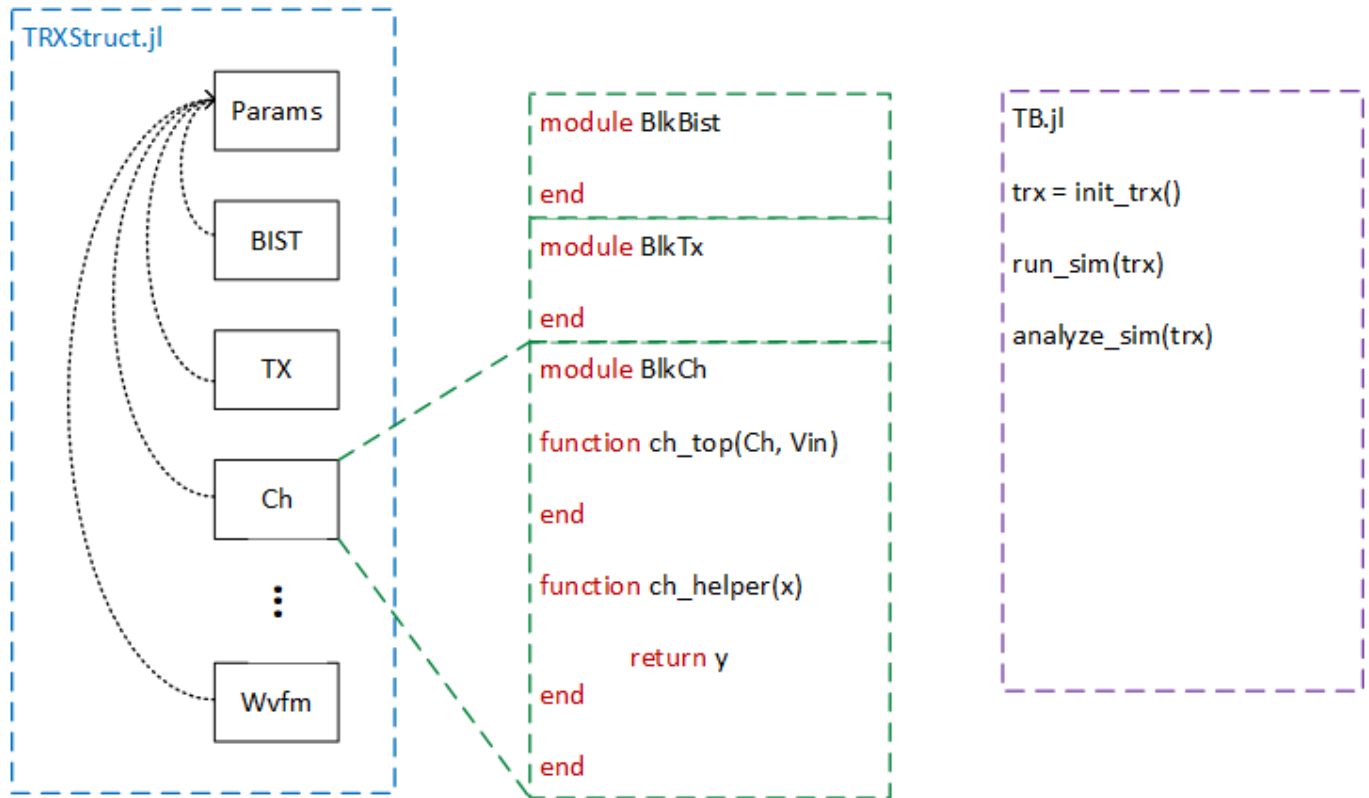


Building SerDes Models in Julia, pt3 - Data and Code Structures



Julia is not an Object Oriented Programming (OOP) language. Unlike Python where Class and Object definitions are supported, Julia's main focus is dealing with data (closer to MATLAB in this aspect even though MATLAB has OOP support too). OOP typically is good at managing and scaling complex software systems, but usually at the cost of performance (here is a famous [video rant](#) on OOP)

In reality, OOP is just a programming paradigm that focuses on encapsulating *states* and handling (im)mutability at a large scale. For our purpose, we do have quite some states to take care of for each circuit module (e.g. seed in BIST, channel memory/ISI, CDR accumulators, and even jitter), so how can we achieve this without OOP in Julia?

