GNU Radio 入门

GNU Radio / USRP / OpenBTS

Version 0.99

译作者：黄琳 等

7/21/2010

目　录

第1章　GNU Radio 是什么

GNU Radio 是一个通过最小程度地结合硬件（主要是 USRP），用软件来定义无线电波发射和接收的方式，搭建无线电通信系统的开源软件系统 。也就是说， 现在那些高性能的无线电设备中所遇到的数字调制问题将变成软件问题。

我们都知道 GNU 这个非常成功的自由软件项目，它包括 Linux 这样的操作系统软件，也包括大批应用软件。但 GNU 在涉及硬件开源尤其是无线射频方面还是存在着很大的盲区。 Gnu Radio 通过提供一套信号处理软件模块和相关联硬件（自由的软件，价格合理便宜的硬件）给大众，以图填充这个空白。

GNU Radio 应用程序用 Python 语言来编写，真实的信息处理过程是由 C++浮点扩展库来实现的。因此开发者可以获得实时高效的可复用的应用开发环境。虽然 GNU Radio 并不是主要用于仿真，但也可以不用真实硬件，而使用预先记录或生成的数据来开发信号处理算法。

让我们来给几个简单的例子，看看 GNU Radio 可以用来做什么：

- 学生和研究人员用它来开发物理层信号处理算法，MAC 层甚至更上层的协议。因为所有的通信协议，从上至下都是 PC 机上的软件代码。你可以像使用普通软件一样快速自如的修改、编译和运行，可以灵活地在多个协议层之间互操作。当你撰写学术论文的时候，这些真实的实验结果常常能够为你的论文增色不少。
- 创业型小公司或者学校里做横向开发项目的人，他们通常用它来开发原型设备（prototype）。比如做一个支持多种制式的家庭网关，因为所有的东西都是“软”的，所以开发起来非常快，出现问题的时候也容易修改。
- 用来做高校里的教学用实验平台。比如做通信原理实验，现在大部分实验都是用 Matlab 仿真来做的，当有了 GNU Radio，你就可以看到真正的信号星座图，频率漂移等现象。而且它可以是一个远程的平台，供很多学生同时使用。
- 业余无线电爱好者，他们用 GNU Radio 来搭建自己的电台。我猜想，它可以让你同时在多个频道上呼叫。不过我不知道这是否违反无线电使用规定。
- 黑客！这是用户中很大的一个群体。特别是 OpenBTS，也就是 GNU Radio 上的 GSM 基站开发出来之后，加上 GSM 加密的破解算法，这吸引了很多对 GSM 网络感兴趣的人。当然，相反的，反黑客的人，我们的无线电监控部门，军方的实验室，也对此很有兴趣。

GNU Radio 还可以做什么呢？发挥你的想象力吧。

本章将简单从软件无线电的基本概念开始，介绍 GNU Radio 的各个部分。[1]

[1] 本章的余下部分主要翻译节选自 Eric Blossom 的“Exploring GNU Radio”

1.1 软件无线电的基本思想

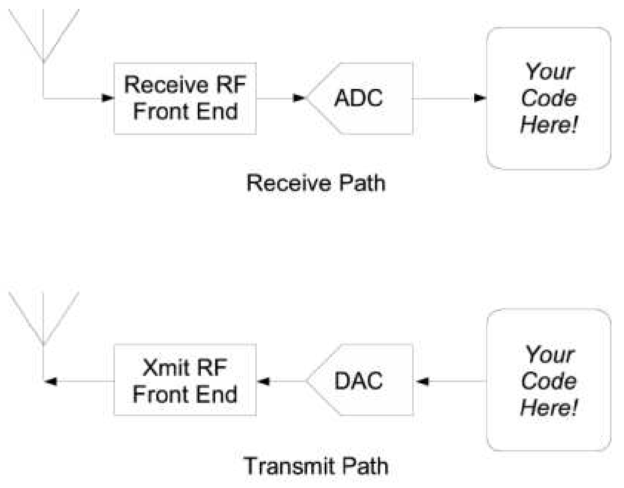

图 1-1 典型的软件无线电处理流程

上图表示一个典型的软件无线电处理流程图。为了理解无线电的软件模块，首先需要理解和其关联的硬件。在这个图中的接收路径上，能够看到一个天线，一个神奇的 RF 前端，一个模拟数字转换器 ADC 和一堆代码。ADC 是一个连接连续模拟的自然世界和离散的数字世界的桥梁。

ADC 有两个主要特性，抽样率和动态范围。抽样率是 ADC 测量模拟信号的速度，动态范围是 ADC 区别最低信号值和最大信号值的精度，这决定 ADC 数字信号输出的比特数（位数）。例如，8 位的 AD 转换器最多能代表 256 个信号层次，而一个 16 位的转换器能够代表 65536 个层次信号。总的来说，ADC 的物理特性和价格决定了抽样率和动态范围。

在我们深入研究软件之前，先来了解一些理论知识，在 1927 年，出生于瑞典的物理和电子学家 Harry Nyquist 提出了如果 AD 转换想没有混叠现象发生，那么抽样率至少是目标信号带宽的 2 倍。

混叠现象就像是车子重复的碾在过去的车轮印迹上一样，让你分不清楚是本次的还是以前碾过的车轮印迹。假设我们要处理一个低通信号，我们感兴趣的信号带宽是 0 到 fmax，按照 Nyquist 理论，抽样率必须至少是 2*fmax。如果我们的 ADC 工作在 20MHZ，但是我们想收听 92.1MHZ 的 FM 电台，我 们该怎么办呢？答案是使用 RF 前端，接收机的 RF 前端能够把它接收到的高频段信号下变频到一个低频段信号后输出。例如，我们能让 RF 前端把 90- 100MHZ 频段内的信号下变频到 0-10MHZ 的低频范围内，那么我们的 20MHZ 的 ADC 就能够派上用场了。

大多数情况下，我们把 RF 前端当作一个信号控制的黑盒子，负责处理输入信号的中心频率。举一个具体例子，一个调制解调器的调制模块能够把 50M 到 800M 之间的 6M 带宽的信号下变频到一个中心频率是 5.75MHZ 的信号输出。这个输出的中心频率通常称作中频(IF)。

按照越简单越易用的原则，RF 前端最好也能够被去掉。有一个 GNU Radio 用户已经成功的使用一个 100 英尺（译注：一英尺等于 12 英寸，合 0.305 米）的天线直接连接到一个 20M 抽样率的 ADC 上收听到了 AM 和短波广播。

1.2 软件架构

1.2.1 “Hello World”－ Dial tone

GNU Radio 提供一个信号处理模块的库，并且有个机制可以把单个的处理模块连接在一起形成一个系统。编程者通过建立一个流向图（flow graph）就能搭建成一个无线电系统。

信号处理模块是使用 C++来实现的，理论上说，信号数据流不停的从输入端口流入从输出端口流出。信号处理块 (blocks)的属性包括输入和输出的端口数，流过它们的数据的类型。

经常使用的数据流的类型是短整型(short),浮点型(float),和复数 （complex）类型。一些处理模块仅仅有输出端口或者输入端口，它们分别成为信号源(data source)和信号接收器(sink)。有的信号源从文件或者 ADC 读入数据，信号接收器写入文件或者 DAC 或者 PC 的多媒体接口。

GNU Radio 提供了超过 100 个信号处理块，并且扩展新的处理模块也是非常容易的。软件图形化接口和信号处理模块的链接机制是通过 python 脚本语言来进行的，

例 1 是一个 GNU Radio 的“Hello World”的例子。它产生两个 sine 波形并且把他们输出到声卡，一个输出到声卡的左声道，一个输出到右声道。

