如何编写 GNU Radio 信号处理功能块

Eric Blossom

<eb@comsec.com>

修正历史	
修正版 0.1	2005-01-20
修正版 0.2	2005-02-02
针对 SWIG 1.3.24 的更新	
修正版 0.3	2006-07-21
Clarification of 1:1 fixed rate vs item size	

摘要

此文阐述如何为 GNU Radio 编写信号处理模块。

目录

GNURadio 中文社区
GNURadio: 中文维客
标题索引: GNURadio
GNURadio: 安装指导
GNURadio: 常见问题
GNURadio: 讲义教程
GNURadio: 社区贡献
GNURadio 最新发布
GNURadio 3.6.3
GNURadio 3.6.3rc0
GNURadio - OpenBTS 中文维客
(Range) - OpenBTS 中文社区
SCA - OSSIE 中文维客

序言 - Prerequisites

此文基于读者具备初步掌握 GNU Radio 并且已经阅读并理解 GNU Radio 探秘（*Exploring GNU Radio*）。

随同还发布一个 tarball 格式的文档，它将包含所有的例程，DocBook 源码以及所有的 Makefiles 等等。到 ftp://ftp.gnu.org/gnu/gnuradio 或镜像站点便可找到一个文件名为 gr-howto-write-a-block-X.Y.tar.gz （所需的都将包含其中）。挑出最新版本即可。有关 CVS 读取方法可参阅 http://comsec.com/wiki?CvsAccess 。

引言 - Introduction

GNU Radio 提供一个构建软件无线电的框架。Waveforms -- 信号处理应用程序 -- 是结合 Python 代码用于构架高层次的组织、策略、GUI 及其它非性能相关的功能，性能相关的信号处理模块是由 C++ 构建的。

站在 Python 的角度来看，GNU Radio 是数据流的抽象。其基本概念是连接起来的信号处理模块。这些工作是基于 Python 的 gr.flow_graph 类来实现的。每个（信号处理）模块都有一组输入和输出端口。每个端口都有与其关联的数据结构。最常见的端口类型是 float 和 gr_complex (等同于 std::complex<float>)，当然其它的类型也被用到，诸如：structures, arrays 或其它打包数据（packetized data）等的表述.

从宏观的来看，（GNU Radio）是无尽的流自模块端口的数据流。基于 C++ 的观点，数据流是采用合适的大小处理数据块，以连续的数组格式表述其数据类型。

综览 - The View from 30,000 Feet

这篇文章通过组建一些简单的信号处理模块来阐述所用到的技巧和术语。最后将涉及关联混合 Python/C++ 环境、性能测试和优化条件下的信号处理模块的诊断。

例程模块将基于延伸 GNU Radio 的方式构建。也就是其构建于 GNURadio-core 的构建树之外，构建为能够被 Python 使用"import"机制调入的共享库。SWIG （the Simplified Wrapper and Interface Generator）用来充当粘合作用，其目的是让 Python 能够使用调用用户的代码。

C++ 的类 **gr_block** 是 GNU Radio 的所有的信号处理模块的基石。编写一个新的信号处理模块涉及 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 粘合此新的类同 Python 的 **.i** 文件。新的类必须是从 gr_block 衍生出来或是其子类。

这里的例程将直接从 gr_block 衍生出来。最后将会查看一些通用场合的子类的例程。

Autotools, Makefiles, 及目录结构（Directory Layout）

在染指代码之前，先略表所涉及的构建环境和目录结构的概貌。

为了减少 Makefile 带来的为数众多的麻烦，也是为了加强不同系统之间的可移植性，采用 GNU autoconf, automake, 和 libtool 的工具便是其原因。它们被统称其为 autotools。一旦渡过“初识”的不适，它们便会成为你的朋友。（好的消息是提供模板几乎可以被原封不动的使用。）

- automake

automake 和 configure 一同来产生来源于以高级描述方式包含在 Makefiles.am 的 GNU 相关的 Makefiles 文件。Makefile.am 描述了将构建的库和程序和源码文件用来构建它们。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 包含成千上万的规则来做合适的事情：构建、检查和安装代码。产生 5 或者 6 倍于 Makefile.am 代码的 Makefile 是件很寻常的结果。

- **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++ 代码中来控制构建的变量和定义。如果所需的特征没有被发现， configure 将输出错误（error）信息并终止工作。

- **libtool**

  libtool 默默地并神奇地为系统多方位构建共享库。

表 1, “目录结构 - Directory Layout” 展示将要使用的目录结构（Directory Layout）和常规文件。相应的更换 *topdir* 的名称后，出现在具体的项目之中。在随后相应的文件出现时，将会对其进行详述。