Struct - packing parameters, data and states

Enters `struct` - a composite data type in Julia. Think of it as a collection of any data that you want to throw at it, or object with only *attributes/properties*. MATLAB also has `struct`, and the closest native

data type in Python might be dictionary (though they are still quite different). Here is a Julia struct definition:

```
1 struct Params_im
2     osr::Int64      #oversampling ratio for simulator
3     data_rate::Float64 #nominal data rate
4     pam::Int8       #number of data levels
5     blk_size::Int64  #number of symbol in a block
6     subblk_size::Int64 #number of symbols in a sub-block
7     fbaud::Float64   #baud rate
8     fnyq::Float64    #nyquist frequency
9     tui::Float64     #unit interval time
10    dt::Float64       #simulation time step
11    # ... and more
12 end
```

```
1 p_im = Params_im(32, 10e9, 2, 2^10, 16, 10e9, 10e9/2, 1/10e9, 1/10e9/32);
```

Julia Tips

Each field can be declared with a specific type (like Float64) or left empty (to be determined at runtime, but often less performant). Try giving the compiler as much as information for the code to be optimized, especially when for arrays/matrices

Just like that, we have successfully instantiated our first struct, packing all the global simulation parameters and convenience variables together. Each field can be accessed with the "." operator.

1.0e10

```
1 p_im.data_rate #access field
```

However, struct in Julia is **immutable** by default - you can only read and not modify its internal fields once instantiated

setfield!: immutable struct of type Params_im cannot be changed

Stack trace

Here is what happened, the most recent locations are first:

1. setproperty! @ Base.jl:41

2. (This cell: line 1

```
1 p_im.osr = 16 #this will throw an error!
```

```
1 p_im.osr = 16 #this will throw an error!
```

The reason is that immutability lets compilers use memory efficiently. But don't worry, Julia have a mutable struct! (note: they will be "slower" than immutable structs)

```
1 mutable struct Params_m
2     osr::Int64          #oversampling ratio for simulator
3     data_rate::Float64  #nominal data rate
4     pam::Int8           #number of data levels
5     blk_size::Int64     #number of symbol in a block
6     subblk_size::Int64  #number of symbols in a sub-block
7     fbaud::Float64      #baud rate
8     fnyq::Float64       #nyquist frequency
9     tui::Float64        #unit interval time
10    dt::Float64         #simulation time step
11    # ... and more
12 end
```

```
1 p_m = Params_m(32, 10e9, 2, 2^10, 16, 10e9, 10e9/2, 1/10e9, 1/10e9/32);
```

1.0e10

```
1 p_m.data_rate #so far so good
```

16

```
1 p_m.osr = 16 #yay!
```

But wait, there are certain things that likely won't change once we set them. Can we have the best of both worlds? Fortunately, Julia now supports immutable fields inside a mutable struct w/ the `const` keyword.

```

1 mutable struct Params_m2
2     const osr::Int64          #oversampling ratio for simulator
3     const data_rate::Float64  #nominal data rate
4     const pam::Int8           #number of data levels
5     const blk_size::Int64     #number of symbol in a block
6     const subblk_size::Int64  #number of symbols in a sub-block
7     const fbaud::Float64     #baud rate
8     const fnyq::Float64      #nyquist frequency
9     const tui::Float64       #unit interval time
10    const dt::Float64         #simulation time step
11    cur_blk::Int
12    # ... and more
13 end

```

```

1 p_m2 = Params_m2(32, 10e9, 2, 2^10, 16, 10e9, 10e9/2, 1/10e9, 1/10e9/32, 0);

```

setfield!: const field .osr of type Params_m2 cannot be changed

Stack trace

Here is what happened, the most recent locations are first:

1. `setproperty!(x::Main.var"workspace#6".Params_m2, f::Symbol, v::Int64)`
@ `Base.jl:41`
2. `This cell: line 1`

```

1 p_m2.osr = 16 #oops error again