例子 1.输出拨号音

```
#!/usr/bin/env python
#
# Copyright 2004,2005,2007 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published
by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#
```

```
from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
from optparse import OptionParser

class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

        parser = OptionParser(option_class=eng_option)
        parser.add_option("-O", "--audio-output", type="string",
default="",
                          help="pcm output device name.  E.g., hw:0,0
or /dev/dsp")
        parser.add_option("-r", "--sample-rate", type="eng_float",
default=48000,
                          help="set sample rate to RATE (48000)")
        (options, args) = parser.parse_args ()
        if len(args) != 0:
            parser.print_help()
            raise SystemExit, 1

        sample_rate = int(options.sample_rate)
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350,
ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440,
ampl)
        dst = audio.sink (sample_rate, options.audio_output)
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))


if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

在这个例子中，我们首先调用 gr_sig_source_f 产生两个 sine 波形模块，src0 和 src1。后缀 f 表面这个信号源的数据类型是浮点型的，一个波形是 350HZ，另外一个是 440HZ，合在一起他们听起来像一个美国电话拨号音。audio_sink 是一个接收器，它把接收到的信号输入到声卡 中。

我们把 3 个信号处理模块用流向图的 connect 方法连接到一起。connect 方法有两个参数，源端点和目的端点，用来建立一个从源到目的处理模块的链路。每个端点(endpoint)有两个成员：一个信号处理模块和一个端口号。端口号表示哪个输入或者输出端口应该被连接的。

通常端点使用 python 语言的 tuple 来表示，像：（block,port_number）。当端口号是 0 时，port_number 可以被省略。例如下面的两个表示方法是一样的:

fg.connect ((src1, 0), (dst, 1))

fg.connect (src1, (dst, 1))

一旦流向图被建立了，我们调用 start 生成一个或者多个线程去运行它，按下任意键这个程序就会退出。

1.2.2 Flow graph（流图）和 Block（模块）

GNU Radio 的编程基于 Python 脚本语言和 C++的混合方式。Python 用来构造流图。C++由于具有较高的执行效率，被用于编写各种信号处理模块，如：滤波器、FFT 变换、调制 / 解调器、信道编译码模块等，GNU Radio 中称这种模块为 block。Python 是一种新型的脚本语言，具有无须编译、语法简单以及完全面向对象的特点，因此被用来编写连接各个 block 成为完整的信号处理流程的脚本，GNU Radio 中称其为 graph。

GNU Radio 的软件结构顶层是面向用户的 block 及其“粘合剂”——graph。用户除了能够开发自己的 block 之外，还可使用 GNU Radio 所包含的丰富的 block，包括各种滤波器、FFT 变换、调制 / 解调模块、时频同步模块等等，其中一些利用了 CPU 的增强指令集(如：MMX、SSE、3D Now!)进行了优化，以提高性能。

在用户用 block 和 graph 构造的应用程序下面是 GNU Radio 的运行支持环境，主要包括缓存管理、线程调度以及硬件驱动。GNU Radio 中巧妙地设计了一套零拷贝循环缓存机制，保证数据在 block 之间高效的流动。多线程调度主要用于对信号处理流程进行控制以及各种图形显示，GNU Radio 对此也提供了支持。GNU Radio 的硬件驱动包括 USRP、AD 卡、声卡等等，用户也可根据需求进行扩充。

GNU Radio 应用程序的图形化接口是使用 python 来实现的，接口能使用 python 的任何 toolkit 来实现，我们推荐使用 wxPython，它能最大化的实现跨平台应用。GNU Radio 提供由 C++到 Python 的连接机制。

GNU Radio 除了支持 Linux 的多种发行版本之外，还被移植到 Mac OS X、NetBSD 以及 Windows 等操作系统上，这也意味着它也支持多种类型的计算机系统。

1.3 硬件架构

GNU Radio 是硬件独立的。也就是说除了我们熟悉的 USRP 以外，还有其他一些硬件可以用 GNU Radio：比如说一种 cable modem tuner（型号 mc4020，用来接收 FM 广播）；另外还有一些业余无线电设备也用 GNU Radio。

如今带有单独浮点运算单元的上 GHZ 的 CPU 已经很常见了，这给台式机实现数字信号处理功能带来了可能。一个 3G 的 Pentium 或者 Athlon 处理器能够每秒处理 30 亿次浮点 FIR 运算。我们现在能够在 PC 上建立软件通讯系统，这是几年前所不敢想象的。

你的硬件要求依赖于你想做什么。总的来说，一个 1G 或者 2G 带有 256M 内存的机器应该是足够了。你也需要一些其他的模拟外设连接在你的 PC 上，包括内置 的声卡或者 96 kHz, 24-bit 高保真声卡。使用这些模拟设备，你只能处理有限的窄带信号。另外的方案是使用高速的 PCI 模拟到数字的外设，这些设备可

以达到 20M 的抽样率，但是他们是比较昂贵的，差不多是买一台 PC 的价钱。使用这些高速板，调制解调器可以作为 RF 前端来使用。

为了有一个性价比更好的外设，Ettus 等人设计了一个通用软件无线电外设（USRP）。这也是目前绝大部分 GNU Radio 用户所用的硬件。

1.3.1 USRP，从启动过程说起

USRP 是 Matt Ettus 的杰作，USRP 是一个非常灵活的 USB 设备，它把你的 PC 连接到 RF 世界。

USRP 包含一个小的母板，母板包含 4 个 12bit /64M 抽样率的 ADC，4 个 14bit/128M DAC，一个百万门的 FPGA 芯片和一个可编程的 USB2.0 控制器。每个 USRP 母板支持 4 个子板，2 个接收，2 个发射。

RF 前端是实现在子板上的，不同的子板处理不同的频率带宽。作为业余无线电使用，可以选择低能量的能够接收和发送 440MHZ 和 1.24GHZ 的子板。一个仅有接收功能的基于 cable modem 调谐的子板覆盖 50MHZ 到 800MHZ 频率范围，方便做电视信号接收的实验。

USRP 的灵活性得益于 2 个可编程的元件，通过他们和 PC 上的 host 端交互。为了对 USRP 有一个初步印象，让我们看一下它的启动过程。

USRP 本身不含有 ROM，仅仅有一个存储 VendorID 和 productID 和版本号信息的 EEPROM。当 USRP 插到 PC 的 USB 口上以后，主机上的程序通过 VID，PID 和版本号识别这是一个未配置的 USRP，主机上的程序第一步要做的是下载一个 8051 固件到 USB 控制芯片上，这个固件控制 USB 的行为。

当 USB 固件下载好后，USRP 模拟一个 USB 设备的重枚举过程，此后主机识别到一个不同的设备，VID,PID 和版本号都不同了。现在这个 USB 固件定义 USB 端口，接口和用户自定义命令。其中一个命令是 load FPGA，收到这个命令后 USB 设备就能够把 FPGA 配置 bitstream 下载到 FPGA 芯片中开始工作。

FPGA 是一个通用硬件，它的行为完全由配置的 bitstream 来决定，你可以把 bitstream 看作是一个目标码。这个 bitsteam 是由一个高级硬件描述语言编译得到的，在 USRP 里面这是由 verilog 硬件描述语言来实现的。这些代码是开源的，和其他的 GNU Radio 代码一样，是基于 GNU 的 GPL 协议的。

1.3.2 数字中频——FPGA 的职责

简单的说，USRP 上的 FPGA 的职责就是做上下变频，在数字中频和基带信号之间进行转换。

FPGA 像一个小的、高性能的并行计算机一样，可以完成你设计的任务，设计 FPGA 需要一些技能，并且如果不慎还会烧坏你的板子。还好我们已经提供一个标准的适应性很广的 FPGA 配置。

通过使用一个好的 USB 控制器，USRP 能够有 32M/s 的处理能力，USB 是半双工的，基于你的需要，USB 能够在传输和接收之间转换。

在接收模式（receiver）下，标准的 FPGA 配置能够允许你选择你感兴趣的频率，同时完成基带化和抽取滤波的工作。和 RF 前端处理方法一样，但现在做的是在数字域的采样。执行这些功能的代码叫做数字下变频转换器 DDC，利用这些处理我们能够在数字域快速的改变中心频率。

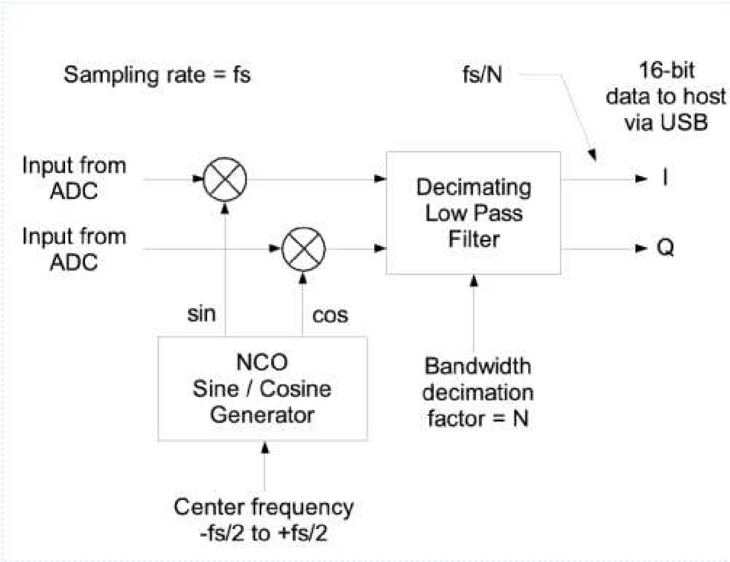

图 1-2 在 FPGA 上做下变频

在发射模式（transmitter）下，执行一个完全反过来的过程。FPGA 包含多个数字上下变频实例，根据需要这些实例连接到相同或者不同的 ADC 上面。鉴于篇幅，这里不能介绍所有的理论，在 GNU Radio 的 wiki 上有更加多的信息可以参考。

第2章　　USRP，GNU Radio 的硬件平台[2]

本章主要介绍 GNU Radio 的硬件平台——USRP。

USRP（Universal Software Radio Peripheral，通用软件无线电外设）旨在使普通计算机能像高带宽的软件无线电设备一样工作。从本质上讲，它充当了一个无线电通讯系统的**数字基带和中频部分**。

USRP 背后的基本设计理念是在主机 CPU 上完成所有波形相关方面的处理，比如调制和解调 。所有诸如数字上下变频、抽样和内插等高速通用操作都在 FPGA 上完成。

USRP 的真正价值是它能使工程师和设计师以低预算和最少的精力进行创造。为数不少的开发者和用户贡献了大量的代码库，并为软件和硬件提供了许多实际应用。灵活的硬件、开源软件和拥有经验丰富用户社区群的强强联合，使它成为您软件无线电开发的理想平台。

2.1　USRP 母板

USRP 有 4 个高速模拟数字转换器（ ADCs ），每符号 12 比特， 64M 符号/秒。另有 4 个高速数字模拟转换器（ DACs ），每符号 14 比特，128M 符号/秒。这 4 个输入和输出通道连接到 Altera 的 Cyclone EP1C12 FPGA 上。FPGA 进而连接到 USB2 接口芯片—— Cypress FX2，并接至计算机上。USRP 只通过高速 USB2.0 接口连接到计算机，不能使用 USB1.1。

因此，原则上，如果使用实采样的话，有 4 个输入和 4 个输出通道。但是如果使用复采样（IQ），可以有更大的灵活性（和带宽）。此时必须对它们进行配对，这样就能获得 2 个复输入和 2 个复输出。

[2] 本章主要内容翻译自 Firas Abbas Hamza 的“The USRP under 1.5X Magnifying Lens!”。

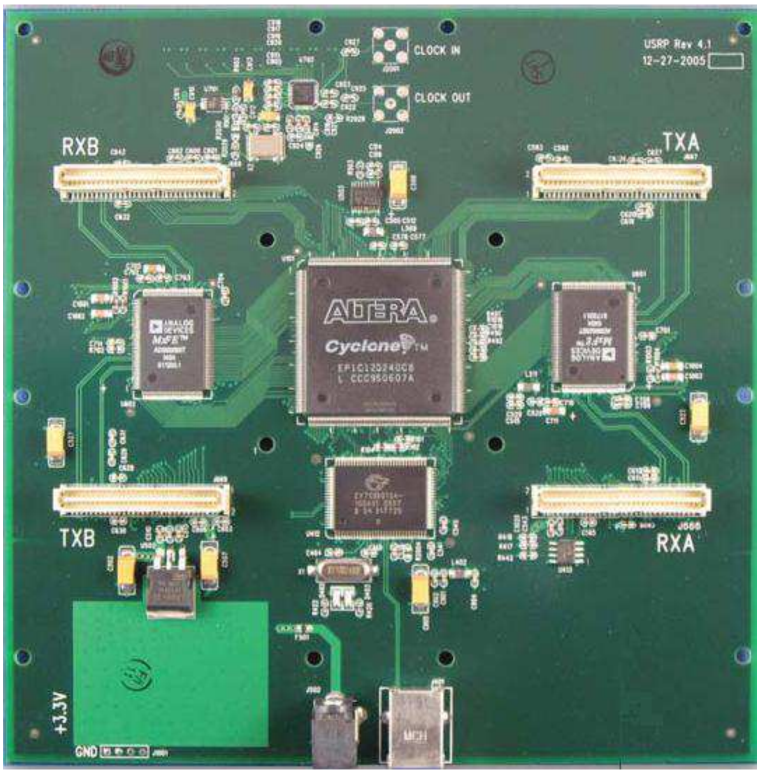

图 2-1USRP 母板

2.1.1 模数转换器部分

USRP 有 4 个高速的 12 位模数转换器。采样速率是每秒 64M 符号。从原则上讲，它可以作为数字化 32MHz 带宽。AD 转换器可以带通滤波约达 200MHz 的采样信号。如果可以接受几个分贝的损失，可以数字化高达 500 MHz 的中频频率。然而，如果采样信号的中间频率大于 32MHz ，我们将引入量化噪声，实际的有用信号带宽被映射到-32MHz 和 32MHz 之间 。有时候，这可能是有用的，例如，我们可以在没有任何射频前端的情况下收听调频广播电台。被采样信号的频率越高，抖动带来的信噪比损失越多。建议上限为 100MHz。

模数转换器的范围是 2V 峰峰值，输入是差分 50 欧姆。这就是 10mW 的功率 ，或 10dBm。在 ADCs 之前有一个可编程增益放大器（ PGA ）用于放大输入信号，以便在输入信号较弱的情况下使用到 ADCs 的整个输入范围。 PGA 最大 20dB。增益设置为零，最大输入为差分 2V 峰峰值。当设定为 20 分贝，只需要 0.2 V 峰峰值差分输入信号，便可达到最大范围。这个 PGA 是可软件编程的。

如果信号是交流耦合的，在内部缓冲打开的情况下，没有必要提供直流偏置。它将提供约 2V 的偏置。如果信号是直流耦合的，应给正负极同时提供一个 VCC/2（ 1.65V ）的直流偏置，而且要关闭内部缓冲区。ADC VREF 提供了一个 1 伏参考电压。

2.1.2 数模转换器部分

在传输路径上也有 4 个高速 14 位数模转换器。 DAC 时钟频率为 128 MS / s，所以奈奎斯特频率为 64MHz 。然而，我们可能会希望低于该频率以使滤波容易。一个有用的输出频率范围是从 DC 到 44MHz 。DACs 可为 50 欧姆或 10mW

（ 10dBm ）差分负载提供峰值 1V 的电压。DAC 之后也使用了 PGA 用于提供高达 20dB 增益。PGA 是软件可编程的。DAC 信号（ IOUTP_A / IOUTN_A 和 IOUTP_B / IOUTN_B ）是电流输出的，每个介于 0 和 20 毫安之间。它们可以通过一个电阻转换成差分电压。

2.1.3 辅助模拟 I/O 端口

有 8 个辅助的模拟输入通道连接到 10 位低速 ADC 输入（标记为 AUX_ADC_A1_A ， AUX_ADC_B1_A ， AUX_ADC_A2_A ， AUX_ADC_B2_A ， AUX_ADC_A1_B ， AUX_ADC_B1_B ， AUX_ADC_A2_B ，和 AUX_ADC_B2_B ），它们可以被软件读取。这些 ADCs 可以转换高达 1.25MS/S，其带宽约 200KHz。这些模拟通道可用于感知 RSSI 信号水平，温度，偏置水平等等。

此外，有 8 个模拟输出通道连接 8 位低速 DAC 输出。他们是 AUX_DAC_A_A， AUX_DAC_B_A， AUX_DAC_C_A， AUX_DAC_A_B， AUX_DAC_B_B 和 AUX_DAC_C_B 。这些 DACs 可用于提供各种控制电压，如外部可变增益放大器控制。此外，还有两个额外的 DACs （标记为 AUX_DAC_D_A 和 AUX_DAC_D_B ），这是由 12 位 Sigma - Delta 调制器与外部简单的低通滤波器构成。

USRP 母板连接器（ RXA 和 TXA）共享一组 4 个模拟输出通道（ 从 AUX_DAC_A_A 到 AUX_DAC_D_A 用于 RXA 和 TXA），每个有 2 个独立的模拟输入通道（ AUX_ADC_A1_A 和 AUX_ADC_B1_A 用于 RXA 以及 AUX_ADC_A2_A 和 AUX_ADC_B2_A 用于 TXA） 。RXB 和 TXB 共用他们另外的独立的一组。如果必要，另有 AUX_ADC_REF 可以提供增益设置的一个参考等级。

2.1.4 辅助数字 I/O 端口

USRP 母板有一个高速 64 位数字 I / O 端口。这些被分为二组（32 位用于 IO_RX，32 位用于 IO_TX ） 。这些数字 I / O 引脚连接到子板接口连接器（ RxA ，TxA， RxB 和 TxB ） 。所有这些连接器有 16 位数字 I / O 位。这些信号可以由软件通过读/写特殊的 FPGA 寄存器来控制，而且每个都可以被独立配置为数字输入或数字输出。

其中一些引脚来用于控制所安装子板上的特定操作，如控制选择接收射频输入的端口，在自动发送/接收模式控制不同的 Tx 和 Rx 部件的供电电源，合成器锁定检测等。它也可被用于实现 AGC 处理。当连接到逻辑分析仪时，它非常有助于 FPGA 上的调试。

2.1.5 FPGA

对 GNU Radio 用户来说，或许最重要的部分是理解 USRP FPGA 上所发生的事情。如下图所示，所有的 ADCs 和 DACs 都连接到 FPGA。这块 FPGA 在 USRP 系统中起着关键作用。基本上他们所做的是执行高带宽下的数学运算，并减少数据传输速率，使其至少可以在 USB2.0 上传送。FPGA 连接到 USB2 接口芯片

——Cypress FX2 。通过 USB2 总线，所有（ FPGA 电路及 USB 微控制器）都是可编程的。

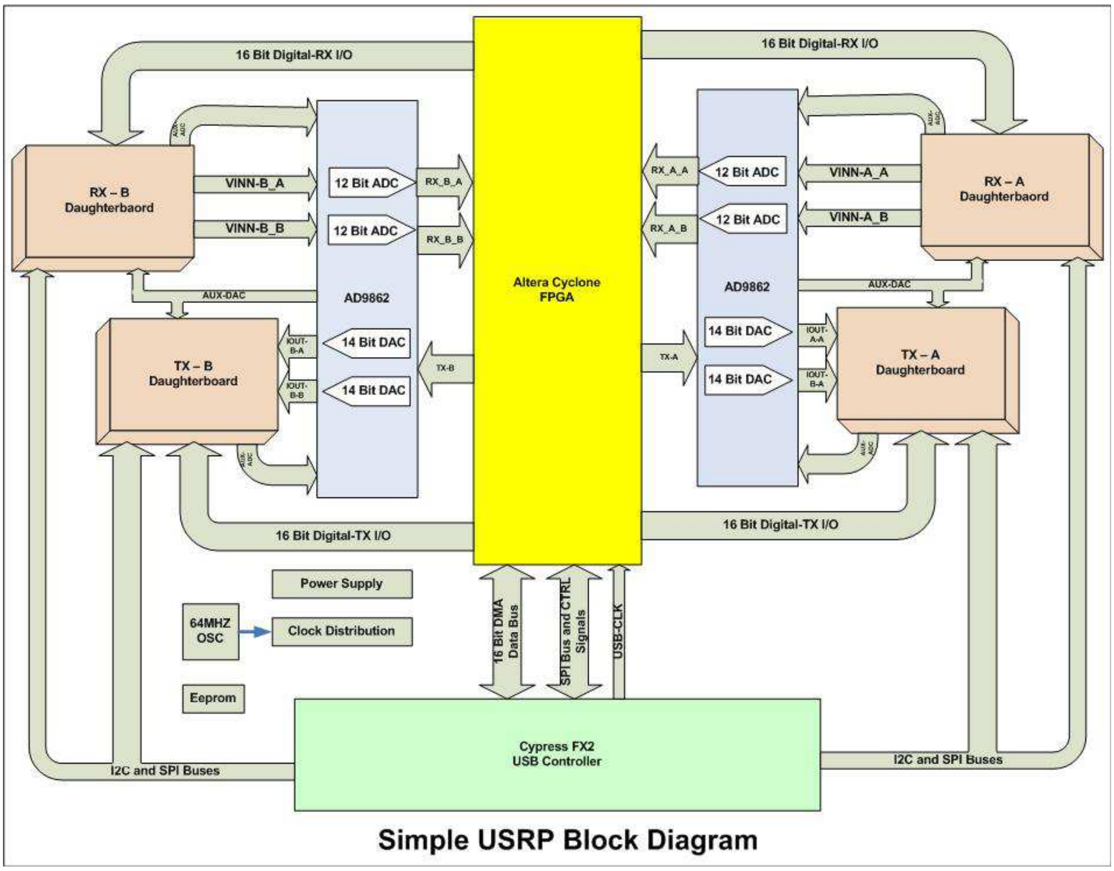

图 2-2USRP 的模块结构图

标准的 FPGA 配置包括由 4 级级联积分梳状滤波器（ CIC ）实现的数字下变频器（ DDC）。CIC 滤波器是只使用加法器和延时器的非常高性能的滤波器。对于频谱成形和带外信号抑制，也有一个 31 抽头半带滤波器与 CIC 滤波器级联，形成完整的 DDC 状态。标准 FPGA 配置包含 2 个完整的数字下变频器（ DDC 的）。另有一种配置带 4 个 DDCs 但没有半带滤波器，从而拥有 1、2 或 4 个不同的接收信道。

在 4 DDC 的实现中，我们在接收路径上有 4 个 ADCs 和 4 个 DDCs。每个 DDC 有两个输入 I 和 Q。每个 ADCs 都可以连接到 4 个 DDCs 的任何一个的 I 或者 Q 支路的输入。这样可以在同一个 ADC 采样流中有多种信道选择。

下图为 USRP 数字下变频器的框图。

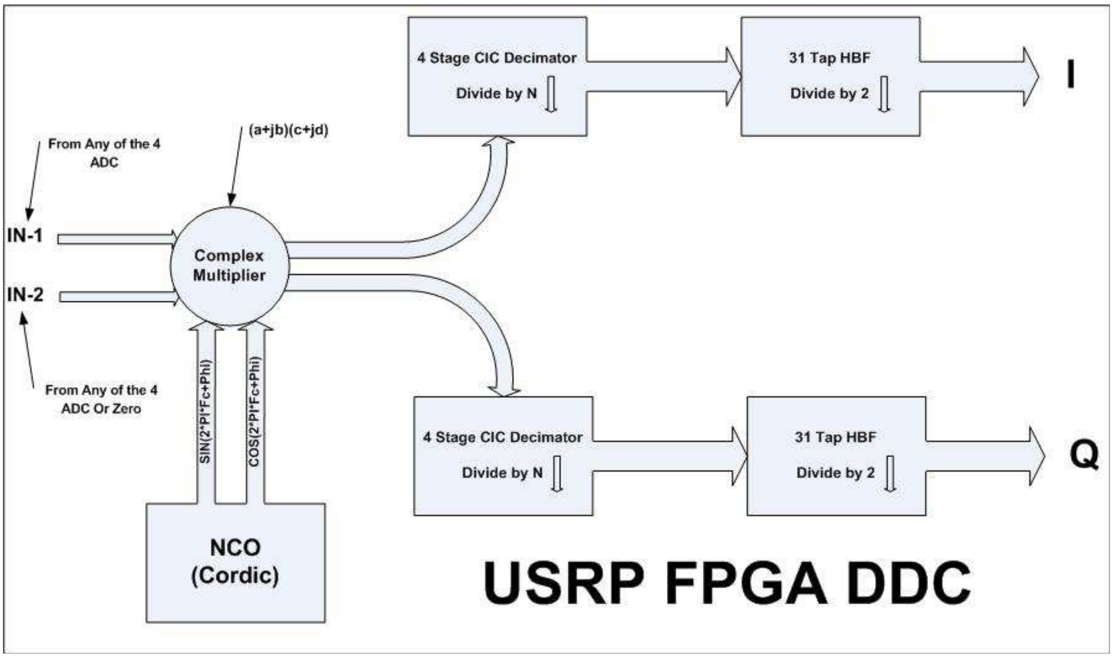

图 2-3USRP 数字下变频器

现在，让我们看看数字下变频器（ DDC）。首先，它把信号从中频波段转换到基带。第二，它采样信号，使数据传输速率可以适应 USB 2.0 以及计算机的计算能力。复输入信号（ IF ）乘以固定频率（通常也是中频）指数信号。由此产生的信号也是复的，而且集中在 0 频。然后，我们以因子 N 对信号进行抽样。抽样可视为一个低通滤波器接一个下采样器。假设抽样因子是 N。如果我们看一下数字频谱，低通滤波器挑出带宽[-Fs/N, Fs/N],，然后下采样器去掉了从[-Fs, Fs]到[-Fs/N, Fs/N]的频谱。所以实际上我们已经以因子 N 缩小了有用数字信号的带宽。

关于带宽，我们可以在 USB 上保持 32MB/sec 。USB 接口上发送的所有符号都是以 16 位有符号整数组成的正交格式，比如 16 位 I 和 16 位 Q 数据（复信号）意味着每个复采样 4 字节。这导致 USB 上的符号速率为 8M 符号/秒（ 32MByte 每秒/4 字节 ）。由于使用复处理，根据奈奎斯特准则，这将提供最大有效总频谱带宽约为 8MHz。当然，我们可以通过改变采样率选择更窄的范围。例如，假设我们要设计一个调频接收器，调频电台的带宽一般为 200 kHz。因此，我们可以选择抽取因子为 250 ，则 USB 上的数据传输速率是 64MHz / 250 = 256 kHz，这非常适合 200 kHz 的带宽而不会丢失任何频谱信息。抽样率必须在[8 ， 256]之间 。最后，复正交信号通过 USB 进入电脑。那就是软件的世界了！

请注意，当有多个信道时（最多 4 个） ，信道被交织。例如，4 信道时，USB 上的发送序列将是 I0 Q0 I1 Q1 I2 Q2 I3 Q3 I0 Q0 I1 Q1……等等。在多个接收信道（ 1,2 或 4 ）情况下 ，所有输入信道必须是相同的数据速率（即同样的抽样率） 。

发射路径上的情况几乎完全一样，除了它是依次反过来出现。我们要发送基带正交信号给 USRP 板子。数字上变频器（DUC）将对信号进行内插，上变频到中频频段，并最终通过 DAC 发送。

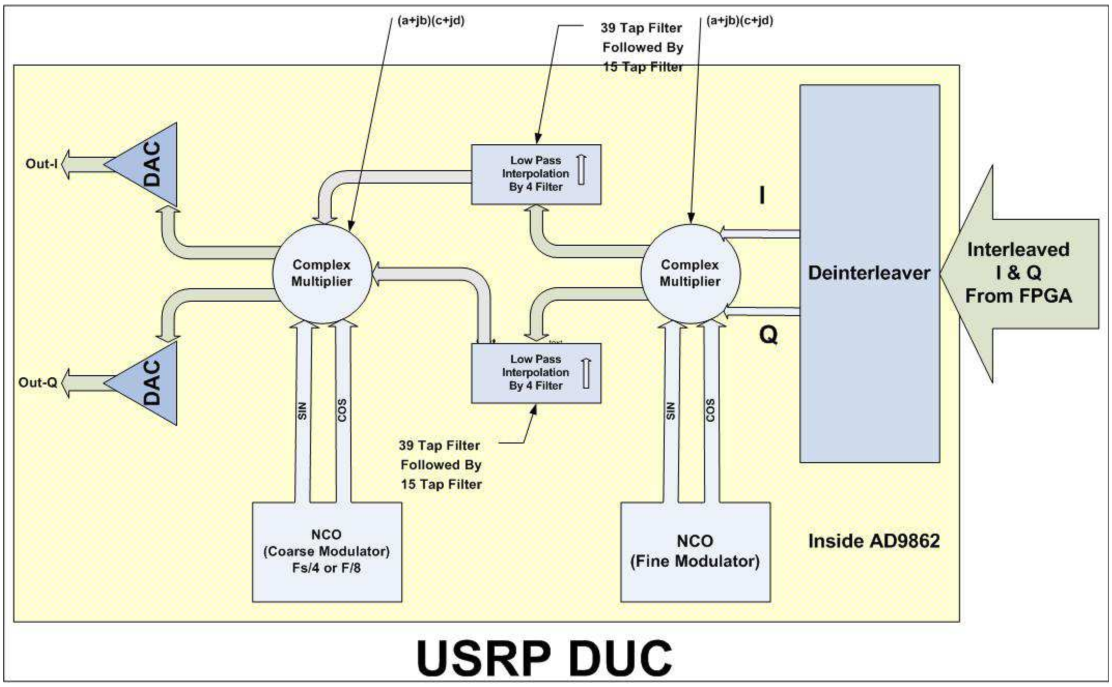

图 2-4USRP 数字上变频器

传输端的数字上变频器（DUC）实际上是包含在 AD9862 CODEC 芯片里，而不是在 FPGA 里（如上图所示）。FPGA 上的唯一传输信号处理模块是 CIC 插值器。插值输出可发送到 4 个 CODEC 输入的任何一个。

在多发送信道（1 或 2）情况下，所有输出信道必须是相同的数据速率（即同一插值比例）。请注意，发送速率可能不同于接收速率。USRP 可在全双工模式下工作。在这种模式下，发送和接收端是完全独立的。唯一要考虑的是，总线上合并后的数据速率必须为 32MB/s 或更低。

2.2 USRP2 母板

USRP2 基于 USRP 的成功经验，以非常低的价格提供更高的性能和更大的灵活性。更高速度和更高的精度 ADC 和 DAC 允许使用更宽波段的信号，增加了信号的动态范围。针对 DSP 应用优化了的大型现场可编程门阵列（FPGA）可以在高采样率下处理复杂波形。千兆以太网接口，使应用程序可以使用 USRP2 同时发送或接受 50 MHz 的射频带宽。在 USRP2 中，FPGA 出现了诸如数字上变频器和下变频器等高采样率处理器。较低采样率的操作可在主机电脑上，甚至可以在具有 32 位 RISC 微处理器和有很大用户设计自由空间的 FPGA 上做。更大的 FPGA 使得 USRP2 可以在没有电脑主机的情况下作为一个独立的系统运行。USRP2 的配置和固件被存储在一个 SD 闪存卡里，无需特别的硬件就可以轻松编程。

多个 USRP2 系统可以连接在一起形成最多可达 8 天线 MIMO 的全相关多天线系统。主振荡器可以被锁定到一个外部参考，并有一个每秒 1 个脉冲（1PPS）的输入用于对精确定时有需求的应用。

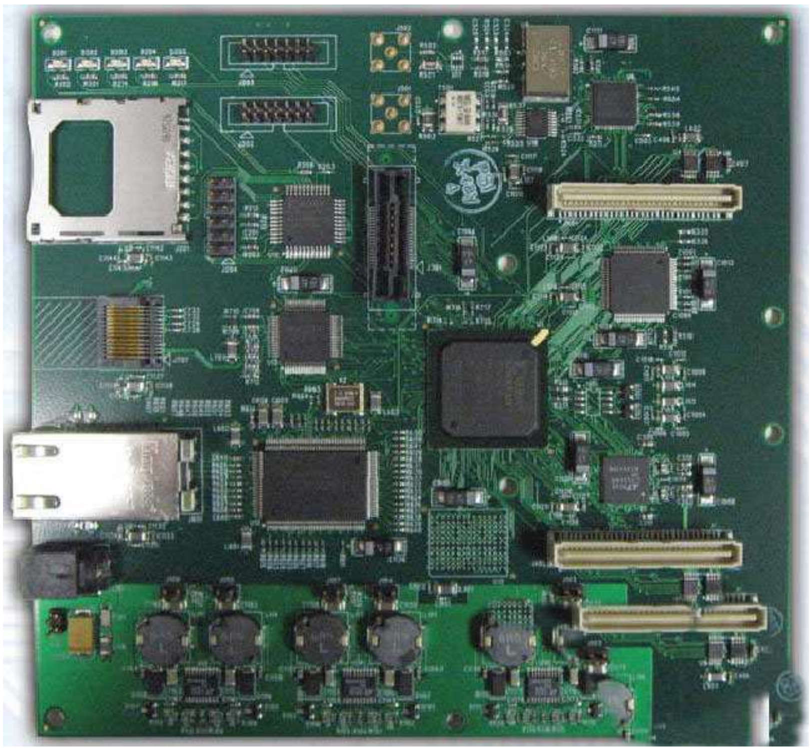

图 2-5 USRP2

USRP2 的主要特性：

- ✓ 两个 100 MS/s 的 14 位模数转换器
- ✓ 两个 400 MS/s 的 16 位数模转换器
- ✓ 可编程控制抽样率的数字下变频器
- ✓ 可编程控制插值率的数字上变频器
- ✓ 千兆以太网接口
- ✓ 2 Gbps 的高速串行接口用于扩展
- ✓ 能处理的信号带宽高达 100 MHz
- ✓ 模块化的架构，可以支持更多的射频子板
- ✓ 附属的模拟和数字 I/O 支持复杂的无线电控制，例如 RSSI 和 AGC
- ✓ 多达 8 天线的全相关多信道系统（支持 MIMO）
- ✓ 1 兆字节的板载高速 SRAM

2.3 子板

母板有四个插槽，可以插入 2 个基本接收子板和 2 个基本发送子板，或者 2 个 RFX 板子。子板是用来装载 RF 接收接口或者调谐器和射频发射机的。有 2 个标注为 TXA 和 TXB 的插槽用于连接 2 个发送子板，相应的，有 2 个标注为 RXA 和 RXB 的接收子板插槽 。每个子板插槽 可以访问 4 个高速 AD/DA 转换器其中的 2 个（ DAC 输出用于发送， ADC 输入用于接收） 。

这使得每个使用实（不是正交）采样的子板有 2 个独立的射频部分，和 2 个天线（ 整个系统一共有 4 个） 。如果使用复正交采样，每个板子可支持一个单一的射频部分，整个系统一共 2 个。通常，我们可以看到每个子板有两个 SMA 连接器。我们通常会使用它们连接输入或输出信号。USRP 母板上没有提供抗混叠或重建滤波器。这样可以在子板频率规划时获得最大的灵活性。

每个子板有一个板载 I2C 的 EEPROM （ 24LC024 或 24LC025 ）使系统能够识别子板。这使得主机软件能够根据所安装的子板自动设置合适的系统。EEPROM 也可以保存一些校准值比如直流偏置或者 IQ 不平衡(IQ unbalance)。如果这个 EEPROM 没有被编程，每次 USRP 软件运行时会打印出一个警告消息。

每个发送子板有一对差分模拟输出，其更新速率为 128MS/s。信号（ IOUTP_A / IOUTN_A 和 IOUTP_B / IOUTN_B ）为电流输出。同样每个接收子板也有 2 个差分模拟输入（ VINP_A / VINN_A 和 VINP_B / VINN_B ），其抽样率为 64MS/s。

2.3.1 Basic TX/RX 子板

1 MHz - 250 MHz 发射机和接收机。

BasicTX 和 BasicRX 用做外部射频前端的中频（IF）接口。ADC 输入和 DAC 输出直接变压器耦合到 SMA 连接器（50 欧姆阻抗）而不通过混频器、滤波器或者放大器。

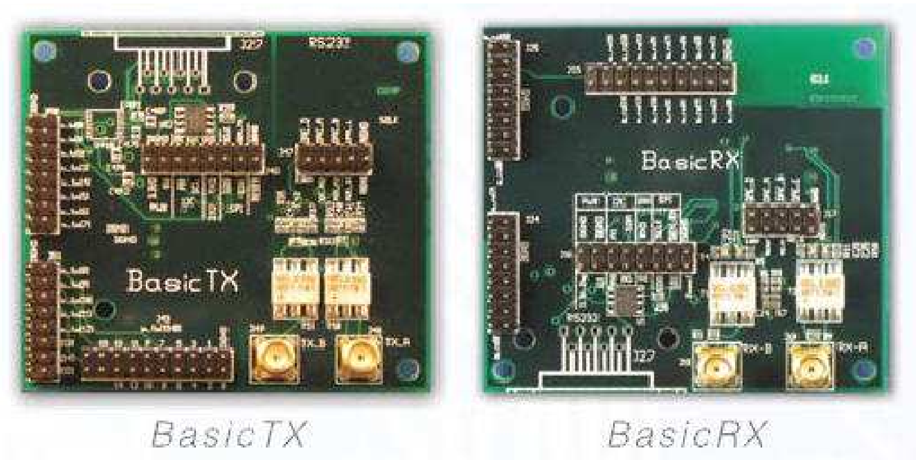

图 2-6 BasicTX 和 BasicRX

BasicTX 和 BasicRX 可以直接访问子板接口上的所有信号（包括 16 位数字 I/O，SPI 和 I2C 总线以及低速 ADC 和 DAC）。每个 Basic TX/RX 板子带有逻辑分析仪连接器用于 16 个通用 IO。这些引脚可以通过访问内部信号来帮助您调试 FPGA 设计。

2.3.2 低频发射/接收子板

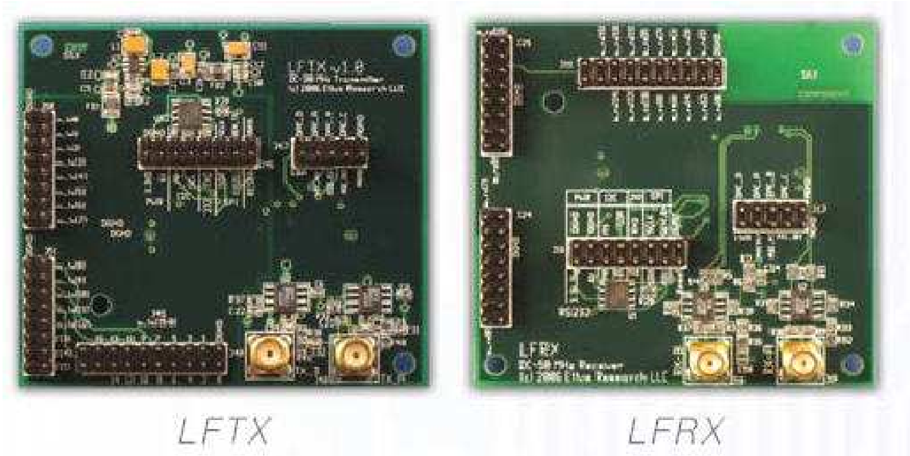

图 2-7 LFTX 和 LFRX

LFTX 和 LFRX 与 BasicTX 和 BasicRX 很相似，主要有两个不同，由于 LFTX 和 LFRX 使用差分放大器而不是变压器，它们的频率响应可以达到直流。LFTX 和 LFRX 也有一个 30 MHz 低通滤波器用于抗锯齿化。

2.3.3 TVRX 子板

图 2-8 TVRX

这是一个只有接收功能的子板。它是基于一个电视调谐器模块的完整的甚高频（VHF）和超高频（UHF）接收系统。射频频率范围从 50MHz 到 860MHz ，IF 带宽为 6MHz 。

所有的调谐和自动增益控制（AGC）功能可以通过软件控制。典型的噪声系数为 8dB。这是唯一不支持 MIMO 的子板。

注：TVRX 是唯一不支持 MIMO 的子板

2.3.4 DBSRX 子板

图 2-9 DBSRX

类似 TVRX 子板，该板也是只有接收功能。它是一个从 800 MHz 到 2.4 GHz 的完整接收系统，其噪声系数为 3-5dB。DBSRX 有一个软件可控的窄至 1 MHz 或宽至 60 MHz 的信道滤波器。 DBSRX 支持 MIMO，并能通过 SMA 为有源天线供电。

DBSRX 频率范围涵盖了许多知名波段，包括所有 GPS 和伽利略波段、902-928 MHz ISM 波段、蜂窝和 PCS、氢和羟基射电天文学波段、DECT（Digital Enhanced Cordless Telecommunications, 数字增强型无绳电话通信）等更多波段。

2.3.5 RFX 系列子板

RFX 系列子板是一个完整的 RF 收发系统。他们拥有独立的用于发送和接受的本地振荡器（射频合成器），这可以支持分频操作。此外，它有一个内置的收发开关，发送和接收信号可以使用同一射频端口（连接器），或者在只能接收时可以使用辅助接收端口。

RFX 子板的主要特性如下：

- ✓ 30 MHz 收发带宽
- ✓ 全同步设计， 支持 MIMO
- ✓ 所有功能可由软件或 FPGA 控制
- ✓ 接收机和发射机的独立本地振荡器（LO）使其支持分频运行
- ✓ 小于 200 微秒锁相环锁定时间，可用于跳频 PLL（Phase Lock Loop，锁相环）
- ✓ 内置收/发开关
- ✓ 发射机和接收机使用同一连接器或使用辅助接收机端口
- ✓ 16 个数字 I / O 线来控制外部设备比如天线开关等

- ✓ 内置的模拟 RSSI（Received Signal Strength Indication 接收信号强度指示）测量
- ✓ 70 分贝的 AGC 范围
- ✓ 可调发射功率
- ✓ 支持全双工功能（有某些限制）

RFX 系列子板有以下型号：

- WBX0510

频率范围：50 MHz-1 GHz

发射功率：100mW (20dBm)

WBX0510 的频率范围覆盖许多知名波段，包括广播电视、公共安全、陆地移动通信、低功耗免许可设备、无线传感器网络、手机，以及五个业余无线电波段。

- RFX900

频率范围：750-1050 MHz

发射功率：200mW (23dBm)

RFX900 装配了一个 902-928 MHz 的 ISM 波段滤波器用于过滤强的带外信号（如寻呼机），这个滤波器可以很容易的绕过，以便可在整个频率范围内使用，与除了 ISM 波段之外的蜂窝、寻呼、motes 和双向无线电一起使用。

- RFX1200

频率范围：1150-1450 MHz

发射功率：200mW(23dBm)

覆盖导航、卫星和业余波段

- RFX1800

频率范围：1.5-2.1 GHz

发射功率：100mW(20dBm)

覆盖 DECT、US-DECT 和 PCS 频段（包括免许可波段）

- RFX2400

频率范围：2.3-2.9 GHz

发射功率：50mW(17dBm)

RFX2400 带有一个 ISM 波段（2400-2483 MHz）附近的带通滤波器。该滤波器可被轻易绕过以便覆盖全部频率范围。

- XCVR2450

频率范围：2.4-2.5 GHz 和 4.9-5.9 GHz

发射功率：100mW(20dBm)

XCVR2450 覆盖 2.4 GHzISM 波段和整个 4.9-5.9 GHz 波段，包括公共安全、UNII（Unlicensed National Information Infrastructure，免许可国家信息基础设施）、ISM 和日本无线波段。

2.4 电源

USRP 由 6V 4A 交流/直流电源转换器供电。该转换器能够在 90 - 260VAC，50/60Hz 运作，因此应可在任何国家使用。如果要使用其他电源，连接器是一种标准的 2.1mm/5.5mm 直流电源连接器。USRP 主板本身只需要 5V 的电源，但总体需要 6V 的电源为子板供电。因为它提取约 1.6A 固定为 2 块子板供电。可以通过观察 LED 的闪烁检查连接到 USRP 的电源。如果 LED 不闪烁，检查所有的电源连接，并检查保险丝的连接（ F501 ，靠近电源连接器的地方）。如果需要更换保险丝，请注意保险丝的大小为 0603，额定电流 3 安培。

当首次上电，USRP 上的 LED 应闪烁约 3-4 次每秒。这表明，处理器正在运行，并把设备置于低功耗模式。一旦固件下载到 USRP ，LED 的闪烁速度将放慢。

2.5 时钟同步问题

整个收发器链拥有一个同步的时钟，可以建立 MIMO 系统，智能天线，相控阵和干涉仪。以下是一些如何使用它的注意事项。

2.5.1 同步所有子板本地晶振

如果您的系统有多个子板并希望得到相干相位，根据你的子板，你得采取这些步骤：

- BasicRX， BasicTX， LFRX， LFTX

这些板子没有本地晶振，所以没有必要。如果客户有自己的外部射频部分，只能靠客户自己来实现同步。

- TVRX -TVRX 不适合同步的多天线应用。

TVRX 有自己的振荡器。没有办法使这些板相位相干。

- DBSRX

DBSRX 始终使用主 USRP 时钟，因此它始终是相位相干的。

- Flex400， Flex900， Flex2400

如果是 USRP 版本 Rev 4 的板子，要修改这些板子才能用于相干应用（最近的板子不需要这些改动，只有早期的 RFX 板需要）

1. 移动 R64 到 R84， 移动 R142 到 R153。这将禁用子板时钟
2. 移动 R35 到 R36， 移动 R117 到 R115。
3. 这将连接子板到母板上的时钟。这些都是 0 欧姆的电阻，因此，如果您丢了一个，只须用合适的垫片短路。

4. 把板子插进 USRP 的 A 侧并执行下列命令之一以重编 dboard 的 EEPROM：
    - usrp/host/apps/burn-db-eeprom -A -t flex_400_mimo_b --force
    - usrp/host/apps/burn-db-eeprom -A -t flex_900_mimo_b --force
    - usrp/host/apps/burn-db-eeprom -A -t flex_2400_mimo_b --force

2.5.2 同步多个 USRP

为了有效地同步多个 USRP 板，有两个因素需要考虑：时钟同步（这是下面要详细列出的）；板间符号的匹配。

在多个 USRP 之间同步时钟，必须把它们转换为使用一个共同的参考时钟。操作如下：

- USRP 版本 4

USRP 版本 4 引入一些改进，可以使一个板子作为主时钟，所有其他板子作为从属。

主时钟板子：

1. 焊一个 SMA 连接器到 J2002 。这是主时钟输出。焊 SMA 连接器时要小心，不要破坏了从 J2002 以 R2028 的微细连接。 请注意，一旦焊上了，您将无法在 RXB 插槽上插 TVRX 了。

从板：

1. 焊一个 SMA 连接器到 J2001。这是时钟输入。焊 SMA 连接器时要小心，不要破坏了从 J2001 以 C927 的微细连接。
2. 移动 R2029 到 R2030 。这将禁用板载时钟。R2029/R2030 是一个 0 欧姆的电阻。
3. 移动 C925 到 C926 。
4. 移除 C924 。
5. 如果您想串联另一个 USRP 板子到这个上，您可以使用 J2002 以提供时钟输出。

第3章　GNU Radio 安装

首先要强调一下 GNU Radio 的网站，永远是最标准的参考文献。安装指南、编程指南和 Maillist 是最有用的几个地方之一。

3.1　安装需求

一个最小的 GNU Radio 开发环境包括一台主机（台式机或笔记本均可），至少一套含 USRP 母板（Motheroard）的 USRP1-PKG 或者 USRP2-PKG，至少一块子板（Daughterboard）。

注意：

1. 如果搭配 USRP1 使用，USB 接口必须是 USB2.0。
2. 如果搭配 USRP2 使用，网卡必须是千兆以太网卡（1G）

3.2　Ubuntu 下安装

如果你是个 Linux 新手，那推荐你用 Ubuntu 吧，最傻瓜的 Linux 的系统，也是目前 GNU Radio 用户中使用最普遍的。

Ubuntu 如何安装在此就不详述了。

下面的步骤，讲述了一种最简单快速的安装方法：以在 Ubuntu9.04 上安装 GNU Radio 3.2.2 为例。（英文链接见：http://gnuradio.org/redmine/wiki/gnuradio/UbuntuInstall）

- 第一步：使用 apt-get 安装需要的库（把网页上长长的命令直接 copy 过来就好了）

```
sudo apt-get -y install swig g++ automake1.9 libtool python2.5-dev fftw3-dev \
libcppunit-dev libboost1.35-dev sdcc-nf libusb-dev \
libsdl1.2-dev python-wxgtk2.8 subversion git guile-1.8-dev \
libqt4-dev python-numpy ccache python-opengl libgsl0-dev \
python-cheetah python-lxml doxygen qt4-dev-tools \
libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools
```

- 第二步：建一个文件夹，使用 svn 获得最新的 gnuradio 源代码。

```
svn co http://gnuradio.org/svn/gnuradio/trunk gnuradio
```

或者你可以去下载一个源码的压缩包，在这里：

ftp://ftp.gnu.org/gnu/gnuradio/gnuradio-3.2.2.tar.gz

解压缩就可以了。

- 第三步：进入源代码的根目录（/gnuradio-3.2.2），编译安装。依次执行以下命令。

```
./bootstrap
```

```
./configure
make
make check
sudo make install
```

到这里，可以说 GNU Radio 就装好了。还有其他一些操作可以让你更方便的使用。比如使非 root 用户可以使用 USRP，如果不做这一步，那么在每次执行的时候加 sudo 也可以。

```
sudo addgroup usrp
sudo usermod -G usrp -a <YOUR_USERNAME>
echo 'ACTION=="add", BUS=="usb", SYSFS{idVendor}=="fffe", SYSFS{idProduct}=="0002", GROUP:="usrp", MODE:="0660"' > tmpfile
sudo chown root.root tmpfile
sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
```

使改动生效

```
sudo udevadm control --reload-rules
```

or

```
sudo /etc/init.d/udev stop
sudo /etc/init.d/udev start
```

如果没有生效，重启计算机

3.3 Fedora 下安装

Fedora 的安装指南在这里

http://gnuradio.org/redmine/wiki/gnuradio/FedoraInstall

现在可以直接在线安装了，所以最简单的办法就是 yum 一下。

```
yum install gnuradio usrp
```

如果你嫌 yum 太慢或者版本不够新的话，就下载源代码安装。

给一个以前在 Fedora10 上安装的例子吧，有点 old 了，仅供参考。

安装过程：

- 首先，光盘安装 FC10。然后，允许 root 登录

（懂得如何使 root 登录的，请自动跳过下面部分。）

第一步：利用其他用户进行登陆

第二步：打开终端->运行 su->键入密码后

键入命令 gedit /etc/pam.d/gdm

然后会弹出文本编辑器（此处注意不能直接去目录查找文件，然后用文本编辑器打开，因为目前你是没有权限修改这个文件的）

第三步：将 user!=root 的那一行注释

注释的语法如下

#auth requir................................

第四步：保存文件退出，注销系统，切换用户即可用 root 登录

- 设置软件源。如果不能直接连接到 Fedora 的源，那就需要设置 yum 代理。

（懂得设置 yum 代理，请自动跳过）

修改 /etc/yum.conf。在 http 代理加入下面这一行

proxy=http://IP 地址:端口

- 安装必须的库

```
$ yum groupinstall "Engineering and Scientific" "Development Tools"
$ yum install fftw-devel cppunit-devel wxPython-devel libusb-devel \
  guile boost-devel alsa-lib-devel numpy gsl-devel python-devel pygsl \
  python-cheetah python-lxml
```

- 安装 sdcc

```
$ yum install sdcc
$ export PATH=/usr/libexec/sdcc:$PATH
```

- 下载源代码，安装 gnuradio

下载链接在这里

http://gnuradio.org/redmine/wiki/gnuradio/Download

把文件包解压缩以后，进入源代码根目录。

