CENTRE TECNOLÒGIC DE TELECOMUNICACIONS DE CATALUNYA

Notes on raw data ingestion in GNSS-SDR

Carles Fernández-Prades

April 27, 2015

CONTENTS

1 INTRODUCTION

A GNSS software receiver is a complex system, which description needs to be addressed at different abstraction layers. Hereafter, we discuss some aspects of the software architecture implemented in GNSS-SDR, that is based on GNU Radio[1], a well-established framework that provides the signal processing runtime and processing blocks to implement software radio applications. Frameworks are a special case of software libraries – they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries: the overall program's flow of control is not dictated by the caller, but by the framework; and it can be extended by the user usually by selective overriding or specialized by user code providing specific functionality. Software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time. GNSS-SDR proposes a software architecture that builds upon the GNU Radio framework in order to implement GNSS software defined receivers.

1.1 TOP-BOTTOM OVERVIEW

The view from ten thousand meters of GNSS-SDR's architecture is as follows:

- **The Control Plane** is in charge of creating a *flow graph* in which a sample stream goes through a network of connected signal processing blocks up to the position fix. The nature of a GNSS receiver imposes some requirements in the architecture design: since the composition of the received GNSS signals will change over time (initially, some satellites will be visible, and after a while, some satellites will not be visible anymore and new ones will show up), some channels will lose track of their signals and some new channels will have to be instantiated to process the new signals. This means that the receiver must be able to activate and deactivate the channels dynamically, and it also needs to detect these changes during runtime.
- **The Signal Processing Plane**, consisting of a collection of blocks that actually implement digital signal processing algorithms. Efficiency is specially critical before and during correlations (the most complex operation in terms of processing load, but from which sample rate decreases three orders of magnitude), and even a modern multi-purpose processor must be properly programmed in order to attain real-time.

Watching it more closely, GNSS-SDR is a C++ program. When running, it reads samples from a *Signal Source* (an abstract concept that can represent a radio frequency front-end, or a file, or a combination of them) of GNSS signals, and performs all the signal processing up to the computation of a position fix.

As in any C++ program, the `main` method is called at program start up, after initialization of the non-local objects with static storage duration. It is the designated entry point to a program

[1] See `http://gnuradio.org`

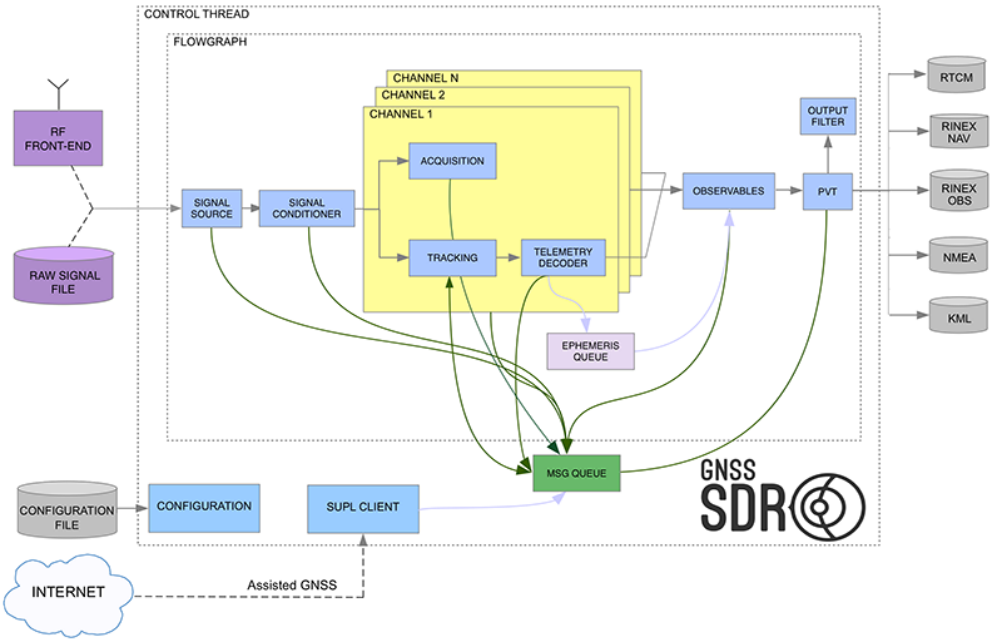

Figure 1.1: GNSS-SDR's general block diagram. Different implementations can be defined for each of the processing blocks (blue boxes).

