# AbstractSDRs: Bring down the two-language barrier with Julia Language for efficient SDR prototyping

Corentin Lavaud, Robin Gerzaguet, Matthieu Gautier, Olivier Berder

# AbstractSDRs: Bring down the two-language barrier with Julia Language for efficient SDR prototyping

Corentin LAVAUD*, Robin GERZAGUET*, Matthieu GAUTIER*, Olivier BERDER*.
* Univ Rennes, CNRS, IRISA, surname.name@irisa.fr

***Abstract*—This paper proposes a new methodology based on the recently proposed Julia language for efficient Software Defined Radio (SDR) prototyping. SDRs are immensely popular as they allow to have a flexible approach for sounding, monitoring or processing radio signals through the use of generic analog components and lot of digital signal processing. As, in this paradigm, most of the processing is done at software levels (i.e. on a CPU), an efficient software methodology has to be envisioned. Right now, most of the existing methods focus on low-level languages (C or C++) for good runtime performance (at the cost of easy prototyping) or high-level language (such as Python) for flexibility (at the price of runtime performance). In this article we propose a new methodology based on Julia language that addresses this *two-language* problem and paves the way for efficient prototyping without giving up runtime performance. To prove the benefits of the proposed approach, a performance benchmark with several optimisation levels compares the Julia approach with C++ and Python.**

***Index Terms*—Software Defined Radio, Julia language, fast prototyping, benchmark**

## I. Introduction

It has been almost thirty years that the first definition of a Software Defined Radio (SDR) has been proposed [1]. The ideal paradigm of the SDR were introduced: every processing part of the radio defined in digital domain with the exception of the antenna and the analog-to-digital converters. Mitola stressed the need of specific prototyping methodology, generic hardware architectures and efficient software means. Through years of research and thorough analysis, the myth of a pure digital architecture fizzled out and analog and hardware parts are still a major area of investigations joined together with software methodologies [2].

To bridge the gap between hardware and software, many different SDR architectures have been proposed: some based on General Purpose Processor (GPP), Digital Signal Processor (DSP) or on specific hardware such as Field Programmable Gate Array (FPGA) [3]. More recently the advances in hardware integration allow to have embedded SDR architectures based on System on Chip (SoC) combining software and hardware computational resources. This enlarges the scope of embedded SDR applications as now they are capable to capture modern wireless standards [4], exhibit cybersecurity vulnerabilities [5] or apply specific real-time processes to signals [6]. In any case, SDR has proved to be a precious tool for prototyping (due to its reasonable cost and simple configuration) and has also strong assets for pedagogy [7].

The question of its programmability is still open. On one hand the general purpose processors embed more and more computational capacity making a pure GPP-based approach performant. On the other hand, hardware accelerators and co-processors are still necessary when massive bandwidths and harsh real-time constraints must be fulfilled. Some of the modern SDR programming approaches propose to reconfigure also the hardware parts through bindings or generic glue interface [8]. Regarding programming semantic, low-level languages (e.g. C++/C, Rust) offers very good runtime performance but does not offer a good prototyping experience. High-level languages (e.g. Python, Matlab) offer the desired flexibility at the price of runtime performance. In practice, it means that using SDR for efficient prototyping is often done in two phases: first an exploration in a high-level language and secondly an optimisation in a low-level language. This is called the *two-language problem*. Note that it is sometimes possible to encapsulate the optimized algorithms (at low-level) into the high-level language but not without strong efforts and non-negligible re-programming [9].

This is where Julia language [10] comes into play. Julia is a programming language whose syntax is really close to scripting languages but that offers really good performance for various platforms as it is compiled through LLVM [11]. The Just In Time (JIT) compilation and the Multiple Dispatch (MD) feature bridge the gap between on one side interactivity and code concision (required for easy exploration and prototyping) and on the other side efficient compiled code (for good runtime performance). To the best of the authors knowledge, only few unmaintained and incomplete works (such as *SoapySDR.jl* and *liquidSDR.jl*) are dedicated to SDR prototyping in Julia language. We aim to fill this gap by proposing *AbstractSDRs.jl*, a package to monitor, control and interact with many different SDRs.