```
./bootstrap
./configure
 make
 sudo make install
```

3.4 *装好之后可以做的第一件事*

进入/usr/local/bin 目录，这里有一些 GNU Radio 的可执行程序。你可以逐个实验一下。其中有些是需要连接 USRP 才能运行的，有些不需要。

3.4.1 如果你有 USRP

3.4.1.1运行 usrp_fft.py

首先连上 USRP，插上一块子板，比如我们插上一块 RFX2400 子板。

$./usrp_fft.py –f 2500M –R A

其中-f 选项后面跟接收频段的中心频率。-R 选项后面跟用 USRP 母板上的哪一侧子板接收，缺省是 A 侧。

然后就会跳出一个下面这样的窗口：

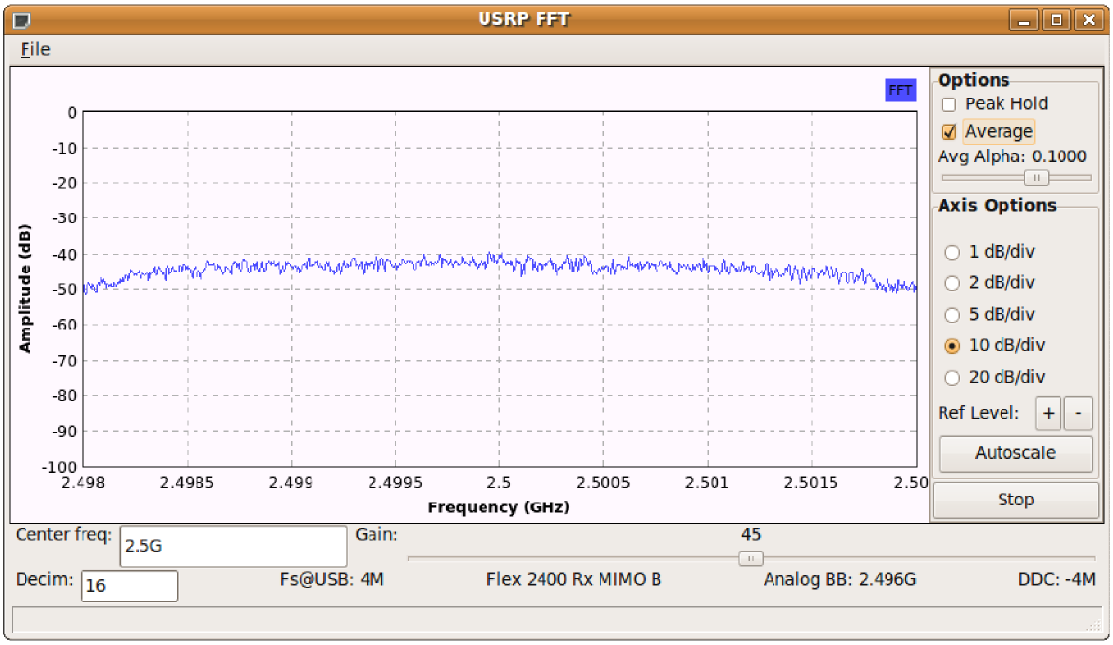

图 3-1 运行 usrp_fft.py 显示的频谱图

如果在 2.5GHz 上没有信号的话，就会如上图显示出一个平坦的白噪声谱。如果你所在的地方有 wifi 信号的话，选择有 wifi 信号的频段，就会看到 wifi 的频谱。Wifi 的频谱是突发的，如果没有数据传输的话，只能看到周期性的 beacon 信号。

如果你有 900MHz 频段的子板，用 usrp_fft.py 来观察一下 GSM 信号的频谱，就会看到非常明显的 200kHz 宽度的 GSM 信号。

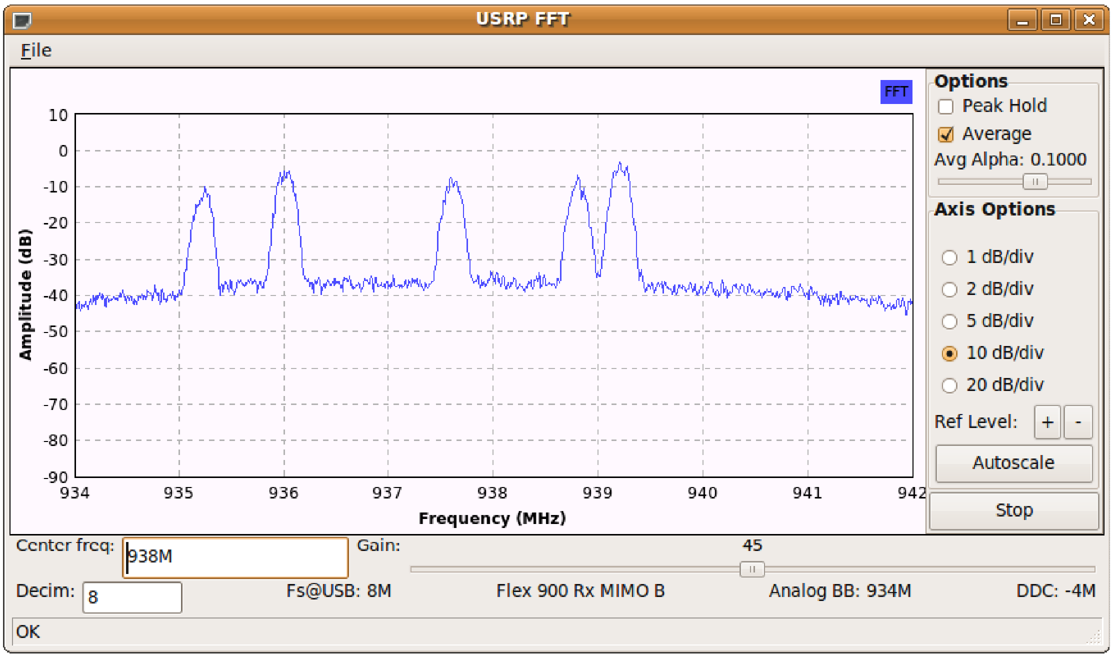

图 3-2 使用 usrp_fft.py 观察 GSM 频谱

Usrp_fft.py 的成功运行，可以证明 GNU Radio 的安装没有问题，USRP 的母版和子板的接收功能工作正常。

3.4.2 如果你没有 USRP

3.4.2.1运行 dial_tone.py

如果你手头没有 USRP 呢，可以尝试/gnuradio-3.2.2/gnuradio-examples/python/audio 下面的例子，比如 dial_tone.py。它产生两个 sine 波形并且把他们输出到声卡，一个输出到声卡的左声道，一个输出到右声道。

$./dial_tone.py

你会听到声卡发出声音。按下任意键程序就可以退出。

这个实验可以证明 GNU Radio 的安装没有问题。如果 dial_tone.py 无法运行的话，除了 GNU Radio 有问题，还有可能是你的声卡相关的库安装不全。

3.4.2.2运行 grc

你还可以去尝试一下 grc，这个例子不需要声卡。在/usr/local/bin 下面，可以找到 grc。它可以看作是 GNU Radio 的图形界面版，有点像 Matlab 的 Simulink。

在终端中运行 grc。

$./grc

会弹出如下图的窗口。

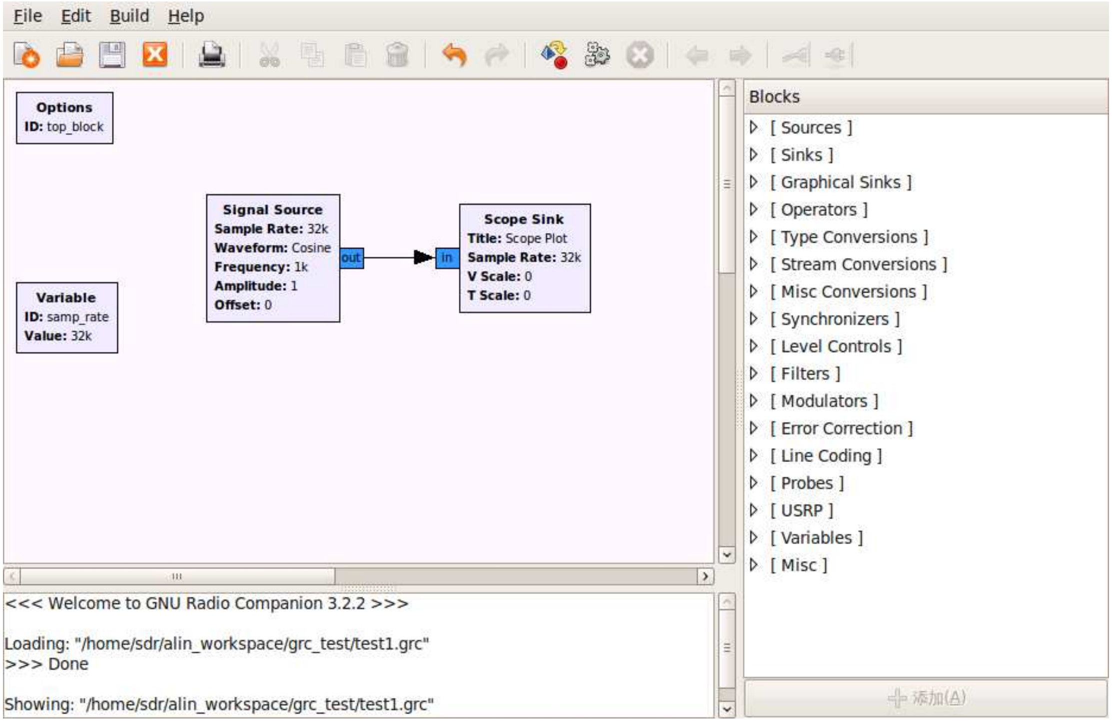

图 3-3 GRC

可以看到右边窗口中有各种 block 供你选择，双击一个 block，就会放到左边的图中。上图建了一个简单的图，一个正弦波的信号源，连接到一个示波器。点击工具栏上的“运行”按钮，就会弹出另一个窗口（如下图），我们可以看到示波器上显示出正弦波形。

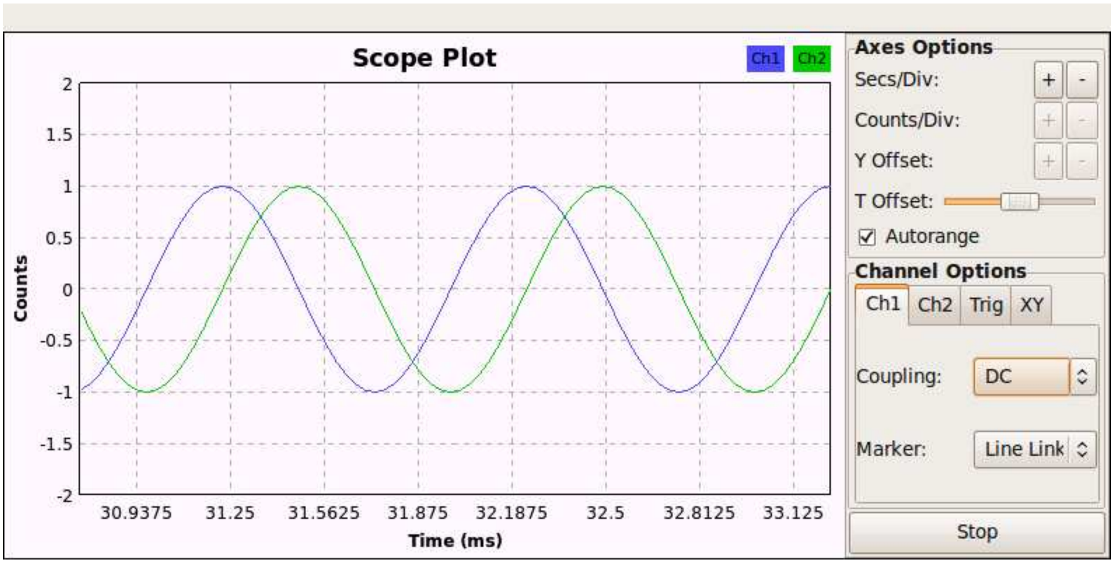

图 3-4 在 GRC 中运行一个流图

Grc 只有一些非常简单的 block，功能也很弱。因此我们不建议使用 grc 来开发你自己的应用程序。

第4章　GNU Radio 编程基础[3]

4.1　*在使用 GNU Radio 之前的预备知识*

GNU Radio 要求使用者有较强的计算机操作能力和较全面的通信和数字信号处理知识。这一节列出了很多有用的资源，包括文档、web 链接和在线教程。文章的目的就是为了帮助 GNU Radio 爱好者在使用之前做足准备。

相信您一定已经发现了 GNU Radio 的有趣之处并渴望能够尽快的熟练操作它。可惜的是，除了有趣之处外您还面临着更多的困难。你需要广泛的了解各个领域的知识，包括（无线）通信系统、数字信号处理、基础硬件电路设计、面向对象程序设计等等。然而你的兴趣和热情能够帮助你战胜困难。在这篇文章中，列出了一些有用的文章和资源，在您投入到 GNU Radio 的开发中之前，或许它们对您会有所帮助。

4.1.1 对 GNU Radio 做一个更清晰的认识

如果对 GNU Radio 还没有一个清晰的认识，请先阅读这篇在线文章——Eric Blossom, ”Exploring GNU Radio”。这篇文章对软件无线电做了简明而要的介绍。Eric 是整个 GNU Radio 项目的创始人。你必须了解 ADC 是如何工作的以及为什么我们需要一个 RF 前端。回顾一下您在信号与系统课程中学习的采样理论。然后仔细阅读“通用软件无线电外设”和“FPGA 内部做了什么（What Goes in the FPGA）”这两部分。文章中还提供了两个例子，一个简单的拨号音输出和一个 FM 接收器。至少您需要了解第一个例子。如果不能明白 FM 接收器这个例子，没有关系，请阅读第二篇文章——Eric Blossom, ” Listen to FM Radio in Software, Step by Step”。你无须一行一行的看，但是你需要知道信号是如何从空中进入声卡的。然后你最好了解一下 USRP 是做什么的，”USRP wiki”和”USRP User’s Guide”这两页会非常有用。如果您已经浏览过上述的文章，那么可以到”GNU Radio wiki”这一页去查看更多的信息。

要想真正的“玩”GNU Radio，你应该能够写自己的代码。“Exploring GNU Radio”这篇文章将会告诉你，GNU Radio 的软件结构包含两层。所有的信号处理模块都用 C++语言写，而 Python 则用作创建网络和流图并将这些模块连接在一起。GNU Radio 提供了很多有用的，使用频率高的模块，所以在很多情况下你无须接触 C++就可以创建你自己的模块，只需要用到 Python 就可以完成你的任务。当然，如果设计较为复杂，那么你还是得用 C++去创建你自己的模块。这时 Eric Blossom, ”how to write a block”这篇文章可以帮助你。现在你也许要问，GNU Radio 到底给我们提供了哪些模块？很遗憾，不像其他的开发工具例如 TinyOS，目前 GNU Radio 还没有一套像样的说明文档。但你可以参考一下由 Doxygen 生成的两篇文档，它们还是非常有用的。在安装了”gnuradio-core”和 ”usrp”的模块后，你会在以下路径找到两个 html 包：

- /usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html

[3] 本章内容主要翻译自 Dawei Shen 的系列 Tutorial

• /usr/local/share/doc/usrp-x.xcvs/html/index.html

虽然它们说明的不够清楚，但还是能提供相当多的信息的。第一个网页也可以在线浏览。

如果你还没有使用过 Python 语言，请阅读 Python 的在线教程。以下几个教程是最重要的：

• Section 2: Using the Python Interpreter（使用 Python 解释器）

• Section 3 An Informal Introduction to Python（Python 简介）

• Section 6: Modules（模块）

• Section 7: Input and Output（输入和输出）

• Section 9: Classes（类）

在 GNU Radio 的编程中会经常用到它们。如果你不熟悉面向对象编程(Object-Oriented Programming)，则需要仔细阅读第九章。以下的链接也许会帮助你了解 oop 的精髓。

• Lesson: Object-Oriented Programming Concepts（面向对象编程的概念）

• Introduction to Object-Oriented Programming Using C++（基于 C++的面向对象编程）

• The Object Oriented Programming Web（OOP 网站）

无论怎么说，Python 都是重点，所以需要掌握好它。

4.1.2 数字信号处理（DSP）知识

我想我们中的大部分人都学习过《信号与系统》这门课。《信号与系统》的知识在这里及其重要。然而这还不够。当信号在模拟和数字间转换，亦或在时域和频域间转换时，请确保自己不会犯糊涂。底线就是，你要懂什么是采样理论，什么是 z 变换，如何得到信号的频谱以及 FIR 和 IIR 滤波器的概念。在这里推荐几本经典的书：

• ”Signals and Systems (2nd edition)” - Alan V. Oppenheim, Alan S. Willsky

• ”Discrete-Time Singal Processing (2nd edition)” - Alan V. Oppenheim, Ronald W.

Schafer, John R. Buck

• ”Digital Signal Processing: Principles, Algorithms and Applications (3rd edition)”

- John G. Proakis, Dimitris Manolakis

请阅读离散时域傅里叶变换和 FIR、IIR 滤波器这几章。也许你会认为书很贵而且乏味，那么这里还有一些有用的在线资源：

• Digital Signal Processing Tutorial

• The Scientist and Engineer’s Guide to Digital Signal Processing

4.1.3 通信系统知识

我们都知道发送和接收的实信号不能是基带的。它们需要通过调制和解调才能传送。相信你在一些课程中已经学习了 AM 和 FM 调制。它们都属于模拟调制。为了开发更多有用而有趣的系统，我们需要发展数字通信系统。数字通信系统关注和感兴趣的是数字调制、解调和同步的问题。在你高年级的课程中，“通信系统”这门课可能会引起你的兴趣。那么我推荐你阅读以下这本书的第 4 章和第 5 章：

- Chapters 4 and 5, ”Digital Communications (4th edition)” - John G. Proakis

以上列出了很多预备知识。但并不意味着你必须一个一个的读完才能开始使用 GNU Radio。其实在这个阅读的过程中你就可以慢慢的学习使用 GNU Radio 了。但至少你应阅读 GNU Radio 的相关知识。接下来你可以试着做一些“作业”：

”gnuradio-examples”模块中，有一些示例代码，在 gnuradio-example/python/usrp/ 路径下。

你能阅读如下的两个程序并逐行理解吗？

- gnuradio-examples/python/usrp/am rcv.py
- gnuradio-examples/python/usrp/wfm rcv gui.py

如果可以，那么恭喜你又跨出了一大步。

4.2 如何编写 Python 应用程序——逐行学习[4]

Python 在 GNU Radio 编程中扮演着重要角色。GNU Radio 为构建软件无线电提供了一个设计平台。信号处理应用软件是由 Python 代码构建而成，Python 用于实现高级的组织、策略、图形界面和其他一些对运算性能要求不高的功能，而信号处理模块则由 C++语言编写。从 Python 角度来看，GNU Radio 提供了数据流的提取。本节着眼于 Python 语言层面，介绍了一些基本的 Python 语言使用方法以及在 GNU Radio 中 Python 是如何连接信号处理模块和控制数据流向的。本文中将介绍一个经典的例子：基于 GUI 的 FM 接收机的实现，会对代码逐行进行解释。同时还会介绍 Pyhon 的语法和软件无线电的概念。所以本节可以看做是 Python 的简明教程。至于如何编写 C++模块和其他一些高阶问题将会在后续章节中阐述。

4.2.1 概述

GNU Radio 的软件部分由双重结构组成。所有 performance-critical 信号处理模块用 C++语言编写，而高级的组织，连接和粘合操作都由 Python 实现。GNU Radio 软件向用户提供了很多优秀的使用频率较高的模块。

[4] 注意：这一小节的源代码版本较早，请对照书中附上的源码阅读。但这不影响我们对 GNU Radio 编程方法的理解。

这个结构与 OSI 的 7 层结构有一些相似之处。底层向高层提供服务，而高层则无需关注底层的执行细节，但需要关注必要的接口和函数的调用。在 GNU Radio 中，这种层次间的透明性与 OSI7 层结构相似。从 Python 的角度看，它要做的就是选择合适的信源，信宿和处理模块，设置正确的参数，然后将它们连接起来形成一个完整的应用程序。实际上，所有的信源，信宿和模块都是由 C++编写的类。然而，在 Python 层面是无法看到 C++程序的工作过程的。一段较长的，复杂的而功能强大的 C++代码，在 Python 中只能体现为一行语句。

因此，无论应用程序多复杂，Python 代码几乎总是比较短而且很简洁。繁重的任务则交给 C++。必须记住一个原则：对于很多应用软件，我们在 Python 层面上要做的只是在心中画一个信号流向图，然后用 Python 这杆好用的笔将它们连接起来，有时需要用到图形用户界面（GUI）。

显然，在学习 GNU Radio 过程中，Python 非常重要。Python 是个强大且灵活的编程语言，要掌握它还有很多要学习的。但如果你有足够的 C/C++知识，那这就是小菜一碟了。考虑到在 Python 应用到 GNU Radio 中时，Python 具有一些特别的性质，同时在 GNU Radio 中有一些不常见的特性也很少被应用，文章将 Python 编程技术同软件无线电的概念结合起来讲。获取新知识中精髓的最佳办法就是通过例子说明，而不是单调的啃书本中的概念。所以我们将围绕一个经典的例子来说明：基于 GUI 的 FM 接收机的实现。我们将逐行的分析这个例子的代码，并逐一讨论 Python 编程、信号处理技术、软件无线电概念和一些硬件配置。

这个例子基于 GUI 工具实现了一个 FM 接收机。FM 信号被 USRP 接收下来，然后在 USRP 和计算机中进行处理。最终解调的信号通过声卡播放出来。整个设计对天线没有特别的要求。只要插入一根铜线到 basic RX 子板就可以接收到质量很高的 FM 信号。代码可以在'gnuradio-examples/python/usrp/wfm_rcv_gui.py'找到。请对照下面的源码以保证我们讨论的是同一段代码。

4.2.2 FM 接收机源代码

The source code of “wfm_rcv_gui.py”

```
#!/usr/bin/env python


from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
```

```
from gnuradio.wxgui import stdgui, fftsink
import wx
class wfm_rx_graph (stdgui.gui_flow_graph):
    def __init__(self,frame,panel,vbox,argv):
        stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

        IF_freq = parseargs(argv[1:])
        adc_rate = 64e6

        decim = 250
        quad_rate = adc_rate / decim # 256 kHz
        audio_decimation = 8
        audio_rate = quad_rate / audio_decimation # 32 kHz

        # usrp is data source
        src = usrp.source_c (0, decim)
        src.set_rx_freq (0, IF_freq)
        src.set_pga(0,20)

        guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

        # sound card as final sink
        audio_sink = audio.sink (int (audio_rate))

        # now wire it all together
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))

        if 1:
            pre_demod, fft_win1 = \
            fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                     512, quad_rate)
            self.connect (src, pre_demod)
            vbox.Add (fft_win1, 1, wx.EXPAND)
```

```
        if 1:
            post_deemph, fft_win3 = \
                        fftsink.make_fft_sink_f (self, panel, "With Deemph",
                                                    512, quad_rate, -60, 20)
            self.connect (guts.deemph, post_deemph)
            vbox.Add (fft_win3, 1, wx.EXPAND)

        if 1:
            post_filt, fft_win4 = \
                    fftsink.make_fft_sink_f (self, panel, "Post Filter",
                                                512, audio_rate, -60, 20)
            self.connect (guts.audio_filter, post_filt)
            vbox.Add (fft_win4, 1, wx.EXPAND)
def parseargs (args):
    nargs = len (args)
    if nargs == 1:
        freq1 = float (args[0]) * 1e6
    else:
        sys.stderr.write ('usage: wfm_rcv freq1\n')
        sys.exit (1)
    return freq1 - 128e6
if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

4.2.3 第一行

如果你已经读过其他例子的代码，你可以发现这些程序的第一行几乎都是

#!/usr/bin/env python

如果我们将这一行放在脚本的开始处并且给此文件一个可执行模式，就可以直接执行 Python 脚本。用'chmod'命令可使脚本为可执行模式：

$ chmod +x wfm_rcv_gui.py

现在脚本 wfm_rcv_gui.py 变为可执行模式了。你可以在 shell 中运行你的程序，用如下命令：

$ ./wfm_rcv_gui.py arguments

启用 Python 解释器，脚本中的代码将被逐行执行。Python 语言是一种解释性的语言，像 Matlab 一样。没有编译和链接这些步骤。有很多种方式来启用 Python 解释器，你可以用：

$ python ./wfm_rcv_gui.py arguments

而无需专门给脚本设置可执行模式。

你还可以用交互模式，只需要敲入命令行：

$ python

然后 Python 解释器环境启用，这时你可以一行一行的输入你的代码了。但是，这显然不方便。我们很少用到交互式的解释器，除非写一些测试功能或者用它来当计算器。大多数时候，都是将代码写进后缀为.py 的文件中，使它能够自动执行，这样更加方便。

4.2.4 导入需要的模块（module）

接下来我们来看一些重要的东西：

```
from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
from gnuradio.wxgui import stdgui, fftsink
import wx
```

理解这些语句要求知道一些有关 Python 语言中“模块（module）”和“包（package）”的概念。学习它们的最好方法就是 Python 教程的第六章。以下是一个简单的介绍。

如果我们关闭 Python 解释器再启动它，所有的已经定义的函数和变量都会丢失。因此，我们希望能写一些稍微长点的程序，将它保存为脚本，包括了函数和变量的定义，可能还包含一些可执行语句。脚本语言可以看做是 Python 解释器的输入。我们也希望能在多个程序中调用同一个函数，而不用在每个程序中都拷贝这个函数。

鉴于这种情况，Python 提供了一个 **module/package** 组织系统。“模块(module)”文件包含了一些 Python 的定义和声明，是后缀为'.py'的文件。模块的名称是一个全局变量'__name__'。模块的定义语句可以被导入到其他模块或者顶层模块。“包（packages）”是有相似功能的模块的集合，这些模块通常都放在相同目录下。需要'__init__.py'文件来告诉 Python 要将这些目录看成 packages。一个 package 包含了模块和子 package（能包含 sub-sub-packages）。

我们用“package.module”的方式来构建 Python 的模块名称空间。例如，模块名 A.B 指明了子模块 B 在 A package 中。

一个模块包括可执行的声明和函数的定义。每个模块都有自己的私有成员变量表，它是一个全局变量表，所有在模块内定义的函数都可以使用它。模块的编写者可以在模块中使用全局变量，而不用担心与模块使用者的全局变量发生的冲突。作为模块使用者，我们可以利用`modname.itemname'来使用模块的函数和全局变量。这里的’itemname’可是成员函数也可以是成员变量。

用‘import’命令在模块中导入其他的模块。习惯上是将`import'加到模块的前面。要注意，导入操作是很灵活的。我们可以导入一个 package、一个模块或者只是一个模块中的定义。当我们尝试从一个 package 导入一个模块，我们可以用’import packageA.moduleB’或者’from package A import module B’。当用`from package import item'时，'item'可以是 package 的模块，或者子 package，或者是其他在模块中定义的名字，像函数、类或变量等。下面详细介绍一下例子中的模块，因为这些 module 和 package 在 GNU Radio 中将会经常遇到。顶层的 GNU Radio package 是'gnuradio'，它包括了所有与 GNU Radio 相关的模块。它在如下的路径中（这台机器装的是 python2.4）：

/usr/local/lib/python2.4/site-packages

有时候，这个路径没有包括在 Python 的搜索路径中。我们需要将这个路径导出到环境变量'PATHONPATH'。所以我们常常将下面这一行加到用户.bash_profile 文件中：

export PATHONPATH=/usr/local/lib/python2.4/site-packages

来保证 Python 解释器能找到 gnuradio package。

“gr”是 gnuradio 中重要的一个子 package，也是 GNU Radio 软件的核心。“流图（flow graph）”类的类型在 gr 中定义，它对安排信号流向来说非常关键。

“eng notation”模块能给工程师的标记带来方便，使用它可以按照工程惯例赋一些常量字或者常量字符。音频模块是与声卡连接的接口，而 usrp 中带有接口用以控制 USRP。音频模块和 usrp 模块通常是当做信源和信宿。在稍后我们将具体讨论。blks 是一个子 package，如果你查看这个路径，它几乎是空的。实际上它将任务交给了 gnuradio 中的子 package blksimpl，用__init__.py 文件描述。blksimpl 可以实现一些有用的应用，例如 FM 接收机，GMSK，等等。在这个例子中，FM 接收机实信号处理部分就在这个 package 中实现。看下面这一行，更加有趣：

from gnuradio.eng_option import eng_option

这正是我们刚才提到的，我们即可以导入一个完整的 module/sub-package，也可以只是一个函数、一个类或一个变量的定义。*gnuradio.eng_option* 模块中定义了类 *eng_option*。我们无需将整个模块导入，只要导入一个单独的类定义。*gnuradio.eng_option* 模块为 optparse 工程标记提供支持。OptionParser。

以下这行似乎与上面一行有相同格式：

from gnuradio.wxgui import stdgui, fftsink

但意义上稍有不同，gnuradio.wxgui 是一个 sub-package 不是一个模块，而 stdgui 和 fftsink 是 sub-package 中的两个模块。因为不用导入整个 sub-package，因此只导入我们需要的。gnuradio.wxgui 为 GNU Radio 提供可视化的工具，是由 wxPython 构建的。Python 的导入功能使得导入操作更加灵活和方便。

最后，optparse, math, sys, wx 都是 Python 或 wxPython 的内置模块或 sub-packages，而不是 GNU Radio 的一部分。

针对这一点，再强调一下，以上导入的模块包含了可执行语句以及函数和类的定义。模块导入后那些可执行语句将被立即执行。在导入这些模块和 packages 后，即初始化了其中的很多变量、类和模块。所以不要认为什么都没做，实际上，大量的工作都是这一阶段承担的。现在就等你输入命令了。好！目前为止，我们简略的了解了大部分使用频率较高的 GNU Radio 模块以及他们的组织方式。也许这听起来太抽象了，你可能对它们的用法仍然很迷惑。没关系，我们马上来看看。

4.2.5 顶层模块 wfm_rx_graph

熟悉面向对象程序设计（OOP）对理解这一章非常重要。OOP 本身说来话长，显然不是我们要关注的。但是学习过程中仍要介绍一些 OOP 的概念。如果遇到与数字信号处理相关的代码，还会回顾一些信号处理技术。

4.2.5.1类的定义

在这个例子中，大部分代码由类'wfm_rx_graph'定义。以下语句：

```
class wfm_rx_graph (stdgui.gui_flow_graph):
```

定义了一个新的类'wfm_rx_graph'，它继承于基类（base class），或者叫做'father class（父类）'——'gui_flow_graph'。父类 gui_flow_graph 由刚刚从 gnuradio 中导入的 stdgui 模块定义。根据名称空间的规定，将其写为 stdgui.gui_flow_graph。

4.2.5.2 'flow graph' 类

有一种重要的类在 GNU Radio 中非常关键：'flow graph（流图）'类。GNU Radio 中定义了一系列与'GRAPH（图表）'相关的类。我们会发现 stdgui.gui_flow_graph 是继承自 gr.flow_graph 的，而 gr.flow_graph 是在 sub-package gr 中定义的。另外，gr.flow_graph 又是继承于'root' 类 gr.basic_flow_class。在 GNU Radio 中，还有很多别的类是继承于 gr.basic_flow_graph。这个大的'GRAPH family'使 GNU Radio 编程变得简洁且简单，同时使信号处理的顺序看起来清晰而直接。

那么'graphs'是做什么的呢？假设你用一些商业软件设计一个电路，例如 Pspice。首先你可能要建一个原理图（schematic）或者一个'canvas'，然后将所有要用的电路元件放到上面，比如电阻、放大器或者一些直流电源。最后用线将他们连接起来完成电路设计。这种方式非常适用于 GNU Radio。一个流图就像是一个原理图或是 canvas。电路元件就是 GNU Radio 中的信源，信宿和信号处理模块。最后，这些'wires（线）'作为图标类的连接方式将这些模块连接起

来。有时，在另一个原理图中集成电路也可以作为一个元件，称作 sub-circuit。在 GNU Radio 中，一个 sub-graph 可以当做另一个 graph 的模块。

在我们的例子中，wfm_rx_graph 属于 graph class 族，基于 GUI 的支持。稍后我们将介绍如何用'connect'命令来粘合 FM 接收机中的必要模块。

4.2.5.3初始化函数: init

接下来我们执行类 wfm_rx_graph 中的方法（函数）'__init__'。定义一个新方法（函数）的语句如下：

```
def funcname(arg1 arg2 ...)
```

__init__方法在任何类中都是非常重要的。在定义了一个类之后，我们可以用这个类去初始化一个实例。__init__用于在已知初始化阶段创建一个对象。类初始化将自动为新创建的类实例引用__init__函数。实际上在我们的例子中，__init__是唯一在类 wfm_rx_graph 中定义的方法。

在讨论方法__init__之前，Python 的一个重要特性值得一提。我们注意到，在这段代码中并没有标明类在哪里定义，函数从哪里开始或结束。通常在其他编程语言中例如 C++或 Pascal，会用一对'begin'和'end'或者一对'f'和'g'来标明一组语句的开始和结束。然而在 Python 中则不需要。在 Python 中没有这样明确的标记，是用“行缩进”的方式来标明一组语句的开始和结束。所以当你用 Python 语言写程序时一定要注意代码的编辑和布局。有时候你会发现程序执行时出现一些莫名其妙的错误，这时候你可以去检查一下是不是缩进有错误。

