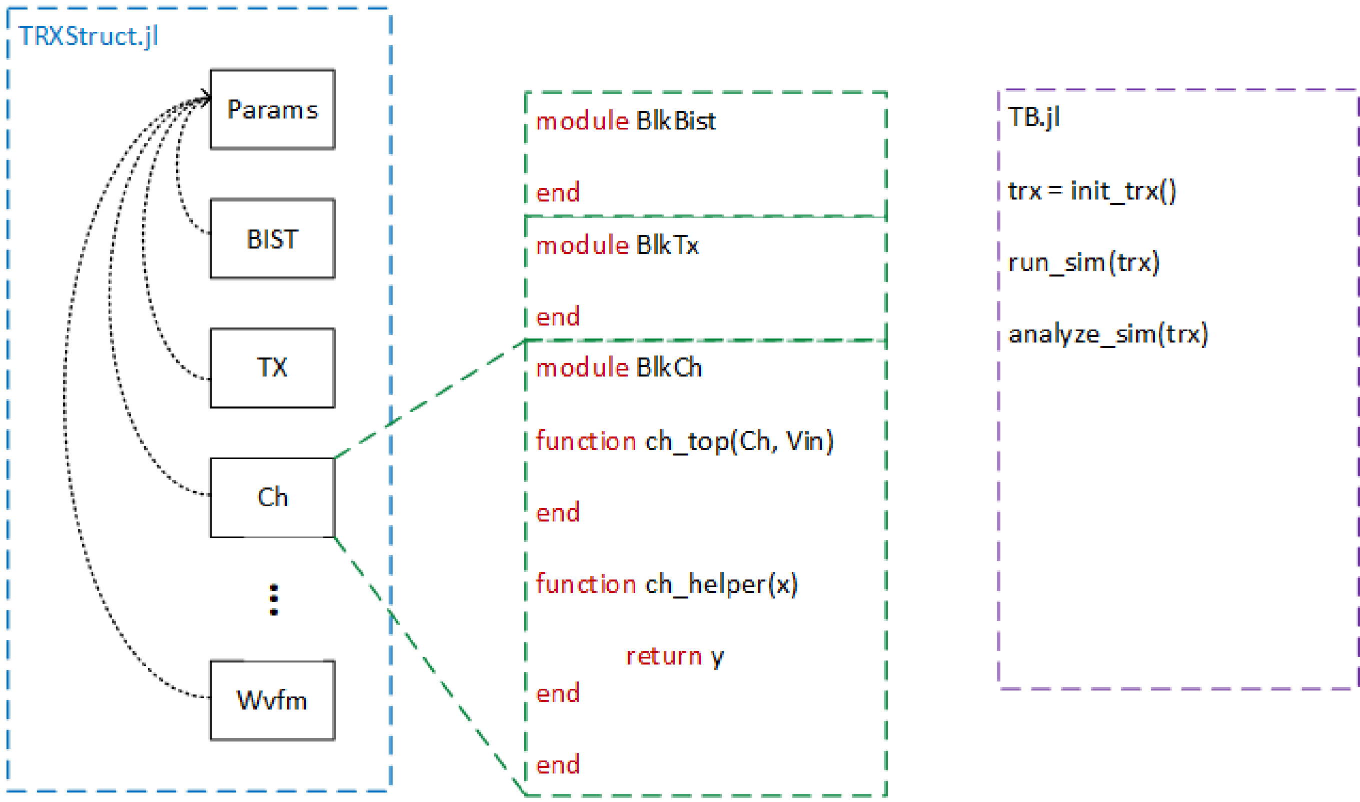
Building SerDes Models in Julia, pt3 - Data and Code Structures



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:

```
struct Params_im
    osr::Int64           #oversampling ratio for simulator
    data_rate::Float64   #nominal data rate
    pam::Int8            #number of data levels
    blk_size::Int64      #number of symbol in a block
    subblk_size::Int64   #number of symbols in a sub-block
    fbaud::Float64       #baud rate
    fnyq::Float64        #nyquist frequency
    tui::Float64         #unit interval time
    dt::Float64          #simulation time step
    # ... and more
end
```

```
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 convience variables together. Each field can be accessed with the "." operator.

