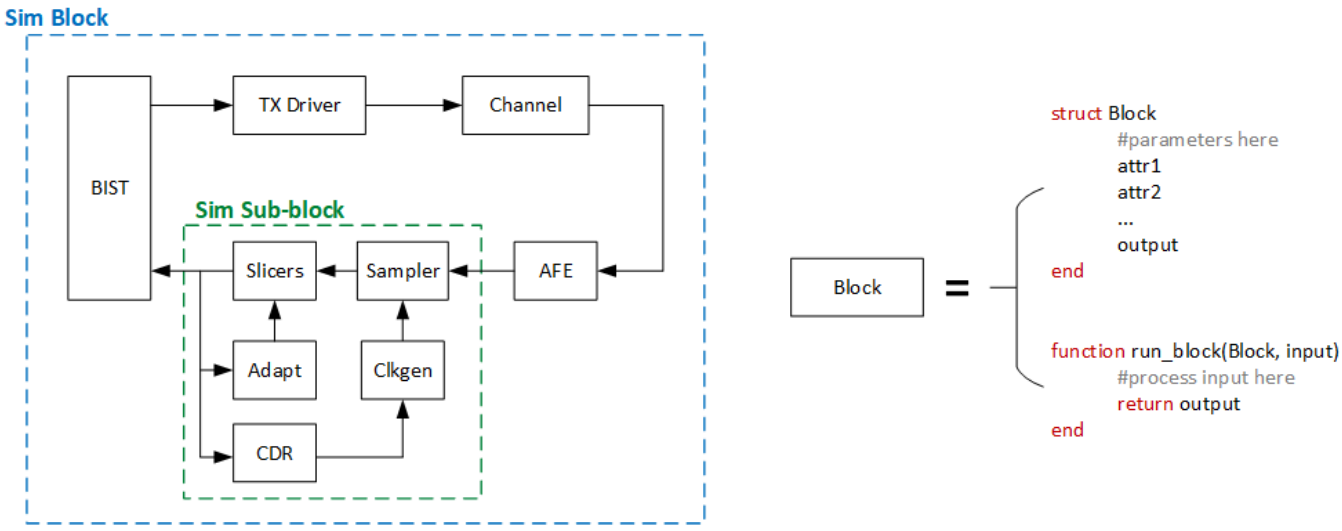
Building SerDes Models in Julia, pt2 - Simulation Framework



For the uninitiated, SerDes systems have evolved to become extremely complicated as the transmission speed increases (>100Gbps) and transmission medium changes (from copper to fiber). The challenge lies in the intersection between continuous- and discrete-time signal processing. One example is when clocks are used to sample signals (either implicitly with a data slicer or explicitly with data converters nowadays). Complex digital adaptation and clock-data recovery (CDR) algorithms become more interwined because all loops are running at the same time. How to effectively model and simulate all these components together has always been an interesting topic.

While many different analysis methods exist, including frequency and statistcal analysis, time domain results remain the final sign-off. Therefore, this framework focuses on transient simulation first and hopefully other analysis tools will be added in the near future.

Key Simulation Parameters - time step and block size

In order to simulate time domain waveforms, we resort to arrays (or vectors) to represent such waveforms. Now in order to build an *efficient* model, we first need to decide the granuarity of a simulation step, as well as the block size to simulate in batch. If we want to simulate a sequence of random bits, they need to be converted to continuous time waveform first with a finite time step.

Then the step size choice directly impacts the accuracy and the speed of your sim. Allow me to illustrate

Simulation time step

```
tui = 1/10e9; # Unit Interval time for 10Gbps data
```

```
osr1 = 4; # Oversampling ratio per symbol, i.e. # of points to simulate per symbol
```

```
osr2 = 32;
```

```
dt1 = tui/osr1; # Simulation time step
```

