思

mkdir build %重新建立build文件夹

cd build

cmake ..

make

sudo make install

sudo ldconfig

（7）更改module

Removing and re-creating the *build/* directory may be necessary before recompiling and reinstalling the module depending on the scope of the change:

$ rm -rf gr-customModule/build

$ mkdir gr-customModule/build

gr\_modtool使用指南

add Adds a block to the out-of-tree module.

disable Disable selected block in module.

info Return information about a given module

makeyaml Generate YAML files for GRC block bindings.

newmod Create new empty module, use add to add blocks.

rename Rename a block inside a module.

rm Remove a block from a module.

update Update the grc bindings for a block

1. C++ OOT设计：

以

为例：

#ifdef HAVE\_CONFIG\_H

#include "config.h"

#endif

#include <gnuradio/io\_signature.h>

#include "square1\_ff\_impl.h"

namespace gr {

namespace mymod {

square1\_ff::sptr

square1\_ff::make()

{

return gnuradio::get\_initial\_sptr

(new square1\_ff\_impl());

}

/\* 私有构造函数 所有信号处理块都是 gr::block 或其子类的派生类\*/

square1\_ff\_impl::square1\_ff\_impl()

: gr::block("square1\_ff",

gr::io\_signature::make(1, 1, sizeof (float)),

gr::io\_signature::make(1, 1, sizeof (float)))

{ }// 构造函数留空,因为该block不需要设置任何东西。

square1\_ff\_impl() 函数中第一个参数就是模块的名字，后两个函数参数依次用来定义输入和输出端口的属性，gr::io\_signature::make 函数前两个参数分别代表该模块所允许的最少/最多的输入/输出端口数，第三个参数代表**一个数据项（items）的字节数**（这里注意一个数据项可以是一般的int、float、char 等数据类型，也可以是一个向量（如一个 OFDM sambol）等数据类型，一堆数据项组成一个数据流）。

/\* 虚拟析构函数 \*/

square1\_ff\_impl::~square1\_ff\_impl()

{

}

/\* forecast函数作用为：告诉调度器有多少输入items被用来生成多少个输出items，这里他们是一样的 \*/

void

square1\_ff\_impl::forecast (int noutput\_items, gr\_vector\_int &ninput\_items\_required)

{

ninput\_items\_required[0] = noutput\_items;// 索引0代表这是第一个输入端口，这里我们只有一个输入端口

}

forecast函数原型如下：

virtual void gr::block::forecast(int noutput\_items,

gr\_vector\_int & ninput\_items\_required )

参数 noutput\_items 代表产生的输出数据项的数量，ninput\_items\_required 表示需要从输入端口中使用多少个数据项。在代码运行过程中，系统需要知道需要多少数据才能确保每个输入数组的有效性，forecast() 方法提供了此信息，因此必须在编写 gr::block 派生类的任何时候重写这个函数（对于 sync 类型的块，这是隐式的，文章结尾会提到这种 block）。函数的作用为评估生成 noutput\_items 个输出，需要多少（ninput\_items\_required）个输入 items 。不过官方文档中特别说明了这个估计值不一定要准确，但应该接近。感觉系统可能会根据我们在代码中指定的输入输出的 items 的数量关系来设置一个具有盈余的存储空间吧，这样可以提升系统的鲁棒性。

// general\_work()是定义在基类gr::block中的纯虚函数，因此在子类中必须进行重写

// general\_work()是执行实际信号处理任务的函数

int

square1\_ff\_impl::general\_work (int noutput\_items,

gr\_vector\_int &ninput\_items,

gr\_vector\_const\_void\_star &input\_items,

gr\_vector\_void\_star &output\_items)

{

const float \*in = (const float \*) input\_items[0];

float \*out = (float \*) output\_items[0];

for(int i = 0; i < noutput\_items; i++) {

out[i] = in[i] \* in[i];

}

// Tell runtime system how many input items we consumed on

// 每个输入数据流需要消耗多少items，由于ninput\_items=noutput\_items，这里填noutput\_items，意味着每生成一次数据（noutput\_items个输出），都消耗掉了noutput\_items个输入。

consume\_each (noutput\_items);

consume函数：

void gr::block::consume(int which\_input,

int how\_many\_items )