Table 1. 目录结构 - Directory Layout

File/Dir Name	Comment
topdir/Makefile.am	Top level Makefile.am - 最高层的 Makefile.am
topdir/Makefile.common	Common fragment included in sub-Makefiles - Makefiles 的通用片段
topdir/bootstrap	Runs autoconf, automake, libtool first time through
topdir/config	Directory of m4 macros used by configure.ac
topdir/configure.ac	Input to autoconf - autoconf 的输入
topdir/src	
topdir/src/lib	C++ code goes here - C++ 代码在此
topdir/src/lib/Makefile.am	
topdir/src/python	Python code goes here - Python 代码在此
topdir/src/python/Makefile.am	
topdir/src/python/run_tests	Script to run tests in the build tree

命名规则 - Naming Conventions

遵从 GNU Radio 所使用的一组命名规则，有助理解代码以及粘合 C++ 和 Python 在一起。

- 再见大小写相间的命名格式 - *Death to CamelCaseNames!*

  已经回到一个更加友善、更加文明的时代。我们使用的是经过略加修改的"STL style"的命名规则，其原因是并没有用到名字空间（namespaces）。

  特例只限于宏（macros）和恒定数值（constant values），所有其它的标识符（identifiers）都是使用小写格式如此：words_separated_like_this 。

  宏 - macros 和恒定数值 - constant values （诸如：枚举数值、静态恒定整数 FOO = 23）是采用大写格式如此：UPPER_CASE。

- 全局命名 - Global Names

  所有的全局可见名称（类型、函数、变量、常量、等等）都以“软件包的前缀”开头，以下划线随后。数量众多的 GNU Radio 的代码隶属于 "gr"(软件)包，所以所见都是诸如：gr_open_file (...)的形式出现的。

  也有为数不少的的代码使用其它的包前缀，以便为使其规范化特列单如下。

- 软件包前缀 - Package Prefixes

  目前所用的软件包前缀如下:

  - gr_

    几乎是涵盖一切。

  - gri_

    构建原函数（Implementation primitives）。有时会出现两个诸如 gr_*foo* 和 a gri_*foo* 。于此， gr_*foo* 源自于 gr_block 而 gri_*foo* 便是相对底层的函数的核心。

  - atsc_

    同 Advanced Television Standards Committee HDTV 相关联的代码。

  - usrp_

    同 Universal Software Radio Peripheral 相关联的代码.

  - qa_

    质量保证相关代码（测试目的） - Quality Assurance (Test code.)

- 类数据成员（实例变量） - Class Data Members (instance variables)

  所有的类的数据成员应当以 d_*foo* 形式开头。

  （这样做）最大的收益是，当面对一段代码块时，一眼便知那些被永久的在块外赋值。这样也防止“创造性”的使用类的方法（methods）和构造函数的参数名。可以使用同实例变量（instance variable）一样仅仅没有 d_ 前缀相同的名称。

```
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)
    {
        ...
    }
    ...
};
```

- 类静态数据成员（类变量）- Class Static Data Members (class variables)

  所有的类静态数据成员（类变量）应当以 s_*foo* 形式开头

- 文件命名 - File Names

每个关键的类都应当包含在它自己的文件中。类 gr_foo 的声明的声明应当由 gr_foo.h 实现，类的定义由 gr_foo.cc 来完成。

- 后缀 - Suffixes

  常规意义上而言，信号处理模块的输入和输出类型通过其名称的后缀被码制化。其后缀通常占一到两个字符。“源”和“漏”的后缀各占一个字符。常规的模块含有输入和输出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，第二个字符表示输入的数据流的类型。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）。

  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)

  进而，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。gr_fft_vcc 是使用它的一个例子。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。

第一个模块：howto_square_ff

第一个例程模块要做的是计算单浮点输入的平方值。该模块接纳单浮点输入数据流，而后产生单浮点输出数据流。

遵从命名规则，使用 howto 作为包前缀，模块命名为 howto_square_ff 。

（程序的布局）做如此安排，此模块，连同此文的其它模块将被终结为 gnuradio.howto 类型的 Python 模块。（这样一来）便可以如下这般从 Python 的角度来引用：

```
from gnuradio import howto
sqr = howto.square_ff ()
```

- 测试受驱使的程序 - Test Driven Programming

  一开始便一头扎入 C++ 代码之中吗？是的，是可以的。但是，作为一个经过多年积累的成熟的程序员，理应从测试代码开始。总之，良好的习惯还是理应的：输入接收单浮点数据流，输出产生单浮点数据流。输出是输入放入平方值。

  难吗？查看完例程 1，“qa_howto.py (first version)”，会发现其实它很简单！

  例程 1. qa_howto.py (first version)