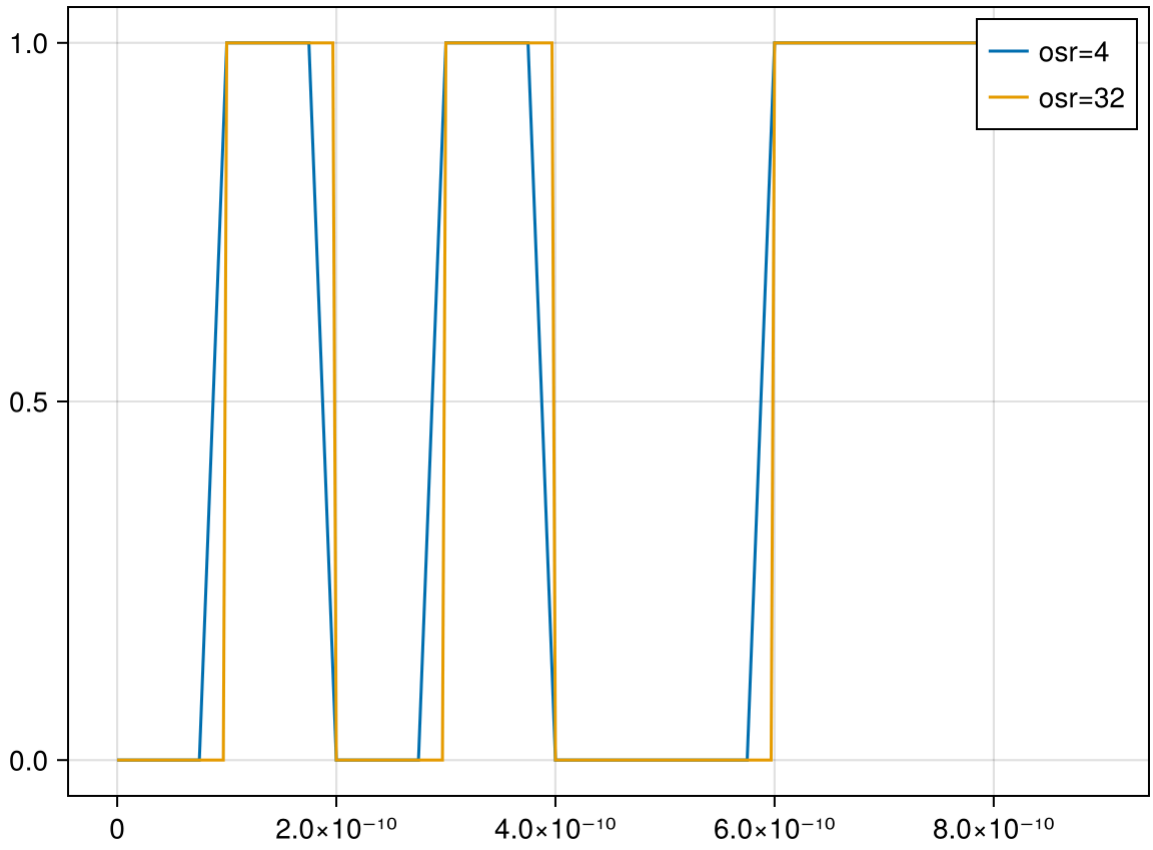
```
dt2 = tui/osr2;
```

```
bits = [0,1,0,1,0,0,1,1,1]; #arbitrary bits; use bitrand(N) to generate random bits
```

```
tt1, Vbits1 = gen_wvfm(bits, tui=tui, osr=osr1);
```

```
tt2, Vbits2 = gen_wvfm(bits, tui=tui, osr=osr2);
```

When `osr` is higher, the resultant waveform looks more like perfect square waveforms. Lower `osr` results in waveform distortion.



Here is what happens when we pass this waveform through a "channel" by means of convolution. With a larger time step, the convolution finishes faster because the array size is obviously smaller. However, there are noticeable kinks in the waveform, which might lead to simulation inaccuracy later. The accuracy increase also plateaus once the curve looks smooth enough.

Note that the increase in computation time doesn't increase linearly w.r.t array size becasue the convolution method in DSP.jl package uses FFT. Therefore, it's often more beneficial to just pick a bigger `osr` because Nlog(N) is cool!

```
bw_ir = 8e9;
```

```
tlen_ir = 20*tui; #20UI long impulse response
```

```
ir1 = gen_ir_rc(dt1, bw_ir, tlen_ir);
```