that is executed in a hosted environment (that is, with an operating system). GNSS-SDR's `main` method processes the command line flags, if any, provided by the user, and initializes the logging library. Then, it records the starting time and instantiates a ControlThread object. Its constructor reads the configuration file, creates a control queue and creates a flow graph according to the configuration. Then, the program's main method calls the `run()` method of the instantiated object, an action that connects the flow graph and starts running it. After that, and until a `stop` message generated by some processing block is received, the control queue reads control messages sent by the receiver's modules through a safe-thread queue and processes them. The flow graph continually processes the sample stream, moving samples from the output of one block to the input of the following one(s), performing all the signal processing up to the computation of Position, Velocity and Time. Finally, when a `stop` message is actually received, the main method executes the destructor of the ControlThread object, which deallocates memory, does other cleanup and exits the program.

1.2 The configuration mechanism

The Control Plane is in charge of creating a flow graph according to the configuration and then managing the modules. Configuration allows users to define in an easy way their own custom receiver by specifying the flow graph (type of signal source, number of channels, algorithms to be used for each channel and each module, strategies for satellite selection, type of output format, etc.) in a single configuration file. Since it is difficult to foresee what future module

implementations will be needed in terms of configuration, we used a very simple approach that can be extended without a major impact in the code. This can be achieved by simply mapping the names of the variables in the modules with the names of the parameters in the configuration.

Properties are passed around within the program using the ConfigurationInterface class. There are two implementations of this interface: FileConfiguration and InMemoryConfiguration. FileConfiguration reads the properties (pairs of property name and value) from a file and stores them internally. InMemoryConfiguration does not read from a file; it remains empty after instantiation and property values and names are set using the set_property method. FileConfiguration is intended to be used in the actual GNSS-SDR application whereas InMemoryConfiguration is intended to be used in tests to avoid file-dependency in the file system. Classes that need to read configuration parameters will receive instances of ConfigurationInterface from where they will fetch the values. For instance, parameters related to *SignalSource* should look like this:

```
SignalSource.parameter1=value1
SignalSource.parameter2=value2
```

Listing 1: Configuration file syntax: setting parameters of the SignalSource module.

The name of these parameters can be anything but one reserved word: *implementation*. This parameter indicates in its value the name of the class that has to be instantiated by the factory for that role. For instance, if we want to use the implementation *DirectResampler* for module *SignalConditioner*, the corresponding line in the configuration file would be

```
SignalConditioner.implementation=Pass_Through
```

Listing 2: Configuration file syntax: setting the implementation of the SignalConditioner module.

Since the configuration is just a set of property names and values without any meaning or syntax, the system is very versatile and easily extendable. Adding new properties to the system only implies modifications in the classes that will make use of these properties. In addition, the configuration files are not checked against any strict syntax so it is always in a correct status (as long as it contains pairs of property names and values in INI format[2]).

This configuration system allows the full specification of a GNSS receiver from a single configuration file. You can find examples of those files at the `conf/` folder of the source tree. Then, users can run its own receiver by typing in a terminal:

```
$ gnss-sdr --config_file=/path/to/my_receiver.conf
```

[2] An INI file is an 8-bit text file in which every property has a name and a value, in the form `name = value`. Properties are case-insensitive, and cannot contain spacing characters. Semicolons (`;`) indicate the start of a comment; everything between the semicolon and the end of the line is ignored.

2 UNDERSTANDING DATA TYPES

The input of a software receiver are the raw bits that come out from the front-end's analog-to-digital converter (ADC), as sketched in Figure 2.1. Those bits can be read from a file stored in the hard disk or directly in real-time from a hardware device through USB or Ethernet buses.