```

```

1 p_m2.osr = 16 #oops error again

```

```

1

```

```

1 p_m2.cur_blk = 1 #this field is mutable

```

Cool! This is all pretty intuitive, but something feels off. When we instantiated a struct instant, the arguments need to be in order and w/o keywords. If the struct becomes bigger, the code isn't exactly easy to read and maintain. Also, some variables are dependent on some other variables (e.g. dt is calculated from tui and osr). A default calculation would be nice. The `@kwdef` macro from the `Parameters.jl` package solves both problems.

```

1 using Parameters

```

Param

```
1 @kwdef mutable struct Param
2     const osr::Int64          # needs initialization
3     const data_rate::Float64  # needs initialization
4     const pam::Int8 = 2
5     const bits_per_sym::Int8 = Int(log2(pam))
6     const fbaud::Float64 = data_rate/bits_per_sym
7     const fnyq::Float64 = fbaud/2
8     const tui::Float64 = 1/fbaud
9     const dt::Float64 = tui/osr
10
11     const nsym_total::Int64    # needs initialization
12     const blk_size::Int64      # needs initialization
13     const blk_size_osr::Int64 = blk_size*osr
14     const nblk::Int64 = Int(round(nsym_total/blk_size))
15     const subblk_size::Int64 = blk_size
16     const subblk_size_osr::Int64 = subblk_size*osr
17     const nsubblk::Int64 = Int(blk_size/subblk_size)
18
19     cur_blk::Int = 0
20     # ... and more
21 end
```

@kwdef allows fields to have default values, and the struct constructor becomes keyword based. Now the way to instantiate the struct becomes much self explanatory:

```
1 param = Param(
2     osr = 32,
3     data_rate = 10e9,
4     pam = 4,
5     blk_size = 2^10,
6     nsym_total = Int(1e6));
```

5.0e9

```
1 param.fbaud #access the instantiated fields here and see their values
```

Note that the default calculations are done just once during instantiation. That's why using `const` is also important to make sure the calculated values remain intact. For example, if `pam` field is not constant and instantiated to be 2 and later changed to 4, any dependent field (like `bits_per_sym`) won't be updated automatically. There is a way to allow mutability and automatic updates, which we will cover in the future, but it's still best practice to keep constants, well, constant.

Let's now define a `Bist` struct that contains parameters and states for our PRBS generator/checker in the previous notebook

Bist

```
1 @kwdef mutable struct Bist
2     const param::Param
3
4     #shared
5     const polynomial::Vector{UInt8}
6     const order::UInt8 = maximum(polynomial)
7     const inv = false
8
9     #generator
10    gen_seed = ones(Bool, order)
11    gen_gray_map::Vector{UInt8} = []
12
13    #checker
14    chk_seed = zeros(Bool, order)
15    chk_gray_map::Vector{UInt8} = gen_gray_map
16    chk_lock_status = false
17    chk_lock_cnt::Int64 = 0
18    const chk_lock_cnt_threshold::Int64 = 256
19    ber_err_cnt::Int64 = 0
20    ber_bit_cnt::Int64 = 0
21
22    #input/output vectors
23    So_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)
24    So::Vector{Float64} = zeros(param.blk_size)
25    Si = CircularBuffer{UInt8}(param.blk_size)
26    #CircularBuffer is a special data type from the DataStructures package
27    Si_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)
28
29 end
30
```

```
1 bist = Bist(
2     param = param, #the param variable defined above
3     polynomial = [28,31]); #that's it!
```

Oh btw, a struct can have a field that points to a struct too. When we instantiate the Bist struct, the params passed into the Bist field is the reference (or pointer) to the variable. We can check the equivalence of the param instance and the param inside bist

true

```
1 param === bist.param #equivalence check
```

If something inside param is changed, it can be accessed through bist as well

```
1 param.cur_blk = 1;
```

```
1 println(bist.param.cur_blk)
2 #Pluto can't track changes in a struct field, so if you changed params.cur_blk above, remember to rerun this code cell to see the change (like Jupyter)
```

1



This is useful when global states need to be passed around, but we don't want our specialized functions (say a `prbs_gen` function operating only on the `bist` object) to take the `param` object as an explicit argument too.