```
1.0e10
```

```
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

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

```
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)

```
mutable struct Params_m
    osr::Int64            #oversampling ratio for simulator
    data_rate::Float64    #nominal data rate
    pam::Int8             #number of data levels
    blk_size::Int64       #number of symbol in a block
    subblk_size::Int64    #number of symbols in a sub-block
    fbaud::Float64        #baud rate
    fnyq::Float64         #nyquist frequency
    tui::Float64          #unit interval time
    dt::Float64           #simulation time step
    # ... and more
end
```

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

```
1.0e10
```

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

```
16
```

```
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.

```julia
mutable struct Params_m2
    const osr::Int64            #oversampling ratio for simulator
    const data_rate::Float64    #nominal data rate
    const pam::Int8             #number of data levels
    const blk_size::Int64       #number of symbol in a block
    const subblk_size::Int64    #number of symbols in a sub-block
    const fbaud::Float64        #baud rate
    const fnyq::Float64         #nyquist frequency
    const tui::Float64          #unit interval time
    const dt::Float64           #simulation time step
    cur_blk::Int
    # ... and more
end
```

```julia
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#66".Params_m2, f::Symbol, v::Int64)`
   @ *Base.jl:41*
2. *This cell: line 1*
   ```julia
   p_m2.osr = 16 #oops error again
   ```

```julia
p_m2.osr = 16 #oops error again
```

1

```julia
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.

```julia
using Parameters
```

Param

```
@kwdef mutable struct Param
    const osr::Int64            # needs initialization
    const data_rate::Float64    # needs initialization
    const pam::Int8 = 2
    const bits_per_sym::Int8 = Int(log2(pam))
    const fbaud::Float64 = data_rate/bits_per_sym
    const fnyq::Float64 = fbaud/2
    const tui::Float64 = 1/fbaud
    const dt::Float64 = tui/osr

    const nsym_total::Int64     # needs initialization
    const blk_size::Int64       # needs initialization
    const blk_size_osr::Int64 = blk_size*osr
    const nblk::Int64 = Int(round(nsym_total/blk_size))
    const subblk_size::Int64 = blk_size
    const subblk_size_osr::Int64 = subblk_size*osr
    const nsubblk::Int64 = Int(blk_size/subblk_size)

    cur_blk::Int = 0
    # ... and more
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:

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

5.0e9

```
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

```
@kwdef mutable struct Bist
    const param::Param

    #shared
    const polynomial::Vector{UInt8}
    const order::UInt8 = maximum(polynomial)
    const inv = false

    #generator
    gen_seed = ones(Bool, order)
    gen_gray_map::Vector{UInt8} = []

    #checker
    chk_seed = zeros(Bool, order)
    chk_gray_map::Vector{UInt8} = gen_gray_map
    chk_lock_status = false
    chk_lock_cnt::Int64 = 0
    const chk_lock_cnt_threshold::Int64 = 256
    ber_err_cnt::Int64 = 0
    ber_bit_cnt::Int64 = 0

    #input/output vectors
    So_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)
    So::Vector{Float64} = zeros(param.blk_size)
    Si = CircularBuffer{UInt8}(param.blk_size)
    #CircularBuffer is a special data type from the DataStructures package
    Si_bits::Vector = zeros(Bool, param.bits_per_sym*param.blk_size)

end

```

```
bist = Bist(
            param = param, #the param variable defined above
            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

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

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

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

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

```julia
println(bist.param.cur_blk)
#Pluto can't tract 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)
```

```julia

```

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...)

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

1

```julia
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*

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

```julia
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.

```julia
drv = Drv(
        param = param,
        #TX driver params here
    );
```

```julia
ch = Ch(
        param = param,
        #Channel params here
    );
```

```julia
cdr = Cdr(
        param = param,
        #CDR params here
    );
```

```julia
trx = (;param, bist, drv, ch, cdr); #add more circuit modules if needed
#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 ...)`.


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

```
true
```

```julia
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)
```

```julia
function double_return(S)
    return 2*S
end
```

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

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

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

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

```
0.002777 seconds (2 allocations: 7.629 MiB)
```

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

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

```
0.000962 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)

```
function pam_gen_top!(bist)
    @unpack pam, bits_per_sym, blk_size = bist.param
    @unpack polynomial, inv, gen_seed, gen_gray_map = bist
    @unpack So, So_bits = bist

    #generate PRBS bits
    So_bits, gen_seed = bist_prbs_gen(poly=polynomial, inv=inv,
                                    Nsym=bits_per_sym*blk_size, seed=gen_seed)

    #generate PAM symbols
    fill!(So, zero(Float64)) #reset So to all 0
    for n = 1:bits_per_sym
        @. So = So + 2^(bits_per_sym-n)*So_bits[n:bits_per_sym:end]
    end

    #gray encoding
    if ~isempty(gen_gray_map)
        for n in 1:blk_size
            So[n] = gen_gray_map[So[n] + 1]
        end
    end

    @. So = 2/(pam-1)*So - 1 #convert to analog voltage levels in +/-1 range

    return nothing
end
```

ber_checker_top! (generic function with 1 method)

```
function ber_checker_top!(bist)
    @unpack cur_blk, pam, bits_per_sym = bist.param
    @unpack gen_gray_map, chk_start_blk, Si, Si_bits = bist


    if cur_blk >= chk_start_blk #make start blk a parameter later
        if ~isempty(gen_gray_map)
            for n in 1:blk_size
                Si[n] = gen_gray_map[Si[n] + 1]
            end
        end

        Si_bits .= vec(stack(int2bits.(Si, bits_per_sym)))

        ber_check_prbs!(bist)
    end

    return nothing
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#375".TrxStruct
```

```
Main.var"workspace#425".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 a 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 constrast to conventional OOP).

