地磁信息注入流程：

PX4Magnetometer文件修改：

1. 定义好需要外部注入的msg文件的内容（消息格式），如：

（Firmware\msg\rfly_ctrl.msg）

```
uint64 timestamp            # time since system start
(microseconds)        //时间戳
uint32 flags                # control flag
//控制flag
uint8 modes                 # mode flag
//模式flag
float32[16] controls        # 16D control signals
//16位控制参数
```

2. 在写故障注入的源码的头文件中添加订阅的msg的头文件引用并定义将要订阅的消息体，以便于接下来的引用，如地磁信息：

（Firmware\src\lib\drivers\magnetometer\PX4Magnetometer.hpp）

```
//省略部分代码
#include <uORB/topics/rfly_ctrl.h> //msg格式的头文件
#include <uORB/Subscription.hpp> //订阅操作相关的头文件
//省略部分代码
private:
    /*订阅外部uorb消息rfly_ctrl_s用于触发故障*/
    // 1、声明结构体参数
    rfly_ctrl_s rflydata;
    //2、订阅rfly_ctrl的uorb消息
    uORB::Subscription _rfly_ctrl_sub{ORB_ID(rfly_ctrl)};
//省略部分代码
```

3. 在要注入的源文件中添加注入操作，如：

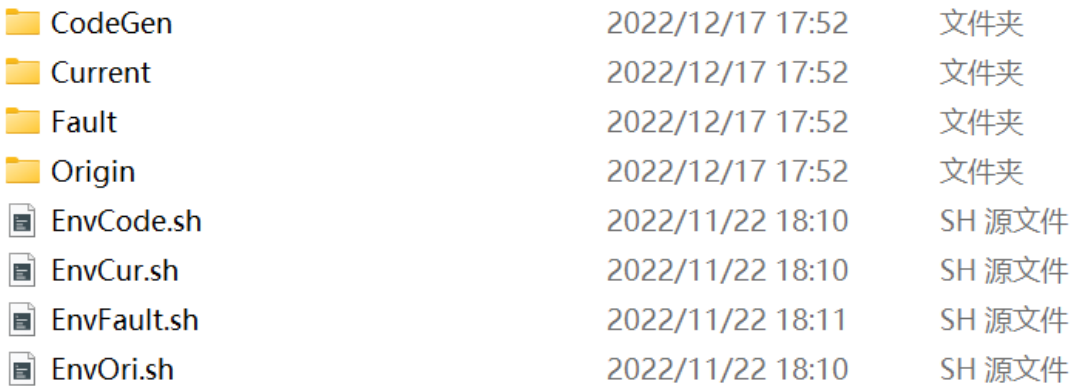
（Firmware\src\lib\drivers\magnetometer\PX4Magnetometer.cpp的update函数中）

```
//省略部分代码

    _rfly_ctrl_sub.copy(&rflydata); // 取出uorb的值 ，取出消息订阅的值

    // if(abs(rflydata.controls[0]-123455)<0.01 )
    if (int(rflydata.controls[0] - 123455) == 0) // 判断故障ID，符合
进入故障
    {
```

```
        //如果故障模式为1，则为覆盖模式，直接将输出值替换成故障注入中的值
        if (int(rflydata.controls[1] - 1) == 0)
        {
            report.x = rflydata.controls[2];
            report.y = rflydata.controls[3];
            report.z = rflydata.controls[4];
        }

         //如果故障模式为2，则为叠加模式，直接将输出值替换成故障注入中的值和传感器自身值的和
        if (int(rflydata.controls[1] - 2) == 0)
        {
            report.x = report.x + rflydata.controls[2];
            report.y = report.y + rflydata.controls[3];
            report.z = report.z + rflydata.controls[4];
        }

        //如果故障模式为0，则为拦截状态，即直接拦截传感器的值，即传感器的状态不更新，默认丢失
        if (int(rflydata.controls[1] - 0) != 0)
        {
            _sensor_pub.publish(report);
        }
    }
    else
    {   ///如果故障模式输入其它的值则为正常模式，不做处理
        _sensor_pub.publish(report);
    }

//省略部分代码
```

故障注入替换模式：

1. 一般我们要修改文件从而实现故障注入功能，但是我们不会去直接修改源码区的源码文件，因为这样很容易破坏文件结构，于是我们设计了一个修改文件备份机制，这个备份机制的文件夹路径为：C:\PX4PSP\Firmware\BkFile，一般我们需要修改或者添加源文件的操作，都可以直接在这个文件夹的内容里面操作，如你要替换某个文件，则直接从源码中找到该文件并拷贝一份至该目录下对应文件夹（Fault或CodeGen）下。
2. 这个文件夹内有着四个文件夹和四个执行脚本，文件夹名称和脚本名称一一对应：



3. CodeGen文件夹内存放是在通过Matlab/Simulink自动化代码生成时要替换的代码的源文件，例如，当你要替换某个文件从而生成故障的时候，就将你修改的源文件放置在CodeGen文件夹中，这样Matlab/Simulink编译模型的时候就会自动化替换。如果你不需要任何替换，则保持文件夹为空。
4. Fault文件夹内存放的是通过手动编译时要替换的代码的源文件，例如，当你要手动编译替换某个文件从而生成故障的时候，就将你修改的源文件放置在Fault文件夹中，然后再手动执行EnvFault.sh脚本，执行源代码替换操作，之后再通过make命令来编译生成目标文件。
5. Origin文件夹内保存着原始文件，方便用于需要撤消替换文件回归原始文件时调用，通过手动执行EnvOri.sh脚本就能撤消之前的替换动作，还原被修改的代码。
6. Current文件夹是Matlab/Simulink自动化代码生成时生成的临时文件，不需要人为改动。

发送故障注入信息流程（**UDP**模式）：

```
import time
import math
import sys

import PX4MavCtrlV4 as PX4MavCtrl

#Create a new MAVLink communication instance, UDP sending port
(CopterSim's receving port) is 20100
mav = PX4MavCtrl.PX4MavCtrler(20100)

#Turn on MAVLink to monitor CopterSim data and update it in real time.
# 可以监听数来自20100端口，主要是outHILStateData结构体
mav.InitMavLoop()
time.sleep(0.5)
```

```
# 开始监听数据来自40100端口，主要是PX4ExtMsg，来自PX4内部向外发布的数据
mav.InitTrueDataLoop()
time.sleep(3)

#第0位为故障id，第1位为故障模式，第2位及往后为故障参数
ctrls=[12345,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0]      //根据注入格式设定注入参数

# 发送SendHILCtrlMsg数据，在PX4内部产生rfly_ctrl的uORB消息
mav.SendHILCtrlMsg(ctrls)


time.sleep(3)
#Display Position information received from CopterSim
print(mav.uavPosNED) # 飞控数据来自20100端口

time.sleep(3)
print(mav.truePosNED)# 真值数据来自30100端口
```

故障测试流程：

1. 电脑连接好px4硬件，通过设备管理器查看com口，运行.bat脚本启动硬件在环仿真，输入PX4COM的数字编号
2. 飞机正常连接后，点击起飞，并确认起飞。
3. 通过地面站（QGC）的mavlink console控制台可以使用listener + msg名称的方式来监听信息，如：

   ```
   listener sensor_mag //监听（订阅）传感器信息

   listener rfly_ctrl  //监听（订阅）注入信息
   ```

4. 通过地面站（QGC）的mavlink inspector可以查看传感器的输出信息