In this paper we introduce the key features that makes Julia appealing for efficient prototyping with SDR. We introduce the proposed ecosystem (e.g. *AbstractSDRs.jl*) and we compare the performance results with a low-level language (C++) and a high-level language (Python). To better stress the prototyping capacity we also expose a minimal Julia code that instantiates a radio and does some processing in a very concise manner. We demonstrate the benefit of the proposed approach with various levels of optimisation, keeping a simple syntax but approaching the benchmark rate obtained with a low-level language.

The paper is divided into four main parts. In Section II we introduce the main useful characteristics of Julia language and why it is suitable for efficient SDR prototyping. We present the proposed SDR ecosystem in Julia in Section III and exposes the benefit of the code concision. In section IV we assess the performance of the proposed library and compare it other languages. Section V eventually draws some conclusions.

## II. Meaningful pointers on SDR with Julia

Julia is a recent language that allows good performance while keeping a flexible and convenient syntax. The purpose here is not to present the language itself, and the interested readers can have a look at [10] or [12], but to rather give insights on why it is suitable for SDR prototyping.

### A. Multiple dispatch

*1) What it is:* Julia has been built around several key ideas. The main one is to propose a compiled language with multiple dispatch, meaning that a function can be dynamically dispatched based on the runtime type of its argument.

*2) Why it is important:* For performance. MD is the baseline of Julia language and it has been demonstrated that Julia uses it more than other languages [10]. It allows to have high performance while keeping the code execution paths tight and minimal. In our scope, it will also ensure the support of many different SDR bindings trough a common Application Programming Interface (API) while being sure that appropriate function call is done, and this, at runtime. The second key advantage is the possibility to efficiently use fixed point processing [10] which is a often used output ADC format.

### B. Dataflow type inference

*1) What it is:* the typing of code is determined by the flow of data through it. It means that the code is applied to types and not to the associated values. Note that the types can be concrete, and, as it is stated in [10], ability to concretely type code is closely related to being capable to properly execute performance-critical code.

*2) Why it is important:* For performance. Vectorization is not mandatory and we can write efficient loop in Julia. It particularly eases the prototyping as efficient code does not require intensive optimisation. This is particularly an advantage for SDR as we will often have to cope with buffers (i.e. data signals). With the use of this kind of efficient loops, effortless porting of computational intensive processing can be achieved.

### C. Call to low-level languages

*1) What it is:* In Julia, C and Fortran functions can be directly called without any additional glue code. It means that calling low-level C function requires no code generation nor additional compilation.

*2) Why it is important:* For portability. It has two key properties. First, it will ease the use of heavily optimized functions (already written in those languages) as bindings. Well known and established libraries can thus be easily ported to Julia. This is for instance the case of the FFTW library associated to Fast Fourier Transform (FFT) [13]. Secondly, as most of the SDR drivers are written in C language, it will also pave the way for the integration of SDR bindings in Julia without any performance penalty.

### D. Multi-architecture and co-processors support

*1) What it is:* Julia supports different architectures. Regarding GPP, it allows to use Julia on both x86 and ARM. Due to the use of LLVM compilation engine, it is also possible to compute Julia code on GPU leveraging the use of co-processors [14].

*2) Why it is important:* For scalability. SDRs have many different architectures and the same Julia code can be run on these different devices. For instance, some devices are based on ARM Cortex A9 with 32 bits architecture on which the long term support of Julia works. In addition, as Julia is also very effective on parallelisation (with same code enhanced through macros), one can envision to use the same code as for high-level simulations on computation grid as real-time processing on embedded SDR.

### E. Plotting tools and user interfaces

*1) What it is:* Julia language offers easy integration with a strong ecosystem of plots (with various backends such as ones from Python or GR) and the possibility to create Web applications with custom parameters (such as sliders, database exploration...). It means that a custom web app can be easily deployed based on the core calculation (which is computational effective) and a limited number of external packages dedicated to web app porting (namely Interact.jl and Blink.jl).

*2) Why it is important:* For interactivity. Albeit not being as powerful and flexible as Gnuradio with the Gnuradio companion initiative, this kind of integration is a precious tool for prototyping. Indeed, it allows to have efficient and rapid tools for monitoring, exploring and debugging data. On the other hand it also paves the way for simple yet functional demonstrators for dissemination or pedagogical purposes.