现在让我们来看看函数__init__中发生了什么。

```
def __init__(self,frame,panel,vbox,argv):
```

以上语句初始化了函数__init__的四个变量。通常，所有函数的第一个变量称作 self。这只不过是一个约定：self 这个名字对 Python 没有什么特殊的意义。然而，若函数要调用其他的函数，则需要利用 self 参量的函数属性，例如'self.connect()'，我们稍后会看到。__init__函数首先要做的是调用初始化函数 stdgui.gui_flow_graph，即它的父类'father class'，其中的 4 个参数是相同的。

```
stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)
```

你可能想了解一下 stdgui.gui_flow_graph 的初始化函数。因为 wfm_rx_graph 是继承于它的，所以我们可以将 wfm_rx_graph 看成特殊的 gui_flow_graph。所以子类的形式自然也应该与父类相似，然后才稍作修改。

4.2.5.4定义常数

我们从实信号的输入开始学习，

```
IF_freq = parseargs(argv[1:])
```

从 **parseargs** 函数的返回值来设置 IF 频率。

IF 频率表示什么？大概来说 IF 是中频的缩写，它是信号带宽的中心频率。我们将实信号带宽，即 RF 带宽（通常频率非常高）搬移至中频带（IF），在中频带 ADC 在奈奎斯特准则下进行采样。

在我们的例子中，**parseargs** 函数会在类 wfm_rx_graph 之后定义。当程序在框架阶段执行时，它可以接收用户输入的参量。稍后会讨论。

另外，你可能已经注意到 Python 并不要求变量或参数的声明。这与 C 语言中的“使用之前必须声明”的概念完全不同。

```
adc_rate = 64e6
```

以上一行定义了 AD 转换器的采样频率为 64MHz。根据奈奎斯特定理，信号带宽的最高频率成分要低于 32MHz 才能在采样后无失真。

```
decim = 250
quad_rate = adc_rate / decim # 256 kHz
```

抽取这个概念属于 DSP 领域。在对模拟信号采样之后，得到一个高速率的数字信号，对于 CPU 和存储器都是很大的负担。通常，我们可以降采样率（抽取）而不损失频谱信息。在例子中，抽取率为 250，则数据率降为 256K 每秒，这样 CPU 就可以接收数据了。**quad_rate** 表示 **quadrature** 数据率。为什么称作 **quadrature_rate** 将在稍后解释。

`# 256 kHz'是注释语句。在 Python 中注释语句前加‘#’号。所有‘#’号后面的语句都会被 Python 解释器忽略。

```
audio_decimation = 8
audio_rate = quad_rate / audio_decimation # 32 kHz
```

处理数字 FM 信号后，希望用计算机的声卡来播放这个信号。然而声卡能接收的数据率相当有限。256kHz 还是太高了而且没有必要。所以需要继续降低数据的速率。多数声卡的速率为 32kHz。

4.2.5.5信号源

以下几行是非常典型的信号处理程序的例子，它包括三个基本要素：信源、信宿和一系列信号处理模块。这个例子给信号处理模块取了个好听的名字：guts。在例子中FM接收机的信号源是USRP。

```
# usrp is data source
src = usrp.source_c (0, decim)
src.set_rx_freq (0, IF_freq)
src.set_pga(0,20)
```

USRP 用 RX 子板接收到模拟的 FM 信号，然后用 AD 转换器以 64MHz 的速率对信号采样。输出的数字序列进入 FPGA 芯片。在 FPGA 中信号根据用户设定的抽取率（在本例中是 250）进行下变频。另一个由 FPGA 执行的数字信号处理操作：实 IF 信号变为复基带信号，由两路 I/Q quadrature components 组成。这也解释了为什么在抽取后数据速率称为`quadrature rate'。当然其中的运算是非常复杂的。最后，复基带型号通过 USB2.0 数据线被送到计算机中的软件模块。复数用一对 real/imag 表示，里面包含有两个实值。

好了，让我们来看代码。一开始我们就导入了 usrp 模块。usrp 模块在如下路径中：

/usr/local/lib/python2.4/site-packages/gnuradio/usrp.py

这表明 usrp 是 USRP 信宿（发送）和信源（接收）封装。当一个信宿或信源作为实例时，usrp 模块将探测 USB 端口以定位使用的板子的编号，然后选择合适的信源或信宿。usrp 模块中定义了一个叫 source_c 的函数。它返回一个对象用以表示数据源。后缀 c 表示信号的数据类型为复数（complex），因为进入计算机的是复信号（实际上是 real/imag 的形式）。相应的，在 usrp 模块中还定义了另一个叫 source_s 的函数，主要用于 16bit 短整型的数据源。

本例中，source_c 有两个参量。`0'用于指定启动哪一个 USRP。如果我们只有一个 USRP，那么就设置为 0。第二个参量用于设定 USRP 的抽取率。set_rx_freq 和 set_pga 是源`src'的两个函数。set_rx_freq 可设定 USRP 的 IF 频率。刚刚我们已经了解了 USPR 将实 IF 信号变为两路复基带信号 I/Q quadrature components 的过程。所以需要告知 USRP，IF 频率的有关信息。

pga 是“可编程增益放大器（Programmable Gain Amplifier）”的缩写。可以用 set_pga 函数来设置它的值（以 dB 为单位），本例中设置为 20dB。

这些函数在哪里定义呢？目前这个阶段，用 Python 语言还不足以解释这些。实际上所有这些函数都是由 C++实现的。SWIG 为 C++和 Python 之间提供了接口。同时还有一个聪明的指针系统（pointer system）降低了 C++和 Python 的相互影响。所以这个内容看起来相当复杂。所以我们略过执行细节，只管在 Python 用这些函数就好。

由 Doxygen 生成的关于 USRP 的一篇文档很重要，在以下路径中：

/usr/local/share/doc/usrp-x.xcvs/html

你可以在这篇文档中查找到所有 USRP 提供的函数。USRP 的高层接口是 usrp_standard_rx 和 usrp_standard_tx. 。我们来关注一下它们的基类，usrp_basic_rx, usrp_basic_tx 和 usrp_basic。还有很多其他的函数，用以控制和配合 USRP。请不要担心 Python 和 C++之间的接口问题，在讨论用 C++编写模块时还会继续讲解。

4.2.5.6信号处理模块 'gut'

guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

`guts'是 FM 接收机的核心处理模块。所有的信号处理模块包括非相干解调模块都包含在里面。更多具体的内容可以在以下路径中找到：

/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/wfm_rcv.py

FM 接收技术的有关细节将在后面讨论。抛开这些底层的细节，让我们来试试看。blksimpl 模块定义了类 wfm_rcv。类 wfm_rcv 的基类是 hier_block，此基类在模块 gr.hier_block 中定义。gr.hier_block 可以看成是一个子流图（sub-graph），包含了很多信号处理模块，在更大的流图中，可以将它看成是一个复杂的模块。在这个语句中，我们创建了一个对象`guts'，将它作为类 wfm_rcv 中的一个模块。所有实信号的处理都将在这个大模块中完成。

4.2.5.7信宿

最后，我们用声卡来播放解调后的 FM 信号。所以在本例中，信宿就是音频设备：

```
# sound card as final sink
audio_sink = audio.sink (int (audio_rate))
```

在音频模块中，sink 函数返回的对象是信宿模块。*audio_rate* 这个参数规定了信号进入声卡的速率，本例中的参数值为 32kHz。

4.2.5.8 粘合模块

以下两行最终完成了信号流图的构建：

```
# now wire it all together
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

刚才我们已经讨论过了流图类，新类 wfm_rx_graph 是属于流图类的。所有这些流图类都继承自`root'类 gr.basic_flow_graph。连接函数（connect）在 gr.basic_flow_graph 中定义。这个函数用于为流图连接模块。我们会在后面具体的探讨流图类和它们的函数。

现在信号流图就完成了！

好了！让我们总结一下。剩下的代码需要一些高阶的知识。我们将要学习非常棒的 GUI 工具，它可以帮助设计 FM 接收机。GUI 由 gr-wxgui 构建，而 gr-wxgui 则是基于 wxPython 和 FFTW 的。现在大家已经学习了如何获取参数，以及如何启动流图。然而可能大家对类与类之间，函数与函数之间参数的传递还不是很清楚。这些高阶的问题将会在后面介绍。

这一节主要介绍了 Python 的基本语法和 GNU Radio 中 Python 的作用。文章还提到了一些软件无线电的概念和信号处理技术。希望在阅读过这篇文章后，你能有所收获。

4.3 流图，模块和连接的原理

前面我们已经了解了流图类（family of `flow graph' classes），它使高层信号处理的调度和组织变得便捷和灵活。一般来说，一个流图可以包含多个信源、信宿和信号处理模块，然后将它们连接起来构成一个完整的应用程序。本节中，我们将做更深入的了解。文中引用了另一个例子`dial tone（拨号音）'进行说明。这是一个非常简单的`Hello World!'式的例子。但它已经足以说明 GNU Radio 中流图机制的优秀和强大。本例中将产生两路频率不同的正弦波并通过声卡推动扬声器发声。流图中只有信源和信宿，不包含信号处理模块。

这个例子的源代码在以下路径：'gnuradio-examples/python/audio/dial_tone.py'。请参阅下面的源代码以确保我们讨论的代码是同一版本。运行这个例子，无需

使用 USRP。但是你的计算机必须带有声卡。只需要在 shell 中输入命令 $./dial_tone.py，你就能从扬声器听到清晰的拨号音。

- dial_tone.py 源代码

```
#!/usr/bin/env python
from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
from optparse import OptionParser

class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

        parser = OptionParser(option_class=eng_option)
        parser.add_option("-O", "--audio-output", type="string", default="",
                          help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
        parser.add_option("-r", "--sample-rate", type="eng_float", default=48000,
                          help="set sample rate to RATE (48000)")
        (options, args) = parser.parse_args ()
        if len(args) != 0:
            parser.print_help()
            raise SystemExit, 1

        sample_rate = int(options.sample_rate)
        ampl = 0.1

        src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink (sample_rate, options.audio_output)
        self.connect (src0, (dst, 0))
        self.connect (src1, (dst, 1))


if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

4.3.1 顶层模块 my_top_block

```
class my_top_block(gr.top_block):

    def __init__(self):
```

这个程序的最顶层模块叫做 my_top_block，初始化函数_init_()中可以从命令行获得 audio-output 和 sample-rate 这两个参数。Sample-rate 的默认值是 48000，也就是输入声卡的数字信号的速率默认为 48kHz。

另外，初始化函数中还设置了一个常量。

```
ampl = 0.1
```

它表示正弦波的幅值设为 0.1v。

4.3.1.1信源

在my_top_block中创建了两个数据源。在上一篇教程，我们已经学习了一类信源：*usrp*。它使得USRP成为了一个输入信号。在本例中，信源更加简单：由计算机产生的正弦波：

```
src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
```

如前所述，在 GNU Radio 中，所有 block 都由 C++来实现。SWIG 为 Python 和 C++之间提供了接口。因此在 Python 中你可以直接使用 C++编写的类定义和函数。我们将在后面讨论如何用 C++编写 block 以及如何用 SWIG 构建接口。到那时就可以理解整个设计过程了。而现在，让我们先把它放在一边，只管在 Python 中使用这些 block。

要想知道 GNU Radio 执行了什么 block 最好的方法是查看 Doxygen 生成的 GNU Radio 文档。安装了 GNU Radio 之后，在如下路径可以找到它：

/usr/local/share/doc/gnuradio-core-2.5cvs/html/index.html

在线也可以获得这些文档。

找到 gr_sig_source_f 以后，打开这个类的参考文档，你可以看到 gr_sig_source_f 的层次结构。`root'类是 gr_block。所有的信号处理模块都是继承于 gr_block 的。gr_sync_block 是一个重要类，继承自 gr_block。这些模块都以前缀`gr_'开始，由 C++实现。当使用 SWIG 来做 C++和 Python 之间的接口时，SWIG 后台做了一些事情，使得我们可以在 Python 中以 gr.sig_source_f()的形式接入 gr_sig_source_f。这里的后缀`_f'指明了信源的数据类型，本例中是浮点型。SWIG 到底做了什么我们将在学习用 C++编写 block 时讨论，所以如果你对这点有疑问也请不要担心。

4.3.1.2信宿

在讲解 FM 接收机的时候我们就已经遇到过相同的信宿：音频设备。我们需要通过声卡来播放这个音频设备：

```
dst = audio.sink (sample_rate, options.audio_output)
```

audio 模块（module）在如下路径中：

/usr/local/lib/python2.4/site-packages/gnuradio/audio.py

模块 *audio* 与 *usrp* 十分相似，它是音频信源和信宿的封装。你可以在 Python 中直接利用 audio.sink()来创建一个音频模块了。`sink()'和`source()'实际上是模块 *audio* 中定义的函数，它们与支持声卡的 OSS 或 ALSA 相连，假设你已经安装了 *gr-audio-oss* 或 *gr-audio-alsa* 模块。这两个 method 将返回一个类实例作为音频信宿或者信源。

4.3.1.3连接

最后，我们将这些已经定义的模块连接起来完成这个流图：

```
self.connect (src0, (dst, 0))
self.connect (src1, (dst, 1))
```

`connect()'的用法是非常简单的。但是还有一些东西值得一提。让我们再看一下 top_block.py 中 connect()的定义：

```
def connect(self, *points):
    '''connect requires one or more arguments that can be coerced to endpoints.
    If more than two arguments are provided, they are connected together successively.
    '''
    if len (points) < 1:
        raise ValueError, ("connect requires at least one endpoint; %d provided." % (len (points),))
      else:
        if len(points) == 1:
            self._tb.connect(points[0].basic_block())
        else:
            for i in range (1, len (points)):
                    self._connect(points[i-1], points[i])
```

参数前的`*'号是 Python 的一个特殊功能。它表示函数调用时，参数可为任意。这些参数将被包含在一个元组中。参量的数量是可变的，可能会出现 0 或者更多常规参量。所以 connect() method 实际上能够接受两个以上的参量。如果有两个以上的 block 作为 connect()的参量，它们将会一个接一个的成功连接起来，代码中显示的非常清楚。

另外一个隐藏的问题就是，当我们连接这些点时，它们首先被强制（`coerced'）成为节点实例（`endpoint' instance）。让我们看看方法（method）——_coerce_endpoint()的定义：

```
def _coerce_endpoint(self, endp):
        if hasattr(endp, 'basic_block'):
            return (endp, 0)
        else:
            if hasattr(endp, "__getitem__") and len(endp) == 2:
                return endp # Assume user put (block, port)
            else:
                raise ValueError("unable to coerce endpoint")
```

显然，我们不一定非要以“节点实例”作为方法_connect()的参量（实际上这样做会让代码显得很麻烦）。我们可以设一个二元参数组，也就是(block name, port No.)。方法_coerce_endpoint()会把它变成一个节点实例。在本例中，节点为(dst, 0)和(dst, 1)。如果 block 实例没有提供端口序号，方法_coerce_endpoint()将

设置端口号为默认值 0 然后返回一个节点实例，像本例中的 src0 和 src1。也可以说是(src0, 0)。因为 block 只有一个端口，可以直接用 block 名作为 connect()中的参数。

要注意的是，目前，流图只是在逻辑上创建并连接起来，在物理上还没有完成。相当于只得到了一个草图。最终，my_top_block()函数将返回一个我们创建的流图实例。

4.3.2 运行程序

现在可以运行流图：

```
if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

第一行是什么意思呢？每一个模块（module）都有一个属性`__name__'，用来指明当前模块的名称。很多模块都包含了下面一段：

```
if __name__ == '__main__':

The testing code

...
```

出于测试的目的，当导入一个模块时，会在模块名称空间创建`__name__'来存放模块文件的名称。所以模块导入时，解释器会忽略测试代码，因为模块的名称__name__怎么也不会变成`__main__'。然而，如果像可执行文件一样直接运行这个模块，像./dial_tone.py，或者用 python dial_tone.py 去运行脚本，而不是导入，那么模块的名称空间就是全局的。Python 自动将__name__设为__main__。这样，if 后面的代码就将生效。

my_top_block().run()调用了我们刚刚定义的函数，run()即可运行流图。

本文中，我们通过一个非常简单的例子中，分析了流图机制是如何使得 GNU Radio 编程变得方便和灵活的。可能目前对于在 GNU Radio 中用 Python 编程来说，文中所提到的一些东西并不是必须的。但要理解整个编程的结构，它们还是非常有用的。

让我们继续讨论 FM 的例子 wfm_rcv_gui.py。本文将阐述基于 wxPython 的 GUI 工具的使用。我们还将介绍一些有用的关于变量描述方面的编程要点。如果你对使用甚至是开发 GUI 工具感兴趣，这篇文章会对你有所帮助。

4.4 图形界面的使用

这一部分，将继续 FM 接收机代码 usrp_wfm_rcv.py 的讨论。告诉你如何利用 GNU Radio 强大的 GUI 工具，显示和分析信号，并可以仿真频谱分析仪和示波器。文中将介绍基于 wxPython 的 GUI 工具的使用方法。然后会讨论如何使用 Python 中的命令行参量。

最感性直接的信号分析方式就是用图像来显示它，包括时域和频域。在实际的研发工作中，则有频谱分析仪和示波器可以帮助我们。在软件无线电领域，我们为有这样一个优秀的工具而感到高兴，感谢 wxPython，它提供了一个非常灵活的方式来构建 GUI 工具。

4.4.1 频谱分析仪

请看 usrp_wfm_rcv.py 中的以下代码：

```
if 1:
            self.src_fft = fftsink2.fft_sink_c(self.panel, title="Data from USRP",
                                       fft_size=512, sample_rate=usrp_rate,
                                       ref_scale=32768.0, ref_level=0, y_divs=12)
            self.connect (self.u, self.src_fft)
            vbox.Add (self.src_fft.win, 4, wx.EXPAND)
```

这就是软件频谱分析仪，它是基于数字序列快速傅里叶变换（FFT）的。频谱分析仪模块通常作为 sink 模块。这就是为什么要将它命名为`fft_sink'的原因。它是在模块 wxgui.fftsink2.py 中定义的。

4.4.2 wxPython 是如何工作的

现在我们来了解一些有关 wxPython 的知识，wxPython 是一个 GUI 工具箱。感兴趣的读者可以访问 wxPython 的网站或在线教程以获得更多信息。

让我们花些时间学习 wxPython 的组织机制。首先要做的是导入所有 wxPython 部件到当前工作空间，也就是`import wx'。每一个 wxPython 都要从 wx.App 继承一个类并为这个类生成一个方法 OnInit()。系统将调用这个方法作为它启动/初始化（startup/initialization）过程的一部分，用 wx.App.__init()__来实现。OnInit()的主要目的是创建框架、窗口等元素来使得程序得以运行。在定义了这样一个类后，需要实例化这个类的一个对象，并通过调用它的方法 MainLoop()来启动这个应用程序，方法 MainLoop()的作用是处理事件。在 FM 接收机的例子中，类在哪里定义呢？请看下面三行：

```
if __name__ == '__main__':
    app = stdgui2.stdapp (wfm_rx_block, "USRP WFM RX")

    app.MainLoop ()
```

实际上，这个类称作 stdapp，在导入 stdgui2 模块时就创建了。

```
from gnuradio.wxgui import stdgui2, fftsink2, form
```

在所有 wxPython 应用程序中，最后两行都是相同的。我们只是简单的创建了一个应用程序类的实例，接着调用它的方法 MainLoop()。MainLoop()是应用程序的核心，用来处理事件和分配事件到各个应用程序窗口。后台还做了一些我们

不了解的工作。请不要着急。让我们看看/gnuradio/wxgui/stugui2.py 中 stdapp 的定义：

```
class stdapp (wx.App):
    def  __init__  (self,  top_block_maker,  title="GNU  Radio",
nstatus=2):
        self.top_block_maker = top_block_maker
        self.title = title
        self._nstatus = nstatus
        #  All  our  initialization  must  come  before  calling
wx.App.__init__.
        # OnInit is called from somewhere in the guts of __init__.
        wx.App.__init__ (self, redirect=False)

    def OnInit (self):
        frame   =   stdframe   (self.top_block_maker,   self.title,
self._nstatus)
        frame.Show (True)
        self.SetTopWindow (frame)
        return True
```

stdapp 是继承自 wx.App 的类。它的初始方法__init__()有两个参量：top_block_maker，它是属于流图族的一个类；title，是整个应用程序的标题（本例中是 WFM RX）。在方法 OnInit()中，这两个参量进一步用于创建 **stdframe** 创建的对象。

```
class stdframe (wx.Frame):
    def  __init__  (self,  top_block_maker,  title="GNU  Radio",
nstatus=2):
        # print "stdframe.__init__"
        wx.Frame.__init__(self, None, -1, title)

        self.CreateStatusBar (nstatus)
        mainmenu = wx.MenuBar ()

        menu = wx.Menu ()
        item = menu.Append (200, 'E&xit', 'Exit')
        self.Bind (wx.EVT_MENU, self.OnCloseWindow, item)
        mainmenu.Append (menu, "&File")
        self.SetMenuBar (mainmenu)

        self.Bind (wx.EVT_CLOSE, self.OnCloseWindow)
        self.panel = stdpanel (self, self, top_block_maker)
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(vbox)
        self.SetAutoLayout(True)
        vbox.Fit(self)

    def OnCloseWindow (self, event):
        self.top_block().stop()
        self.Destroy ()

    def top_block (self):
```

```
        return self.panel.top_block
```

现在可以花点时间来了解一下 wxPython GUI 的布局。在 wxPython 中，wx.Window 能在计算机屏幕上创建一个显示空间。因此 wx.Window 是一个基类，所有可视元素都继承自它——包括输入域，下拉菜单，按钮等等。类 wx.Window 定义了所有可视 GUI 元素的公共行为，包括 positioning, sizing, showing, giving focus 等等。

如果要找一个对象在屏幕上表示出一个窗口，不要在 wx.Window 找而要到 wx.Frame 去找。wx.Frame 继承于 wx.Window。它可实现所有的行为特别是屏幕上的窗口的行为。一个`Frame（框架）'就是一个窗口，用户可以改变的它的大小和位置。它通常带有粗的边框和一个标题栏，并可以随意放置菜单栏、工具栏和状态栏。这个`Frame（框架）'可以容纳任何窗口，但这个窗口不是一个框架或者对话框。因此要在屏幕上创建一个窗口，你可以创建一个 wx.Frame（或者它的一个子类，例如 wx.Dialog），而不是创建 wx.Window。

在这个框架内，你要用很多 wx.Window 的子类去充实这个框架的内容，例如 wx.MenuBar、wx.StatusBar、wx.ToolBar、wx.Control 的子类（例如 wx.Button, wx.StaticText, wx.TextCtrl, wx.ComboBox 等等），或者 wx.Panel，它可以容纳各种 wx .Control 对象。所有可视元素（wx.Window 对象和它们的子类）都可以包含子元素（sub-element）。一个 wx.Frame 可以包含很多 wx.Panel 对象，而 wx.Panel 又依次包含大量的 wx.Button, wx.StaticText 和 wx.TextCtrl 的对象。

在本例中，stdframe——wx.Frame 的子类，用于创建`frame（框架）'。我们用 frame.Show（True）来显示这个框架。方法 SetTopWindow()告诉我们这个框架是应用程序的主框架中的一个（本例中只有一个主框架）。请注意 wx.Frame 的构造函数的形式：

```
wx.Frame(Parent, Id, "title")
```

wxPython 的大多数构造函数都有如下的形式：parent 作为第一个参数，Id 作为第二个参数。像本例中所示，None 和-1 都可能作为默认的参数，这表示对象没有 parent 并且有一个系统定义的 Id。

在 stdframe.__init__()中，我们创建了一个控制面板（panel）并放置在框架内部。

```
self.panel = panel
```

请注意，面板的 parent 是我们刚刚创建的框架对象，意味着这个面板是框架的子元件（subcomponent）。框架利用一个 wx.BoxSize 即 vbox 把面板放置在其内部。box sizer 的基本概念是窗口可以以简单的几何图形展开，经典的形式是一行、一列或是嵌套。一个 wx.BoxSizer 会将它的项目排列成一行或一列，这取决于传递到构造函数的目标参数。

例如：在我们的例子中有，

```
hbox = wx.BoxSizer(wx.HORIZONTAL)
```

这句告诉构造函数这个 sizer 是水平摆放的。代码中还有一个 vbox，它是这个应用程序窗口的主框架，是垂直摆放的。对于一个 sizer，最重要最有用的方法是 add()，它可以附加一个项目给 sizer。它的句法如下：

Add(self, item, proportion=0, flag=0)

代码中有

```
vbox.Add (self.src_fft.win, 4, wx.EXPAND)
```

`item'就是你要附加给 sizer 的项目，通常是一个 wx.Window 项目。它也可以是一个 child sizer。proportion 与 sizer 的子 sizer（child）能否在 wx.BoxSize 的主方向上改变它的大小有关。关于 flag，wx 中定义了很多标志（flag），它们用于确定 sizer 项目（item）的行为和窗口的边界。我们例子中的 wx.EXPAND 表示其中的项目会铺满整个空间。

现在我们来考虑另一个问题：我们定义了一个“大”类 wfm_rx_block，但是我们该在哪里使用它？为什么从未创建这个类的实例？其中的秘密与 stdpanel.__init__()有关。wfm_rx_block 的实例在这里创建同时流图也被启动了。

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, top_block_maker):
        # print "stdpanel.__init__"
        wx.Panel.__init__ (self, parent, -1)
        self.frame = frame

        vbox = wx.BoxSizer (wx.VERTICAL)
        self.top_block = top_block_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)

        self.top_block.start ()
```

我们在框架中放置了一个面板，但是在面板中放置什么呢？首先为面板创建一个新 sizer vbox。接着创建 top_block_maker 的实例，而 vbox 作为传递给它的参数（同时还有框架和面板本身）。在 wfm_rx_block.__init__()中，vbox 将利用 vbox.Add()附加一些频谱分析仪或示波器（wx.Window 对象）到 sizer。然后，

面板利用 sizer vbox 确定所有子窗口的位置和大小。最后，用 self.top_block.start() 启动流图，相应的数据会动态的显示在屏幕上。

让我们回顾一下 FM 接收机的代码：

```
if 1:
            self.src_fft      =      fftsink2.fft_sink_c(self.panel,
title="Data from USRP",fft_size=512, sample_rate=usrp_rate,
ref_scale=32768.0, ref_level=0, y_divs=12)
            self.connect (self.u, self.src_fft)
            vbox.Add (self.src_fft.win, 4, wx.EXPAND)
```

面板是作为 fft sink 的`parent'被传递给 fft_sink 模块的。于是 fft_sink 的输出就显示在面板上了。

关于类 fft_sink_c 有一些比较复杂的问题，这个函数被包装了很多层，在此我们就不详细去读代码了。但着重学习了它与 Python 以及 wxPython 之间的接口。实际上，另一个叫`Numeric' 的 Python package 也很有用。然而，我们无需知道所有的细节。只要知道在 Python 层面它是如何与 wxPython 以及其他 block 相配合的就足够了。

4.4.3 示波器

另一个重要的 GUI Radio 是软件示波器——scope_sink。在我们的例子中没有用到它，但如果你想在时域观察信号波形，它非常有用。它的使用方法与 fft_sink 非常相似。例如，我们可以看一下/gr-utils/src/python/usrp_fft.py 中的这一段：

```
elif options.oscilloscope:
            self.scope       =       scopesink2.scope_sink_c(panel,
sample_rate=input_rate)
```

这一段的功能就是直接把时域波形直接显示在屏幕上。

4.5 处理命令行参数

Python 完全支持在命令行运行所创建的程序，包括相应的命令行参数以及或短型或长型的标志（flag），标志用于指定各种操作。在 usrp_wfm_rcv.py 的例子中，我们用：

```
stdgui2.std_top_block.__init__ (self,frame,panel,vbox,argv)
```

每一个传送到程序的命令行参数都保存在 argv 中，argv 只是一个列表。在这个列表中，argv[0]就是命令本身（在我们的例子中是 usrp_wfm_rcv.py）。所以实际上所有的参量保存在 argv[1:]。

你可能想用短型或长型的标志（flag）来添加各种操作，像`-v'或`--help'。那么你需要模块 **optparse** 来帮助你。**optparse** 是一个强大且灵活的命令行解释器。例如代码中的这一段，设置了各种命令行参数：

```
parser=OptionParser(option_class=eng_option)
parser.add_option("-R",       "--rx-subdev-spec",       type="subdev",
default=None,
                          help="select   USRP   Rx   side   A   or   B
(default=A)")
        parser.add_option("-f",       "--freq",       type="eng_float",
default=100.1e6,
                          help="set      frequency      to      FREQ",
metavar="FREQ")
        parser.add_option("-g",       "--gain",       type="eng_float",
default=40,
                          help="set gain in dB (default is midpoint)")
        parser.add_option("-V",      "--volume",      type="eng_float",
default=None,
                          help="set volume (default is midpoint)")
        parser.add_option("-O",     "--audio-output",    type="string",
default="",
                          help="pcm  device  name.   E.g.,  hw:0,0  or
surround51 or /dev/dsp")
```

总结：这部分完成了对 FM 接收机例子的讨论。我们主要学习了在 GNU Radio 中 wxPython 是如何发挥它的作用的。在理解类的组织方式时我们可能有些困难，但只要我们有足够的耐心一定能够弄明白。而且，好在我们可以简单地以模板的形式使用这些代码，无需了解太多执行细节。

4.6 GNU Radio 中常用的 block

在用 Matlab 做仿真时，为了清楚有效地写代码，我们需要记住一些 Matlab 自带的函数和工具箱并能熟练使用它们。对于 GNU Radio 同样如此。GNU Radio 大约带有 100 个使用频率较高的模块。对一些确定的应用程序，我们可以在 Python 层面编程时使用这些现成的 block，而无需自己编写 block。所以在本文中，我们将介绍 GNU Radio 中常用的 block。

通过学习 Doxygen（文档生成器）生成的说明文档来了解 block 是个好方法。假设你已经安装了 Doxygen，那么在安装 gnuradio-core 和 usrp packages 时文档就已经生成了。它们在如下路径中：

/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html

/usr/local/share/doc/usrp-x.xcvs/html/index.html block 是由 C++编写的。例如，在 FM 接收机例子中(/gnuradio-core\src/python/gnuradio/blks2impl/wfm_rcv.py)，我们使用的 block gr.quadrature_demod_cf。这个 block 相当于 C++编写的类 gr_quadrature_demod_cf。SWIG 提供了它到 Python 的接口。SWIG 做了一些工作，以致从 Python 的角度来看，block 变成了在模块 gr 中定义的类 quadrature_demod_cf，这样我们就可以在 python 中，以 gr.quadrature_demod_cf 来接入这个 block 了。查看 Doxygen 生成的文档时，前缀 gr, qa, usrp 在 Python 中相当于模块名，前缀之后的部分是模块中 block 的名称，例如 quadrature_demod_cf, fir_filter_fff。所以如果编写一个 block 为`A_B_C_...'，那在 Python 中就称为`A.B_C_...'。

注意：这一节（4.6 节）中的代码版本较老，但是用于介绍模块的基本信息足够了。若要使用，应该很容易在新版本中找到。

4.6.1 信号源

4.6.1.1 正弦信源和常量信源

Block：gr.sig_source_X

用法：

gr.sig_source_c [f, i, s] (double sampling_freq,
gr_waveform_t waveform,
double frequency,
double amplitude,
gr_complex [float, integer, short] offset)