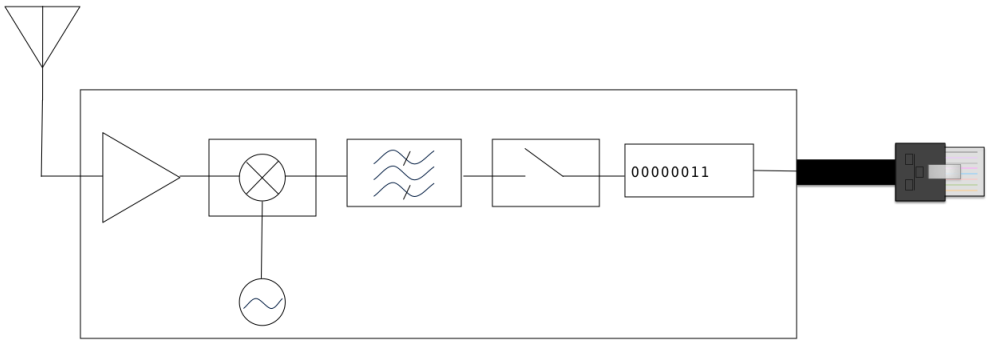

Figure 2.1: Simplified lock diagram of a generic radio frequency front-end, consisting of an antenna, an amplification stage, downshifting from RF to an intermediate frequency, filtering, sampling, and an interface to a host computer for real-time processing mode, or to an storage device for post-processing.

GNSS-SDR is designed to work with a wide range of radiofrequency front-ends, each one with its own parameters of sampling frequency, number of bits per sample, signal format (baseband or passband), etc. When the sample stream enters the host computer, there is a first processing block, called *Signal Conditioner* (see Section 3), which is in charge of accommodate the sample stream in a format tractable by a computer. The containers of data in a computer system are called data types.

2.1 DATA TYPE DEFINITION

A *type* is a set of possible values which an object, reference, function or expression can possess, and it is defined as its representation and a set of operators manipulating these representations. The type is a property which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

The C++ type system defines the following fundamental types:

- The type `void` specifies that the function does not return a value.
- The null-pointer type `std::nullptr_t`
- Arithmetic types
  - Floating-point types (`float`, `double`, `long double`)
  - Integral types
    - The type `bool`

* Character types
  - Narrow character types (`char`, `signed char`, `unsigned char`)
  - Wide character types (`char16_t`, `char32_t`, `wchar_t`)
* Signed integer types (`short`, `int`, `long`, `long long`)
* Unsigned integer types (`unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`)

Group	**Type names**	**Notes on size / precision**
Void type	void	no storage
Null pointer	decltype(nullptr)	
Boolean type	bool	
Character types	char	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char. At least 16 bits.
	char32_t	Not smaller than char16_t. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	signed char	Same size as char. At least 8 bits.
	signed short *int*	Not smaller than char. At least 16 bits.
	signed int	Not smaller than short. At least 16 bits.
	signed long *int*	Not smaller than int. At least 32 bits.
	signed long long *int*	Not smaller than long. At least 64 bits.
Integer types (unsigned)	unsigned char	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	float	
	double	Precision not less than float
	long double	Precision not less than double

Table 2.1: List of fundamental types in C++. Within each of the groups, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties. The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., `signed short int` can be abbreviated as `signed short`, `short int`, or simply `short`; they all identify the same fundamental type.

Table 2.1 shows a list fundamental types in C++. Note that other than `char` (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run.

In C++, the `typedef` keyword allows the programmer to create new names for types, as shown in Listing 3.

```
// simple typedef
typedef unsigned long ulong;

// the following two objects have the same type
unsigned long l1;
ulong l2;
```

Listing 3: Example of using `typedef`

2.2 Data types in GNSS-SDR

In the C and C++ programming languages, `stdint.h` is the name of the header file that allows programmers to write more portable code by providing a set of typedefs that specify exact-width integer types, together with the defined minimum and maximum allowable values for each type. This header is particularly useful for embedded programming which often involves considerable manipulation of hardware specific I/O registers requiring integer data of fixed widths, specific locations and exact alignments. The naming convention for exact-width integer types is `intN_t` for `signed int` and `uintN_t` for `unsigned int`. Among others, `stdint.h` defines the following typedefs:

- `int8_t` Signed integer type with a width of *exactly* 8 bits.
- `int16_t` Signed integer type with a width of *exactly* 16 bits.
- `int32_t` Signed integer type with a width of *exactly* 32 bits.