作用是告诉调度器每生成一次输出数据，数据流“which\_input”（即哪个输入端口）消耗了多少个输入数据项（how\_many\_items个）。该函数用来设置单个输入端口，并用在 general\_work() 中调用，告诉调度程序处理的输入数据项（items）的数量。但是如果所有的输入端口的输入输出比是一样的话，就可以调用 consume\_each() 函数进行统一设置。该函数的原型如下：

void gr::block::consume\_each( int how\_many\_items )

// Tell runtime system how many output items we produced.

return noutput\_items;

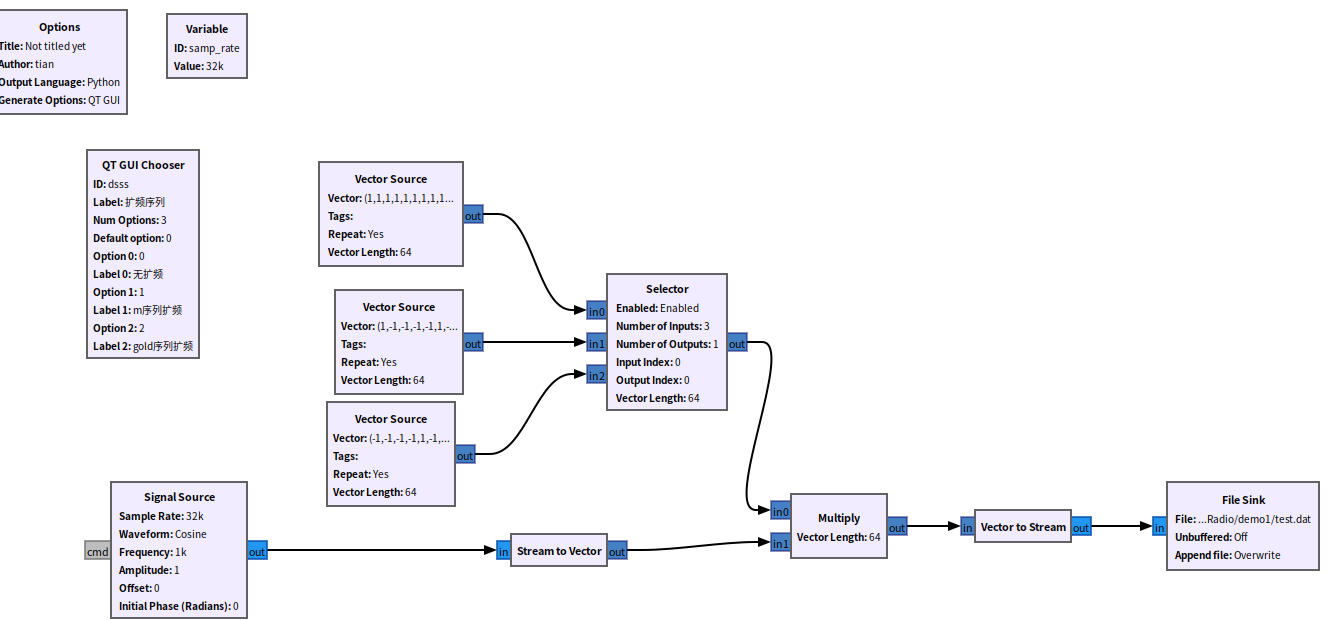
general\_work() 函数的返回值可以是输出数据项的数量也可以是 WORK\_CALLED\_PRODUCE 或者 WORK\_DONE 。WORK\_CALLED\_PRODUCE 用于并非所有输出都产生相同数量的数据项的场合，WORK\_DONE 表示这个块将不再产生数据。

}

} /\* namespace mymod \*/

} /\* namespace gr \*/

1. 多个block合成一个：
2. 先建立完整框图



该框图可以实现通过choose控制selector

1. 将需要变成一个block的blocks选中，右键，more,create hier，此时生成了一个新的grc，输入输出为pad
2. 双击options，ID和title改成集成block的ID.catorgy是集成block在右边哪个module里找到，默认为[GRC Hier Blocks]可不修改。
3. 