注：前缀 X 表示信号源的数据类型。它可以是 c (复数型), f (浮点型), i (4 字节整型) or s (2 字节短整型)。偏移参量（offset argument）与信号类型相同。波形参量指明波形的类型。gr_waveform_t 是枚举类型，在 gr_sig_source_waveform.h 中定义。它的值可以是：

gr.GR_CONST_WAVE

gr.GR_COS_WAVE

gr.GR_SIN_WAVE

使用 gr.GR_CONST_WAVE 时，输出为一个直流，即幅值+偏移量。

4.6.1.2 噪声源（Noise sources）

Block：gr.noise_source_X

用法：

gr.noise_source_c [f, i, s] (gr_noise_type_t type,
float amplitude,
long seed)

注：前缀 X 表示信号源的数据类型。它可以是 c (复数型), f (浮点型), i (4 字节整型) or s (2 字节短整型)。类型参量指明噪声的类型。gr_noise_type_t 是在 gr_noise_type.h 定义的枚举类型。它的值可以是：

GR_UNIFORM

GR_GAUSSIAN

GR_LAPLACIAN

GR_IMPULSE

GR_UNIFORM 用于产生在[-amplitude, amplitude]上均匀分布的信号。GR_GAUSSIAN 产生 0 均值方差为 amplitude 的高斯噪声类似的，GR_LAPLACIAN 和 GR_IMPLUSE 分别表示 Laplacian 分布的噪声和 impulse 脉冲噪声。所有这些噪声信源都基于伪随机序列发生器 block gr.random。你可以查看/gnuradio-core/src/lib/general/gr_random.h(cc)，它会告诉你如何产生各种分布的随机序列。

4.6.1.3空信源（Null sources）

Block：gr.null_source

用法：

gr.null_source (size_t sizeof_stream_item)

注：gr.null_source 用于产生常数 0。参量 sizeof_stream_item 表示 0 数据流的数据类型，例如 gr_complex（复数型）, float（浮点型）, integer（整型）等等。

4.6.1.4矢量信源（Vector sources）

Block：gr.vector_source_X

用法：

gr.vector_source_c [f, i, s, b] (const std::vector< gr_complex > & data,
bool repeat = false)

(gr_complex can be replaced by float, integer, short, unsigned char)

注：前缀 X 表示信号源的数据类型。它可以是 c (复数型)、f (浮点型)、i (4 字节整型)、s (2 字节短整型)或 b（1 字节无符号字符型）。这些信源的数据是矢量。参量 **repeat** 决定矢量数据是否重复的产生。例如，可以在 Python 中这样用此 block：

src_data = (-3, 4, -5.5, 2, 3)

src = gr.vector_source_f (src_data, TRUE)

于是这个 src 就会连续的产生“-3, 4, -5.5, 2, 3, -3, 4, -5.5, 2, 3,…”。

4.6.1.5文件信源（File sources）

Block：gr.file_source

用法：

```
gr.file_source ( size_t itemsize,
                 const char * filename,
                 bool repeat )
```

注：gr.file_source 的作用是从文件中读出数据流。文件名由 **filename** 指定。第一个参量 **itemsize** 表示信号流的数据类型，例如 gr_complex（复数型），float（浮点型），unsigned char（无符号字符型）。参量 **repeat** 决定文件的数据是否重发。例如可以在 Python 中这样用此 block：

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/payload.dat", TRUE)
```

4.6.1.6音频信源（Audio source）

Block：gr.audio_source

用法：

```
gr.audio_source (int sampling_rate)
```

注：audio_source 的作用是从音频线读入数据。参量 sampling_rate 指明信源的数据速率，单位是采样值/每秒。

4.6.1.7USRP 信源（USRP source）

Block：usrp.source_c [s]

用法：

```
usrp.source_c (s) (int which_board,
                   unsigned int decim_rate,
                   int nchan = 1,
                   int mux = -1,
                   int mode = 0 )
```

这是最常用的一种数据源了。用 GNU Radio 设计接收机，即工作在 RX 通道的设备，你就会用到 USRP 作为数据源。前缀 c (complex（复数型）), or s (short（短整型）)表示 USRP 数据流的数据类型。复数信源（通过数字下变频得到的 I/Q quadrature）更常用。which_board 表示将启用哪个 USRP，如果只有一个 USRP 此参量可能为 0。decim_rate 指明数字下变频的抽取因子 D。**nchan** 则指

明了信道的数量，1、2 或 4。**mux** 用于设置输入 MUX，它决定了哪一个 ADC（或常量 0）与 DDC 输入端相连。`-1'表示保持默认设置。**mode** 设置了 FPGA 的模式，接触的较少。其默认值是 0，表示标准模式。通常我们只设置前两个参数，其他的使用默认值。例如：

```
usrp_decim = 250
```

```
src = usrp.source_c (0, usrp_decim)
```

4.6.2 信宿（Signal Sinks）

4.6.2.1空信宿（Null sinks）

Block：gr.null_sink

用法：

```
gr.null_sink ( size_t sizeof_stream_item )
```

注：gr.null_sink 什么都不做，只是接收数据流。参量 sizeof_stream_item 指定了0 数据流的数据类型，例如 gr_complex（复数型）, float（浮点型）, integer（整型）等等。Null_sink 通常用来给一个没有结尾的流图收尾，否则最后一个模块的输出节点就会“悬空”，这样流图是无法运行的。

4.6.2.2矢量信宿（Vector sinks）

Block：gr.vector_sink_X.

用法：

```
gr.vector_sink_c [f, i, s, b] ()
```

注：前缀 X 表示信号源的数据类型。它可以是 c (复数型)、f (浮点型)、i (4 字节整型)、s (2 字节短整型)或 b（1 字节无符号字符型）。这些信宿的数据是矢量。例如，可以在 Python 中这样使用此 block：

```
dst = gr.vector_sink_f ()
```

4.6.2.3文件信宿（File sinks）

Block：gr.file_sink

用法：

```
gr.file_sink ( size_t itemsize,
               const char * filename )
```

注意：gr.file_sink 用于向文件写入数据流。文件名由 **filename** 指定。第一个参量 **itemsize** 决定了输入数据流的数据类型，如 gr_complex（复数型）, float（浮点型）, unsigned char（无符号字符型）。如例所示，以在 Python 中这样使用此 block：

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/rx.dat")
```

4.6.2.4音频信宿（Audio sink）

Block：gr.audio_sink

用法：

gr.audio_source (int sampling_rate)

注：当完成信号处理并想通过扬声器播放信号时，你需要用到音频信宿。audio_sink 以 sampling_rate 指定的采样率向声卡输出数据。

4.6.2.5USRP 信宿（USRP sink）

Block：usrp.sink_c [s]

用法：

```
usrp.sink_c (s) ( int which_board,
                  unsigned int interp_rate,
                  int nchan = 1,
                  int mux = -1 )
```

注：用 GNU Radio 设计发射机时，即工作在 TX 通道的设备，可能你需要 USRP 作为数据信宿。后缀 c (complex（复数型）), or s (short（短整型）)指明了进入 USRP 的数据流的数据类型。常用的数据类型是复数型。which_board 表示将启用哪个 USRP，如果只有一个 USRP 此参量可能为 0。interp_rate 指明了插值器的差值因子 I。插值速率 I 必须在[4, 512]范围内且必须是 4 的倍数。**nchan** 则指明了通道的数量，1 或 2。**mux** 用于设置输入 MUX，它决定了哪一个 DAC（或常量 0）与插值器输出端相连。`-1'表示使用默认值。通常我们只设置前两个参数，其他的使用默认值。例如：

```
usrp_interp = 256
src = usrp.sink_c (0, usrp_interp)
```

4.6.3 简单运算（Simple Operators）

4.6.3.1添加常量（Adding a constant）

Block：gr.add_const_XX

用法：

gr.add_const_cc [ff, ii, ss, sf] (gr_complex [float, integer, short] k)

注：前缀XX包含两个参数。第一个指明输入数据流的类型，而第二个指明输出数据流的类型。它可以是cc (复数到复数), ff (浮点到浮点), ii (4字节整型到4字节整型), ss (2字节整型到2字节整型) 或sf (短整型到浮点型)。gr.add_const_XX block添加了一个常量到输入数据流以便信号有一个不同的偏移量。请注意对于gr.add_const_sf，参量k是浮点型。我们向短整型数据流添加一个浮点常量，输出数据流就会变成浮点型。

4.6.3.2加法器

Block：gr.add_XX

用法：

gr.add_cc [ff, ii, ss] ( )

注：前缀XX指明输入和输出数据流的数据类型。gr.add_XX用于将所有的输入流加在一起。每一个输入流的类型必须相同。当在Python中使用它时，多个上级block都可以与它连接，例如：

```
adder = gr.add_cc ()
fg.connect (stream1, (adder, 0))
fg.connect (stream2, (adder, 1))
fg.connect (stream3, (adder, 2))
fg.connect (adder, outputstream)
```

加法器的输出为stream1+stream2+stream3。

4.6.3.3减法器（Subtractor）

Block：gr.sub_XX

用法：

gr.sub_cc [ff, ii, ss] ( )

注：前缀XX指明了输入和输出流的数据类型。gr.sub_XX可以计算input_0与所有输入流相减。output = input_0 - input_1 - input_2....每个输入流的类型必须相同。当在Python中使用它时，多个上级模块都可以和它连接，例如：

```
subtractor = gr.sub_cc ()
fg.connect (stream1, (subtractor, 0))
fg.connect (stream2, (subtractor, 1))
fg.connect (stream3, (subtractor, 2))
fg.connect (subtractor, outputstream)
```

减法器的输出为stream1-stream2-stream3。

4.6.3.4数乘（Multiplying a constant）

Block：gr.multiply_const_XX

用法：

gr.add_const_cc [ff, ii, ss] (gr_complex [float, integer, short] k)

注：前缀XX指定了输入输出数据流的数据类型。gr.multiply_const_XX block使输入数据流与常数k相乘。

4.6.3.5乘法器（Multiplier）

Block：gr.multiply_XX

用法：

gr.multiply_cc [ff, ii, ss] ( )

注：前缀XX指定了输入输出数据流的数据类型。gr.multiply_XX可以计算输入信号流的乘法。每个输入流的类型必须相同。当在Python中使用它时，多个上级模块都可以与它连接，例如：

```
multiplier = gr.multiply_cc ()
fg.connect (stream1, (multiplier, 0))
fg.connect (stream2, (multiplier, 1))
fg.connect (stream3, (multiplier, 2))
fg.connect (multiplier, outputstream)
```

乘法器的输出为stream1*stream2*stream3。

4.6.3.6除法器（Divider）

Block：gr.divide_XX

用法：

gr.divide_cc [ff, ii, ss] ( )

注：前缀XX指定了输入输出数据流的数据类型。gr.divide_XX可计算输入信号流的除法。output = input_0 / input_1 / input_2.... 每个输入流的类型必须相同。当在Python中使用它时，多个上级模块都可以与它连接，例如：

```
divider = gr.divide_cc ()
fg.connect (stream1, (divider, 0))
fg.connect (stream2, (divider, 1))
fg.connect (stream3, (divider, 2))
fg.connect (divider, outputstream)
```

除法器的输出为stream1/stream2/stream3。

4.6.3.7log 函数（Log function）

Block：gr.nlog10_ff

用法：

gr.nlog10_ff (float n, unsigned vlen)

注：gr.nlog10_ff可以计算n*$\log_{10}(\mathit{input})$。输入和输出都是浮点数。若忽略参量**vlen**，即矢量长度，则默认值为1。

4.6.4 类型转换（Type Conversions）

4.6.4.1复数型的转换（Complex Conversions）

Block：

gr.complex_to_float

gr.complex_to_real

gr.complex_to_imag

gr.complex_to_mag

gr.complex_to_arg

用法：

gr.complex_to_float(unsigned int vlen)

gr.complex_to_real(unsigned int vlen)

gr.complex_to_imag(unsigned int vlen)

gr.complex_to_mag(unsigned int vlen)

gr.complex_to_arg(unsigned int vlen)

注：参量**vlen**表示矢量长度，我们几乎总是使用默认值1。所以可以忽略此参量。这些block可以将复信号转变成实部和虚部分离的浮点数据流，即将复信号变为实部+虚部，或复信号的模+相位角的形式。请注意gr.complex_to_float block可以有1或2个输出。如果只有一个输出，那么此输出端是复信号的实部。它的结果与gr.complex_to_real等效。

4.6.4.2浮点类型的转换（Float Conversions）

Block：

gr.float_to_complex

gr.float_to_short

gr.short_to_float

用法：

gr.float_to_complex ( )

gr.float_to_short ( )

gr.short_to_float ( )

注：这些block就像`adapters（适配器）'，可提供两个不同类型block之间的接口。请注意gr.float_to_complex block可以有1或2个输入。如果只有一个，那输入信号就是输出信号的实部，而虚部为常0。如果有两个输入，则分别为输出信号的实部和虚部。

4.6.5 滤波器（Filters）

4.6.5.1FIR 设计器（FIR Designer）

Block：gr.firdes

注：gr.firdes有若干个静态公共元函数，用于设计不同类型的FIR滤波器。这些函数能返回一个包含FIR系数的矢量。这个返回的矢量通常作为其他FIR滤波器block的一个参量。

4.6.5.2低通滤波器（Low Pass Filter）

用法：

```
vector< float > gr.firdes::low_pass ( double gain,
                                      double sampling_freq,
                                      double cutoff_freq,
                                      double transition_width,
                                      win_type window = WIN_HAMMING,
                                      double beta = 6.76) [static]
```

注：low_pass是类gr.firdes中的一个静态公共元函数。它可以设计FIR滤波器系数（抽头）。这里的参量**window**是用于FIR滤波器设计的窗口的类型。有效值包括：

WIN_HAMMING

WIN_HANN

WIN_BLACKMAN

WIN_RECTANGULAR

4.6.5.3高通滤波器（High Pass Filter）

用法：

```
vector< float > gr.firdes::high_pass ( double gain,
                                       double sampling_freq,
                                       double cutoff_freq,
                                       double transition_width,
                                       win_type window = WIN_HAMMING,
                                       double beta = 6.76) [static]
```

4.6.5.4带通滤波器（Band Pass Filter）

用法：

```
vector< float > gr.firdes::band_pass ( double     gain,
                                       double     sampling_freq,
```

double low_cutoff_freq,
double high_cutoff_freq,
double transition_width,
win_type window = WIN_HAMMING,
double beta = 6.76) [static]

4.6.5.5带阻滤波器（Band Reject Filter）

用法：

vector< float > gr.firdes::band_reject (double gain,
double sampling_freq,
double low_cutoff_freq,
double high_cutoff_freq,
double transition_width,
win_type window = WIN_HAMMING,
double beta = 6.76) [static]

4.6.5.6希尔伯特滤波器（Hilbert Filter）

用法：

vector< float > gr.firdes::hilbert (unsigned int ntaps,
win_type windowtype = WIN_RECTANGULAR,
double beta = 6.76) [static]

注：gr.firdes::hilbert用于设计希尔伯特变换滤波器。**ntaps**指明了抽头个数，必须为奇数。

4.6.5.7升余弦滤波器（Raised Cosine Filter）

用法：

vector< float > gr.firdes::root_raised_cosine (double gain,
double sampling_freq,
double symbol_rate,
double alpha,
int ntaps) [static]

注：gr.firdes::root_raised_cosine用于设计平方根余弦滤波器。参量**alpha**是多余带宽因子。**ntaps**为抽头个数。

4.6.5.8高斯滤波器（Gaussian Filter）

用法：

```
ctor< float > gr.firdes::gaussian ( double          gain,
                                    double          sampling_freq,
                                    double          symbol_rate,
                                    double          bt,
                                    int             ntaps ) [static]
```

注：gr.firdes::gaussian用于设计高斯滤波器。参量**ntaps**为抽头个数。

4.6.5.9FIR 抽取滤波器（FIR Decimation Filters）

Block：

gr.fir_filter_ccc

gr.fir_filter_ccf

gr.fir_filter_fcc

gr.fir_filter_fff

gr.fir_filter_fsf

gr.fir_filter_scc

用法：

```
gr.fir_filter_ccc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_ccf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fcc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_fff (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fsf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_scc (int decimation,
                   const std::vector< gr_complex > & taps )
```

注：这些block都有一个三元（3-character）的前缀。第一个和第二个分别表示输入和输出信号的数据类型，第三个表示FIR滤波器抽头的数据类型。每个block有两个参量。一个是FIR滤波器的抽取率。如果是1，那么它就是一个1:1的FIR滤波器。另一个参量**taps**是FIR抽头系数矢量，可以从gr.firdes block获得。

4.6.5.10 FIR 插值滤波器（FIR Interpolation Filters）

Block：

gr.interp_fir_filter_ccc

gr.interp_fir_filter_ccf

gr.interp_fir_filter_fcc

gr.interp_fir_filter_fff

gr.interp_fir_filter_fsf

gr.interp_fir_filter_scc

用法：

gr.interp_fir_filter_ccc (unsigned interpolation,
const std::vector< gr_complex > & taps)

gr.interp_fir_filter_ccf (unsigned interpolation,
const std::vector< float > & taps)

gr.interp_fir_filter_fcc (unsigned interpolation,
const std::vector< gr_complex > & taps)

gr.interp_fir_filter_fff (unsigned interpolation,
const std::vector< float > & taps)

gr.interp_fir_filter_fsf (unsigned interpolation,
const std::vector< float > & taps)

gr.interp_fir_filter_scc (unsigned interpolation,
const std::vector< gr_complex > & taps)

注：与抽取滤波器类似，这些block都有一个三元（3-character）的前缀。第一个和第二个分别表示输入和输出信号的数据类型，第三个表示FIR滤波器抽头的数据类型。

每个 block 有两个参量。一个是 FIR 滤波器的插值率。另一个参量 **taps** 是 FIR 抽头系数矢量，可以从 gr.firdes block 获得。

4.6.5.11 带有 FIR 抽取滤波器的 DDC（数字下变频器）

Block：

gr.freq_xlating_fir_filter_ccc

gr.freq_xlating_fir_filter_ccf

gr.freq_xlating_fir_filter_fcc

gr.freq_xlating_fir_filter_fcf

gr.freq_xlating_fir_filter_scc

gr.freq_xlating_fir_filter_scf

用法：

gr.freq_xlating_fir_filter_ccc [ccf, fcc, fcf, scc, scf]
(int decimation,

const std::vector< gr_complex [float] > & taps,

double center_freq,

double sampling_freq)

注：这些block都有一个三元（3-character）的前缀。第一个和第二个分别表示输入和输出信号的数据类型，第三个表示FIR滤波器抽头的数据类型。这些block是与频率变换相结合的FIR滤波器。在FPGA中带有数字下变频器（DDC），将信号从中频搬移到基带。接着后面的抽取器对信号进行降采样并用低通滤波器选出一段窄带信号。这些block除了软件上稍有不同，功能是一样的。这些类有效的将FIR滤波器和抽取技术与频率变换（典型的是下变频）相结合。它非常适合于`channel selection filter（信道选择滤波器）'，并且能够有效地从宽带输入信号中选择和抽取窄带信号。

4.6.5.12 希尔伯特变换滤波器（Hilbert Transform Filter）

Block：gr.hilbert_fc

用法：

gr.hilbert_fc(unsigned int ntaps)

注：gr.hilbert_fc是希尔伯特变换器。输出的实部是输入信号的延时。虚部是输入信号的希尔伯特变换（90度相移）。当使用此block时，只需指明抽头的个数**ntaps**。Hilbert filter设计器gr.firdes::hilbert通常在block gr.hilbert_fc执行过程中用到。

4.6.5.13 延时组合滤波器（Filter-Delay Combination Filter）

Block：gr.filter_delay_fc

用法：

gr.filter_delay_fc(const std::vector< float > & taps)

注：这是一个滤波器延时组合模块。这个block带有一个或两个浮点数据流并输出一个复数流。如果只有一个浮点流输入，输出的实部是输入信号的延时，而虚部是滤波后的输出。如果有两个浮点流连接到输入，那么输出的实部是第一个输入的延时，而虚部是滤波后的输出。实部通道的延时可以解释由虚部通道滤波器引入的群延时。在初始化此block之前需要利用gr.firdes计算滤波器抽头。

4.6.5.14 IIR 滤波器

Block：gr.iir_filter_ffd

用法：

gr.iir_filter_ffd (const std::vector< double > & fftaps,

const std::vector< double > & fbtaps)

注：前缀ffd表示浮点输入、浮点输出和双抽头。此IIR滤波器采用直接I型结构，其中**fftaps**是前馈抽头，**fbtaps**是反馈抽头。**fftaps**和**fbtaps**必须有相等的抽头数。输入和输出满足差分方程：

$$y[n]-\sum_{k=1}^{N}a_k y[n-k]=\sum_{k=0}^{M}b_k x[n-k]$$

相应的系统函数为：

$$H(z)=\frac{\sum_{k=0}^{M}b_k z^{-k}}{1-\sum_{k=1}^{N}a_k z^{-k}}$$

请注意，有些文章中定义系统函数时分母中带有`+'号。如果你的定义习惯也是如此，那么需要去掉反馈抽头。

4.6.5.15 单极 IIR 滤波器（Single Pole IIR Filter）

Block：gr.single_pole_iir_filter_ff

用法：

gr.single_pole_iir_filter_ff (double alpha, unsigned int vlen)

注：这是一个带有浮点输入和浮点输出的单极IIR滤波器。输入输出满足差分方程：

$$y[n]-(1-\alpha)y[n-1]=\alpha x[n]$$

相应的系统函数为：

$$H(z)=\frac{\alpha}{1-(1-\alpha)z^{-1}}$$

请注意，有些文章中定义系统函数时分母中带有`+'号。如果你的定义习惯也是如此，那么需要去掉反馈抽头。**vlen**表示矢量长度。通常为默认值1，所以不用管它。

4.6.6 FFT

Block：

gr.fft_vcc

gr.fft_vfc

用法：

```
gr.fft_vcc ( int fft_size,
             bool forward,
             bool window )
gr.fft_vfc ( int fft_size,
             bool forward,
             bool window )
```

注：这些block用于计算输入序列的快速傅里叶变换。对于gr.fft_vcc，它计算正向或反向FFT，复矢量输入复矢量输出。而对于gr.fft_vfc，它计算正向FFT，浮点矢量输入复矢量输出。

4.6.7 其他一些有用 block

4.6.7.1FM 调制与解调（FM Modulation and Demodulation）

Block：

gr.frequency_modulator_fc

gr.quadrature_demod_cf

用法：

gr.frequency_modulator_fc (double sensitivity)

gr.quadrature_demod_cf (float gain)

注：gr.frequency_modulator_fc block是FM调制器。输出复信号的瞬时频率随输入浮点信号成比例变化。我们在FM接收机例子中遇到过gr.quadrature_demod_cf。它可以计算输入信号的瞬时频率。

4.6.7.2数控振荡器（Numerically Controlled Oscillator）

Block：gr.fxpt_nco

用法：此block用于产生正弦波。我们可以通过类的若干个公共元函数，设置或改变振荡器相位和频率。函数包括：

void set_phase (float angle)

void adjust_phase (float delta_phase)

void set_freq (float angle_rate)

void adjust_freq (float delta_angle_rate)

void step ()

void step (int n)

float get_phase () const

float get_freq () const

void sincos (float *sinx, float *cosx) const

float cos () const

float sin () const

用cos()或sin()函数来获得正弦信号。它们根据当前相位计算正弦值和余弦值。**freq**为连续采样值之间的相位差。调用方法step()时，当前相位增加**freq**。

4.6.7.3数字传输 block（Blocks for digital transmission）

Block：

gr.bytes_to_syms

gr.simple_framer

gr.simple_correlator

用法：

gr.bytes_to_syms ( )

gr.simple_framer (int payload_bytesize)

gr.simple_correlator (int payload_bytesize)

注：gr.bytes_to_syms ( )将一字节序列（例如一无符号字符流）转换成一个二进制序列，即数字二进制符号。gr.simple_framer以payload_bytesize的长度将一字节数据打包。必要的同步和控制信息加在头部（head）。gr.simple_correlator是序列相关器，它使符号同步且帧同步，这样就可以正确的检出信息了。这些block的实际执行过程有点复杂。请参阅FSK的例子fsk_r[t]x.py和这些block的源代码。当你想设计数字传输系统时，这些block非常有用。

本节介绍了一些GNU Radio中常用的block。熟练的运用这些block很重要。无疑我们只讨论了很小的一部分。还有很多更高阶的block和使用频率较低的block我们没有讨论。同时，有越来越多的block添加到GNU Radio中来（也许有一天你的模块也被加入进来了）。本文的介绍比较简单。如果你想知道更多block的细节，请到它的说明文档页面直接阅读源代码。源代码是深入理解block的最佳工具。通过学习一些例子来理解block也是非常好的方式。

4.7 如何编写C++ blocks

4.7.1 最简单的方法——利用模板

安装好 gnuradio 以后，我们可以调用里面的数字信号处理模块，也可以通过将几个模块级联生成一个新的模块。这些都可以通过 python 的强大的粘合功能实现。但是如果要生成一个全新的数字信号处理模块，就不能简单的通过 python 语言来实现了。我们需要自己编写 c 语言的源码程序，最后编译成可以调用的 python 模块，而具体如何实现呢？如果源码程序(.h, .cc)，编译文件(Makefile)都要自己编写，首先需要很强的专业知识，而且非常的繁琐。

最简单的方法就是下载一个模板，然后进行简单的修改，得到自己所需要的模块。gnuradio 也提供了编写自己信号处理模块的模板，我们可以在下面的地址上下载到：

ftp://ftp.gnu.org/gnu/gnuradio/

下载最新版本的 gr-howto-write-a-block-3.2.2.tar.gz，然后修改源码，得到你自己定义的模块就是这么简单。

下面简单说一下怎么修改源码。

4.7.1.1 如何用模板编写自己的函数

我们用 howto_square_ff 来复制一个新的函数 howto_add_ff，一个输入为浮点数的加法器，相当于输入的数值乘以 2。

我们把整个 gr-howto-write-a-block-3.2.2 重命名为 test_example。那么，在这个目录下，需要改动的文件一共有 4 个。

test_example/src/lib/howto_add_ff.h	新建
test_example/src/lib/howto_add_ff.cc	新建
test_example/src/lib/howto.i	修改
test_example/src/lib/Makefile.am	修改

- 编写 howto_add_ff.h

修改.h 和.cc 文件，就是一个创建自己的函数模块的过程。你可以设计自己的接口参数，在 general_work()部分用 c++语言实现自己需要的功能。不过这里我们首先来用一个简单的加法的例子来说明一下，没有修改函数接口。

把 howto_square_ff.h 另存为 howto_add_ff.h。把所有的“square_ff”替换为“add_ff”，注意区分大小写。

- 编写 howto_add_ff.cc

把 howto_square_ff.cc 另存为 howto_add_ff.cc。把所有的“square_ff”替换为“add_ff”，注意区分大小写。修改函数的功能，把乘法改成加法：

```
int
howto_add_ff::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];
  }
```

- 修改 howto.i 文件

修改这个文件的目的是把 c 和 python 联系在一起。首先把 howto_add_ff.h 头文件添加进去。

```
#include "howto_add_ff.h"
```

然后添加函数的接口定义。把 howto_square_ff 函数的部分复制一份，粘贴在最后，把 square 都替换成 add。

```
GR_SWIG_BLOCK_MAGIC(howto,add_ff);
```

```
howto_add_ff_sptr howto_make_add_ff ();

class howto_add_ff : public gr_sync_block
{
private:
  howto_add_ff ();
};
```

➢ 修改 Makefile.am 文件

Makefile.am 文件是用来生产 makefile 的。因此要把编译需要的.h 和.cc 文件添加进去。

添加头文件。

```
# C/C++ headers get installed in ${prefix}/include/gnuradio
grinclude_HEADERS =             \
        howto_square_ff.h       \
        howto_square2_ff.h      \
        howto_add_ff.h
```

添加.cc 文件。

```
# additional sources for the SWIG-generated library
howto_la_swig_sources =         \
        howto_square_ff.cc      \
        howto_square2_ff.cc     \
        howto_add_ff.cc
```

好了，现在就可以编译了。回到 test_example 目录下，运行：

$ ./bootstrap

$ ./configure

$ make

为了检验模块编写得是否正确，有一个 qa_howto.py 文件专门用来做测试。我们在 test_example/src/python/qa_howto.py 中添加一段测试代码：

```
def test_003_add_ff (self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (-6, 8, -11, 4, 6)
        src = gr.vector_source_f (src_data)
        sqr = howto.add_ff ()
        dst = gr.vector_sink_f ()
        self.tb.connect (src, sqr)
        self.tb.connect (sqr, dst)
        self.tb.run ()
        result_data = dst.data ()
        self.assertFloatTuplesAlmostEqual (expected_result, result_data,6)
```

然后在 test_example 下运行：

$ make check

或者在 test_example/src/python 下运行：

$ ./qa_howto.py

如果程序没有错误，就会看到以下结果：

```
Ran 3 tests in 0.005s

OK
PASS: run_tests
==================
All 1 tests passed
==================
```

测试通过了！这样你就可以把这个模块安装到 GNU Radio 的安装目录去了。

$ make install

不过，我们经常是懒得写测试程序的，而且，有些测试程序必须以前面模块的输出为输入。因此，我们常常是 make install 以后，在开发的流图中测试。

4.7.1.2如何用模板创建自己的 package

前面我们讲解了如何编写自己的函数。可是这个新函数是放在 howto 这个 package 里的，使用的时候引用为 howto.add_ff。我们当然不希望用这么个名字。所以，通常会创建一个自己的 package。下面我们就把这个 howto_add_ff 这个函数放到 test_example 这个 package 里。

- 去掉模板中的函数

首先，把 test_example/src/lib 目录下其他两个函数 howto_square_ff 和 howto_square2_ff 相关的文件删掉。howto.i, Makefile.am 和 qa_howto.py 中相关的内容也删掉。只留下 howto_add_ff。

- 重命名自己创建的函数

把 howto_add_ff.h，howto_add_ff.cc，howto.i 文件重命名为 test_example_add_ff.h, test_example_add_ff.cc, test_example.i。

- 把剩下的所有文件中的 howto 替换为 test_example

接下来，把所有文件中的 howto 替换为 test_example。其实，只是有些地方需要替换，并不是所有的“howto”都是起作用的。但是为了操作简单，而且也为了以后的程序看起来清楚明了，就简单的用查找替换把 howto 全部换掉了。

好了，现在编译测试：

$ ./bootstrap

$ ./configure

$ make check

输出是这样的：

```
Ran 1 tests in 0.001s
```

```
OK
PASS: run_tests
==================
All 1 tests passed
==================
```

最后，安装：

$ make install

OK，经过了这两个小节的各个步骤，一个新的 package 叫做 test_example 就创建好了，它里面只有一个函数叫做 add_ff。在 python 脚本中，我们可以这样使用它：

import test_example

…

my_add = test_example.add_ff()

如果你觉得上面这种方法麻烦的话，可以自己写一个小程序，批处理查找替换和重命名。那样就更方便快捷了。

4.7.2 block 的结构和原理

前面我们讲了如何利用模板创建一个block，有点像“照葫芦画瓢”，但是知其然不知其所以然。在这一节中，将介绍关于GNU Radio的block的基本结构，是如何与python联系在一起的，block的运行机制等等。

从 Python 角度来看，GNU Radio 提供了数据流的抽象。基本概念是信号处理模块之间的连接。这种抽象由类 gr.flow_graph 实现。每个 block 都有一套输入输出端口。每个端口都有相应的数据类型结构。最常见的数据类型是浮点型和 gr_complex（相当于 std::complex<float>）。

从宏观的角度来看，端口的数据流是无限的。而在 C++的角度，数据流是一系列一定大小一定类型的“段”排列组成的。

写 block 时，将它们变成共享的库，这样就能用导入机制，动态的载入到 Python 中。SWIG（Simplified Wrapper and Interface Generator），简化封装接口生成器，用于生成和粘合代码以便在 Python 中使用。写新的信号处理 block 包括创建 3 个文件：.用于定义一个新的类的 h 和.cc 文件，.和用于促使 SWIG 粘合此新的类与 python 的.i 文件。新类必须衍生自 gr_block 或它的某个子类。

4.7.2.1所有信号处理 block 的基类：gr_basic_block/gr_block

C++类 gr_basic_block 是所有信号处理 block 的基础。要创建的新 block 类必须继承于 gr_basic_block 或它的某个子类。gr_block 派生自 gr_basic_block。所以首先通过 gr_basic_block.h 和 gr_block.h 学习一下 block 的基础的成员变量和成员函数。

让我们先看一看gr_basic_block定义的成员变量：

```
protected:
    friend class gr_flowgraph;
    friend class gr_flat_flowgraph; // TODO: will be redundant

    enum vcolor { WHITE, GREY, BLACK };

    std::string          d_name;
    gr_io_signature_sptr d_input_signature;
    gr_io_signature_sptr d_output_signature;
    long                 d_unique_id;

    vcolor               d_color;
```

d_name 是一个字符串用于保存 block 的名称。**d_unique_id** 是长整型，作为 block 的`ID'。为的是方便调试。

什么是 d_input_signature 和 d_output_signature？d_input_signature 的数据类型是 gr_io_signature_sptr，这个情况有点复杂，将在下一小节解释。

gr_block 的成员变量有：

```
private:

  int                    d_output_multiple;
  double                 d_relative_rate;      // approx output_rate / input_rate
  gr_block_detail_sptr  d_detail;                  // implementation details
  unsigned               d_history;

  bool                   d_fixed_rate;
```

后面将具体解释其中的一些成员变量。

4.7.2.2`IO signature'类: gr_io_signature

类gr_io_signature在/src/lib/runtime/gr_io_signature.h中被定义。如它名称所示，它像是block输入和输出流的一个说明，告诉我们一些基本信息。以下是gr_io_signature.h的部分定义：

```
class gr_io_signature {
  int                   d_min_streams;
  int                   d_max_streams;
  std::vector<int>      d_sizeof_stream_item;

  gr_io_signature(int min_streams, int max_streams,
                  const std::vector<int> &sizeof_stream_items);

  friend gr_io_signature_sptr
  gr_make_io_signaturev(int min_streams,
                        int max_streams,
                        const std::vector<int> &sizeof_stream_items);

 public:

  static const int IO_INFINITE = -1;

  ~gr_io_signature ();

  int min_streams () const { return d_min_streams; }
  int max_streams () const { return d_max_streams; }
```

```
  int sizeof_stream_item (int index) const;
  std::vector<int> sizeof_stream_items() const;
};
```

对于一个block的输入（输出），类定义了数据流个数的上限和下限（d_min_streams和d_max_streams）。数据流中一个数的大小（所占字节数）由d_sizeof_stream_item给出。创建block时，需要为输入和输出流指明`signature'。

4.7.2.3智能指针：Boost

gr_io_signature_sptr是什么数据类型？让我们研究一下。gr_runtime_types.h包括在gr_basic_block.h中。在gr_runtime_types.h，gr_io_signature_sptr定义如下：

```
typedef boost::shared_ptr<gr_io_signature>      gr_io_signature_sptr;
```

另外，gr_runtime_types.h 首先要包含 gr_type.h。gr_type.h 有一个头文件：

```
#include <boost/shared_ptr.hpp>
```

这样GNU Radio就可以利用**Boost**智能指针了。

➢ 那么什么是 Boost?