Julia Tips

Julia passes mutable variables by their references/pointers into functions, known as "passing by sharing". In contrast, MATLAB passes values of the arguments ("passing by value"), so a NEW copy is made every time the `param` instance is passed into another constructor or function (thus leading to memory and speed penalties).

Julia still creates copies of a immutable variable (like numbers). For more information, [click here](#)

Named Tuples

Another useful and flexible data structure in Julia is Named Tuple. It's essentially a tuple with keyword fieldnames but it's immutable. You instantiate with `(kw1=val1, kw2=val2...)`

```
1 np_test = (a = 1, b=2, c="Foo");
```

1

```
1 np_test.a
```

setfield!:: immutable struct of type NamedTuple cannot be changed

Stack trace

Here is what happened, the most recent locations are first:

1. **setproperty!** @ Base.jl:41

2. **(** This cell: line 1

```
1 np_test.c = "can't change"
```

```
1 np_test.c = "can't change"
```

For NP, there is no need to predefine the field names like a struct. So we can just pack different things into a NP at will. Eventually, we will create a big `trx` (for transceiver) named tuple that stores all the structs that correspond to each circuit module, like `bist`, `drv`, `ch`, `cdr`, etc.

```
1 drv = Drv(  
2     param = param,  
3     #TX driver params here  
4 );
```

```
1 ch = Ch(  
2     param = param,  
3     #Channel params here  
4 );
```

```
1 cdr = Cdr(  
2     param = param,  
3     #CDR params here  
4 );
```

```
1 trx = (;param, bist, drv, ch, cdr); #add more circuit modules if needed  
2 #We can already "feel" that our transceiver is being built up 🙌
```

Julia Tips

Julia has a shorthand for constructing named tuple using keywords. Any statement after ";" will be expanded into (`kw1 = kw1`, `kw2 = kw2`, ...) The line above is the same as `trx = (param=param, bist=bist, drv=drv ...)`.

```
1 trx.param.osr #example of accessing parameters
```



```
true
```

```
1 trx.param == drv.param #still referencing the same param
```

Mutating functions

The simulation framework fully exploits the mutability and "passing by sharing" in Julia, so the input/output vectors are also stored in the struct (i.e., So_bits, So, Si in Bist). Let's define two test functions to see the advantage of mutating functions

double_return (generic function with 1 method)

```
1 function double_return(S)
2     return 2*S
3 end
```

double_mutate! (generic function with 1 method)

```
1 function double_mutate!(S)
2     @. S = 2 * S
3     # the @. macro broadcasts the . to all operations. Equivalent to S .= 2 .* S
4     return nothing
5 end
```

```
1 S1 = randn(1000000);
```

```
1 @time Sx2 = double_return(S1);
```

```
0.002650 seconds (2 allocations: 7.629 MiB)
```

```
1 S2 = randn(1000000);
```

```
1 @time double_mutate!(S2)
```

```
0.000708 seconds
```

As we can see, the non-mutating version takes an input, creates and returns a new array with the modified value. It takes 2 allocations for the computation and ~3x longer. That's why you will see many functions with "!" in Julia, like sort!, push!, append!, etc along with their non-mutating counterparts.

Since our framework simulates on a per-block basis and the vector size for each block is fixed, we will directly operate on the struct's pre-allocated input/output vectors w/ mutating functions. Here are example pam_gen_top! and ber_checker_top! functions taking a Bist struct as an input