生成hier block的流程图

（5）reload原流程图，右边GRC Hier Blocks出现hier block 的 title，

1. OTFS、OFDM设计

OTFS的matlab代码：

N为子载波数，M为符号数

X = fft(ifft(x).').'/sqrt(M/N); %%%ISFFT

s\_mat = ifft(X.')\*sqrt(M); % Heisenberg transform

s1 = s\_mat(:); %一列一列取

可以拆成：（黄色的两次.’相互抵消了）

a=ifft(x,N);%%%ISFFT

b=a.';

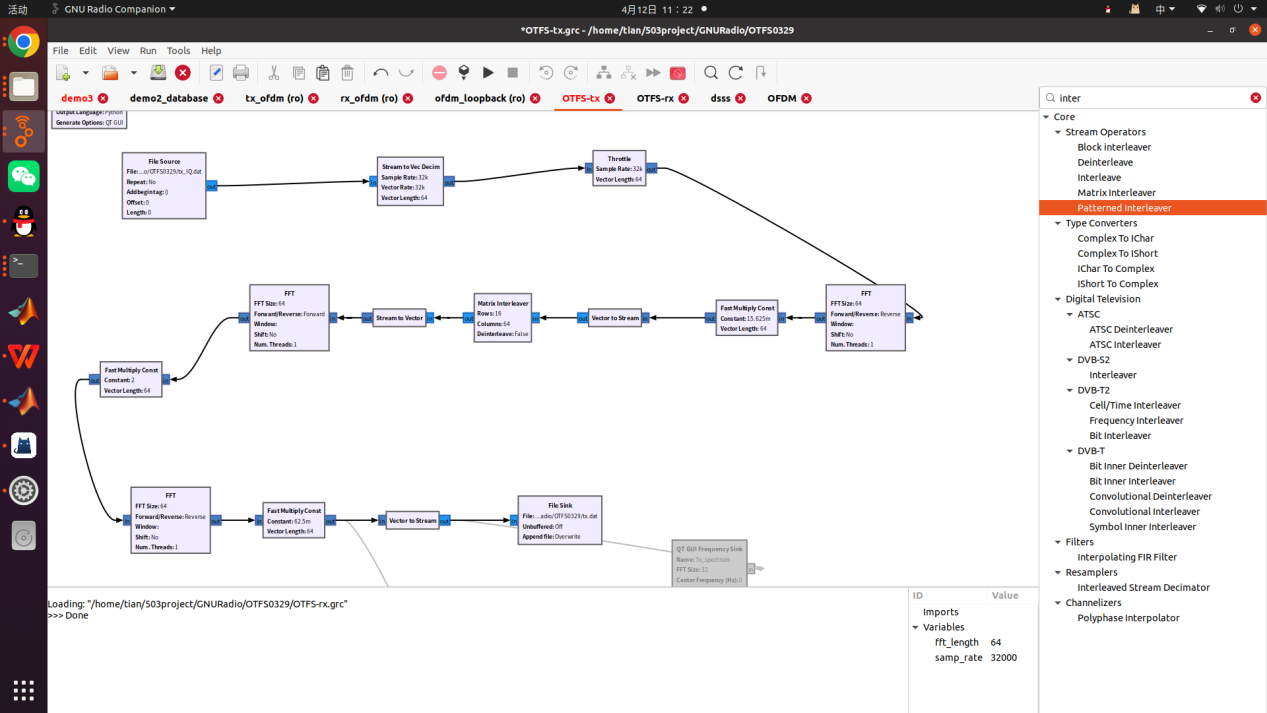
c=reshape(b,N,[]);

d = fft(c,N)/sqrt(M/N);

s\_mat = ifft(d,N)\*sqrt(M); % Heisenberg transform

s = s\_mat(:); %一列一列取

变成gnuradio形成：



**OFDM发送：**

Stream CRC32:stream后面加4byte校验位，stream tag长度会更新

packet header generator:

**digital.packet\_header\_ofdm**(occupied\_carriers, n\_syms=1, len\_tag\_key=length\_tag\_key, frame\_len\_tag\_key=length\_tag\_key, bits\_per\_header\_sym=header\_mod.bits\_per\_symbol(), bits\_per\_payload\_sym=payload\_mod.bits\_per\_symbol(), scramble\_header=False)

**digital.packet\_header\_ofdm**里面的参数：

n\_syms:header里面的ofdm符号（通常为1）如果48个数据子载波，一个packet，header就是48，如果两个packet，header就是96.

len\_tag\_key:length\_tag\_key

frame\_len\_tag\_key The tag key used for the frame length (number of OFDM symbols, this is the tag key required for the frame equalizer etc.)