To conclude, thanks to several key properties, Julia offers a highly appealing solution for SDR prototyping: high performance without code rewriting, portability through low-level call, scalability with multi-architecture support and interactivity through the large portfolio of plotting tools and web-based frameworks. As there are still no initiative for SDR support we propose in the next section an ecosystem for SDR management in Julia.

## III. Proposed ecosystem

### A. Introducing AbstractSDRs.jl package

We introduce here a common API to monitor several different SDR architectures. Each radio type is managed by its own sub-package and a master package (*AbstractSDRs.jl*) is dedicated to the gathering in a common interface. This kind of nested package architecture guarantees both flexibility (each sub-package can be independently modified) and extendability (other packages can be added afterwards). The package is fully open source and the current version can be found here [15]. In particular, the proposed approach is capable to monitor, configure and transmit/receive samples:

- with Universal Software Radio Peripheral (USRP) from Ettus Research. These SDRs are immensely popular and several radios with various architecture (FPGA-based, ARM-based) and can be monitored through the use of UHD. In the proposed approach, the sub-package *UHD-Bindings* wraps the C API in order to use all functions defined in the low-level interface.
- with Analog Device Active Learning Module Pluto (ADALM-Pluto), an SDR proposed by Analog Device that uses a cortex A9 and the well known AD9361 as the transceiver. The proposed sub-package *AdalmPluto.jl* use some low-level C bindings of the driver supplied by Analog device (IIO library).
- with data exchange between a host computer and a remote computer (sub-package *SDROverNetwork.jl*). This interface requires a Julia session with *AbstractSDRs.jl* and is done through the use of ZeroMQ sockets in order to configure the radio from the host PC and send/receive the samples. This has two key advantages: i) allowing efficient use of SoC-based SDR with x86 or ARM (for instance the case the embed series of the USRP e.g. USRP e310/e320) ii) enforcing the scalability of the proposed approach with tree-based SDR network topology.

- with RTL-SDR dongles. Contrary to the other proposed packages, the *RTLSDR.jl* sub-package has not been proposed by the authors but has been incorporated into the master package.

All these different use-cases are encapsulated in a common API in *AbstractSDRs.jl* which can pave the way for easy switch on different radios for prototyping. It is also to note that extensions to support other SDR boards is quite straightforward thanks to the sub-module encapsulation and MD.

### B. Simple syntax example

To better stress the prototyping through code concision, we present here a simple example written in Julia code that opens a radio, configure it and get samples. We also compute the square modulus of the FFT as the processing unit. The code example is depicted below:

```
using AbstractSDRs  # SDR integration
using FFTW          # FFT support
function main();
    # --- Radio parameter
    sdr = :uhd;     # Targeting USRP
    fc  = 2400e6;   # Carrier Frequency  [Hz]
    bw  = 16e6      # Bandwidth [Hz]
    g   = 30;       # Gain [dB]
    N   = 1024;     # Packet size [Samples]
    # --- Opening radio
    radio = openSDR(sdr,fc,bw,g
                    ;args="addr=192.168.10.13");
    # --- Loop on getting samples and processing
    try
        while(true)
            y = recv(radio,N);  # Get samples
            z = abs2.(fft(y));  # Computation
        end
    catch exception
        # Waiting for <ctrl-c> from user
        @info "Getting interruption";
    end
    # --- Close the radio
    close(radio);
end
```

Several important remarks can be made. The specific SDR targeting is done through the symbol `:sdr`. Then, when instantiating the SDR, `;` is used to add special keyword arguments. These arguments are optional and can be used for specific parameters associated to the SDR (e.g FPGA bitstream path, radio IP address,...). Third, processing part leads to extreme code readibility while ensuring very good runtime performance (see Section IV). Finally, note that the code proposed in example is the more readable but not the more efficient as there are allocations at every loop (in buffer and FFT). Several optimisations can be done when calculating the square modulus as described in next Section.

Finally, the proposed ecosystem has be extended with application oriented packages, related to spectral analysis (with *AbstractSDRsSpectrum.jl*) or FM radio receiver (with *AbstractSDRsFMReceiver.jl*).

## IV. Benchmark and performance assessment

In this section we propose some performance evaluation using the proposed approach and compared to other classical approaches used in the literature.