pam_gen_top! (generic function with 1 method)

```
1 function pam_gen_top!(bist)
2   @unpack pam, bits_per_sym, blk_size = bist.param
3   @unpack polynomial, inv, gen_seed, gen_gray_map = bist
4   @unpack So, So_bits = bist
5
6   #generate PRBS bits
7   So_bits, gen_seed = bist_prbs_gen(poly=polynomial, inv=inv,
8                                     Nsym=bits_per_sym*blk_size, seed=gen_seed)
9
10  #generate PAM symbols
11  fill!(So, zero(Float64)) #reset So to all 0
12  for n = 1:bits_per_sym
13    @. So = So + 2^(bits_per_sym-n)*So_bits[n:bits_per_sym:end]
14  end
15
16  #gray encoding
17  if ~isempty(gen_gray_map)
18    for n in 1:blk_size
19      So[n] = gen_gray_map[So[n] + 1]
20    end
21  end
22
23  @. So = 2/(pam-1)*So - 1 #convert to analog voltage levels in +/-1 range
24
25  return nothing
26 end
```

ber_checker_top! (generic function with 1 method)

```
1 function ber_checker_top!(bist)
2   @unpack cur_blk, pam, bits_per_sym = bist.param
3   @unpack gen_gray_map, chk_start_blk, Si, Si_bits = bist
4
5
6   if cur_blk >= chk_start_blk #make start blk a parameter later
7     if ~isempty(gen_gray_map)
8       for n in 1:blk_size
9         Si[n] = gen_gray_map[Si[n] + 1]
10      end
11    end
12
13    Si_bits .= vec(stack(int2bits.(Si, bits_per_sym)))
14
15    ber_check_prbs!(bist)
16  end
17
18  return nothing
19 end
```

Modules

As we discussed in the beginning, struct is similar to a class with only attributes and no member functions. This means that the functions operating on the struct instances need to live somewhere else.

Julia uses modules to organize code. It's best to go through the official documentation on Modules [here](#). We will create two modules to demonstrate how the framework code is structured. (expand the code below to view details)

```
Main.var"workspace#3".TrxStruct
```

```
Main.var"workspace#3".BlkBIST
```

Julia Tips

The `export` keyword in a module defines the internal functions that can be directly called at parent level without using the namespace. For example, if `BlkBIST` is used in the testbench file, `pam_gen_top!` can be directly called, but we need to use `BlkBIST.bist_prbs_gen` for the non-exported function.

Here a design decision is made to put all custom structs under a single module, and functions in another. The main reason is perhaps due to the No. 1 problem I have so far with Julia: it doesn't completely feel like an interpreted runtime language yet. `Revise.jl` package does a good job tracking real time updates in *functions*, but it can't do it for structs in a module (detailed [here](#)). The solution is to put all structs in a dedicated module and include at the Main namespace [\[ref\]](#). (It's ok if you don't follow fully - the key point is that it's some fundamental limitation to Julia's development environment right now).

Another reason is due to Julia's multiple dispatch capability. It is encouraged to have functions outside of the scope of the custom data type definitions (in contrast to conventional OOP).

Of course, once development is almost complete (or at least when your data types are not changing anymore), we can then fully modularize the custom structs for the final performance squeeze.

File/Code Structure

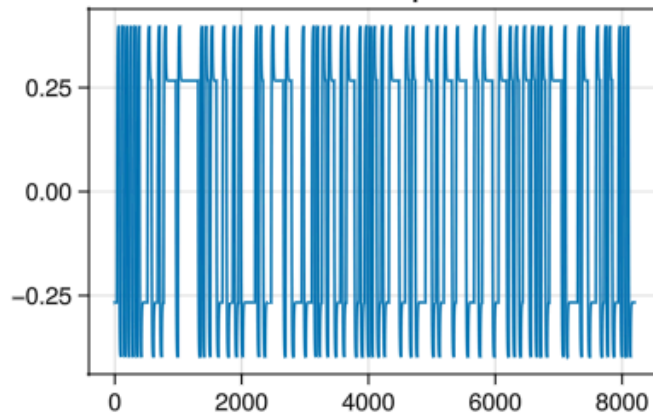
The `src` folder in the code base is then structured as the following

```
src
├── blks
│   ├── BlkBIST.jl
│   ├── BlkCH.jl
│   ├── BlkRX.jl
│   ├── BlkTX.jl
│   └── WvfmGen.jl
├── structs
│   └── TrxStruct.jl
├── tb
│   └── TB.jl
├── util
│   └── Util_JLSD.jl
└── Main.jl
```