num\_tag\_key The tag key used for packet numbering.

bits\_per\_header\_sym Bits per complex symbol in the header, e.g. 1 if the header is BPSK modulated, 2 if it's QPSK modulated etc.

bits\_per\_payload\_sym Bits per complex symbol in the payload. This is required to figure out how many OFDM symbols are necessary to encode the given number of bytes.

scramble\_header Set this to true to scramble the bits. This is highly recommended, as it reduces PAPR spikes.

occupied\_carrier：数据子载波

(list(range(-26, -21)) + list(range(-20, -7)) + list(range(-6, 0)) + list(range(1, 7)) + list(range(8, 21)) + list(range(22, 27)),) 52个载波-4个pilot载波=48个载波

-26 -25 -24 -23 -22 -20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -6 -5 -4 -3 -2 -1 1 2 3 4 5 6 8 9 10 11 12 13 14 15 16 17 18 19 20 22 23 24 25 26

packet header generator里面，formatter object为header\_formatter.base()，其中header\_formatter的形式为：

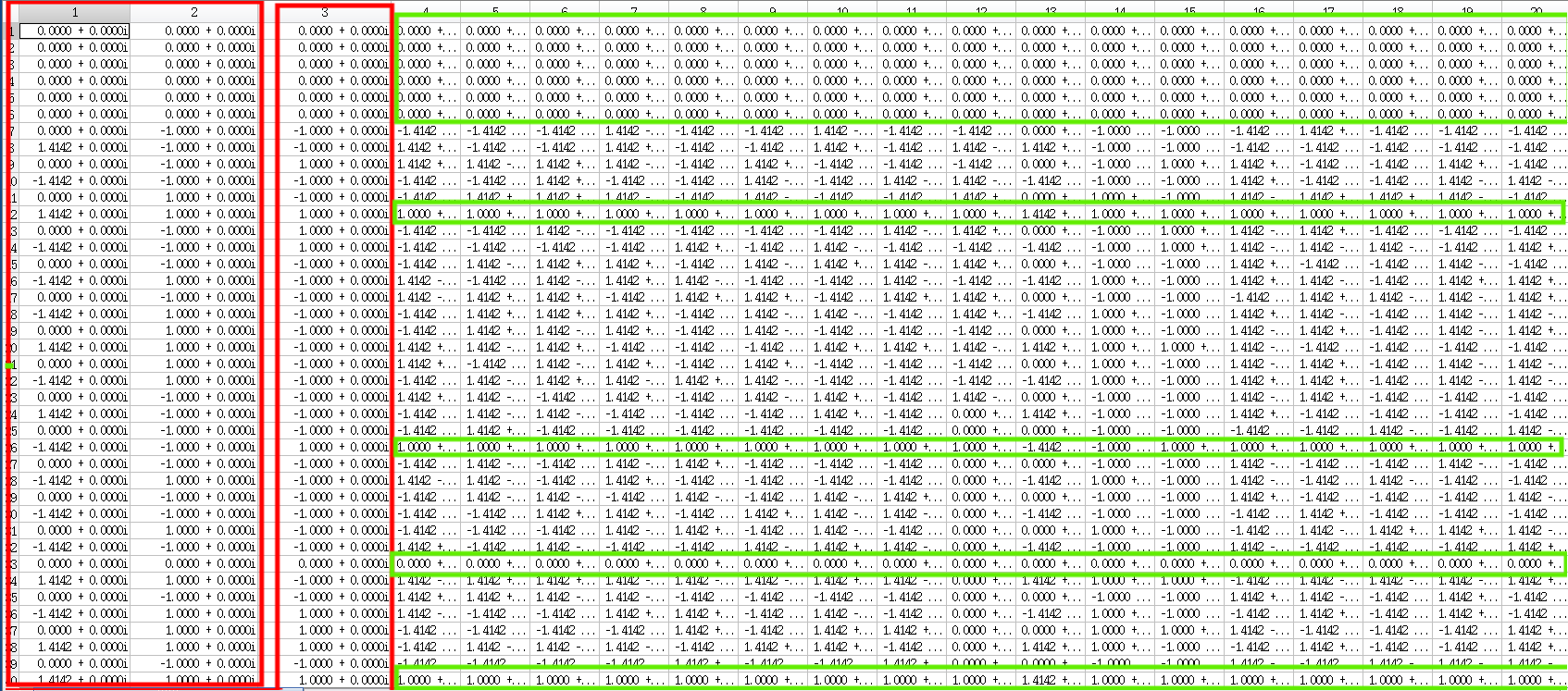
digital.packet\_header\_ofdm(occupied\_carriers, n\_syms=1, len\_tag\_key=length\_tag\_key, frame\_len\_tag\_key=length\_tag\_key, bits\_per\_header\_sym=header\_mod.bits\_per\_symbol(), bits\_per\_payload\_sym=payload\_mod.bits\_per\_symbol(), scramble\_header=False)