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

from gnuradio import gr, gr_unittest
import howto

class qa_howto (gr_unittest.TestCase):

def setUp (self):
self.fg = gr.flow_graph ()

def tearDown (self):
self.fg = None
```

```

def test_001_square_ff (self):
src_data = (-3, 4, -5.5, 2, 3)
expected_result = (9, 16, 30.25, 4, 9)
src = gr.vector_source_f (src_data)
sqr = howto.square_ff ()
dst = gr.vector_sink_f ()
self.fg.connect (src, sqr)
self.fg.connect (sqr, dst)
self.fg.run ()
result_data = dst.data ()
self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)

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

gr_unittest 是标准的 python 模块 unittest 的延伸。gr_unittest 添加支持查看浮点和复合数据类型的元值（tuples）是否大致相同的功能。Unittest 使用 Python 的反射机制来发现所有的表达式以 test_ 开头并运行这些表达式。Unittest 使用泛包裹每个调用来匹配 test_* 从而（决定）调用 setUp 和 tearDown 来完成任务的。详细的 Python 的 unittest 相关文档到 unittest 查看。

每当运行 test 时，gr_unittest.main 便（调用）触发 setUp、test_001_square_ff、和 tearDown 。

test_001_square_ff 构建了一个小流程图（例）其中包含三个节点。**gr.vector_source_f**(src_data)将以 src_data 的元素来“源”数据流便结束它的使命；**howto.square_ff** 是用来测试的模块；**gr.vector_sink_f** 用来 - “漏”接 - 收集 howto.square_ff 的输出数据流。

run （函数）表达式的功用是在流程图的引导下运行所有的所表达的模块。最后所关心的是，查看 square_ff 作用于 src_data 的结果同期望值是否相同。

- 构建树、安装树 - Build Tree vs. Install Tree

构建树 - build tree - 是自 *topdir*（其中包含 configure.ac）往下的一切。通往安装树的路径是指 *prefix*/lib/python*version*/site-packages，其中 *prefix* 是使用 configure 命令时的参数 --prefix （一般缺省是 /usr/local），*version* 是指被安装的 Python 的版本（version）。一个典型的实例如：/usr/local/lib/python2.3/site-packages 。

一般的，在构建树上，通过文档 ~/.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 。

一方面希望写好的应用程序能够在安装树内读取代码和库。另一方面，希望测试代码运行在构建树上，这样一来诊断问题便在安装之前可以做到。

- 进行检测 - make check

使用 **make check** 来进行测试。Make check 触发用来设置PYTHONPATH 环境变量的脚本程序 **run_tests**，这样一来，该测试代码使用的是构建树版本的代码和库。它运行所有以 qa_*.py 形式的文本，并告知一般意义的成功和失败。

使用没有被安装的用户代码颇需一些幕后故事(工作）（参看 runtest 会减少惊奇心态。）

最后，在 Python 命令下运行 **make check** 便会出现如下结果：

```
[eb@bufo python]$ make check
make  check-TESTS
```

```
make[1]: Entering directory `/home/eb/gr-build/gr-howto-write-a-block/src/python'
Traceback (most recent call last):
      File "./qa_howto.py", line 24, in ?
            import howto
ImportError: No module named howto
Traceback (most recent call last):
      File "./qa_howto_1.py", line 24, in ?
            import howto
ImportError: No module named howto
FAIL: run_tests
===================
1 of 1 tests failed
===================
make[1]: *** [check-TESTS] Error 1
make[1]: Leaving directory `/home/eb/gr-build/gr-howto-write-a-block/src/python'
make: *** [check-am] Error 2
[eb@bufo python]$
```

好极了！测试失败，真如估计的一样。出错信息 - ImportError: No module named howto - 表明无法找到名字为 howto 的模块。不要意外，因为它还没有被构建。

- C++ 代码 - The C++ code

到目前为止，如上所编写的测试例程“成功”地回报失败（fails）的结果，这样以来（便有必要探讨）编写 C++ 代码。正如如前所述，所有的信号处理模块衍生于类（class） gr_block 或它的子类 - subclasses 。如下的例程 2，便是关于“gr_block.h”：