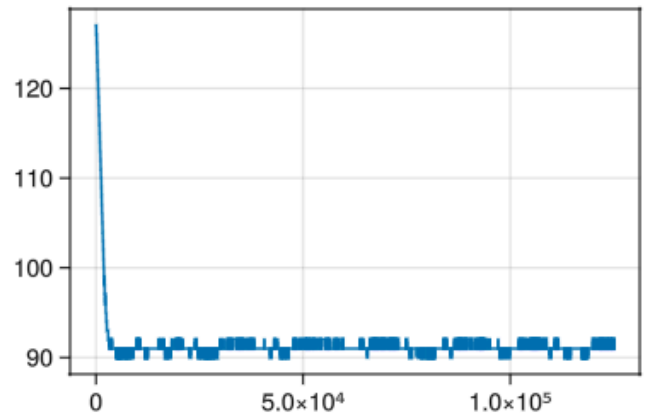
The `blks` folders contain the modules of functions for each circuit block. `structs` contains the modules of data structs for each circuit block. `tb` has the `run_sim` functions that define how the circuit blocks are connected. The `Main.jl` file is main script to be run.

At this point, you should be able to understand the code structure and hopefully the `run_sim()` functions in `TB.jl` as well. Try running `Main.jl` (after Julia is setup) and see the following plot show up (with transmitter jitter turned on). In the next notebook, we will walk through the specifics of the `BlkTX.jl` to build a reasonably complex transmitter model.