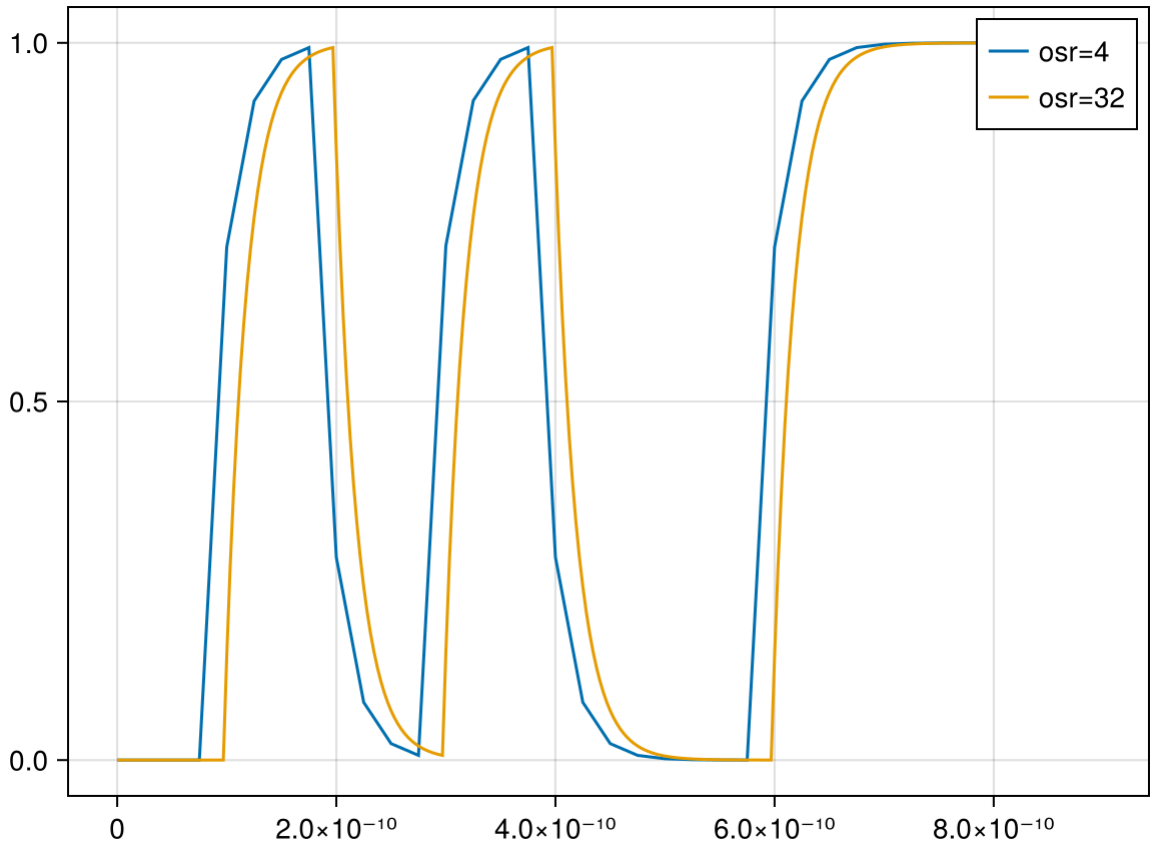
```
@time Vout1 = conv(ir1, Vbits1)*dt1;
```

```
0.002972 seconds (19 allocations: 7.234 KiB)
```

```
ir2 = gen_ir_rc(dt2, bw_ir, tlen_ir);
```

```
@time Vout2 = conv(ir2, Vbits2)*dt2;
```

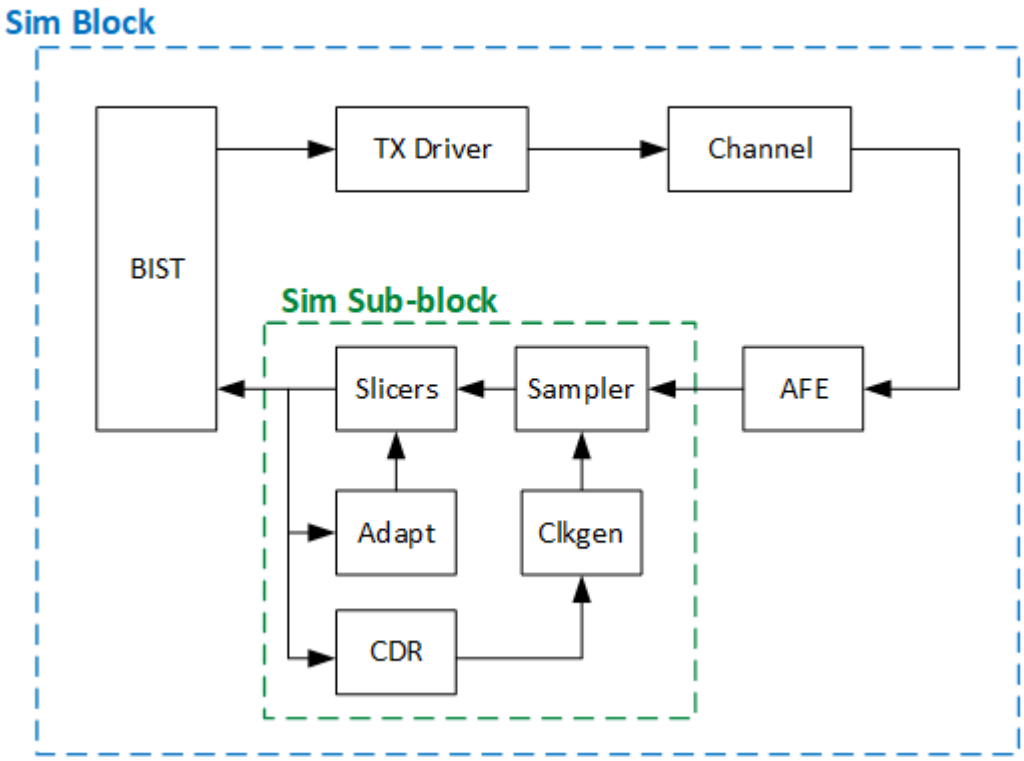
```
0.000477 seconds (19 allocations: 46.016 KiB)
```