Boost是C++库的集合，GNU Radio的一个预安装包。**Boost**从很多方面扩展了C++，例如算法实现，数学/数值，输入/输出，迭代算法等等。感兴趣的程序员可以参阅http://www.boost.org。

➢ 什么是智能指针（smart pointer）？

GNU Radio从Boost引入了一个非常好的东西：smart_ptr库，称之为智能指针。智能指针能将指针储存在动态分配的对象中。它有点像C++中的指针，此外它还能在适当的时候自动删除指向的对象。智能指针非常有用，特别是在异常处理方面，因为它可以确保动态分配对象正确销毁。它们还可以被用作跟踪多个用户共享的动态分配对象。概念上说，智能指针看起来像是拥有了这个所指向的对象，因此当不再需要时可以删除这个对象。实际上，智能指针通常以类的模板的形式被定义。smart_ptr库提供了五个智能指针的类模板，但在GNU Radio中，我们只用到其中的一个：shared_ptr，在<boost/shared_ptr.hpp>中定义。当多个指针指向同一个对象时使用shared_ptr。

➢ 在 GNU Radio 中如何使用智能指针？

首先需要加入头文件<boost/shared_ptr.hpp>。这样就可以用它定义一个智能指针：

```
boost::shared_ptr<T> pointer_name
```

这些智能指针的类模板有个模板参数，T，它表示智能指针所指向的对象的类型。例如boost::shared_ptr<gr_io_signature>声明了一个指针，指向一个类型为gr_io_signature的对象。好了，目前为止已经解释了什么是gr_io_signature_sptr。

现在让我们回头看一下gr-howto-write-a-block-3.2.2//src/lib/howto_square_ff.h中的两段注释：

```
/*
 * We use boost::shared_ptr's instead of raw pointers for all access
 * to gr_blocks (and many other data structures).  The shared_ptr gets
 * us transparent reference counting, which greatly simplifies storage
 * management issues.  This is especially helpful in our hybrid
 * C++ / Python system.
 *
 * See http://www.boost.org/libs/smart_ptr/smart_ptr.htm
 *
 * As a convention, the _sptr suffix indicates a boost::shared_ptr
 */
typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;
/*!
 * \brief Return a shared_ptr to a new instance of howto_square_ff.
 *
 * To avoid accidental use of raw pointers, howto_square_ff's
 * constructor is private.  howto_make_square_ff is the public
 * interface for creating new instances.
 */
howto_square_ff_sptr howto_make_square_ff ();
```

因为 shared_ptr 的这种特点，GNU Radio 用这种指针作为各个 block 的输入输出端口，这些端口所指向的数据会被多个 block 共享，方便 GNU Radio 进行存储管理。这对于 C++ / Python 混合编程非常有帮助。

为了保证 GNU Radio 的 block 都使用这种智能指针。Block 的模板用了一个友元函数的方法。howto_square_ff 的构造函数是私有的。在类外是不能调用私有构造函数的。所以下面构建的 howto_square_ff 实例是不合法的：

```
howto_square_ff* test_block = new howto_square_ff()
```

因为howto_square_ff的构造函数不是公共的。这样做避免了raw C++指针指向块对象。相反，我们用用howto_make_square_ff()函数作为公共接口来创建新的实例。首先声明它为类howto_square_ff的友元函数（friend function），这样它就可以接入所有在howto_square_ff中定义的私有变量和方法。

```
friend howto_square_ff_sptr howto_make_square_ff ();
```

接下来在函数howto_make_square_ff()中，调用howto_square_ff的私有构造函数，通过新命令创建一个实例。但是将返回指针的数据类型从raw C++指针映射到智能指针howto_square_ff_sptr：

```
howto_square_ff_sptr howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}
```

通过以上操作，就确保当 block 创建时就使用了 shared_ptr。

4.7.2.4block 的核心：方法 general work()

gr_block衍生自gr_basic_block，gr_block有一个方法general_work()，它是一个虚函数，我们需要重写它。方法general_work进行实际的信号处理，是block的CPU。

```
virtual int general_work ( int noutput_items,
                           gr_vector_int &ninput_items,
                           gr_vector_const_void_star &input_items,
                           gr_vector_void_star &output_items) = 0;
```

通常，方法general_work()的主要工作是把输入流，经过一系列运算，变成输出流。首先介绍一下它的四个参数。

noutput_items 表示输出 item 的个数，用于记录每一个输出流输出 item 的数量。注意这里的 item 是输入输出的粒度，用户自定义的，可能是一个复数，也可能是一个复数的向量，等等。ninput_items 用于记录输入流的 item 个数。

一个 block 可能有 x 个输入流和 y 个输出流。ninput_items 是一个长度为 x 的整形矢量，元素 *i* 表示第 *i* 个输入端的输入流个数。然而对输出流，为什么 noutput_items 只是一个整形，而不是一个矢量呢？这是因为一些技术上的问题，目前的 GNU Radio 版本只能提供具有相同数据速率的输出流，即所有输出流的 item 个数都必须相同。而输入流的速率可以不同。

input_items 是指向输入流的指针向量，每个输入流一个。output_items 是指向输出流的指针矢量，每个输出流一个。我们用这些指针获取输入数据并将计算后的数据写到输出流中。请注意 ninput_items, input_items 和 output_items 这三个参量都是用地址表示（‘&’），所以可以在 general_work()中修改它们。general_work()的返回值为写入输出流的实际 item 个数，或是 EOF 为-1。返回值也可以小于 noutput_items。

最后值得注意的是，当重写 general_work()时，我们必须调用方法 consume()或 consume_each()来指明每个输入流消耗掉的 item 个数。[5]

```
void consume (int which_input, int how_many_items)
```

方法consume()告诉调度器第*i*个输入流（表示为`which_input'）消耗掉的item个数（表示为`how_many_items'）为多少。如果每个输入流的流个数相同，则用consume_each。

```
void consume_each (int how_many_items);
```

它告诉调度器每个输入流消耗的item个数。调用方法consume()或consume_each()的原因是我们必须告诉调度器输入流消耗的item个数，以便调度器安排上级缓冲器和相应的指针。这两个方法的细节比较复杂。只要记住，在GNU Radio中可以很方便的使用它们，并且在每次重写方法general_work()时记得调用它们。

[5] 如果你的 block 是派生自 gr_sync_block，就可以免去这些操作。

4.7.2.5其他方法和成员变量

下面介绍一下 gr_block 类的一些成员变量。

➢ 方法 forecast()

virtual void forecast (int noutput_items,

gr_vector_int &ninput_items_required);

方法forecast()用于，对于给定的输出，估计输入要多少数据。第一个参量noutput_items在general_work()中介绍过了，是输出流的流个数。第二的参量ninput_items_required是整型矢量，保存输入流的item个数。

当重载方法 forecast()时，需要估计每个输入流的数据，对于给定`noutput_items'item 的输入流。这个估计无需很准确，但是要接近。参量input_items_required 会由调度器传递出去，所以计算出的估计值直接保存到这个变量就可以了。

➢ d_output_multiple 与方法 set output multiple()

现在介绍gr_block中定义的一个成员变量d_output_multiple。它对传送到forecast()和general_work()的参数noutput_items做条件约束，即条件是调度器要确保参量noutput_items是d_output_multiple的整数倍。d_output_multiple的默认值是1。

那么，当我们要设计一个 block 的时候，在重写方法 general_work()和 forecast()时，就会想：谁来调用这些方法以及怎样调用？传送什么值到 noutput_items 参量以及谁完成这个工作呢？答案是：调度器。当程序执行到 general_work()或forecast()时，我们可以认为 noutput_items 和其他参量都已经被调度器设置好了。实际上，对于我们来说，永远不会直接调用这些方法以及设置这些参量。调度器会自己安排好这些事情，根据高层策略和缓冲器分配策略调用这些函数。有很多幕后的细节，我们无需具体了解。当你设计 block 时也无需担心它们。

总之，变量 d_output_multiple 告诉调度器 noutput_items 必须是d_output_multiple 的整数倍。可用方法 set_output_multiple() 设置d_output_multiple 的值。

```
void
gr_block::set_output_multiple (int multiple)
{
  if (multiple < 1)
    throw std::invalid_argument ("gr_block::set_output_multiple");

  d_output_multiple = multiple;
}
```

➢ d_relative_rate 和方法 set_relative_rate()

第五个存储器变量 d_relative_rate 提供了与数据速率相关的信息，它是输出速率比上输入速率的值。

这个变量给缓冲区(buffer)的分配器和调度器提供了信息，分配器和调度器根据这个信息来分配缓冲区并相应的调整参数。对于大多数信号处理 block 来说

d_relative_rate 默认值为 1.0。显然，抽取器(decimator)的 d_relative_rates 应该小于 1.0 而插值器(interpolator)的 d_relative_rates 要大于 1.0。

可以用方法 set_relative_rate()设置 d_relative_rate 的值，用方法 relative_rate()获取它的值。

好了！现在我们已经深入介绍过了类gr_basic_block和gr_block。在这里我们没有介绍d_detail和与它相关的方法。因为它们比较复杂，主要用于内部操作。在设计block时很少遇到它们。请阅读源代码来深入理解这些概念。

4.7.3 命名规则

在学习了类gr_block并阅读了一些源文件后，让我们总结一下GNU Radio的命名惯例。以下的命名惯例有助于我们理解代码基础以及连接C++与Python。在GUN Radio中，除了宏（macros）和常量（constant values），所有其它的标识符（identifiers）都是使用小写，比如：words_separated_like_this 。宏 - macros和常量 - constant values是采用大写，比如：UPPER_CASE。

4.7.3.1Package 前缀

所有全局可见的名称（类型、函数、变量、常量等等）都带有 Package 前缀，后面跟着一条下划线。GNU Radio 中的大部分代码都属于`gr'package，所以所见都是诸如 gr_open_file (...)的形式。

还有其他 package 前缀，以下是一些使用频率较高的名称：

gr_: 几乎出现在 GNU Radio 中所有的命名中

usrp_: 与 USRP 相关的 package

qa_: 质量验证，用于测试代码

4.7.3.2类数据成员变量

如我们所见，所有类数据成员变量应带有前缀`d_'。这样做最大的收益是，当面对一段 block 的代码时，一眼便知哪些应该在 block 之外赋值。这样也免去你在写构造函数时绞尽脑汁去想新的变量名，只需要使用跟成员变量一样的名字，只不过把 d_前缀去掉就可以了。

```
class gr_wonderfulness {
    std::string    d_name;
    double         d_wonderfulness_factor;
public:
    gr_wonderfulness (std::string name, double wonderfulness_factor)
    : d_name (name), d_wonderfulness_factor (wonderfulness_factor)
    ...
};
```

另外，所有类静态数据成员都应以`s_’开头。

4.7.3.3文件命名

所有关键的类都应包含在它自己的文件中。例如类应当在 gr_foo.h 声明并在 gr_foo.cc 中定义。

按照常规，信号处理模块名称的后缀由输入和输出类型确定。其后缀通常占一到两个字符。"信源"和"信宿"的后缀各占一个字符。常规的模块含有输入和输出的，便占两个字符长度的后缀。第一字符表示输入数据 流的类型，第二个字符表示输出的数据流的类型。FIR 滤波器带有三个字符的后缀，它们分别表示输入、输出和抽头（taps）的数据类型。

下面是一些后缀和其含义：

- ✓ f - 单精度浮点（数据类型） - single precision floating point
- ✓ c – 复合<浮点>（数据类型） – complex<float>
- ✓ s – （16 位整数）短型整型（数据类型） – short (16-bit integer)
- ✓ i – （32 位整数）整型（数据类型） – integer (32-bit integer)

另外，对于那些处理向量类型的数据流的模块而言，采用字符 'v' 作为后缀的第一个字符，例如 gr_fft_vcc。FFT 模块在输入端处理由复合数据类型构成的向量类型的数据流，在输出端生成由复合数据类型构成的向量类型的数据流。

4.7.4 如何把 C++与 Python 连接在一起

前面已经讲过，编写一个信号处理 block 要创建 3 个文件：.h 和.cc 用于定义新的 block，.i 则告诉 SWIG 如何将类与 Python 粘合。

SWIG（Simplified Wrapper and Interface Generator），简化封装和接口生成器，就像胶水一样把 C++编写的模块和 python 粘在一起，使 python 可以直接调用 C++模块。.i 文件告诉 SWIG 粘合的方法。

.i 文件可以看做是.h 文件的缩减版本，非常神奇的把 python 和 boost shared_ptr 结合在一起。为了不使代码显得臃肿，我们在.i 文件中只声明那些 python 需要使用的函数。

比如 howto.i，它保存了所有带有`howto_'前缀的 block 的 SWIG 声明，带有前缀`howto_'的类能够从 Python 读取。`howto'在 Python 中相当于一个 package 名。

```
/*
 * First arg is the package prefix.
 * Second arg is the name of the class minus the prefix.
 *
 * This does some behind-the-scenes magic so we can
 * access howto_square_ff from python as howto.square_ff
 */
GR_SWIG_BLOCK_MAGIC(howto,square_ff);

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff : public gr_block
{
private:
```

```
  howto_square_ff ();
};
```

这里的 GR_SWIG_BLOCK_MAGIC 做了一些工作使得我们能够从 Python 中接入 howto_square_ff。从 Python 的角度，howto 是一个 package，而 square_ff()为 howto 中定义的函数。调用此函数将返回一个智能指针 howto_square_ff_sptr，指向新的 howto_square_ff 实例。关于 GR_SWIG_BLOCK_MAGIC 是如何实现的，我们在此就不深入了解了。

4.7.4.1 目录结构

下表“目录结构”展示了模板目录的结构和常用文件。在为 topdir 目录改名之后，就可以在我们的 project 中使用了。

为了减少工作量，最好拷贝整个文件夹作为你的工作空间并根据自己的需要修改文件。而且如果你觉得 makefile 的操作很麻烦，可以把这些文件当成模板，只需在相应的地方替换新的内容即可，就像我们在4.7.1 节中介绍的。

表 4-1 Block 模板的目录结构

文件/目录名称	内容	需要修改吗
Topdir/Makefile.am	顶层的 Makefile.am	否
Topdir/Makefile.common	共用的一些 Makefile	否
Topdir/bootstrap	用于第一次编译时，配置 autoconf，automake，libtool	否
Topdir/config	Configure.ac 将使用这个目录下的内容	否
Topdir/configure.ac	Autoconf 的输入	否
Topdir/src		
Topdir/src/Makefile.am		否
Topdir/src/lib	C++代码，包括.h, .cc, .i 文件等	是
Topdir/src/lib/Makefile.am	C++代码的 Makefile.am	是
Topdir/src/lib/Makefile.swig.gen	配合.i 文件产生.cc, .py 等文件	是（当需要修改 package 名称时）
Topdir/src/python	测试代码	可不用
Topdir/src/python/Makefile.am	测试代码的 Makefile.am	可不用
Topdir/src/python/run_tests	用于执行测试代码	否

4.7.4.2Autotools

为了减少与 makefile 有关的代码工作，也为了提高跨系统的可移植性，使用了 GNU 中的 autoconf, automake, 和 libtool 工具。它们三者和在一起称为 autotools。

- Automake

Automake 和 configure 将共同生成 Makefiles，Makefile.am 是 Makefile 的一种更高级的描述方式。Makefile.am 描述了要构建的库和程序，以及源文件。Automake 读入 Makefile.am 生成 Makefile.in 文件。Configure 读入 Makefile.in 文件生成 Makefile 文件。生成的 Makefile 包含着成千上万的规则(rule)，这些规则用来做如下的事情：构建、检查和安装代码。通常，生成的 Makefile 是 Makefile.am 代码量的 5 到 6 倍。

- Autoconf

autoconf 读取 configure.ac 产生脚本文件 configure。configure 自动测试系统的特征从而设置为数众多的用于 Makefile 和 C++代码中来控制“build”的变量和定义。如果有条件没有满足，比如发现缺少某个依赖的库文件，configure 将输出错误（error）信息并退出。

- Libtool

用于创建共享库。

4.7.4.3编译目录和安装目录

编译目录（build tree）是自 *topdir* (其中包含 configure.ac)往下的一切内容。安装目录（Install tree）是指 *prefix*/lib/python2.x/site-packages，其中 *prefix* 是使用 configure 命令时的参数--prefix （一般默认是 /usr/local），2.x 是指安装的 Python 的版本。一个典型的安装目录是/usr/local/lib/python2.6/site-packages。有时候不是 site-packages，而是 dist-packages。

一般的，在编译目录中，通过在文档~/.bash_profile 或~/.profile 中设置 PYTHONPATH 环境变量。这样使得用户的 Python 的应用程序能够读取所有的标准 Python 库函数，以及新安装的诸如：GNU Radio。

当执行

$ make install

时，就把编译好的文件拷贝到了安装目录。

4.7.4.4代码测试（Make check）

写一段应用程序使其能在安装目录下读取代码和库。另一方面，我们希望自己的代码能在编译目录里运行，在编译目录中运行使我们能在安装前发现问题。这就是`make check'的意义。

为了达到目的，需要写一段带有前缀`qa_'的 Python 测试代码，并把它放到/src/python 目录下。`qa'代表`Quality Assurance'。比如在 qa_howto.py 脚本中，用`howto.square_ff()'测试刚刚创建的 block，并看看它能否正确运行。

接下来就可以用 make check 运行我们的测试程序了。make check 启动/src/python/run_tests 脚本，此脚本设置了 *PYTHONPATH* 环境变量以便我们能够使用我们代码和库所在的编译目录。它将运行所有名称形式为 qa_*.py 的文件，并记录下所有的成功和失败的结果。

OK！到此为止，编写 block 的方法和它背后的基本原理已经介绍完了。

4.8 如何使用外部库文件

有时候我们有一些已经编写好的算法，很复杂，不愿意在 GNU Radio 里再写一遍 block；又或者你想利用一些现成的写得非常好的库，比如 IT++。那么你就可以用外部库文件，把外部程序封装成一个 block。

例如要使用的库文件叫 libspectrum_sensing.so，头文件是 spectrum_sensing.h。我们要把它用到一个叫 example_detection_cf.cc 的 block 中。 那么可以按照下面的步骤做：

- 修改 block 的头文件

在需要调用.so 的 block 的头文件（example_detection_cf.h）中加入 spectrum_sensing.h。

```
#include “spectrum_sensing.h”
```

若.so 是用 C 写的且用 gcc 编译的，则在 include 的时候要特别注意，

```
extern “C”
{
#include “spectrum_sensing.h”
}
```

表示要使用的这个动态库是 C 的库文件。这样一会你编译整个 package 的时候，gcc 才不会出错。

- 修改 block 的 C 文件

在 example_detection_cf.cc 中直接调用 spectrum_sensing.h 中定义的函数。把需要的头文件(spectrum_sensing.h 以及它所 include 的其他头文件)放在适当的地方，本地目录 topdir/src/lib 中，或者/root/gr/include 中，还可以在 makefile.am 文件中的 INCLUDES 变量后面添加你存放头文件的目录。总之就是让编译程序能够找到它需要的头文件。

- 修改 Makefile.swig.gen 文件

把 libspectrum_sensing.so 拷贝到/root/gr/lib 下，并在 topdir/src/lib 下的 Makefile.swig.gen 中_example_la_LIBADD 变量后面添加 －lspectrum_sensing，

表示 example 要链接到 libspectrum_sensing.so 去。在 make 完成之后，可以用命令：

ldd _example.so

查看到_example.so 与 libspectrum_sensing.so 有链接。

- 编译安装

./bootstrap

./configure

make install

第5章　　应用范例解读

本章将详细讲解 GNU Radio 自带的两个例子，一是 OFDM，另一个是 MIMO。因为 MIMO/OFDM 在当前最新一代的通信系统中，几乎是必选的两个关键技术，因此很多做物理层研发的 GNU Radio 用户都会从这两个例子开始着手。希望这章的讲解能够让初学者可以更快的进入状态。

5.1 OFDM Tunnel

Tunnel 是 GNU Radio 中很经典的例子。tunnel 有两个，一个是基于 GMSK 调制的(gnuradio-examples\python\digital)，另一个基于 OFDM 调制的(gnuradio-examples\python\ofdm)。它们都由物理层和 MAC 层构成，提供一个虚拟的 Ethernet 接口，使得基于 IP 的各种应用程序都可以加载在这个 tunnel 上面，它就像一个管道，负责传输你的数据。

首先让我们来运行一下 GMSK tunnel [6]（这个例子运行成功的概率很高☺，OFDM tunnel 有时会由于某些原因不灵）。

1. 用两个 USRP 分别加上子板，例如我们用了两个 RFX900。然后连到两台 PC 上。
2. 在 PC_A 上，打开两个终端窗口：在第一个终端中，输入下列命令。`$./tunnel.py --freq 930M --bitrate 500k –v` 它告诉程序在 930MHz 上建立一个速率为 500kbps 的连接，并把每个数据包的一些信息输出到屏幕。在第二个终端中，输入`$ ifconfig gr0 192.168.200.1` 它的意思是，给接口 gr0 配置一个 ip 地址 192.168.200.1
3. 在 PC_B 上，打开两个终端窗口：在第一个终端中，输入`$./tunnel.py --freq 930M --bitrate 500k -v` 这样，PC_B 就会与 PC_A 一样在 930MHz 上有一个 500kbps 的物理层链路，两者才能够相互通信。在第二个终端中，输入`$ ifconfig gr0 192.168.200.2` 它给 PC_B 的 gr0 配置了另一个 ip 地址 192.168.200.2，以区别于 PC_A。

OK，程序运行起来之后，在 PC_A 的第一个终端上，我们就会看到以下这些输出：

```
# ./tunnel.py --freq 423.0M --bitrate 500k -v
>>> gr_fir_fff: using SSE
bits per symbol = 1
Gaussian filter bt = 0.35
Tx amplitude 0.25
modulation: gmsk_mod
bitrate: 500kb/s
samples/symbol: 2
USRP Sink: A: Basic Tx
```

[6] 运行 GMSK tunnel 这部分内容翻译自 Alalelddin Mohammed 的硕士论文“Studying Media Access and Control Protocols”

Requested TX Bitrate: 500k Actual Bitrate: 500k
bits per symbol = 1
M&M clock recovery omega = 2.000000
M&M clock recovery gain mu = 0.175000
M&M clock recovery mu = 0.500000
M&M clock recovery omega rel. limit = 0.005000
frequency error = 0.000000
Receive Path:
modulation: gmsk_demod
bitrate: 500kb/s
samples/symbol: 2
USRP Source: A: Basic Rx
Requested RX Bitrate: 500k
Actual Bitrate: 500k
modulation: gmsk
freq: 930M
bitrate: 500kb/sec
samples/symbol: 2
Carrier sense threshold: 30 dB

Allocated virtual ethernet interface: gr0
You must now use ifconfig to set its IP address. E.g.,

$ sudo ifconfig gr0 192.168.200.1

Be sure to use a different address in the same subnet for each machine.