Example 2. gr_block.h

```
/* -*- c++ -*- */
/*
* Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/

#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H
```

```

#include <gr_runtime.h>
#include <string>

/*!
* \brief The abstract base class for all signal processing blocks.
* \ingroup block
*
* Blocks have a set of input streams and output streams. The
* input_signature and output_signature define the number of input
* streams and output streams respectively, and the type of the data
* items in each stream.
*
* Although blocks may consume data on each input stream at a
* different rate, all outputs streams must produce data at the same
* rate. That rate may be different from any of the input rates.
*
* User derived blocks override two methods, forecast and general_work,
* to implement their signal processing behavior. forecast is called
* by the system scheduler to determine how many items are required on
* each input stream in order to produce a given number of output
* items.
*
* general_work is called to perform the signal processing in the block.
* It reads the input items and writes the output items.
*/

class gr_block {

public:

virtual ~gr_block ();

std::string name () const { return d_name; }
gr_io_signature_sptr input_signature () const { return d_input_signature; }
gr_io_signature_sptr output_signature () const { return d_output_signature; }
long unique_id () const { return d_unique_id; }

/*!
* Assume block computes y_i = f(x_i, x_i-1, x_i-2, x_i-3...)
* History is the number of x_i's that are examined to produce one y_i.
* This comes in handy for FIR filters, where we use history to
* ensure that our input contains the appropriate "history" for the
* filter. History should be equal to the number of filter taps.
*/
unsigned history () const { return d_history; }
```

```
void set_history (unsigned history) { d_history = history; }

/*!
* \brief return true if this block has a fixed input to output rate
*
* If true, then fixed_rate_in_to_out and fixed_rate_out_to_in may be called.
*/
bool fixed_rate() const { return d_fixed_rate; }

// ----------------------------------------------------------------
// override these to define your behavior
// ----------------------------------------------------------------

/*!
* \brief Estimate input requirements given output request
*
* \param noutput_items number of output items to produce
* \param ninput_items_required number of input items required on each input stream
*
* Given a request to product \p noutput_items, estimate the number of
* data items required on each input stream. The estimate doesn't have
* to be exact, but should be close.
*/
virtual void forecast (int noutput_items,
gr_vector_int &ninput_items_required);

/*!
* \brief compute output items from input items
*
* \param noutput_items number of output items to write on each output stream
* \param ninput_items number of input items available on each input stream
* \param input_items vector of pointers to the input items, one entry per input stream
* \param output_items vector of pointers to the output items, one entry per output stream
*
* \returns number of items actually written to each output stream, or -1 on EOF.
* It is OK to return a value less than noutput_items. -1 <= return value <= noutput_items
*
* general_work must call consume or consume_each to indicate how many items
* were consumed on each input stream.
*/
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;

/*!
* \brief Confirm that ninputs and noutputs is an acceptable combination.
*
* \param ninputs number of input streams connected
* \param noutputs number of output streams connected
*
* \returns true if this is a valid configuration for this block.
*
* This function is called by the runtime system whenever the
* topology changes. Most classes do not need to override this.
* This check is in addition to the constraints specified by the input
* and output gr_io_signatures.
*/
virtual bool check_topology (int ninputs, int noutputs);

/*!
* \brief Called to enable drivers, etc for i/o devices.
*
* This allows a block to enable an associated driver to begin
* transfering data just before we start to execute the scheduler.
* The end result is that this reduces latency in the pipeline when
* dealing with audio devices, usrps, etc.
*/
virtual bool start();

/*!
* \brief Called to disable drivers, etc for i/o devices.
*/
virtual bool stop();

// ------------------------------------------------------------------------

/*!
* \brief Constrain the noutput_items argument passed to forecast and general_work
*
* set_output_multiple causes the scheduler to ensure that the noutput_items
* argument passed to forecast and general_work will be an integer multiple
* of \param multiple The default value of output multiple is 1.
*/
void set_output_multiple (int multiple);
int output_multiple () const { return d_output_multiple; }

/*!
* \brief Tell the scheduler \p how_many_items of input stream \p which_input were consumed.
*/
void consume (int which_input, int how_many_items);

/*!
* \brief Tell the scheduler \p how_many_items were consumed on each input stream.
*/
void consume_each (int how_many_items);

/*!
* \brief Set the approximate output rate / input rate
*
```

```
* Provide a hint to the buffer allocator and scheduler.
* The default relative_rate is 1.0
*
* decimators have relative_rates < 1.0
* interpolators have relative_rates > 1.0
*/
void set_relative_rate (double relative_rate);

/*!
* \brief return the approximate output rate / input rate
*/
double relative_rate () const { return d_relative_rate; }

/*
* The following two methods provide special case info to the
* scheduler in the event that a block has a fixed input to output
* ratio. gr_sync_block, gr_sync_decimator and gr_sync_interpolator
* override these. If you're fixed rate, subclass one of those.
*/
/*!
* \brief Given ninput samples, return number of output samples that will be produced.
* N.B. this is only defined if fixed_rate returns true.
* Generally speaking, you don't need to override this.
*/
virtual int fixed_rate_ninput_to_noutput(int ninput);

/*!
* \brief Given noutput samples, return number of input samples required to produce noutput.
* N.B. this is only defined if fixed_rate returns true.
* Generally speaking, you don't need to override this.
*/
virtual int fixed_rate_noutput_to_ninput(int noutput);

// ----------------------------------------------------------------------------

private:

std::string d_name;
gr_io_signature_sptr d_input_signature;
gr_io_signature_sptr d_output_signature;
int d_output_multiple;
double d_relative_rate; // approx output_rate / input_rate
gr_block_detail_sptr d_detail; // implementation details
long d_unique_id; // convenient for debugging
unsigned d_history;
bool d_fixed_rate;


protected:

gr_block (const std::string &name,
gr_io_signature_sptr input_signature,
gr_io_signature_sptr output_signature);

//! may only be called during constructor
void set_input_signature (gr_io_signature_sptr iosig){
d_input_signature = iosig;
}
```

```

//! may only be called during constructor
void set_output_signature (gr_io_signature_sptr iosig){
d_output_signature = iosig;
}

void set_fixed_rate(bool fixed_rate){ d_fixed_rate = fixed_rate; }

// These are really only for internal use, but leaving them public avoids
// having to work up an ever-varying list of friends

public:
gr_block_detail_sptr detail () const { return d_detail; }
void set_detail (gr_block_detail_sptr detail) { d_detail = detail; }
};

long gr_block_ncurrently_allocated ();

#endif /* INCLUDED_GR_BLOCK_H */
```