Simulation (sub-)block size

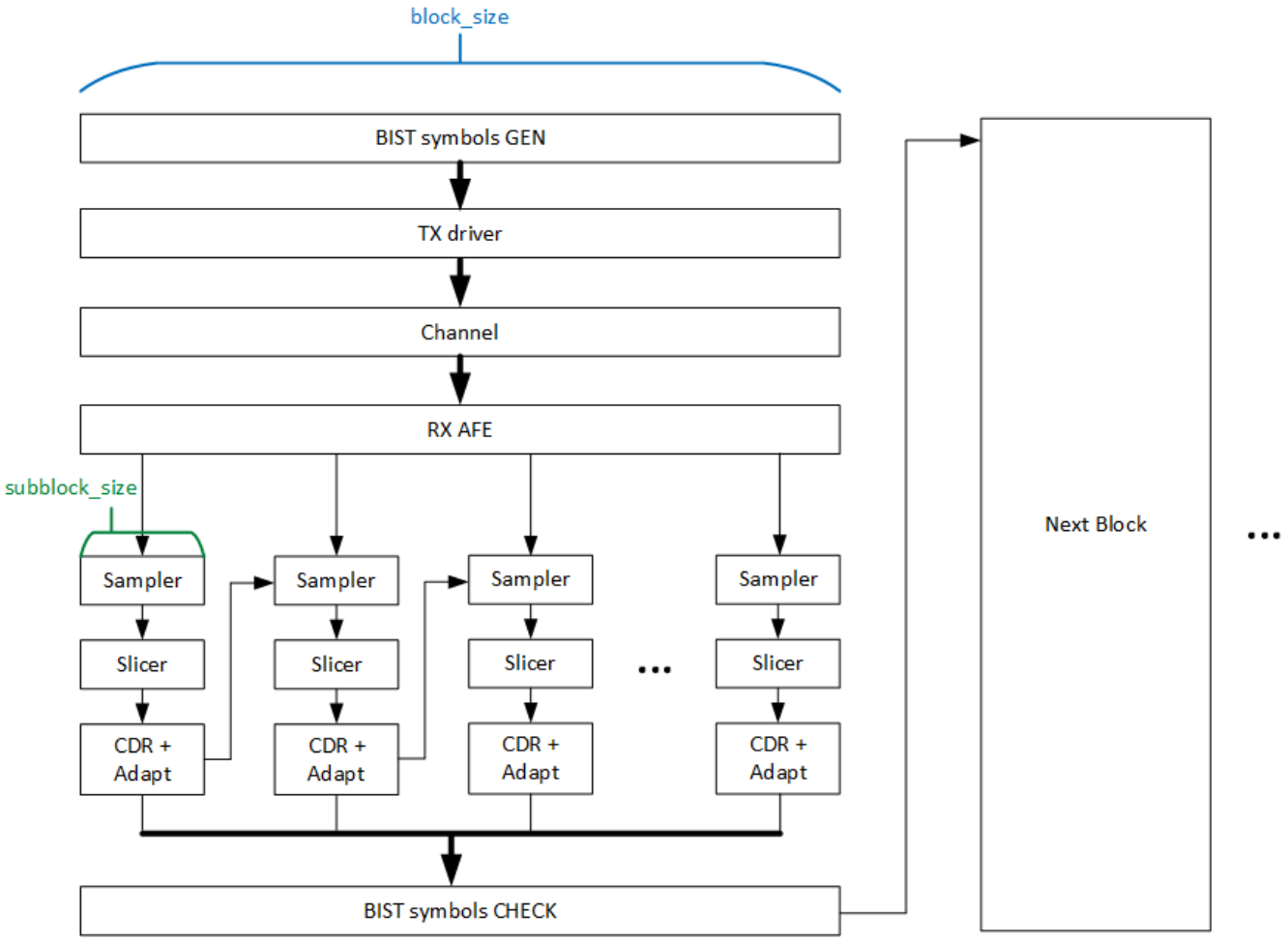
Once the simulation time step is chosen, the next question is the batch size of the signal we want to process. The extreme examples would be when simulating 1 million bits, we can either generate and process *1 bit at a time* or *1 million at once*. This choice will equally impact simulation speed and accuracy. Like all things in engineering, it's always a spectrum and our framework should give the designer the freedom to adjust these parameters.