Tx: len(payload) = 90
Tx: len(payload) = 54
Tx: len(payload) = 153
Tx: len(payload) = 82
Tx: len(payload) = 235
Rx: ok = False len(payload) = 235
Tx: len(payload) = 78
Tx: len(payload) = 235

最后这部分信息说明了PC_A发送了几个数据包，长度分别都说明了；收到了一个长度为235的数据包，但是校验结果是错的。

5.1.1 系统框图和 MAC 帧的构成

下图是 Tunnel 的系统框图。Tunnel 的物理层由发射机，接收机和一个载波侦听（sensing probe）三部分构成，完成由信息比特到基带波形之间的转换，以及通过能量检测判断当前信道是否空闲。MAC 层是一个基于 CSMA 的简单的MAC。MAC 层与 PHY 层之间传递的是一个在 IP 包的基础上加了一些包头和包尾的数据包。

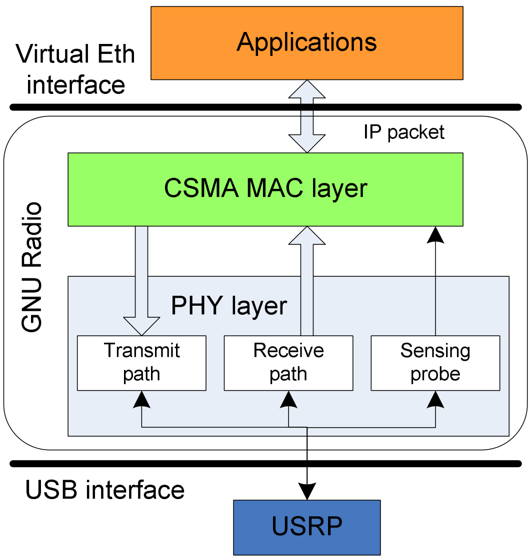

图 5-1 Tunnel 的系统框图

图 5-2 说明了一个 IP 数据包是如何打包成 MAC 数据包的。

首先，IP 包被加上了 4 字节的校验比特，算法是 CRC32。然后数据部分，加上 CRC 比特和尾比特（x55），都被白化处理，使得数据具有随机均匀分布的特性。最后，加上一个 4 字节的包头。包头包含两个信息：白化参数 4 比特和数据包长度 12 比特。包头采用了重复发送的方法，以增加可靠性。到此，一个完整的 MAC 数据包就包装完成了。

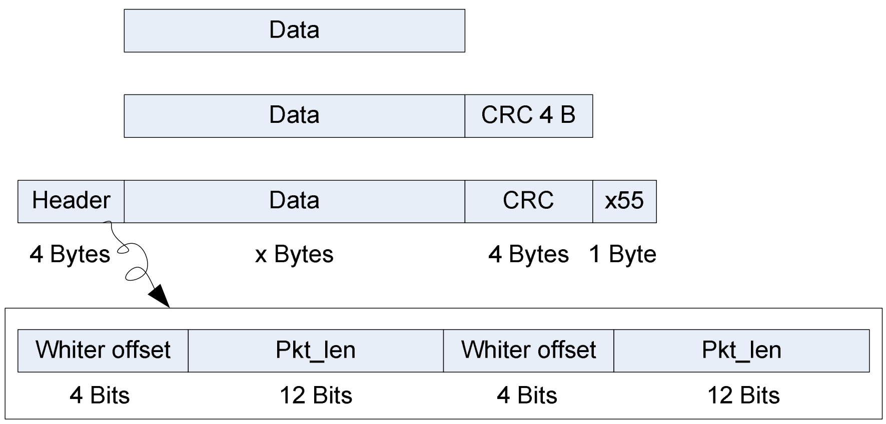

图 5-2 IP 数据包到 MAC 数据包的打包过程

5.1.2 物理层

下面以 ofdm tunnel 为例来解读一下物理层。Ofdm tunnel 的代码除了在 gnuradio-examples\python\ofdm 目录下，还有一些在 gnuradio-core\src\python\gnuradio\blks2impl 目录下。

先来看发射机图 5-3。在 transmit_path.py 中，语句

```
self.connect(self.ofdm_tx, self.amp, self)
```

说明其中包含两个模块，ofdm_tx 是一个 ofdm_mod 类，amp 是一个乘法器。

进入 ofdm_mod 类看一下，其代码在文件 ofdm.py 中。

Ofdm_mod 中，数据包首先经过一个 send_pkt 函数，完成 MAC 包的打包过程。

```
send_pkt(self, payload='', eof=False)
```

然后 MAC 包被放进一个队列

```
self._pkt_input.msgq().insert_tail(msg)
```

后面的 ofdm_mapper_bcv 模块从队列中取出数据包，根据 OFDM 调制的参数映射成一个个 OFDM symbol，再送到后续模块，添加 preamble，IFFT 变换，添加 cyclic prefix，最后调整一下幅度，发送出去。

这里想特别提一下的是，在 ofdm_mapper 之后是流图的形式，在这之前是通过一个 message queue 与 MAC 层联系在一起。这种连接方式使得“异步的”MAC 层数据（而且数据包长不定），跟与系统时钟“同步的”物理层连接在一起。这种连接方式是一个很好的例子，值得参考。

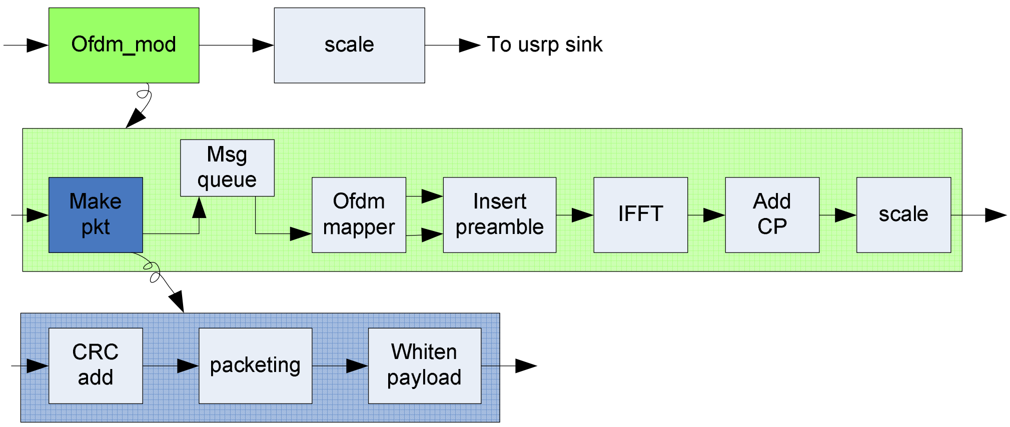

图 5-3 ofdm tunnel 发射机框图

接收机部分见图 5-4。receive_path.py 包含了 ofdm_demod 和 probe 两个模块。Ofdm_demod 显然就是 ofdm 接收机部分。而 probe 是一个信号检测模块，当 usrp 收到的信号幅度大于门限时，就认为无线信道已经被其他用户占用。

Ofdm_demod 类的代码在文件 ofdm.py 中，主要分成同步模块(ofdm_receiver)，解调模块(`ofdm_frame_sink`)，和 MAC 帧拆包部分。与发射部分类似，物理层与 MAC 层也是通过一个队列 `self._rcvd_pktq` 连接在一起的。Ofdm_receiver 部分比较复杂，是用 python 写的，完成了帧同步，频偏估计，频偏纠正，FFT 的功能。ofdm_frame_sink 是一个 C 写成的模块，完成了从调制符号到比特的解映射过程（不确定有没有信道均衡）。

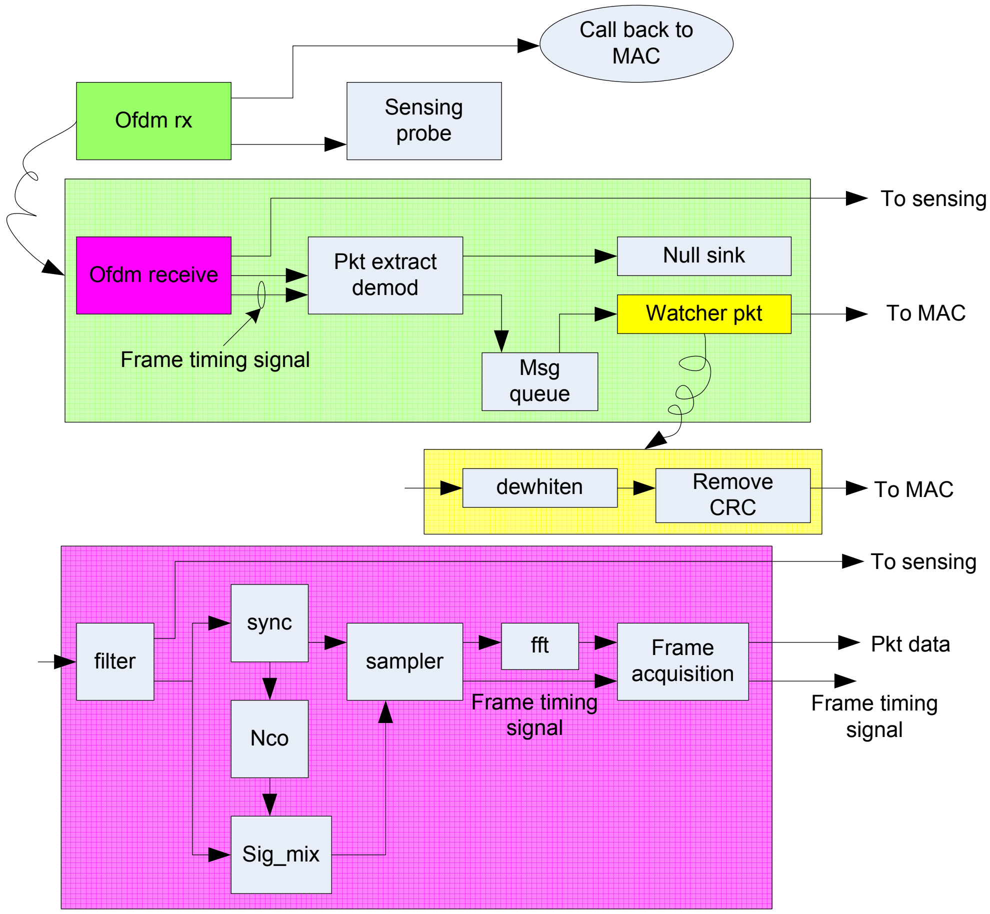

图 5-5 ofdm tunnel 接收机框图

5.1.3 开发和调试方法

整个 ofdm tunnel 的物理层还是比较简单的。它模仿了 802.11 的物理层，在不定长的 burst 前面添加一个定长的 preamble，依靠这个 preamble 完成时间同步和频率同步。但它没有信道编码，因此抗噪声性能较差。

gnuradio-examples\python\ofdm 目录下，除了 tunnel 调用的函数外，还有许多其他的函数。这些函数都是程序的开发过程中需要用到的，它们教会了我们如何一步步的进行程序开发。特别是对于利用 GNU Radio 做物理层研发的人来说，是很好的参考。下面简单说明一下。

ofdm_mod_demod_test.py——用于物理层收发模块的仿真测试。

benchmark_ofdm.py——加上 MAC 层以后，做收发的仿真测试。

benchmark_ofdm_tx.py，benchmark_ofdm_rx.py——加上 USRP 之后，做单向收发的测试。分别测试了连续的数据包传输，和不连续的突发数据包传输。

当单向传输没有问题之后，就可以实验双向的传输了：tunnel.py。

另外，还有一些 matlab 程序，帮助调试程序。当我们把 log 标志设为 True 时，就会产生很多.dat 文件。

```
if logging:
            self.connect(self.chan_filt,         gr.file_sink(gr.sizeof_gr_complex,
```

```
"ofdm_receiver-chan_filt_c.dat"))
            self.connect(self.fft_demod,
gr.file_sink(gr.sizeof_gr_complex*fft_length, "ofdm_receiver-fft_out_c.dat"))
            self.connect(self.ofdm_frame_acq,
                         gr.file_sink(gr.sizeof_gr_complex*occupied_tones,
"ofdm_receiver-frame_acq_c.dat"))
            self.connect((self.ofdm_frame_acq,1),  gr.file_sink(1,  "ofdm_receiver-found_corr_b.dat"))
            self.connect(self.sampler, gr.file_sink(gr.sizeof_gr_complex*fft_length,
"ofdm_receiver-sampler_c.dat"))
            self.connect(self.sigmix,            gr.file_sink(gr.sizeof_gr_complex,
"ofdm_receiver-sigmix_c.dat"))
            self.connect(self.nco,               gr.file_sink(gr.sizeof_gr_complex,
"ofdm_receiver-nco_c.dat"))
```

这些文件把各个 block 的输出都记录下来：同步之前，频率同步之后，FFT 之后，解映射之后等等。然后用 Matlab 程序一一检查，就可以发现究竟哪一步出了问题。

总结这个例子的开发方法，我们建议，如果你要创建一个自定义的无线连接程序，

- 第一步：用 Matlab 写一个物理层收发程序，设计各个功能模块，确定参数等等。
- 第二步：用 GNU Radio 写一个不包括 USRP 的收发程序，与 Matlab 程序一致，方便把 GNU Radio 中的数据导入 Matlab 中调试。
- 第三步：当物理层没有问题之后，再添加 MAC 层。
- 第四步：加入 USRP。先调试单向通信，再调试双向的。

5.2 MIMO

在 gnuradio 的 example 目录中，有两个是与 MIMO 有关的：multi-antenna 和 multi_usrp。顾名思义，Multi-antenna 是同一个 USRP 母板上的两个子板，也就是两天线的 MIMO；multi-usrp 是多个 USRP 母板，这时最多可以有 4 个天线。

在 mutli-antenna 的情况下，两个子板的时钟都来自于母板上的时钟，因此两个子板的信号能够同步。

在 multi-usrp 的情况下，需要把一个母板设为 master，另一个设为 slave，然后从 master 上引出一个时钟信号，接到 slave 上了，从而使两个 USRP 上的信号能够同步。具体的配置方法请看2.5.2 节。

5.2.1 MUX 参数的含义

在讲解 multi-antenna 和 multi-usrp 之前，我们先来了解一下 mux 这个关键的参数，这有助于我们了解在 mimo 的配置下，各个子板上的数据是如何复用和传输的。

首先我们从接收方向看起。在 usrp_standard.h 中，有这样一段说明：

```
/*!
```

```
 * \brief Set input mux configuration.
 *
 * This determines which ADC (or constant zero) is connected to
 * each DDC input.  There are 4 DDCs.  Each has two inputs.
 *
 * <pre>
 * Mux value:
 *
 *    3                   2                   1
 *  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
 * +-------+-------+-------+-------+-------+-------+-------+-------+
 * |   Q3  |   I3  |   Q2  |   I2  |   Q1  |   I1  |   Q0  |   I0  |
 * +-------+-------+-------+-------+-------+-------+-------+-------+
 *
 * Each 4-bit I field is either 0,1,2,3
 * Each 4-bit Q field is either 0,1,2,3 or 0xf (input is const zero)
 * All Q's must be 0xf or none of them may be 0xf
 * </pre>
 */
bool set_mux  (int mux);
```

在 FPGA 上，有 4 个数字下变频器（DDC），每个 DDC 都有两个通道，I 通道和 Q 通道。在 AD 芯片上，有 4 个 ADC 通道。以典型的 2 天线 MIMO 复数采样为例，4 个 ADC 通道分别对应子板 A 的 I 路和 Q 路，以及子板 B 的 I 路和 Q 路。这 4 路信号分别送到 4 个 DDC 通道，下变频之后，放到 USB 上传输。例如，USB 上的发送序列可能是 I0 Q0 I1 Q1 I0 Q0 I1 Q1……。注意：所有输入信道必须是相同的数据速率（即同样的抽样率）。

DDC3		DDC2		DDC1		DDC0	
Q3(4 bits)	I3(4 bits)	Q2(4 bits)	I2(4 bits)	Q1(4 bits)	I1(4 bits)	Q0(4 bits)	Q0(4 bits)

图 5-6 mux 参数

上图是 mux 参数各个字段的含义。Mux 参数共 32 比特，每 4 比特一个值，这个值可以是[0,1,2,3]，表示 ADC0,1,2,3。其中 Q 通道可以是 0xf，即不使用 Q 通道。GNU Radio 规定，Q 通道要么都用，要么都不用。在典型的 2 天线 MIMO 复数采样的情况下，mux 值可以设为 0x0123。它表示 ADC0 连接到 DDC1 的 I 通道，ADC1 连接到 DDC1 的 Q 通道；ADC2 连接到 DDC0 的 I 通道，ADC3 连接到 DDC0 的 Q 通道。这样，你在 python 脚本中，从 usrp_source 收到的数据流就是，子板 A 的 sample（复数，包含 I 路和 Q 路），子板 B 的 sample，交替传输。

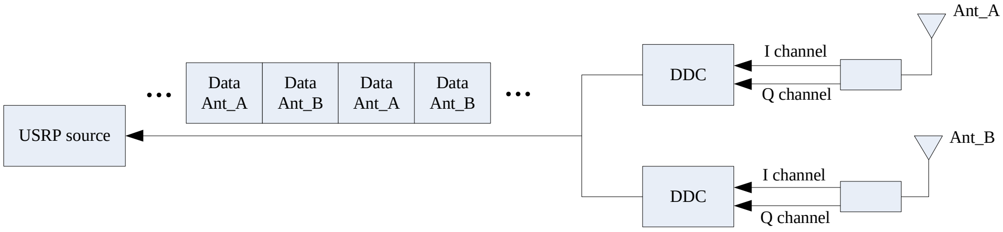

图 5-7 MIMO 数据复用示意图

当然，你也可以随意把 mux 值设成别的配置，比如 0x1320。这样你收到的数据流，就把子板 A 的 I 路和子板 B 的 I 路合成了一个复数，显然这样就没有物理意义了。实际上，你可以试试随意设一个 mux 值，你可能会发现它没有像你期望的那样产生奇怪的结果，这说明 GNU Radio 内部可能对这个参数做了某种约束。

在发射方向上，是类似的。4 个 DUC（数字上变频器）对应 4 个 DAC。在 usrp_standard.h 中有如下说明：

```
/*!
 * \brief Set output mux configuration.
 *
 * <pre>
 *     3                   2                   1
 *   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
 *  +-------------------------------+-------+-------+-------+-------+
 *  |                               | DAC3  | DAC2  | DAC1  |  DAC0 |
 *  +-------------------------------+-------+-------+-------+-------+
 *
 *  There are two interpolators with complex inputs and outputs.
 *  There are four DACs.
 *
 *  Each 4-bit DACx field specifies the source for the DAC and
 *  whether or not that DAC is enabled.  Each subfield is coded
 *  like this:
 *
 *     3 2 1 0
 *    +-+-----+
 *    |E|  N  |
 *    +-+-----+
 *
 *  Where E is set if the DAC is enabled, and N specifies which
 *  interpolator output is connected to this DAC.
 *
 *   N   which interp output
 *  ---  -------------------
 *   0   chan 0 I
 *   1   chan 0 Q
 *   2   chan 1 I
 *   3   chan 1 Q
 * </pre>
 */

bool set_mux  (int mux);
```

以 2 天线 MIMO 复数采样为例，mux 值可以设为 1011 1010 1001 1000，用十六进制表示就是 0Xba98。

不过，我们在配置 mux 参数时，可以用一个比较保险的办法——用函数 determin_rx_mux_value 或者 determin_tx_mux_value 来获得合适的 mux 值。这个函数会自动的根据你的子板类型，你选择的通道数(channel)，然后返回一个 mux 值。

5.2.2 代码示例：2 天线接收

我们来看一个两天线接收的例子。一个 USRP 母板上插了两个 RFX900 的子板，然后我们用 FFT 模块来分别观察两个天线收到的信号，其中一个子板的中心频率设为 939MHz，另一个设为 939.2MHz。

```
#!/usr/bin/env python

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import usrp
from gnuradio.eng_option import eng_option
import math
import sys
from numpy.numarray import fft
from optparse import OptionParser
from gnuradio.wxgui import stdgui2, fftsink2, scopesink2
import wx

class top_graph (stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__ (self, frame, panel, vbox, argv)

        # ------------------ parameters setting ------------------
        sample_rate = 1e6 # 1MHz
        usrp_decim = int(64e6 / sample_rate)
        antenna_num = 2
        freq_a = 939e6
        freq_b = 939.2e6
        db_gain = 45.0

        # ------------------ USRP init ------------------
        self.u = usrp.source_c (0, usrp_decim)
        self.u.set_nchannels(antenna_num)
        subdev_a = usrp.selected_subdev(self.u, (0,0)) # (0,0) is A side; (1,0) is
B side
        subdev_b = usrp.selected_subdev(self.u, (1,0))
        print "A side:", subdev_a.name()
        print "B side", subdev_b.name()

        subdev_a.select_rx_antenna('TX/RX')
        subdev_b.select_rx_antenna('TX/RX')
        subdev_a.set_gain(db_gain)
        subdev_b.set_gain(db_gain)
        subdev_a.set_auto_tr(False)
```

```
        subdev_b.set_auto_tr(False)
        r = self.u.tune(0, subdev_a, freq_a)
        if not(r):
              print "A side: Failed to set initial frequency"
        else:
              print "A side: The carrier frequency is set as ", freq_a
        r = self.u.tune(1, subdev_b, freq_b)
        if not(r):
              print "B side: Failed to set initial frequency"
        else:
              print "B side: The carrier frequency is set as ", freq_b
        mux_val = 0x0123
        self.u.set_mux(mux_val)
        print "mux_val = 0x%x" % gru.hexint(mux_val)

        # ------------------ flow graph ------------------
        # s2p
        self.s2p = gr.deinterleave(gr.sizeof_gr_complex)

        # sink
        self.null_sink = gr.null_sink(gr.sizeof_gr_complex)

        self.connect(self.u, self.s2p)

        # fft scope
       self.spectrum_a = fftsink2.fft_sink_c (panel, fft_size=1024,
sample_rate=sample_rate,title="Spectrum on Antenna A")
        self.connect ((self.s2p,0), self.spectrum_a)
        vbox.Add (self.spectrum_a.win, 1, wx.EXPAND)
        self.spectrum_b = fftsink2.fft_sink_c (panel, fft_size=1024,
sample_rate=sample_rate,title="Spectrum on Antenna B")
        self.connect ((self.s2p,1), self.spectrum_b)
        vbox.Add (self.spectrum_b.win, 1, wx.EXPAND)

def main ():
    app = stdgui2.stdapp(top_graph, "2 Antenna Rx")
    app.MainLoop ()

if __name__ == '__main__':
    main ()
```

这个程序的流图非常简单，如下图。

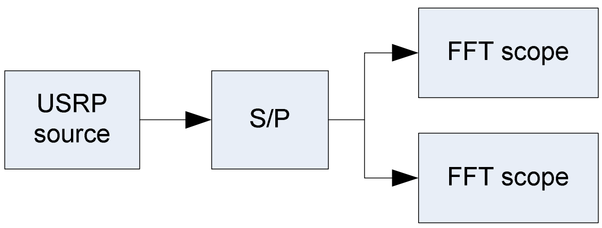

图 5-8 2 天线接收程序的流图

在这个程序中，mux 值设为 0x0123。运行的结果是这样的：

```
Launching a SCIM daemon with Socket FrontEnd...
Loading simple Config module ...
Creating backend ...
Loading socket FrontEnd module ...
Starting SCIM as daemon ...
GTK Panel of SCIM 1.4.7

A side: Flex 900 Rx MIMO B
B side Flex 900 Rx MIMO B
A side: The carrier frequency is set as  939000000.0
B side: The carrier frequency is set as  939200000.0
mux_val = 0x123
```

频谱显示的窗口见下图。

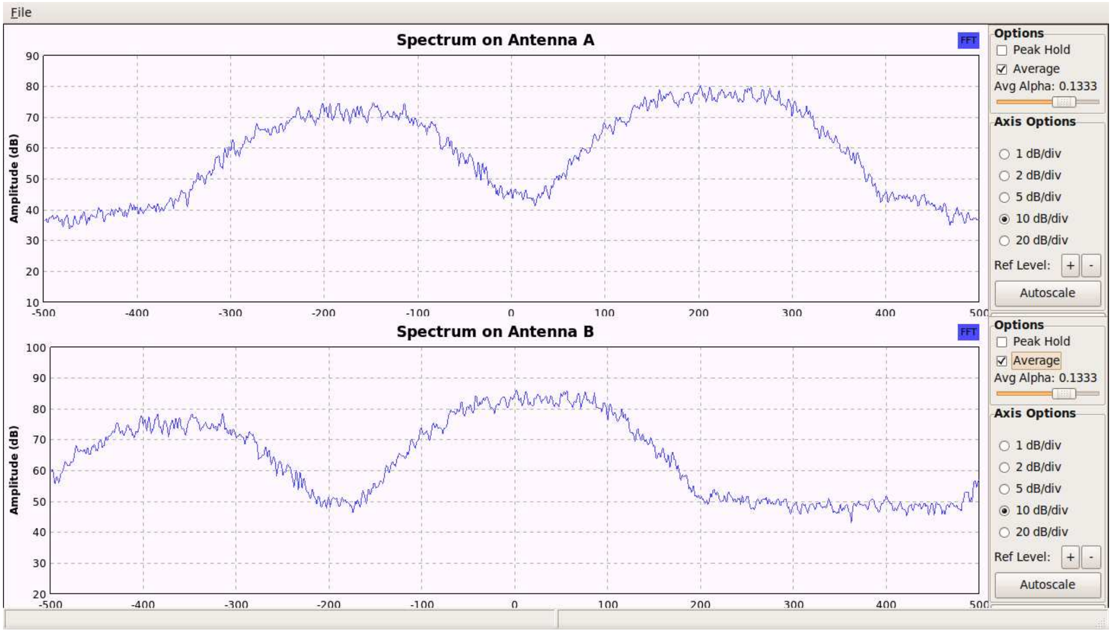

图 5-9 2 天线接收的信号频谱

从图中我们可以看到，两个天线收到的信号的频谱形状是一样的，只是子板 B 的中心频率高了 200Hz。

5.2.3 代码范例：2 天线发射

这一小节是一个 2 发 2 收的例子，但是两天线不是工作在同一频率，因为我们接收端只是用 FFT 观察频谱，如果是同频的话就不能明显的观察出来了，那需要 MIMO 解调。但是改成同一频率也是没有问题的，那样就变成标准的 2x2 MIMO 了。

其实，一个传输链路同时工作在两个频率，也是时下热门的所谓“载波聚合 (Carrier Aggregation)”技术呢，只不过这里的实现方法比较“低级”而已。

这个例子用到两个 USRP，每个 USRP 插了两块 RFX2400 子板。一个天线工作在 2.45GHz，另一个天线工作在 2.46GHz。

发射端的代码如下：

```python
#!/usr/bin/env python

from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio.wxgui import stdgui2, scopesink2
import wx

class top_graph (stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__ (self, frame, panel, vbox, argv)

        # ------------------ parameters setting ------------------
        sample_rate = 1e6 # 1MHz
        usrp_decim = int(64e6 / sample_rate)
        antenna_num = 2
        freq_a = 2450e6
        freq_b = 2460e6

        # ------------------ USRP init ------------------
        self.u = usrp.sink_c (0, usrp_decim)
        self.u.set_nchannels(antenna_num)
        subdev_a = usrp.selected_subdev(self.u, (0,0))
        subdev_b = usrp.selected_subdev(self.u, (1,0))
        print "A side:", subdev_a.name()
        print "B side", subdev_b.name()

        subdev_a.select_rx_antenna('TX/RX')
        subdev_b.select_rx_antenna('TX/RX')
        subdev_a.set_auto_tr(True)
        subdev_b.set_auto_tr(True)

        r = self.u.tune(0, subdev_a, freq_a)
        if not(r):
              print "A side: Failed to set initial frequency"
        else:
              print "A side: The carrier frequency is set as ", freq_a
        r = self.u.tune(1, subdev_b, freq_b)
        if not(r):
              print "B side: Failed to set initial frequency"
        else:
              print "B side: The carrier frequency is set as ", freq_b

        mux_val = 0xBA98
        self.u.set_mux(mux_val)
        print "mux_val = 0x%x" % gru.hexint(mux_val)

        # ------------------ flow graph ------------------
        # source
        src0 = gr.sig_source_c (sample_rate, gr.GR_SIN_WAVE, 20e3, 10000)
        src1 = gr.sig_source_c (sample_rate, gr.GR_SQR_WAVE, 20e3, 10000)

        # p2s
        p2s = gr.interleave(gr.sizeof_gr_complex)
```

```
        self.connect (src0, (p2s, 0))
        self.connect (src1, (p2s, 1))
        self.connect(p2s, self.u)

        # oscilloscope
       self.scope_a = scopesink2.scope_sink_c(panel, sample_rate=sample_rate)
        self.connect (src0, self.scope_a)
        vbox.Add (self.scope_a.win, 1, wx.EXPAND)
        self.scope_b = scopesink2.scope_sink_c(panel, sample_rate=sample_rate)
        self.connect (src1, self.scope_b)
        vbox.Add (self.scope_b.win, 1, wx.EXPAND)


def main ():
    app = stdgui2.stdapp(top_graph, ”2 Antenna Tx”)
    app.MainLoop ()

if __name__ == ’__main__’:
    main ()
```

程序的流图如下。

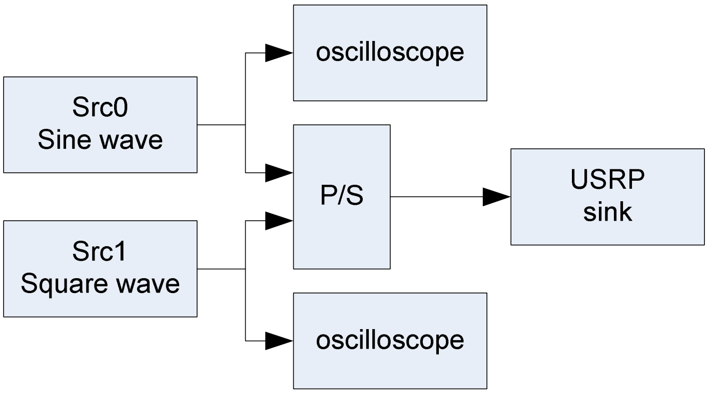

图 5-10 2 天线发射程序的流图

这里 mux 值设为 0xBA98。我还尝试了 0x89AB，但是发现结果不正确。因此，对 mux 值究竟 GNU Radio 做了哪些限制，还有待读者自己去读源代码了。

下图是流图中的两个示波器显示的正弦波和方波波形。蓝色和绿色两条线分别是信号的 I 路和 Q 路，也就是复数的实部和虚部。

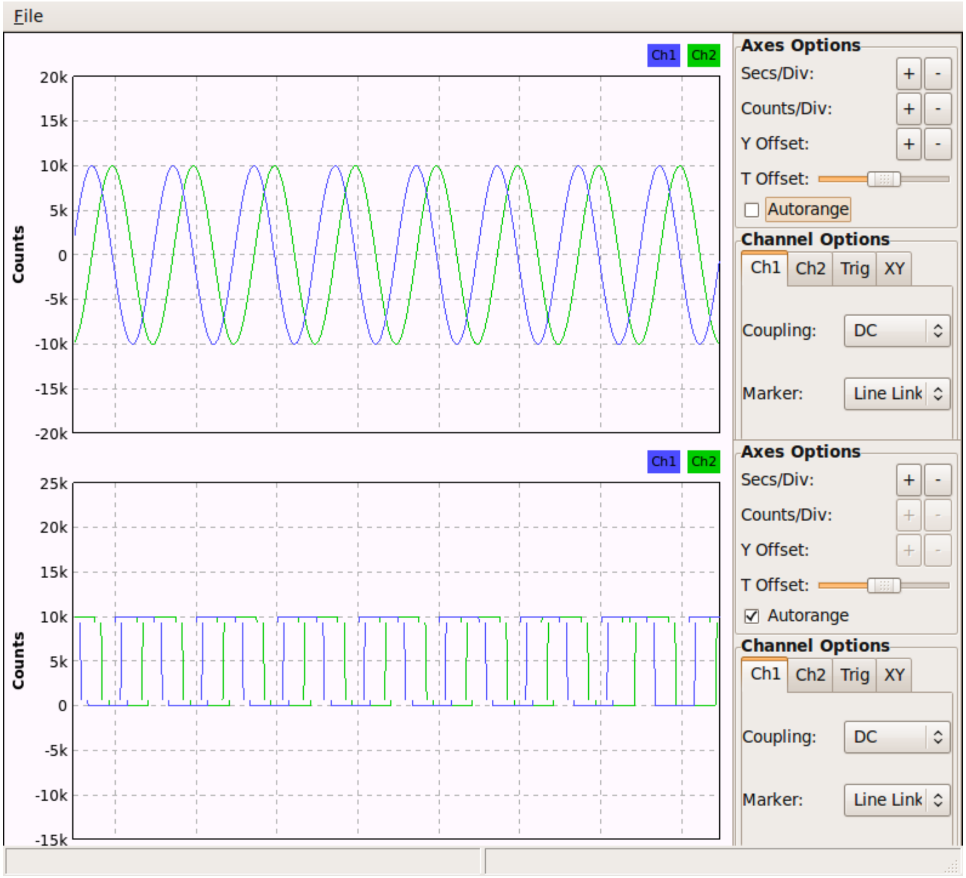

图 5-11 2 天线发射的波形图：正弦波和方波

在接收端，我们运行了一个 2 天线接收程序，使一个天线工作在 2.45GHz，另一个天线工作在 2.46GHz。与5.2.2 节的代码不同是，用“示波器”来显示收到的信号波形，而不是“频谱仪”。我们可以看到方波的波形发生了变化，这主要是因为经过有限带宽的无线信道传输之后，高频成分丢失了。