快速一瞥 gr_block.h 展示了一个这样的事实：因为 general_work 是纯粹的虚拟的（虚函数），所以根本不需关心它。 general_work 是用来进行实际信号处理的表达式。对于该求信号平方的例程而言，只管忽视 general_work 的功用，而只需关心如何提供一个构造和清除函数（constructor and destructor）以及一点点涉及如何运用 boost 和 shared_ptrs 的事情而已。

如下 例程3, “howto_square_ff.h” 和 例程 4, “howto_square_ff.cc” 是头文件和 C++ 源文件.

例程 3. howto_square_ff.h

```
/* -*- c++ -*- */
/*
* Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/
#ifndef INCLUDED_HOWTO_SQUARE_FF_H
```

```
#define INCLUDED_HOWTO_SQUARE_FF_H

#include <gr_block.h>

class howto_square_ff;

/*
* 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 ();

/*!
* \brief square a stream of floats.
* \ingroup block
*
* \sa howto_square2_ff for a version that subclasses gr_sync_block.
*/
class howto_square_ff : public gr_block
{
private:
// The friend declaration allows howto_make_square_ff to
// access the private constructor.

friend howto_square_ff_sptr howto_make_square_ff ();

howto_square_ff (); // private constructor

public:
~howto_square_ff (); // public destructor

// Where all the action really happens

int general_work (int noutput_items,
gr_vector_int &ninput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items);
};
```

```

#endif /* INCLUDED_HOWTO_SQUARE_FF_H */
```

例程 4. howto_square_ff.cc

```
/* -*- c++ -*- */
/*
* Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/

/*
* config.h is generated by configure. It contains the results
* of probing for features, options etc. It should be the first
* file included in your .cc file.
*/
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <howto_square_ff.h>
#include <gr_io_signature.h>

/*
* Create a new instance of howto_square_ff and return
* a boost shared_ptr. This is effectively the public constructor.
*/
howto_square_ff_sptr
howto_make_square_ff ()
{
return howto_square_ff_sptr (new howto_square_ff ());
}

/*
* Specify constraints on number of input and output streams.
* This info is used to construct the input and output
```

```
signatures
* (2nd & 3rd args to gr_block's constructor). The input and
* output signatures are used by the runtime system to
* check that a valid number and type of inputs and outputs
* are connected to this block. In this case, we accept
* only 1 input and 1 output.
*/
static const int MIN_IN = 1; // mininum number of input streams
static const int MAX_IN = 1; // maximum number of input streams
static const int MIN_OUT = 1; // minimum number of output streams
static const int MAX_OUT = 1; // maximum number of output streams

/*
* The private constructor
*/
howto_square_ff::howto_square_ff ()
: gr_block ("square_ff",
gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
// nothing else required in this example
}

/*
* Our virtual destructor.
*/
howto_square_ff::~howto_square_ff ()
{
// nothing else required in this example
}

int
howto_square_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];
}

// Tell runtime system how many input items we consumed on
// each input stream.

consume_each (noutput_items);

// Tell runtime system how many output items we produced.
return noutput_items;
}
```

现在便是可以用 Makefiles.am 来对这些来构建的时候。例程 5, “src/lib/Makefile.am (no SWIG)” 足以展示如何用源文件来构建一个共享库。下面将只添加一点点额外的规则来“牛刀小试” SWIG。如果还是不明白这一切，赶紧到构建树那儿完全的浏览一下 Makefile.am 的内容，这样便有可能明白这一切是如何的结合在一起的。