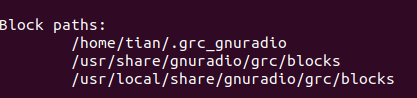
采用OFDM-carrier allocator进行载波分配的时候：变成64\*N的矩阵

前两列：sync\_word1 sync\_word2作同步

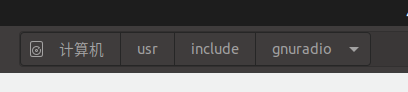
第三列：header

后面的所有列前6行和后6行做保护间隔。中间横着的4个做导频（1，1，1，-1），直流为0

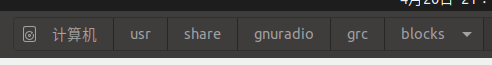


1. GNU Radio各个文件位置

.h的位置：



.yml位置：



.cmake位置：



.py文件位置：



1. 一些block的作用

variable to message:

监测一个variable，如果有改变，则输出message，内容为message variable name

python block使用：

添加message接口

self.message\_port\_register\_out(pmt.intern("out"))

# 检查消息队列是否有消息

msg = self.msgq().peek()

if msg is not None:

# 消息队列有消息，解析并打印

stream to tagged stream:打长度标签

1. 监测界面：

def main():

from whole\_system import whole\_system as gui

# print(type(gui))

app = QApplication(sys.argv)

# 创建QWidget实例

widget = QWidget()

instance=gui()

run=instance.Constel

print(run)

if \_\_name\_\_ == '\_\_main\_\_':

main()

1. 调度程序：

接口输出：

1如果是仿真工作方式：

0

2如果是实时工作方式：

工作方式：1

接收方式：自发自收0/外部接收1

2.1

1 仿真工作方式

0

2 实时工作方式自发自收

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 字典键 | 字典值 | pmt类型 |
|  | 工作方式 | int：1 | pmt.from\_long |
|  | 信号来源 | int：0（自发自收） | pmt.from\_long |
| runSelfReceiveProcess | 接收方式 | int：  0：未知信号参数接收(默认)  1：已知信号参数接收 | pmt.from\_long |
| runFileInput | 信源数据 | string | pmt.intern |
| runChannelCoding | 信道编码 | int：  0：无(默认)  1：LDPC  2：BCH | pmt.from\_long |
| runChannelInter | 信道交织 | int:  0:无(默认)  1:行列交织  2:卷积交织 | pmt.from\_long |
| runConstel | 调制方式 | int:  0:2ASK  1:2FSK  2:BPSK(默认)  3:QPSK  4:16QAM | pmt.from\_long |
| runDsss | 直序扩频 | int:  0:无(默认)  1:m序列  2:gold序列 | pmt.from\_long |
| runWave | 波形类型 | int:  0单载波(默认)  1:多载波 | pmt.from\_long |
| runFhss | 跳频扩频 | int:  0:无跳频(默认)  1:跳频图案1  2:跳频图案2 | pmt.from\_long |
| runSymbolRate | 符号速率 | float | pmt.from\_float |
| runCarrierFreq | 载波频率 | float | pmt.from\_float |
|  | 信道类型 | int:  0:AWGN  1:时选信道  2:频选信道  3:双选信道 | pmt.from\_long |
| SNR | 信噪比 | int | pmt.from\_long |
| ChanModel | 信道模型 | int:  0:瑞利信道  1:时选信道 | pmt.from\_long |
| Multipath | 信道径数 | int | pmt.from\_long |
| Delay | 信道时延 | string | pmt.intern |
| Doppler | 信道多普勒 | string | pmt.intern |

3 实时工作方式外部接收

4 离线工作方式自发自收

5 离线工作方式外部接收