Building upon these definitions, the Vector-Optimized Library of Kernels (VOLK) library[3] defines complex data types. As shown in Listing 4, it loads the header `complex.h`, a file that defines functionality for complex arithmetic (i.e. basic, arithmetic, trigonometric and hyperbolic operations[4], but only for floating-point data types: `float`, `double` and `long double`. This means that complex operations are not defined for integer data types, and for instance the instantiation of an object of type `std::complex<int8>` has undefined behavior. The VOLK library provides definitions for those data types that are missing in C++ in a portable manner.

```
...
#include <complex>
#include <stdint.h>

typedef std::complex<int8_t>  lv_8sc_t;
typedef std::complex<int16_t> lv_16sc_t;
typedef std::complex<int32_t> lv_32sc_t;
typedef std::complex<int64_t> lv_64sc_t;
typedef std::complex<float>   lv_32fc_t;
typedef std::complex<double>  lv_64fc_t;
```

[3] See `http://libvolk.org`

[4] See `http://en.wikibooks.org/wiki/C_Programming/C_Reference/complex.h`

```
template <typename T> inline std::complex<T> lv_cmake(const T &r, const T &i){
    return std::complex<T>(r, i);
}
...
```

Listing 4: Part of `include/volk/volk_complex.h`

As shown in the typedefs of Listing 4, VOLK defines type names for objects holding complex numbers in which their real and imaginary parts are integers of exactly 8, 16, 32 or bits, or floating point numbers of 32 or 64 bits. It also provides a template constructor for them. VOLK is also instrumental in squeezing the processor capabilities by providing with an interface to use Single Input - Multiple Data (SIMD) instructions, which are of special interest for operations that are in the receiver's critical path of processing load. Processors providing SIMD instruction sets compute with multiple processing elements that perform the same operation on multiple data points simultaneously, thus exploiting data-level parallelism, an can be found in most modern desktop and laptop personal computers. In a nutshell, VOLK implements in assembly language optimized versions of computationally-intensive operations for different processor architectures that are commonly found in modern computers. In order to use the most optimized version for the specific processor(s) of the host machine running the software receiver (that is, the implementation than runs the fastest), VOLK provides `volk_profile`, a program that tests all known VOLK kernels (that is, basic processing components like adders, multipliers, correlators, and much more) for each architecture supported by the host machine, measuring their performance. When finished, the profiler writes to `$HOME/.volk/volk_config` the best architecture for each VOLK function. This file is read when using a function to know the best version to execute. In this way, portability is always ensured, since VOLK always provide a generic C implementation that is executed when no SIMD instructions are available at the host machine executing the software receiver, but takes advantage of those instructions when they are actually present. By using VOLK, GNSS-SDR ensures that it will produce optimized executables for a wide range of processors and low-level instruction sets.

Internally, GNSS-SDR makes use of the complex data types defined by VOLK. They are fundamental for handling sample streams in which samples are complex numbers with real and imaginary components of 8, 16 or 32 bits, common formats delivered by GNSS radio frequency front-ends. Table 2.2 shows the data type names that GNSS-SDR exposes through the configuration file.

Type name in conf file	Definition	Sample stream
`byte`	Signed integer, 8-bit two's complement number ranging from -128 to 127. C++ type name: `int8_t`	$[S_0], [S_1], [S_2], ...$
`short`	Signed integer, 16-bit two's complement number ranging from -32768 to 32767 C++ type name: `int16_t`	$[S_0], [S_1], [S_2], ...$
`float`	Defines numbers with fractional parts, can represent values ranging from approx. 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits (32 bits). C++ type name: `float`	$[S_0], [S_1], [S_2], ...$
`ibyte`	Interleaved (I&Q) stream of samples of type `byte` C++ type name: `int8_t`	$[S_0^I], [S_0^Q], [S_1^I], [S_1^Q], [S_2^I], [S_2^Q], ...$
`ishort`	Interleaved (I&Q) samples of type `short` C++ type name: `int16_t`	$[S_0^I], [S_0^Q], [S_1^I], [S_1^Q], [S_2^I], [S_2^Q], ...$
`cbyte`	Complex samples, with real and imaginary parts of type `byte` C++ type name: `lv_8sc_t`	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], [S_2^I + jS_2^Q], ...$
`cshort`	Complex samples, with real and imaginary parts of type `short` C++ type name: `lv_16sc_t`	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], [S_2^I + jS_2^Q], ...$
`gr_complex`	Complex samples, with real and imaginary parts of type `float` C++ type name: `std::complex<float>`	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], [S_2^I + jS_2^Q], ...$

Table 2.2: Data type names used in GNSS-SDR configuration files.