Here we define two types of blocks: an outer loop block and an inner loop sub-block. Each block will contain different modules corresponding to different circuits in your system. Let's look at the example block diagram below.



The outter loop block is suitable for circuits that are "quasi-static" and can process many bits at once. For example, a transmitter can generate waveforms for a large number of bits up front instead of simulating multiple short sequences, since we don't expect the tranmistter parameters to change during simulation (e.g. FIR coefficients and bandwidth). This will save time by doing repetitve convolutions (as we have seen before, longer convolutions become faster due to FFT algorithm).

The inner loop sub-block is more suitable for circuits that require more accuracy and have more complicated loop dynamics. For example, the adaptation and CDR loops normally operate on a parallel bus of data and error slicer outputs. This vector then effectively determine the sub-block size because we want to capture the real dynamics of the feedback loops. Another way to think about this is that we implicitly created a "digital clock" through this sub-block loop, i.e. each loop iteration is a digital clock cycle.



The figure above illustrate the relationships between the big block and sub-blocks. We first batch process a long bit sequence, then iterate over the waveform through the sub-blocks.

Note that although this picture implies that the block size has to be an integer multiple of sub-block size, but in reality it could be extended through an elastic buffer layer in between. The current code only assumes integer relationship between these two numbers, but the elastic buffer approach would be the most general for modeling say frequency offsets between TX and RX (will be addressed in future development)

The run_sim() skeleton

With this in mind, we can start with a run_sim() function pseudo-code already as

```
run_sim (generic function with 1 method)
```

```
function run_sim(params)
    @unpack Nblks, Nsubblks, blk_size, subblk_size = params
    @unpack bist_params, tx_params, rx_params = params

    for n = 1:Nblks
        # run batch processing blocks here
        # e.g.
        # S = bist_gen(bist_params)
        # Vtx = tx_drv(tx_params, Vtx)

        for m = 1:Nsubblks
            # run shorter loops here
            # e.g.
            # Φ = clkgen(pi_code)
            # Ssub = sampler(Vtx, Φ)
            # dvec = slicers(Ssub)
            # ...

            # record waveforms here
        end

        # bist_check(bist_params, data_rcvd)

    end

end
```

That's it! This is pretty much the big picture of the framework, and now it's a matter of adding in the functional model of each circuit module **AS LONG AS** they satisfy this `osr`, `blk_size`, `subblk_size` framework. This usually means dealing with boundary conditions internally, remembering states from the previous (sub-)block, etc.

World's best SerDes

Let's build the world's most ideal SerDes using this framework. What we need are some BIST functions that generates and checks PRBS, an ideal transmitter and slicers. Let's start with the bist_prbs_gen from our first notebook

```
bist_prbs_gen (generic function with 1 method)
```

```
function bist_prbs_gen(;poly, inv, Nsym, seed)
    seq = Vector{Bool}(undef,Nsym)
    for n = 1:Nsym
        seq[n] = inv
        for p in poly
            seq[n] ⊻= seed[p]
        end
        seed .= [seq[n]; seed[1:end-1]]
    end
    return seq, seed
end
```