Example 5. src/lib/Makefile.am (no SWIG)

```
include $(top_srcdir)/Makefile.common

# Install this stuff so that it ends up as the gnuradio.howto module
# This usually ends up at:
# ${prefix}/lib/python${python_version}/site-packages/gnuradio

ourpythondir = $(grpythondir)
ourlibdir = $(grpyexecdir)

INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)

ourlib_LTLIBRARIES = _howto.la

# These are the source files that go into the shared library
_howto_la_SOURCES = \
howto_square_ff.cc

# magic flags
_howto_la_LDFLAGS = -module -avoid-version

# These headers get installed in ${prefix}/include/gnuradio
grinclude_HEADERS = \
howto_square_ff.h

MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

- The SWIG .i file

  在这里，通过编写 SWIG.i 文件来展示一下编译过程。这是一个精简版的 .h 文件，点缀一点 python 结合 boost shared_ptr's 神奇魔力。为了不使代码显得臃肿，在此只声明那些需要读取 Python 的表达式。

  一下程序将要调用很小的 .i 文档 howto.i，使用它持有所有能够从 Python 读取的来自 howto 的类的 SWIG 的声明（the SWIG declarations）：

```
/* -*- c++ -*- */

%include "exception.i"
%import "gnuradio.i" // the common stuff

%{
#include "gnuradio_swig_bug_workaround.h" // mandatory bug fix
#include "howto_square_ff.h"
#include <stdexcept>
%}

// ------------------------------------------------------------------

/*
* 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 ();
};
```

- 综合：将所有这些综合一起 - Putting it all together

  现在通过修改 src/lib/Makefile.am 来运行 SWIG 产生粘合作用，从而生成共享库。

  例程 6. src/lib/Makefile.am (with SWIG)

```
#
# Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
# Boston, MA 02111-1307, USA.
#

include $(top_srcdir)/Makefile.common

# Install this stuff so that it ends up as the gnuradio.howto module
# This usually ends up at:
# ${prefix}/lib/python${python_version}/site-packages/gnuradio

ourpythondir = $(grpythondir)
ourlibdir = $(grpyexecdir)

INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)

SWIGCPPPYTHONARGS = -noruntime -c++ -python $(PYTHON_CPPFLAGS) \
-I$(swigincludedir) -I$(grincludedir)

ALL_IFILES = \
$(LOCAL_IFILES) \
$(NON_LOCAL_IFILES)

NON_LOCAL_IFILES = \
$(GNURADIO_CORE_INCLUDEDIR)/swig/gnuradio.i


LOCAL_IFILES = \
howto.i

# These files are built by SWIG. The first is the C++ glue.
# The second is the python wrapper that loads the _howto shared library
# and knows how to call our extensions.

BUILT_SOURCES = \
howto.cc \
howto.py

# This gets howto.py installed in the right place
ourpython_PYTHON = \
howto.py

ourlib_LTLIBRARIES = _howto.la

# These are the source files that go into the shared library
_howto_la_SOURCES = \
howto.cc \
howto_square_ff.cc

# magic flags
_howto_la_LDFLAGS = -module -avoid-version

# link the library against some comon swig runtime code and the
# c++ standard library
_howto_la_LIBADD = \
-lgrswigrunpy \
-lstdc++

howto.cc howto.py: howto.i $(ALL_IFILES)
$(SWIG) $(SWIGCPPPYTHONARGS) -module howto -o howto.cc $<

# These headers get installed in ${prefix}/include/gnuradio
grinclude_HEADERS = \
howto_square_ff.h

# These swig headers get installed in ${prefix}/include/gnuradio/swig
swiginclude_HEADERS = \
$(LOCAL_IFILES)

MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

make 现在成功的构建了一切，尽管招致几个警告，但这些无关紧要。

切换回目录 Python 之下，再次运行 **make check** 结果如下：

```
[eb@bufo python]$ make check
make   check-TESTS
make[1]: Entering directory `/home/eb/gr-build/gr-howto-write-a-block/src/python'
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

```
PASS: run_tests
==================
All 1 tests passed
==================
make[1]: Leaving directory `/home/eb/gr-build/gr-howto-write-a-block/src/python'
[eb@bufo python]$
```

万岁！ 成功了！

gr_block 模块的额外函数表达式

在如上的例程 howto_square_ff 中，刻意忽视 **general_work** 函数表达式进而实现想要的目的。 gr_block 同时还提供一些其它有用的函数表达式。

- forecast

  查看 **general_work** 时，脑海中也许会产生如此懈虑：系统如何来确定需要多大程度的数据来确保每个输入数组中的数据是有效的。**forecast** 就是用来消除这个懈惑的。

  forecast 的缺省的构建如此描述：对于所需的每个输入和 noutput_items 的比率关系是 1:1。gr_block 的构建函数中的 gr_io_signatures 用来定义名目（items）的大小。gr_io_signatures 定义的每个输入和输出的数据名目（items）的大小当然可以不相等；但此 1:1 的比率关系（名目形成的数据流 - 译者注）是应当被保证的。