3 SIGNAL CONDITIONER

A *Signal Conditioner* block is in charge of adapting the sample bit depth to a data type tractable at the host computer running the software receiver, and optionally intermediate frequency to baseband conversion, resampling, and filtering. Regardless the selected signal source features, the Signal Conditioner interface delivers in a unified format a sample data stream to the receiver downstream processing channels, acting as a facade between the signal source and the synchronization channels, providing a simplified interface to the input signal at a reference, *internal* sample rate f_{IN}. This signal stream feeds a set of parallel *Channels*.

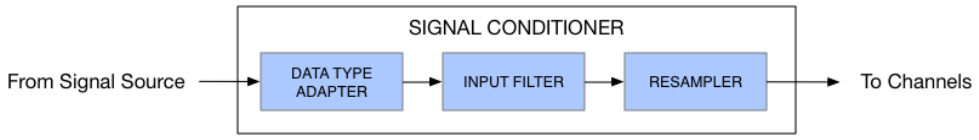

Figure 3.1: The Signal Conditioner is a hierarchical block consisting of a Data Type Adapter, a Filter and a Resampler.

Listing 5 provides an example of Signal Conditioner configuration, and Table 3.1 summarizes all the possible configuration options for the Signal Conditioner block.

```
; ...

;######### SIGNAL_CONDITIONER CONFIG ############
;## It holds blocks to change data type, filter and resample input data.
SignalConditioner.implementation=Signal_Conditioner

;######### DATA_TYPE_ADAPTER CONFIG ############
;## Changes the type of input data.
;#implementation: [Pass_Through] disables this block
DataTypeAdapter.implementation=Ishort_To_Complex

;######### INPUT_FILTER CONFIG ############
;## Filter the input data.
InputFilter.implementation=Fir_Filter
InputFilter.input_item_type=gr_complex
InputFilter.output_item_type=gr_complex
; ... other parameters

;######### RESAMPLER CONFIG ############
;## Resamples the input data.
Resampler.implementation=Pass_Through
Resampler.item_type=gr_complex
Resampler.sample_freq_in=8000000
Resampler.sample_freq_out=4000000

; ...
```

Listing 5: Example of Signal Conditioner configuration

	Data type adapter			Filter			Resampler
	Input type	**Implementation**	**Output type**	**Input type**	**Impl**	**Output type**	**type**
Passband	byte	Pass_Through	byte	byte	Freq_Xlating_Fir_Filter	cbyte	cbyte
	byte	Pass_Through	byte	byte		gr_complex	gr_complex
	byte	Byte_To_Short	short	short		cshort	cshort
	short	Pass_Through	short	short		cshort	cshort
	short	Pass_Through	short	short		gr_complex	gr_complex
	float	Pass_Through	float	float		gr_complex	gr_complex
	gr_complex	Pass_Through	gr_complex	gr_complex		gr_complex	gr_complex
Baseband	ibyte	Ibyte_To_Cbyte	cbyte	cbyte	Fir_Filter	cbyte	cbyte
	ibyte	Ibyte_To_Complex	gr_complex	gr_complex		gr_complex	gr_complex
	ishort	Ishort_To_Cshort	cshort	cshort		cshort	cshort
	ishort	Ishort_To_Cshort	cshort	cshort		gr_complex	gr_complex
	ishort	Ishort_To_Complex	gr_complex	gr_complex		gr_complex	gr_complex
	cshort	Pass_Through	cshort	cshort		cshort	cshort
	cshort	Pass_Through	cshort	cshort		gr_complex	gr_complex
	gr_complex	Pass_Through	gr_complex	gr_complex		gr_complex	gr_complex
	ibyte	Ibyte_To_Cbyte	cbyte	cbyte	Pass_Through	cbyte	cbyte
	ibyte	Ibyte_To_Complex	gr_complex	gr_complex		gr_complex	gr_complex
	ishort	Ishort_To_Cshort	cshort	cshort		cshort	cshort
	ishort	Ishort_To_Cshort	cshort	cshort		gr_complex	gr_complex
	gr_complex	Pass_Through	gr_complex	gr_complex		gr_complex	gr_complex

Table 3.1: Configuration options for Signal Conditioner.