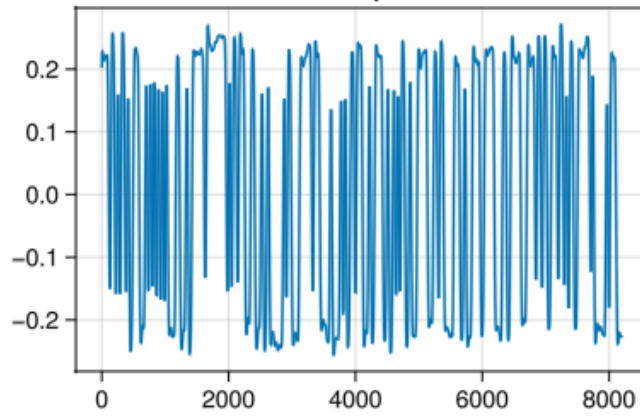
Driver output



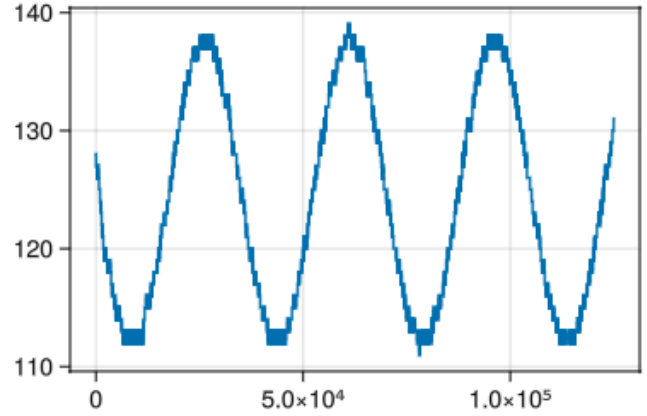
Error Slicer DAC code



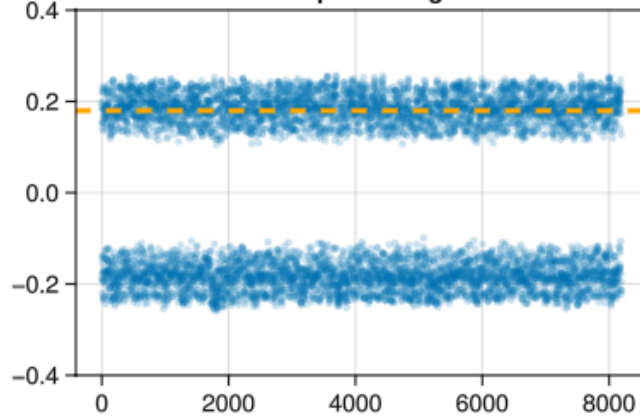
RX input



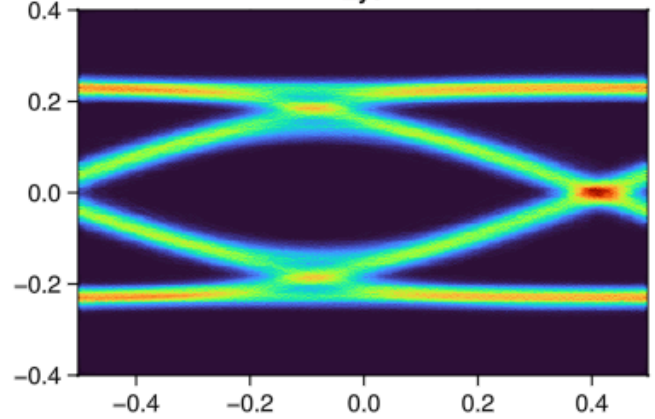
PI code



Sampled voltage



Eye



Helper functions

bist_prbs_gen (generic function with 1 method)

```
1 function bist_prbs_gen(;poly, inv, Nsym, seed)
2     seq = Vector{Bool}(undef,Nsym)
3     for n = 1:Nsym
4         seq[n] = inv
5         for p in poly
6             seq[n] ⊖= seed[p]
7         end
8         seed .= [seq[n]; seed[1:end-1]]
9     end
10    return seq, seed
11 end
```

ber_check_prbs! (generic function with 1 method)

```
1 function ber_check_prbs!(bist)
2     @unpack polynomial, inv, chk_seed, Si_bits = bist
3     nbits_rcvd = lastindex(Si_bits)
4
5     # Uncomment if you want to add artifical BER
6     # err_loc = rand(Uniform(0,1.0), nbits_rcvd).< 1e-4;
7     # Si_bits .= Si_bits .y err_loc
8
9     if bist.chk_lock_status
10         ref_bits, chk_seed = bist_prbs_gen(poly=polynomial, inv=inv,
11                                           Nsym=nbits_rcvd, seed=chk_seed)
12
13         bist.ber_err_cnt += sum(Si_bits .y ref_bits)
14         bist.ber_bit_cnt += nbits_rcvd
15     else
16         for n = 1:nbits_rcvd
17             brcv = Si_bits[n]
18             btst = inv
19             for p in polynomial
20                 btst y= chk_seed[p]
21             end
22
23             #need consecutive non-error for lock. reset when error happens
24             bist.chk_lock_cnt = (btst == brcv) ? bist.chk_lock_cnt+1 : 0
25
26             chk_seed .= [brcv; chk_seed[1:end-1]]
27
28             if bist.chk_lock_cnt == bist.chk_lock_cnt_threshold
29                 bist.chk_lock_status = true
30                 println("prbs locked")
31                 ref_bits, chk_seed = bist_prbs_gen(poly=polynomial, inv=inv,
32                                           Nsym=nbits_rcvd-n,
33                                           seed=chk_seed)
34                 bist.ber_err_cnt += sum(Si_bits[n+1:end] .y ref_bits)
35                 bist.ber_bit_cnt += nbits_rcvd-n
36                 break
37             end
38         end
39     end
40     return nothing
41 end
```

int2bits (generic function with 1 method)

```
1 function int2bits(num, nbit)
2     return [Bool((num>>k)%2) for k in nbit-1:-1:0]
3 end
```

Drv

```
1 @kwdef mutable struct Drv #dummy struct for TX driver
2     param::Param
3 end
```

Ch

```
1 @kwdef mutable struct Ch #dummy struct for channel
2     param::Param
3 end
```

Cdr

```
1 @kwdef mutable struct Cdr #dummy struct for CDR
2     param::Param
3 end
```

```
1 using Makie, CairoMakie
```

```
1 using UnPack, DataStructures, Random, DSP
```

```
1 using BenchmarkTools
```