And here is an example ber_check_prbs function

```
ber_check_prbs (generic function with 1 method)
```

```
function ber_check_prbs(rcvd_bits; poly, inv, seed, lock_status, lock_cnt,
lock_threshold, ber_err_cnt, ber_tot_cnt)

    nbits_rcvd = lastindex(rcvd_bits)

    if lock_status #if prbs already locked, use prbs_gen for reference bits
        ref_bits, seed = bist_prbs_gen(poly=poly, inv=inv,
                                       Nsym=nbits_rcvd, seed=seed)
        ber_err_cnt += sum(rcvd_bits .⊻ ref_bits)
        ber_tot_cnt += nbits_rcvd

    else # if not locked yet, use received bits as seed
        for n = 1:nbits_rcvd
            brcv = rcvd_bits[n]
            btst = inv
            for p in poly
                btst ⊻= seed[p]
            end
            seed .= [brcv; seed[1:end-1]]

            #need consecutive non-error for lock. reset when error happens
            lock_cnt = (btst == brcv) ? lock_cnt+1 : 0

            if lock_cnt == lock_threshold
                lock_status = true
                println("prbs locked")
                #run prbs till end
                ref_bits, seed = bist_prbs_gen(poly=poly, inv=inv,
                                               Nsym=nbits_rcvd-n, seed=seed)
                ber_err_cnt += sum(rcvd_bits[n+1:end] .⊻ ref_bits)
                ber_tot_cnt += nbits_rcvd - n

                break
            end
        end
    end

    return seed, lock_status, lock_cnt, ber_err_cnt, ber_tot_cnt
end
```

We can verify these two functions with direct feedthrough

```
poly = [28,31]; #PRBS31 polynomial
```

```
inv = false;
```

```
Nsym = Int(10e6); #Try changing number of bits
```

```
gen_seed = ones(Bool, 31);
```

```
chk_seed = zeros(Bool, 31);
```

```
bits_test, gen_seed1 = bist_prbs_gen(poly=poly, inv=inv, Nsym=Nsym, seed=gen_seed);
```

```
chk_seed1, lock_status1, lock_cnt1, ber_err_cnt1, ber_tot_cnt1 =
    ber_check_prbs(bits_test; poly=poly, inv=inv, seed=chk_seed,
                   lock_status=false, lock_cnt=0, lock_threshold=256,
                   ber_err_cnt=0, ber_tot_cnt=0);
```

```
prbs locked
```

Try printing lock_status, lock_cnt, etc. below

```
println(ber_err_cnt1)
```

```
0
```

We can now artificially add bit errors to verify our checker

```
ber_target = 1e-4; #play around with this number and Nsym
```

```
err_loc = rand(Uniform(0,1.0),Nsym).< ber_target;
#Uniform is from Distributions package
```

```
bits_werr = bits_test .⊻ err_loc; #flip the bits at error locations
```

```
chk_seed2, lock_status2, lock_cnt2, ber_err_cnt2, ber_tot_cnt2 =
    ber_check_prbs(bits_werr; poly=poly, inv=inv, seed=chk_seed,
                   lock_status=false, lock_cnt=0, lock_threshold=256,
                   ber_err_cnt=0, ber_tot_cnt=0);
```

```
prbs locked
```

```
println("BER ≈ " * string(ber_err_cnt2/ber_tot_cnt2))
```

```
BER ≈ 9.610275814915889e-5
```

Let's now build an ideal transmitter and receiver. The ideal transmitter will simply take the bit sequence and oversample according to `osr` to create the time domain waveform. Our ideal receiver will then take samples from the waveform with direct indexing. Because the waveform is perfect, it doesn't really matter which index we take the samples at.

```
tx_drv_top (generic function with 1 method)
```

```
function tx_drv_top(bits; osr, swing)
    #just like the gen_wvfm helper function, but no need for the time vector
    #change output waveform to analog voltage in +/- swing/2
    #the dots are needed to do elementwise operation on the bits vector
    return kron(sign.(bits.-0.5), swing/2*ones(osr))

end
```

```
rx_slice_top (generic function with 1 method)
```

```
function rx_slice_top(Vin; sample_idx, osr, slice_lvl)
    # sample_idx is a number between 1 and osr, will wrap around for now
    Vsamples =  Vin[mod(Int(sample_idx)-1, osr)+1:osr:end]
    return Vsamples .> slice_lvl
end
```