3.1 Data Type Adapter

This block is in charge of changing the data type of the sample stream. This is the first processing block after the Signal Source, and each king of source can deliver data in different formats.

- Signal Source is delivering samples at a given intermediate frequency:
  - Real samples: `byte`, `short`, `float` (8, 16 and 32 bits, respectively).
  - Intervealed (I&Q) samples: `ibyte`, `ishort`, `gr_complex`
- Signal Source is delivering samples at baseband:
  - Intervealed (I&Q) samples: `ibyte`, `ishort`, `gr_complex`
  - Complex samples: `cbyte`, `cshort`, `gr_complex`

3.1.1 Implementation: Byte_To_Short

This implementation takes samples of type `byte` (8 bits, real samples) at its input and writes samples of type `short` (16 bits, real samples) at its output.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Byte_To_Short
```

Listing 6: Example of Data Type Adapter configuration with implementation `Byte_To_Short`

3.1.2 Implementation: Ibyte_To_Cbyte

This implementation takes samples of type `ibyte` (interleaved I&Q samples, 8 bits each) at its input and writes samples of type `cbyte` (complex samples with real and imaginary components of 8 bits each) at its output. This reduces the sample rate by two.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Ibyte_To_Cbyte
```

Listing 7: Example of Data Type Adapter configuration with implementation `Ibyte_To_Cbyte`

3.1.3 Implementation: Ibyte_To_Complex

This implementation takes samples of type `ibyte` (interleaved I&Q samples, 8 bits each) at its input and writes samples of type `gr_complex` (complex samples with real and imaginary components of 32 bits each) at its output. This reduces the sample rate by two.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Ibyte_To_Complex
```

Listing 8: Example of Data Type Adapter configuration with implementation `Ibyte_To_Complex`

3.1.4 Implementation: Ishort_To_Cshort

This implementation takes samples of type `ishort` (interleaved I&Q samples, 16 bits each) at its input and writes samples of type `cshort` (complex samples with real and imaginary components of 16 bits each) at its output. This reduces the sample rate by two.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Ishort_To_Cshort
```

Listing 9: Example of Data Type Adapter configuration with implementation `Ishort_To_Cshort`

3.1.5 Implementation: Ishort_To_Complex

This implementation takes samples of type `ishort` (interleaved I&Q samples, 16 bits each) at its input and writes samples of type `gr_complex` (complex samples with real and imaginary components of 32 bits each) at its output. This reduces the sample rate by two.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Ishort_To_Complex
```

Listing 10: Example of Data Type Adapter configuration with implementation `Ishort_To_Complex`

3.1.6 Implementation: Pass_Through

This implementation copies samples from its input to its output.