### A. Benchmark properties

For this we choose to compare the performance offered with the Julia-based *AbstractSDRs.jl* package against C++ and Python. For a fair comparison, the following important statements have to be done:

- Performance is compared in terms of output rate after processing. For every language, the number of samples processed in a given amount of time is counted to deduce the rate (or throughput).
- The proposed approach has been tested with the use of a SDR X310 from Ettus Research as it allows the maximal instantaneous bandwidth of 200 MHz.
- Performance is compared with C++ (low-level high performance language) and Python (high-level language with concise semantic). Note that here we do not have added Gnuradio approach as the processing blocks should be written in C++ (using SWIG) nor the Cython approach as they fall into the *two-language* problem we have stated beforehand.
- All the codes have been evaluated using one thread for both sample acquisition and processing, and the `-O3` flag for compilation (C++ and Julia). We also have used different optimisations for both C++, Python and Julia described afterwards.
- Rate performance is achieved by Monte Carlo simulation with 20 independent runs of 10 seconds. All the code used for the benchmark (in all languages) and versions associated to the used modules can be found here [16].
- Sliding average of the square modulus of the FFT is considered as the processing. The sliding window is rectangular with 16 samples. For the three languages, the FFT is computed with the use of FFTW with the same compilation flags (and same output rate). Difference between the languages lies in the implementation of the square absolute and mainly how the sliding mean is implemented.

### B. Code versions definition

We define four versions of the code, the initial one L0 and three optimisation levels (namely L1, L2 and L3). These three optimisations levels are independent and will be sequentially applied:

- L0 corresponds to the rapid prototyping (e.g the minimal code version that allows a proper processing chain).
- L1 corresponds to L0 with algorithmic optimisations namely by removing boundary checks in `for` loops, using buffer pre-allocations and expliciting `for` loops for square absolute and sliding average.
- L2 corresponds to L1 with optimisation on memory side and corresponds to use of low-level containers for buffers (i.e. static arrays).
- L3 corresponds to L2 with optimisation on processor instructions and corresponds to the use of SSE and AVX vectorial instructions [17] on the two `for` loops (absolute value and sliding average).

As additional notes, L2 can not been fully applied on Python and L3 vectorisation has been incorporated through the use of JIT via Numba.

### C. Comparison between versions

We evaluate the benefits of the proposed versions in Fig. 1. In this figure, no radio has been used in order to point out the maximal achievable rate

Regarding C++, the main gain has been achieved with the use of static arrays (i.e. L2). The L3 vectorisation can help to reach the maximal rate of 1.245 GS/s. It is also clear that

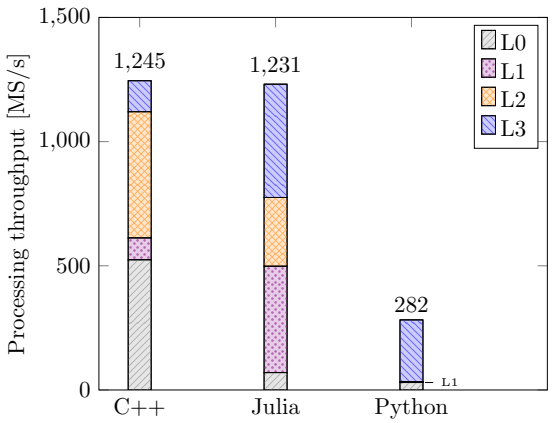


Fig. 1. Benchmark of the different versions.

the main benefit obtained with Python is based on the JIT engine (i.e. L3). Regarding Julia, all the optimisation tricks equally help to increase the throughput. The optimised rate in Julia (1.231 GS/s) is comparable to C++ but with more both rapid development time to L0 and easier application of optimisations (i.e. from L0 to L3).

### D. Benchmark with X310

We now include the SDR and we depict in Fig. 2 the performance obtained with the use of the X310. The output rate (i.e. after processing) in samples per second is evaluated versus the rate provided by the SDR which explains the lower rates encountered compared to Fig. 1. Worst performance is obtained by Python L0 code as expected. Julia without any optimisation performs better than Python (with a very similar syntax) but is far from C++. Without any optimisation, the three languages do not reach the 200 MS/s limit. When it comes to the highest optimisation level, both C++ and Julia achieve the maximal rate. It is not the case of Python (albeit being JIT compiled).