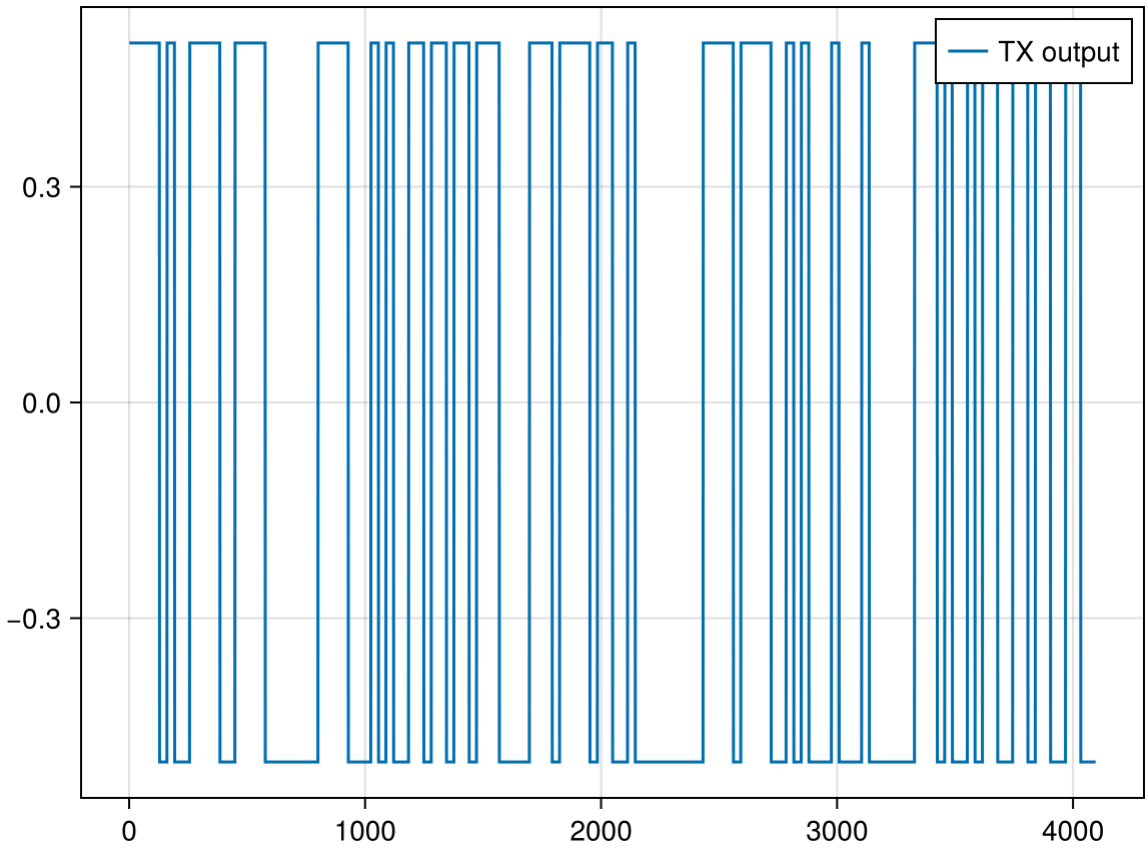
Now let's write a simple run_sim_ideal() function to plug all these functions in. In this example, we don't have a sub-block loop, which would make more sense when CDR and adaptation is introduced.

run_sim_ideal (generic function with 1 method)

```
function run_sim_ideal(osr, blk_size, nsym_tot)

    # initialize parameters
    gen_seed = ones(Bool, 31)
    chk_seed = zeros(Bool, 31)
    poly = [28,31]
    inv = false

    lock_status = false
    lock_cnt = 0
    ber_err_cnt = 0
    ber_tot_cnt = 0

    ber_target = 0e-5;

    Vtx = zeros(blk_size*osr)

    #round nblk to the nearest total # of symbols
    nblk = Int(round(nsym_tot/blk_size))

    # main loop
    for n = 1:nblk
        #generate bits
        bits, gen_seed = bist_prbs_gen(poly=poly, inv=inv,
                                       Nsym=blk_size, seed=gen_seed);

        #generate TX waveform
        Vtx = tx_drv_top(bits, osr=osr, swing=1)

        #sample TX waveform
        bits_rcvd = rx_slice_top(Vtx, sample_idx=osr/2, osr=osr, slice_lvl=0)

        #add random bit error
        err_loc = rand(Uniform(0,1.0),blk_size).< ber_target;
        bits_werr = bits_rcvd .⊻ err_loc;

        #check error
        chk_seed, lock_status, lock_cnt, ber_err_cnt, ber_tot_cnt =
            ber_check_prbs(bits_werr; poly=poly, inv=inv, seed=chk_seed,
                    lock_status=lock_status, lock_cnt=lock_cnt, lock_threshold=256,
                    ber_err_cnt=ber_err_cnt, ber_tot_cnt=ber_tot_cnt);

        #record waveforms here if needed

    end

    println("Sim done")
    println("BER: " * string(ber_err_cnt/ber_tot_cnt))

    return Vtx
end
```

```julia
@time Vtx = run_sim_ideal(32, 128, Int(1e6));
#play around with the osr and blk_size parameters and see how the run time is impacted
```

```
prbs locked
Sim done
BER: 0.0
  0.563659 seconds (4.13 M allocations: 585.194 MiB, 15.79% gc time)
```