```
;######### DATA_TYPE_ADAPTER CONFIG #############
DataTypeAdapter.implementation=Pass_Through
```

Listing 11: Example of Data Type Adapter configuration with implementation `Pass_Through`

3.2 Input Filter

This block is in charge of filtering the incoming signal.

3.2.1 Implementation: Fir_Filter

This implementation, based on the Parks-McClellan algorithm, computes the optimal (in the Chebyshev/minimax sense) FIR filter impulse response given a set of band edges, the desired response on those bands, and the weight given to the error in those bands. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses.

Parameters:

- `dump` [`false`, `true`]: Flag for storing the signal at the filter output in a file. It defaults to `false`.
- `dump_filename`: If `dump` is set to true, path to the file where data will be stored.
- `input_item_type` [`cbyte`, `cshort`, `gr_complex`]: Input data type. This implementation only accepts streams of complex data types.
- `output_item_type` [`cbyte`, `cshort`, `gr_complex`]: Output data type. You can use this implementation to upcast the data type (i.e., from `cbyte` to `gr_complex` and from `cshort` to `gr_complex`).
- `taps_item_type` [`float`]: Type and resolution for the taps of the filter. Only `float` is allowed in the current version.
- `number_of_taps`: Number of taps in the filter. Increasing this parameter increases the processing time.
- `number_of_bands`: Number of frequency bands in the filter.
- `band1_begin`: Frequency at the band edges [**b1** e1 b2 e2 b3 e3 ...]. Frequency is in the range [0, 1], with 1 being the Nyquist frequency ($\frac{F_s}{2}$). The number of `band_begin` and `band_end` elements must match the number of bands. See Figure 3.2 for an example of filter design.
- `band1_end`: Frequency at the band edges [b1 **e1** b2 e2 b3 e3 ...]
- `band2_begin`: Frequency at the band edges [b1 e1 **b2** e2 b3 e3 ...]
- `band2_end`: Frequency at the band edges [b1 e1 b2 e2 **b3** e3 ...]
- `ampl1_begin`: Desired amplitude at the band edges [**a(b1)** a(e1) a(b2) a(e2) ...]. The number of `ampl_begin` and `ampl_end` elements must match the number of bands.
- `ampl1_end`: Desired amplitude at the band edges [a(b1) **a(e1)** a(b2) a(e2) ...].
- `ampl2_begin`: Desired amplitude at the band edges [a(b1) a(e1) **a(b2)** a(e2) ...].

- `ampl2_end`: Desired amplitude at the band edges [a(b1) a(e1) a(b2) **a(e2)** ...].
- `band1_error`: Weighting applied to band 1 (usually 1).
- `band2_error`: Weighting applied to band 2 (usually 1).
- `filter_type` [`bandpass`, `hilbert`, `differentiator`]: type of filter to be used.
  - `passband`: designs a FIR filter, using the weights `band1_error`, `band2_error`, etc. to weight the fit in each frequency band.
  - `hilbert`: designs linear-phase filters with odd symmetry. This class of filters has a desired amplitude of 1 across the entire band.
  - `differentiator`]: For nonzero amplitude bands, it weights the error by a factor of 1/f so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).
- `grid_density`: Determines how accurately the filter will be constructed. The minimum value is 16; higher values makes the filter slower to compute, but often results in filters that more exactly match an equiripple filter.

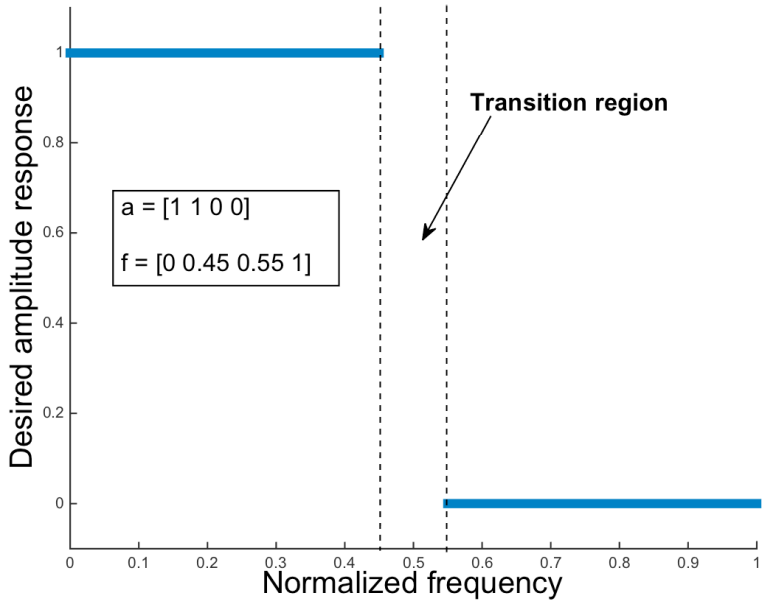

Figure 3.2: The relationship between f = [`band1_begin band1_end band2_begin band2_end`] and a = [`ampl1_begin ampl1_end ampl2_begin ampl2_end`] vectors in defining a desired frequency response for the Input Filter.

If you have access to MATLAB, you can plot easily the frequency response of the filter:

```
f = [0 0.45 0.55 1];
a = [1 1 0 0];
b = firpm(5, f, a);
[h, w] = freqz(b, 1, 512);
plot(f, a, w/pi, abs(h))
legend('Ideal', 'Filter design')
xlabel 'Radian Frequency (\omega/\pi)', ylabel 'Magnitude'
```

Listing 12: MATLAB code for plotting the filter frequency response (5 taps).