Finally, albeit writing high-level code in Julia may not directly lead to high performance, code optimisation (i.e. to increase throughput) can be directly done in Julia language with the use of macros. It makes rapid and straightforward the transition from prototyping to high performance allowing the language to address real-time high bandwidth processing.

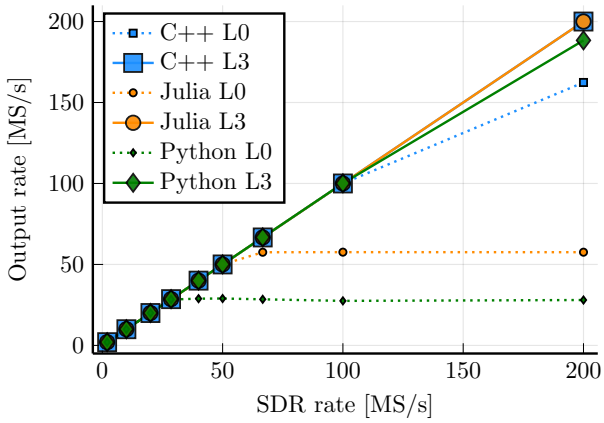


Fig. 2. Benchmark for initial code and highest optimisation level code using X310 device.

## V. Conclusion

In this paper, we have presented a new methodology for efficient SDR prototyping based on Julia language. We have presented the key properties offered by Julia that make this language appealing for SDR, namely high runtime performance, easy portability, strong scalability and convenient interactivity. These strong assets pave the way for efficient prototyping with SDR, addressing the so called *two-language problem* (rewriting high-level code in low-level language for performance). The proposed open source ecosystem offers an easy integration with different SDR types and ensure high performance with extreme code concision. The benefits of the approach (language and ecosystem) has been proven by use of benchmarks that compare the rate performance with the ones of C++ and Python. It means that rapid prototyping of very large bandwidth and extensive computational processing can be envisioned in real-time and through concise code even in embedded SDRs bringing down the classic *two-language* barrier encountered in prototyping.

## References

[1] J. Mitola, "Software radios: Survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, no. 4, pp. 25–36, 1993.

[2] M. Palkovic, P. Raghavan *et al.*, "Future software-defined radio platforms and mapping flows," *IEEE Signal Processing Magazine*, vol. 27, no. 2, 2010.

[3] M. Dardaillon, K. Marquet, T. Risset, and A. Scherrer, "Software defined radio architecture survey for cognitive testbeds," in *Proc. International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2012, pp. 189–194.

[4] S. Barmpounakis, N. Maroulis *et al.*, "Network slicing-enabled RAN management for 5G: Cross layer control based on SDN and SDR," *Computer Networks*, vol. 166, p. 106987, 2020.

[5] C. Lavaud, R. Gerzaguet, M. Gautier, and O. Berder, "Toward Real time interception of Frequency Hopping Signals," in *Proc. IEEE International Workshop on Signal Processing Systems*, 2020.

[6] M. Shi, Y. Bar-Ness, and W. Su, "Blind OFDM systems parameters estimation for software defined radio," in *Proc. IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (NFDSAN)*, 2007, pp. 119–122.

[7] A. M. Wyglinski, D. P. Orofino, M. N. Ettus, and T. W. Rondeau, "Revolutionizing software defined radio: case studies in hardware, software, and education," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 68–75, 2016.

[8] M. Braun, J. Pendlum, and M. Ettus, "RFNoC: RF network-on-chip," in *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2016.

[9] D. M. Beazley *et al.*, "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." in *Tcl/Tk Workshop*, vol. 43, 1996, p. 74.

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 75–86.

[12] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky, "Julia: Dynamism and performance reconciled by design," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–23, 2018.

[13] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381–1384.

[14] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2018.

[15] Julia Telecom, "AbstractSDRs - Common API for Software Defined Radio ," 2020, https://github.com/JuliaTelecom/AbstractSDRs.jl.

[16] C. Lavaud, R. Gerzaguet, M. Gautier, and O. Berder, "AbstractSDRsBenchmark repository," 2020, https://github.com/RGerzaguet/AbstractSDRsBenchmark.

[17] R. Karrenberg and S. Hack, "Whole-function vectorization," in *International Symposium on Code Generation and Optimization (CGO)*, 2011, pp. 141–150.