  ```
  // default implementation: 1:1 - 缺省构建 1:1

  void
  gr_block::forecast (int noutput_items,
  gr_vector_int &ninput_items_required)
  {
  unsigned ninputs = ninput_items_required.size ();
  for (unsigned i = 0; i < ninputs; i++)
  ninput_items_required[i] = noutput_items;
  }
  ```

  尽管此 1:1 的构建关系对 howto_square_ff 有效，但它未必对插值函数 - interpolators、抽取函数 - decimators、或关联更复杂的 noutput_items 和所需输入（input requirements）关系的模块奏效。这就是说，从 gr_sync_block, gr_sync_interpolator 或 gr_sync_decimator来继承（衍生）用户的类，而不是直接从 gr_block 来继承，可以避免构建 **forecast** 。

- set_output_multiple

  当构建 general_work 时，偶尔显得很方便：通过运行时间系统（run time system）来确保所要求的仅仅是综合几个一些特定数值的复合的输出。这中情况常常发生于将算法无意识的应用于固定大小的数据流上。 因此需在构建函数中调用 set_output_multiple 来明确这种要求。此复合输出缺省值是 1 。

通常模块的子类 - Subclasses for common patterns

gr_block 允许十分灵活的方式接纳输入数据流以及产生输出数据流。灵巧的运用 forecast 和 consume 使得一个可变比率（接纳输入数据流和产生输出数据流的比率）的模块能够被构建。构建一个以不同比率接纳每个输入；而产生的输出数据的比率是同输入数据的内容以某种函数关系相联系的模块是可行的。

另一方面，信号处理模块采用固定输入和输出比率是十分常见的。通常采用 1:1，但是采用 1:N 或 N:1 也很常见。

另外一个常规需求是检测多个输入采样从而产生一个单一输出采样。对于输入和输出比率关系而言，这是一个司空见惯的需求。比如，一个没有抽取 - non-decimating、没有插值 - non-interpolating FIR 滤波器而言，产生每个输出样例（output sample）都必须对 N 个输入采样进行分析，这里的 N 便是此滤波

器的抽头（taps）个数。然而，对于那种仅接纳一个输入然后产生一个输出的，这种情况被称之为“历史”。当然也可以被认为是“前瞻”。

- gr_sync_block

  gr_sync_block 是从 gr_block 衍生下来的，并“历史性地”继承了 1:1（入出比率）的关系。既然预先得知输入和输出的比率关系，这样便可以在某种程度上使问题简化。从构建的角度来看，最大的不同是使用 work 函数表达式来替代了 general_work 的构建。 work 函数表达式在调用顺序上略有不同；它省略没有必要的参数： ninput_items ，同时站在用户的立场上安排了调用 consume_each 。

  ```
  /*!
  * \brief Just like gr_block::general_work, only this arranges to
  * call consume_each for you.
  *
  * The user must override work to define the signal processing code
  */
  virtual int work (int noutput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items) = 0;
  ```

  这使得要考虑的因素简化、需要编写的代码也被简化。如果模块所涉及的比率大于 1，在构建函数内调用 set_history 即可，或者做相应次数的变更而已。

  gr_sync_block 也提供处理此“历史性”需求（输入输出比率）的 forecast 版本函数。

- gr_sync_decimator

  gr_sync_decimator 继承于 gr_sync_block 从“历史的角度”来构建 N:1 关系的。

- gr_sync_interpolator

  gr_sync_interpolator 继承于 gr_sync_block 从“历史的角度”来构建 1:N 关系的。

第二个模块：howto_square2_ff

既然已经了解 gr_sync_block， 那么构建 howto_square_ff 不言而喻是应循此子类 gr_sync_block 。

以下的 例程7，“howto_square2_ff.h”、 例程 8,“howto_square2_ff.cc”便是修正版（继承于 gr_sync_block). 随同文件包含额外的测试代码。

例程 7. howto_square2_ff.h

```
/* -*- c++ -*- */
/*
* Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */
#ifndef INCLUDED_HOWTO_SQUARE2_FF_H
#define INCLUDED_HOWTO_SQUARE2_FF_H

#include <gr_sync_block.h>

class howto_square2_ff;

/*
 * 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_square2_ff> howto_square2_ff_sptr;

/*!
 * \brief Return a shared_ptr to a new instance of howto_square2_ff.
 *
 * To avoid accidental use of raw pointers, howto_square2_ff's
 * constructor is private. howto_make_square2_ff is the public
 * interface for creating new instances.
 */
howto_square2_ff_sptr howto_make_square2_ff ();