```
;######### INPUT_FILTER CONFIG #############
InputFilter.implementation=Fir_Filter

InputFilter.dump=false
InputFilter.dump_filename=../data/input_filter.dat

InputFilter.input_item_type=cbyte
InputFilter.output_item_type=gr_complex
InputFilter.taps_item_type=float
InputFilter.number_of_taps=5

InputFilter.number_of_bands=2

InputFilter.band1_begin=0.0
InputFilter.band1_end=0.45
InputFilter.band2_begin=0.55
InputFilter.band2_end=1.0

InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0

InputFilter.band1_error=1.0
InputFilter.band2_error=1.0

InputFilter.filter_type=bandpass
InputFilter.grid_density=16
```

Listing 13: Example of Input Filter configuration with implementation `Fir_Filter`

3.2.2 Implementation: Freq_Xlating_Fir_Filter

This implementation features a frequency-translating FIR filter. This is often used when input data is art an intermediate frequency, as it performs frequency translation, filtering and decimation in one step. The basic principle of this block is to perform:

$$\text{Input signal} \rightarrow \text{BPF} \rightarrow \text{decim} \rightarrow (\text{mult by } 2\pi\frac{f_{IF}}{f_s}\text{*decim}) \rightarrow \text{Output signal.}$$

The BPF is the baseband filter (LPF) moved up to the center frequency $2\pi\frac{f_{IF}}{f_s}$. The block then applies a derotator with $-2\pi\frac{f_{IF}}{f_s}$ to downshift the signal to baseband.

This implementation accepts the same parameters as `Fir_Filter`, with the following differences:

- `input_item_type` [`byte`, `short`, `float`, `gr_complex`]: This implementation accepts as input data type real samples. It also accepts complex samples of the type `gr_complex`, assuming the presence of an intermediate frequency. The filter also works with `IF=0`.
- `IF`: Specifies the intermediate frequency f_{IF}, in Hz.
- `sampling_frequency`: Specifies the sample rate f_s, in samples per second.
- `decimation_factor`: Decimation factor (defaults to 1).

```
;######### INPUT_FILTER CONFIG ############
InputFilter.implementation=Freq_Xlating_Fir_Filter

InputFilter.dump=false
InputFilter.dump_filename=../data/input_filter.dat

InputFilter.input_item_type=byte
InputFilter.output_item_type=gr_complex
InputFilter.taps_item_type=float

InputFilter.number_of_taps=5
InputFilter.number_of_bands=2

InputFilter.band1_begin=0.0
InputFilter.band1_end=0.45
InputFilter.band2_begin=0.55
InputFilter.band2_end=1.0

InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0

InputFilter.band1_error=1.0
InputFilter.band2_error=1.0
InputFilter.filter_type=bandpass
InputFilter.grid_density=16
```

```
InputFilter.IF=2000000
InputFilter.sampling_frequency=8000000
```

Listing 14: Example of Input Filter configuration with implementation `Freq_Xlating_Fir_Filter`

3.2.3 IMPLEMENTATION: PASS_THROUGH

This implementation copies samples from its input to its output, without performing any filtering.

```
;######### INPUT_FILTER CONFIG ############
InputFilter.implementation=Pass_Through
```

Listing 15: Example of Input Filter configuration with implementation `Pass_Through`

3.3 Resampler

This block is in charge of resampling the signal and delivering it to the N parallel processing channels. At the Resampler's output, only complex types are allowed: `cbyte`, `cshort`, or `gr_complex`. This block does not perform any data type conversion.

3.3.1 Implementation: Direct_Resampler

This implementation performs a direct resampling of the incoming signal, without performing any interpolation.

Parameters:

- `sample_freq_in`: Sample rate at the Resampler's input, in samples per second.
- `sample_freq_out`: Sample rate at the Resampler's output, in samples per second.
- `item_type` [`cbyte`, `cshort`, `gr_complex`]: Data type to be resampled. It defaults to `gr_complex`.

```
;######### RESAMPLER CONFIG ############
Resampler.implementation=Direct_Resampler
Resampler.sample_freq_in=8000000
Resampler.sample_freq_out=4000000
Resampler.item_type=gr_complex
```

Listing 16: Example of Resampler configuration with implementation `Direct_Resampler`

3.3.2 Implementation: Pass_Through

This implementation copies samples from its input to its output.

Parameters:

- `item_type` [`cbyte`, `cshort`, `gr_complex`]: Data type to be copied from the input to the output of this block. It defaults to `gr_complex`.

```
;######### RESAMPLER CONFIG ############
Resampler.implementation=Pass_Through
Resampler.item_type=gr_complex
```

Listing 17: Example of Resampler configuration with implementation `Pass_Through`