特别说明一下，在这个接收程序中，我发现 mux 值 0x0123 不灵了，只能用 0x2301。—_—很惭愧，mux 值这个问题我还没彻底弄明白。

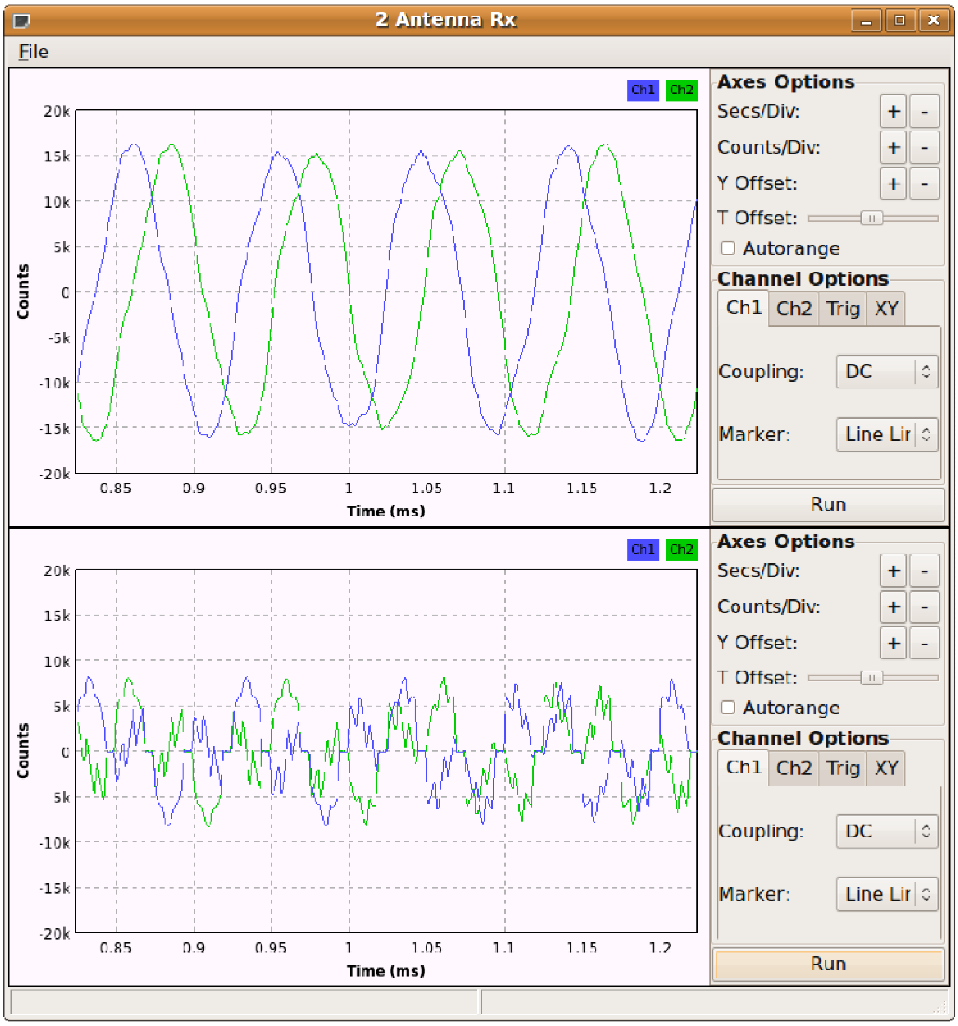

图 5-12 2 天线接收的波形图：正弦波和方波

第6章　GNU Radio 的其他应用

USRP 产品系列在世界各地有着各种各样的应用。除了被用于快速原型设计和科研应用，它已被部署到现实世界的许多商业和国防系统中。

6.1　商业应用

有许多应用 USRP 的商业系统。软件无线电作为通信系统开发和原型设计的理想平台，当一个应用没有足够的空间调整个性化的硬件设计，灵活的 USRP 实现了成本敏感的可部署的系统。

例如，Path Intelligence 有限公司使用 USRP 产品系列跟踪商场里行人的步行情况。Path Intelligence 通过接收手机控制信道的传输信息能够确定顾客的位置。

6.2　国防和国土安全

USRP 产品系列被美国军事和情报服务的所有分支机构广泛使用，许多大的防御合作伙伴和其他北约国家也都在使用 USRP。USRP 主板和子板能够以很低的预算，快速成型和部署先进无线系统。一些应用包括：

- ✓ 信号情报/通信情报
- ✓ 战场网络，生存网络
- ✓ 联合战术无线电系统（JTRS）的研究
- ✓ 公共安全通讯桥梁
- ✓ 应急低功耗灯塔
- ✓ 矿山安全和地下通信
- ✓ 合成孔径雷达
- ✓ 无源雷达

下图是 TD-SCDMA 频段的扫描结果，它说明 USRP 可以被无线电监管部门用于国家无线电安全监控。

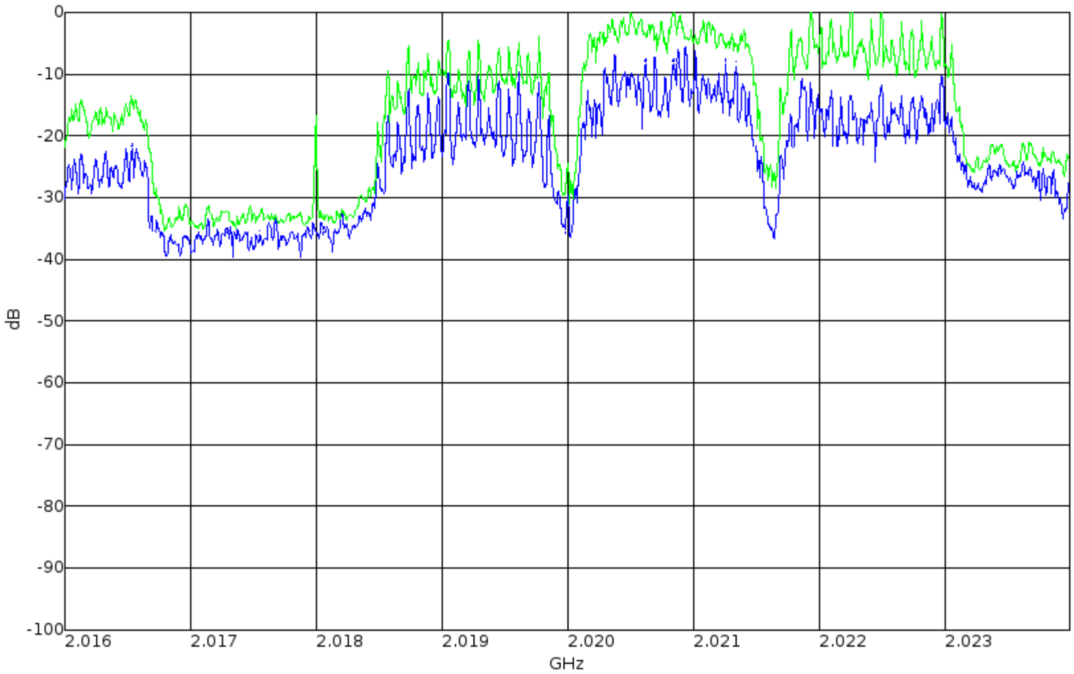

图 6-1 TD-SCDMA 波段频谱扫描

6.3 无线研究

许多无线网络领域的研究者使用 USRP 产品系列研究以下各种课题：

- ✓ 认知无线电
- ✓ MIMO 系统
- ✓ Ad-hoc 和 mesh 网络
- ✓ MAC 层协议
- ✓ 物理层设计
- ✓ 频谱占用，频谱遥感

USRP 产品系列的开放和易于使用，使创新的新型通信系统能够快速形成原型。低成本使得在测试平台可以部署大量节点，以了解大型网络的性能。

6.4 教学

美国和世界各地的许多大学都为学生配置了带 USRP 系统的实验室。USRP 产品系列的低成本、极大灵活性和开源性质，以及 GNU Radio 开源社区的支持使 USRP 成为讲授下列课程的理想选择：

- ✓ 软件无线电
- ✓ 信号与系统
- ✓ 数字信号处理
- ✓ 通信系统
- ✓ FPGA 设计

6.5 其他应用

这些年来，北京海曼无限公司和公司的 USRP 客户已经推出了基于 USRP 系统的大量创新应用，包括软件示波器和软件频谱仪、2G 路测仪、考试防作弊干扰仪、GSM-R 收发平台、802.11 系列协议仿真平台。一些更有趣的例子包括：射电天文学，野生动物跟踪， 射频识别 ，医疗成像，声纳和可定制测试设备等等。

下图显示了一个基于 USRP 的软件示波器的界面，可以看到 FSK 波形以及 FSK 解调之后的信号。

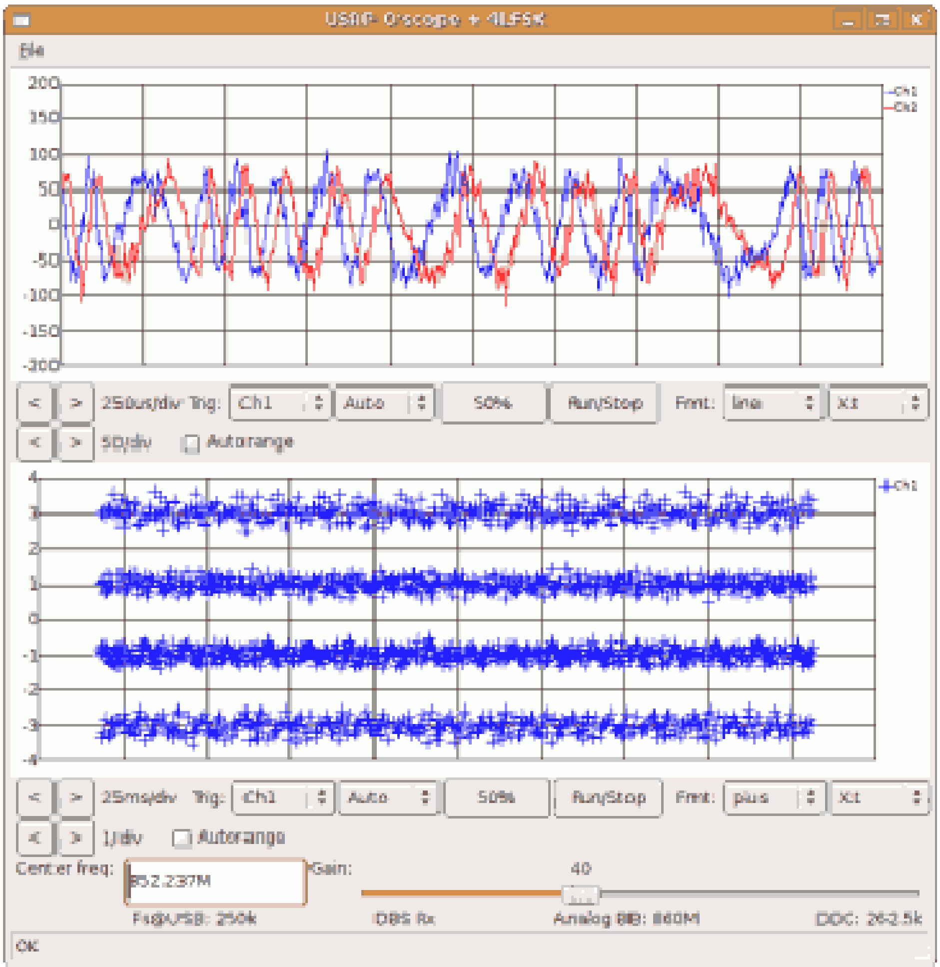

图 6-2 USRP 软件示波器，FSK-4

第7章　其他的 SDR 平台简介

本章将介绍其他一些软件无线电平台，以及具有 SDR 概念的产品。目的是为读者拓宽视野，了解行业现状。

7.1　几种 SDR 平台简介

首先，列举一些 SDR 领域的公司的产品或者高校的科研成果。这里没有包含那些传统的 FPGA、DSP 厂家，以及数据采集卡这样的产品。从广义的范畴来讲，所有这些可编程的且可用于无线通信的产品都可以称为 SDR 产品。不过本书的目的是介绍 GNU Radio，因此主要讨论一些跟 GNU Radio 类似的东西。

下图把下面列举的 SDR 平台归纳在一起，按照“网络侧平台”，“终端侧平台”，“研究用平台”，“商用平台”四种特点分类，说明了他们各自的特点和应用领域。

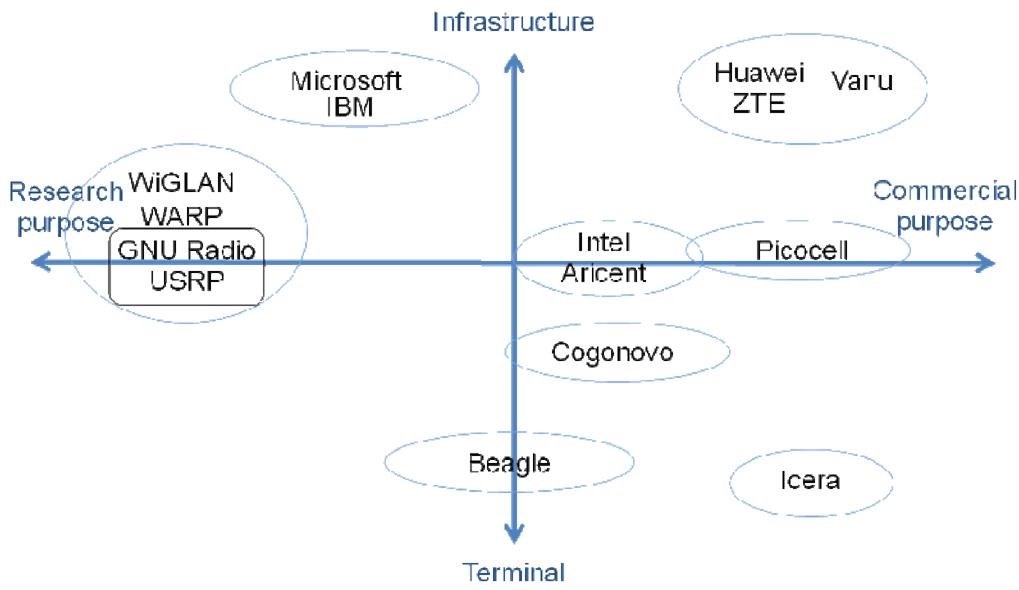

图 7-1 各种 SDR 平台以及他们的应用领域

下面依次简单介绍一下各个平台。

- Vanu（www.vanu.com）：首个采用计算机架构来制造移动通信基站的公司。它是从 MIT 的一个研究项目分离出来的公司。Vanu 生产了第一个通过 FCC 认证的 SDR 的 GSM/CDMA 双模基站。并且在一个叫 Mid-Tex 的小型运营商那里首次得到了商用。我想这也体现了计算机架构的基站的处理能力是很有限的，还不能在用户数量很大的地方使用。
- 华为 / 中兴：华为和中兴也有称之为 SDR 的多模基站产品。不过他们的产品主要特点其实是 RRU-BBU 架构。RRU（Remote Radio Unit）同时支持 GSM 和 3G 两种制式，有一个宽带的功放，比如 20MHz，可以同时支持几个 3G 的载波再加几个 GSM 的载波，将来还可以升级到 LTE 支持一个 20MHz 载波。而 BBU（Base Band Unit）的部分是可以在不同制式

间更换的，目前是通过更换 ASIC 板的方式，将来可以变为例如 FPGA/DSP 甚至计算机架构，直接用软件升级。因为目前基站中最昂贵的部分是射频单元，所以如果射频单元可以保持不更换，就可以节省运营商的投资。因此这种方案适合运营商逐步的升级网络，逐步减少 GSM 载波，更换为 3G 载波，最后换为 LTE 载波。华为和中兴在宽带 RRU 的研发上作出了很大的贡献，使得一个宽带的 RRU 可以保证各种不同制式对于射频指标的要求。

- Picochip
  www.picochip.com
  Picochip 公司的产品都是基于他们的 PicoArray 来做的。PicoArray 是一种 DSP cluster，或者可以叫多核 DSP。通过多个 DSP 并行处理，提供高速的运算能力。Picochip 主要的产品是家庭基站。 这也是由 PicoArray 的特点决定的，体积和功耗较大，不适合做终端，计算能力又不是非常强，所以不适合大规模的宏站。

- Microsoft
  http://research.microsoft.com/en-us/projects/sora/
  微软是做操作系统的，所以它的背景天生就非常适合去开发基于计算机架构“软件”无线电，特别是针对操作系统底层的优化。微软中国研究院开发的 SORA 平台，包括一个硬件的板卡系列和一套软件套件。目前（2009 年底）SORA 上已经实现了 802.11a/g， LTE 的开发好像在进行中。稍后我们会更详细的介绍一下 SORA。

- Intel + Aricent：(http://download.intel.com/netcomms/solutions/ipservices-wireless/322035.pdf)
  Intel 也是非常适合做 SDR 的，因为它知道 CPU 该怎么优化才更适合做信号处理。Aricent 非常擅长做通信协议栈。所以 Intel 和 Aricent 联手提出了所谓 LTE eNodeB solution。但这个解决方案公开的资料非常有限，我们仅仅知道 Intel 的多核处理器加上 Aricent 的软件，就可以实现一个 LTE eNodeB。

- Icera
  www.icerasemi.com
  Icera 是一个 2002 年创立的芯片制造公司，做的是终端侧的 SDR 产品。他们称自己的产品是世界上第一款为终端设计的高性能软件无线电参考设计。产品的核心是一块高性能的 DSP 芯片，上面可以装载各种制式的软件，从 GSM 到 HSPA+等一系列模式都可以支持，适用于智能手机和数据卡。

- Cognovo
  http://www.cognovo.com
  Cognovo 比 Icera 更年轻，是 2009 年从 ARM 分离出来的。他们的理想是做出另一个类似 ARM 的产品，让终端制造商可以他们的平台通过软件开发的方式，快速的生产符合新标准的产品，而无需等待上游芯片厂家开发专用的新标准的芯片。Icera 和 Cognovo 让我们相信，终端侧的 SDR 梦想也不那么遥远了。

- Beagle board
  http://beagleboard.org/
  Beagle board 是一个开放的硬件项目，是一个基于 TI OMAP3530 处理器(ARM Cortex A-8 core)的单板计算机，一个非常廉价的平台。所以很多同样喜欢廉价的 USRP 的人，用它与 USRP 连接，建立一些小型系统，因此在 GNU Radio 社区中备受关注。

- WARP
  http://warp.rice.edu/trac
  WARP 是莱斯大学针对科研工作开发的一个 SDR 平台，主要用于算法验证，它的硬件平台通过。它比 USRP 的性能指标要高，无论是 AD/DA 转换速率还是 FPGA 容量等，包括射频器件的品质都要高一些。但相应的，WARP 的价格也较贵，因此它的用户主要是一些大的高校实验室和大公司，用户群相对 USRP 小一些。

- IBM
  索尼，东芝和 IBM 共同开发的的 cell 处理器，是一种含有多个 SPU（Synergistic Processing Element）的处理器，具有很强的并行处理能力。它主要是为 PlayStation3 设计的，用于处理游戏中大量的图像数据。Cell 处理器也适用于医疗图像处理，雷达和声纳信号处理，IBM 还用它来制造超级计算机。因为 Cell 处理器的这种并行处理能力，所以它非常适合于 SDR 开发，IBM 的研究团队在大约 2007~2008 年，在 PlayStation3 上开发了 WiMAX 的物理层（不包含射频模块）。GNU Radio 社区中也有很多人用 PlayStation3 来开发他们的 SDR 应用，因为 PlayStation3 也可以安装 Linux，而且也有 USB 接口。不过，2009 年 IBM 宣布，要停止 Cell 处理器的研发，不过他们仍然会将 Cell 处理器的技术应用到其他的产品中，结合 GPU（Graphic Processing Univ），把一种混合的架构作为将来的研发方向。

总的来看，首先，现在已经有越来越多的非传统电信厂商，通过 SDR 技术，进入电信领域；其次，原先对于 SDR 技术在终端侧应用大家都有所顾虑，而现在已经有公司专门为终端设计出低功耗小体积的 SDR 平台，也许真正的 SDR 手机不会太遥远了。

7.2 微软的 SORA[7]

如果微软对于 SORA 的研发持续下去的话，SORA 有可能就会流行起来，成为相对 GNU Radio 最有竞争力的一个平台，从而把 Windows 和 Linux 的战争扩展到 SDR 领域。

7.2.1 SORA 上已经实现了什么系统？

目前 SORA 上已经实现了 802.11a/b/g 和 LTE 的物理层上行链路。

[7]本节关于 SORA 的介绍全部摘自微软发表在 NSDI’09 的文章：“Sora: High Performance Software Radio Using General Purpose Multi-core Processors”

在 802.11 的实现上，相比 GNU Radio，SORA 解决了两个重要的问题：第一，高带宽，SORA 的 PCIe 接口使得它可以支持 20MHz 信道带宽的采样；第二，低时延，SORA 可以满足 802.11 几十个 ms 的时延要求，低时延的实现主要归功于 SORA 的高性能计算能力和对于 PCIe 接口的定时的控制能力。

目前在微软研究院的网页上，可以下载到 SORA 的 SDK，802.11b 的代码也将要发布出来。

7.2.2 硬件接口板：RCB（Radio Control Board）

SORA 的硬件部分包括 RCB 板和射频前端两部分。

RCB 板的功能主要是“接口”，可以成为接口板。它负责把输入的基带数据流，通过 PCIe 接口送入内存；它的另外一个功能是用各种 buffer 把严格同步的基带数据跟异步的 CPU 连接在一起，使得 CPU 可以处理一些需要快速响应的操作。因此，RCB 板是一个与 SORA 软件紧密关联的部分。

而射频前端则不同，射频前端与 SORA 实际上的相互独立的。射频前端完成射频信号收发和 AD/DA 转换。SORA 可以连接不同厂家的射频板卡，比如 WARP 和 USRP 的射频板卡都可以。这样做的目的是，让 SORA 仅仅只是一个 platform，可以用在各种不同的系统中。

图 7-2 RCB 板

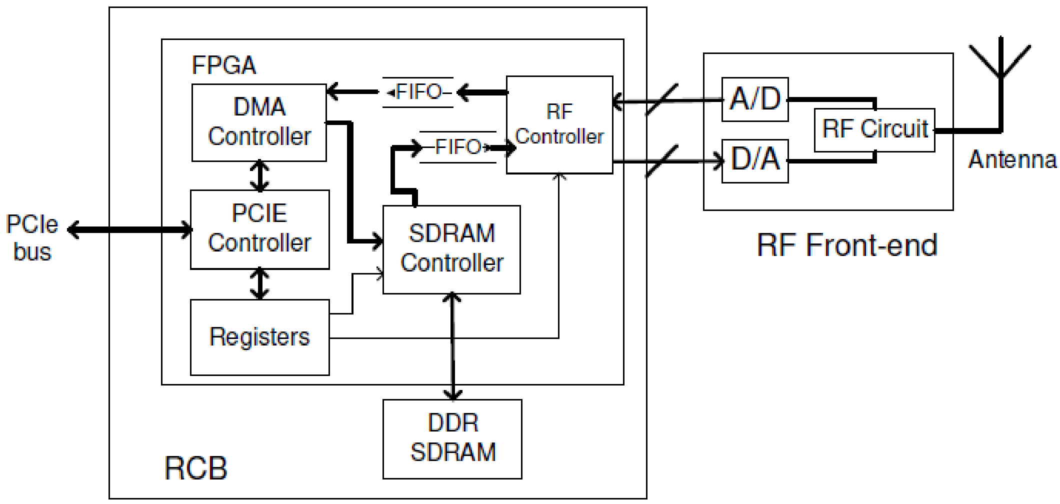

图 7-3 RCB 板和射频前端

RCB 板的 PCIe 接口，理论上可以支持高达 64Gbps（PCIe x32）的吞吐量，目前产品一般可以支持几个 G 左右的吞吐量，对于目前大部分的无线系统，这样的吞吐量应该是足够快的。

RCB 板的价格目前是 2000 美元左右，与 USRP2 价格相当。

7.2.3 软件架构

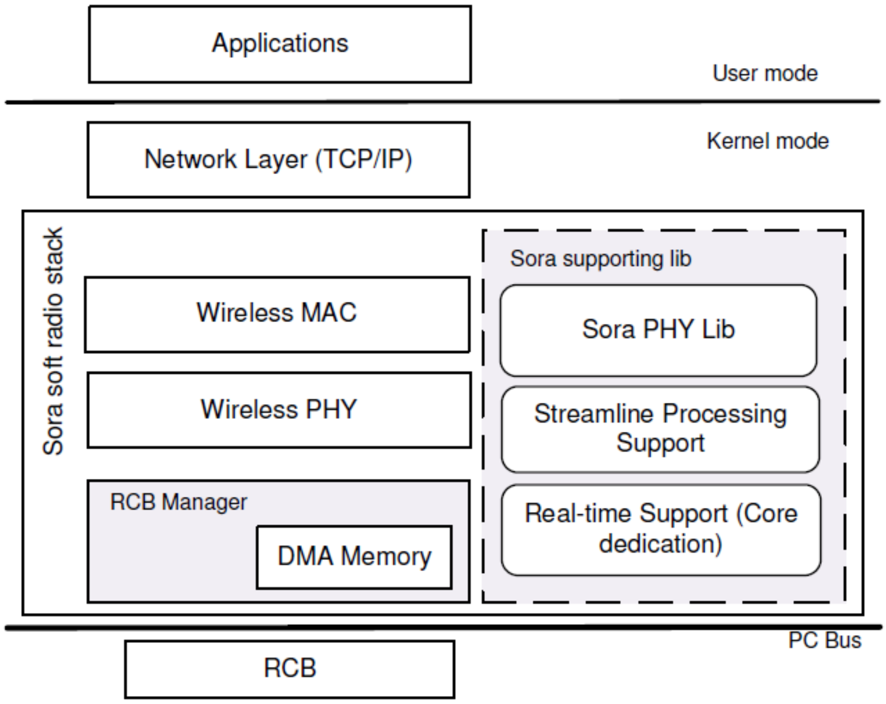

图 7-4 SORA 的软件架构

上图给出了一个 SORA 软件部分的总体架构。SORA 目前的 SDK 应该包括其中的 RCB 管理器和 SORA 支持库部分。

7.2.4 如何提高 CPU 做通信信号处理的速度

我们知道，CPU 的结构天生就不适合用于做实时性很强的通信信号处理。所以，如果我们用 CPU 来做 SDR，经常会遇到，大量的数据不能及时处理，或者虽然能处理，但是时延却比较大。而微软，作为操作系统的专家，利用他们的专长设计了一些“加速”方法，使得 CPU 也能够做实时的通信信号处理。在我看来，这些方法，对于我们 GNU Radio 的使用者来说，也是非常有参考价值的。

下面简单介绍一下 SORA 针对高速计算所使用的一些“加速”方法。

7.2.4.1查找表 LUT(LookUp Table)

把一些算法预先做成查找表，存在 CPU 的 L2 cache 中，这样就可以在很短的时间内（10~20 cycles），算出结果。

比如，通信中常用到的 soft demapper。一般的做法需要计算一个似然比，需要求很多个欧氏距离，还有除法和对数运算。而如果用查找表，我们可以把所有可能的接收信号（一个量化的复数）都算出它的软信息值，存成一个表，然后计算的时候只需要查表就可以了。

比如，802.11a/g 中 64QAM 解映射的查找表，大小为 1.5KB。802.11 中半数以上的物理层算法都可以用 LUT 来实现，加速比大约是 1.5x~50x。802.11a/g 的全部的查找表总共大约 200KB，而 802.11b 大约是 310KB，把他们放在常见的 CPU 的 L2 cache 中，都能装得下。

在传统的 ASIC/FPGA 实现中，不使用这种查找表的原因，一是芯片上设计的算法本身就可以算得很快（3 cycles for 64QAM），二是这样做会占用芯片上宝贵的 RAM，从而增加芯片的面积。

7.2.4.2SIMD

SIMD 是广泛用于图像和视频信号处理的加速方法，当然，也适用于通信信号处理。

那些必须顺序处理的，有迭代特点的算法，比较适合用 SIMD 优化。比如 FIR 滤波器，viterbi 译码等等。

SORA 的 viterbi 译码器据说可以用单个 CPU 核处理 802.11a/g 54Mbps 的吞吐量。

下图列出了一些关键算法在优化之前和之后的运算量比较。

Algorithm	Configuration	I/O Size (bit)		Optimization Method	Computation Required (Mcycles/sec)		
		Input	Output		Conv. Impl.	Sora Impl.	*Speedup*
IEEE 802.11b							
Scramble	11Mbps	8	8	LUT	96.54	10.82	8.9x
Descramble	11Mbps	8	8	LUT	95.23	5.91	16.1x
Mapping and Spreading	2Mbps, DQPSK	8	44*16*2	LUT	128.59	73.92	1.7x
CCK modulator	5Mbps, CCK	8	8*16*2	LUT	124.93	81.29	1.5x
	11Mbps, CCK	8	8*16*2	LUT	203.96	110.88	1.8x
FIR Filter	16-bit I/Q, 37 taps, 22MSps	16*2*4	16*2*4	SIMD	5,780.34	616.41	9.4x
Decimation	16-bit I/Q, 4x Oversample	16*2*4*4	16*2*4	SIMD	422.45	198.72	2.1x
IEEE 802.11a							
FFT/IFFT	64 points	64*16*2	64*16*2	SIMD	754.11	459.52	1.6x
Conv. Encoder	24Mbps, 1/2 rate	8	16	LUT	406.08	18.15	22.4x
	48Mbps, 2/3 rate	16	24	LUT	688.55	37.21	18.5x
	54Mbps, 3/4 rate	24	32	LUT	712.10	56.23	12.7x
Viterbi	24Mbps, 1/2 rate	8*16	8	SIMD+LUT	68,553.57	1,408.93	48.7x
	48Mbps, 2/3 rate	8*24	16	SIMD+LUT	117,199.6	2,422.04	48.4x
	54Mbps, 3/4 rate	8*32	24	SIMD+LUT	131,017.9	2,573.85	50.9x
Soft demapper	24Mbps, QAM 16	16*2	8*4	LUT	115.05	46.55	2.5x
	54Mbps, QAM 64	16*2	8*6	LUT	255.86	98.75	2.4x
Scramble & Descramble	54Mbps	8	8	LUT	547.86	40.29	13.6x

图 7-5 一些关键算法的运算量比较

7.2.4.3多核并行处理

现在的 CPU 一般至少有两个核，将来的 CPU 会有更多的核。因此，利用多个核并行处理，是一种很自然的提速方法。

SORA 使用的是一种静态调度方法。意思是，一个通信系统是由许多个模块构成一个流图，这些模块被固定的分配在各个的 CPU。比如 decimation，FFT 和 demodulation 放在 core 1 上，而 viterbi decoder 放在 core 2 上。内核如何分配是由程序员控制的，不是 SORA 自动分配的。

内核与内核之间的数据传递，采用了 SORA 专门设计的同步 FIFO。据说这种 FIFO 的开销比较小。

为了达到实时处理，SORA 使用了 exclusive thread（独占线程？）。也就是说，每个分配给 SORA 使用的内核，它上面运行的线程都配置为最高优先级，独占了这个内核，使这个内核不响应其他任何进程，甚至所有的硬件中断都不响应。所以，为了保证计算机还能正常工作，SORA 会把 core 0 保留给硬件中断，只使用从 core 1 开始的其余内核。

7.2.5 SORA 对 TDMA 的支持

这里想特别提一下 SORA 的定时机制。与 GNU Radio 的 time stamping 不同，SORA 对 TDMA 的支持是通过 CPU 对 RCB 的快速响应来实现的。

每当 RCB 板卡把数据发送完成之后，就会通知主机。因此主机通过计算发送的数据帧的时长，知道射频侧的定时。而用于 SORA 的内核是独占进程，所以它可以通过计算 sample 的数量来准确的定时。微软的测试表明，这种定时方式的平均误差小于 1us。

简而言之，SORA 平台最核心的东西，是它的 PCIe 接口和与这个接口相匹配的软件。我们希望 SORA 将来能够在开发环境上，变得更友好，使广大的 windows 用户可以像使用 VC 一样开发自己的 SDR 应用。