Congratulations! We have just built your first SerDes model. I admit that this model isn't that useful, and the code itself is not looking that pretty. Trust me, it will get better 🙄.

The main issue here is that we have quite a few *states* that needs tracking (e.g. seed, ber_cnt, etc.), and spelling them out explicitly will get out of hand pretty quickly. In the next notebook, we will introduce **struct** in Julia, which is a lighter weight data structure than "classes" for those who are familiar with object oriented programming.

As a preview, we will use structs to store relevant parameters and states for the corresponding circuit module, and pass the structs around in functions. The TX driver function can then be written as the following

```
tx_drv_top (generic function with 2 methods)
```

```
function tx_drv_top(tx, bits)
    @unpack osr, blk_size, swing = tx #grab the parameters we need from the tx struct

    return kron(sign.(bits.-0.5), swing/2*ones(osr))
end
```

Julia tips

Oops! I named the function tx_drv_top again, but with different arguments. No worries! Julia implements something called multiple dispatch (learn more here). A function can take arguments of different types, and share the same name. This would be particularly helpful when we want to optimize function performance and/or allow different usage cases (e.g. explicit arguments vs. struct). We won't go too much into the details of how to use multiple dispatch and typing to optimize Julia performance, but it will be relevant for improving the framework in the long run.

```
tx_drv_top! (generic function with 1 method)
```

```
function tx_drv_top!(tx, bits)
    #grab the parameters we need from the tx struct
    @unpack osr, blk_size, swing = tx

    #store output directly into the struct
    tx.Vout .= kron(sign.(bits.-0.5), swing/2*ones(osr))
end
```

Julia tips

Julia uses a convention of putting "!" in the function name to denote that it's a mutating function. This means that the argument passed into the function will be directly changed and no explicit values will be returned. Depending on your programming philosophy, there are pros and cons to this approach. Directly mutating the arguments allows more efficient memory usage (i.e., no need to allocate new variables every time a function is called), but one needs to be careful about the sequence of function calls and keep track of the states. This simulation framework choses to mainly to use mutating functions, which we will cover more in the next notebooks.

In the next notebook, we will focus more on the data structures and how the code base will be structured. We will start to build a more complete TX module as we discuss more Julia details.

Helper function section

```
gen_wvfm (generic function with 1 method)
```

```
function gen_wvfm(bits; tui, osr)
    #helper function used to generate oversampled waveform

    Vbits = kron(bits, ones(osr)) #Kronecker product to create oversampled waveform
    dt = tui/osr
    tt = 0:dt:(length(Vbits)-1)*dt

    return tt, Vbits
end
```

Julia tips

The arguments in the gen_wvfm function are separated by a semicolon. This syntax in Julia means that anything after the semicolon is a keyword argument. An explicit "kw = something" statement is needed during function calls. Having keyword argument allows a function to have default parameters as well.

```
gen_ir_rc (generic function with 1 method)
```

```
function gen_ir_rc(dt,bw,t_len)
    #helper function that directly calculates a first order RC response, normalized
    to the time step
    tt = [0:dt:t_len-dt;]

    #checkout the intuitive symbols!
    ω = (2*π*bw)
    ir = ω*exp.(-tt*ω)
    ir .= ir/sum(ir*dt)

    return ir
end
```

Julia tips

Unicode is supported in Julia, thus one can write cleaner code using commonly known symbols like in the gen_ir_rc function.

```
using Makie, CairoMakie
```

```
using DSP, Random, Distributions, UnPack, BenchmarkTools
```