/*!
 * \brief square2 a stream of floats.
 * \ingroup block
 *
 * This uses the preferred technique: subclassing gr_sync_block.
 */
class howto_square2_ff : public gr_sync_block
{
private:
// The friend declaration allows howto_make_square2_ff to
// access the private constructor.

friend howto_square2_ff_sptr howto_make_square2_ff ();

howto_square2_ff (); // private constructor

public:
~howto_square2_ff (); // public destructor

// Where all the action really happens

int work (int noutput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items);
};

#endif /* INCLUDED_HOWTO_SQUARE2_FF_H */
```

例程 8. howto_square2_ff.cc

```
/* -*- c++ -*- */
/*
* Copyright 2004 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 2, 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., 59 Temple Place - Suite 330,
* Boston, MA 02111-1307, USA.
*/

/*
* config.h is generated by configure. It contains the results
* of probing for features, options etc. It should be the first
* file included in your .cc file.
*/
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <howto_square2_ff.h>
#include <gr_io_signature.h>

/*
```

```
* Create a new instance of howto_square2_ff and return
* a boost shared_ptr. This is effectively the public constructor.
*/
howto_square2_ff_sptr
howto_make_square2_ff ()
{
return howto_square2_ff_sptr (new howto_square2_ff ());
}

/*
* Specify constraints on number of input and output streams.
* This info is used to construct the input and output signatures
* (2nd & 3rd args to gr_block's constructor). The input and
* output signatures are used by the runtime system to
* check that a valid number and type of inputs and outputs
* are connected to this block. In this case, we accept
* only 1 input and 1 output.
*/
static const int MIN_IN = 1; // mininum number of input streams
static const int MAX_IN = 1; // maximum number of input streams
static const int MIN_OUT = 1; // minimum number of output streams
static const int MAX_OUT = 1; // maximum number of output streams

/*
* The private constructor
*/
howto_square2_ff::howto_square2_ff ()
: gr_sync_block ("square2_ff",
gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
// nothing else required in this example
}

/*
* Our virtual destructor.
*/
howto_square2_ff::~howto_square2_ff ()
{
// nothing else required in this example
}

int
howto_square2_ff::work (int noutput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items)
{
const float *in = (const float *) input_items[0];
float *out = (float *) output_items[0];

for (int i = 0; i < noutput_items; i++){
out[i] = in[i] * in[i];
}

// Tell runtime system how many output items we produced.
return noutput_items;
}
```

从这儿迈向何方？ - Where to from Here?

At this point, we've got a basic overview of how the system goes together. For more insight, I suggest that you look at the code of the system. The doxygen generated class hierarchy is a useful way to find things that might interest you.

杂类点津 - Miscellaneous Tips

- "源" 与 "漏" - Sources and Sinks

  "源" - sources 与 "漏" - sinks 是源自于 gr_sync_block 。它们的唯一区别是："源" - sources 没有输入；"漏" - sinks 没有输出。它体现在 gr_io_signatures 中并继承于 gr_sync_block 的构建函数中。参阅 gr_file_source.{h,cc} 和 gr_file_sink.{h,cc} 便对此建立直观的感觉。

- 用 gdb 诊断 - Debugging with gdb

  如果代码没有预期一样地工作，使用 Python test 或者在代码中添加几个 printfs 的方法也无能为力，也许只能求助于 gdb 来诊断错误了。 一点稍许的麻烦是 GNU Radio 所有的模块，也包括用户的新模块是被动态的调入 Python 来工作。

  如下尝试一下： 在用户的 Python 代码，在相关的 imports 之后，打印显示过程 ID 并暂停等待回车继续。在另一个窗口运行 GDB 并将此过程 ID 添加到 python 应用之中。在此可以设置断点或一些额外有助诊断的代码(用户代码). 然后返回 Python 窗口，键击回车来继续应用。

  ```
  #!/usr/bin/env python
  from gnuradio import gr
  from gnuradio import my_buggy_module

  # insert this in your test code...
  import os
  print 'Blocked waiting for GDB attach (pid = %d)' %
  (os.getpid(),)
  raw_input ('Press Enter to continue: ')
  # remainder of your test code follows...
  ```

  另外一个可能遇到的麻烦是 gdb 6.2 无法在由 g++ 3.4 生成的构建函数或（垃圾）清理器 - destructors 中设置断点。果真如此，在构建函数中调用 gri_debugger_hook （空）表达式并重新编译。重载代码并在 gri_debugger_hook 中设置断点。

- 用 oprofile 性能监控和评测 - Performance Measurement with oprofile

  Oprofile 是个不错的朋友，请参阅：http://oprofile.sourceforge.net.

后续亮点 - Coming Attractions

- 改良的 type - Improved Type System
- 阶层构架的模块 - Hierarchical Blocks

注：How to Write a Signal Processing Block （原文出处，翻译整理仅供参考!）