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

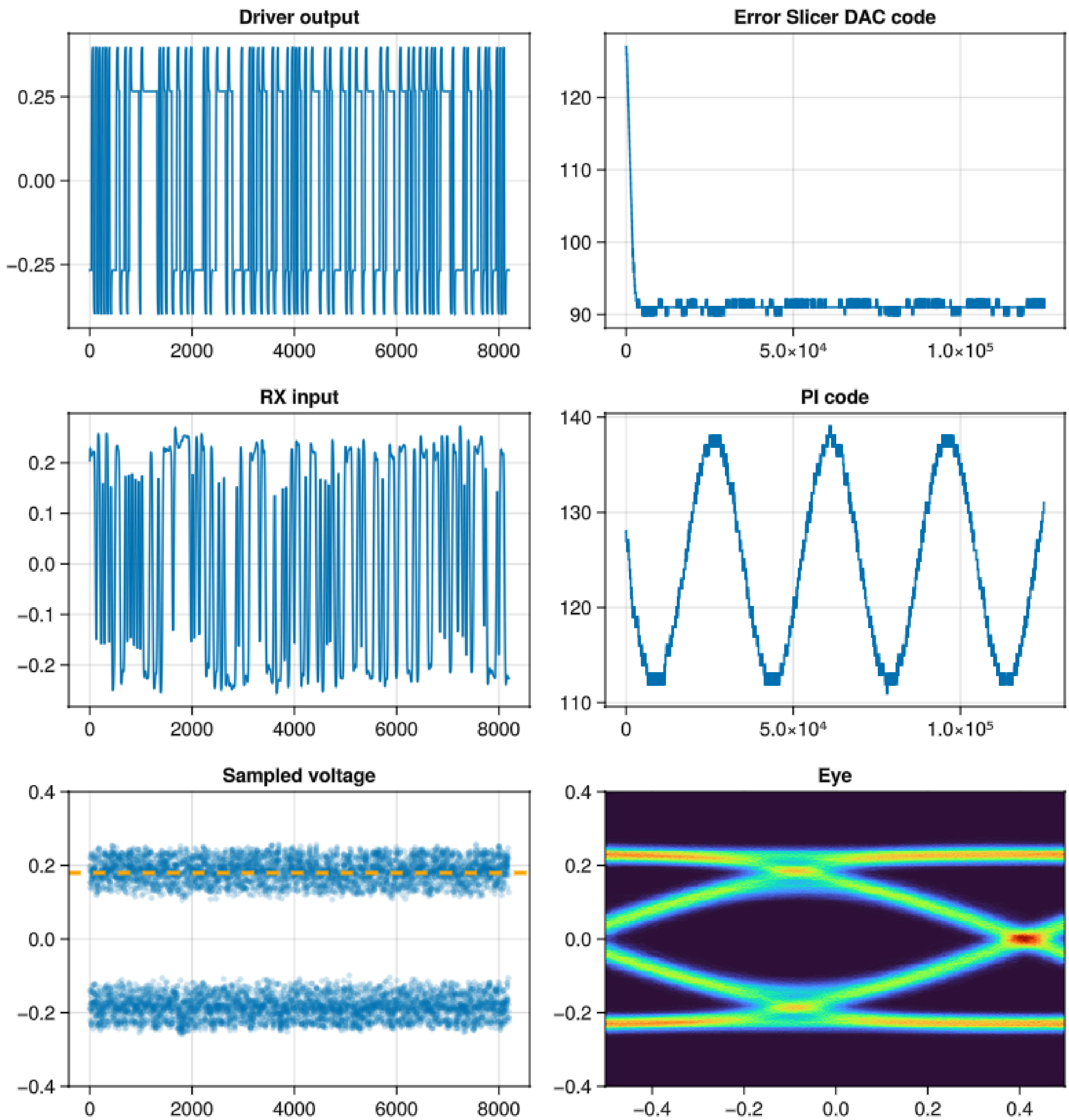
File/Code Structure

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
0.25
0.00
−0.25
0
2000
4000
6000
8000
Error Slicer DAC code
120
110
100
90
0
5.0×10⁴
1.0×10⁵
RX input
0.2
0.1
0.0
−0.1
−0.2
0
2000
4000
6000
8000
PI code
140
130
120
110
0
5.0×10⁴
1.0×10⁵
Sampled voltage
0.4
0.2
0.0
−0.2
−0.4
0
2000
4000
6000
8000
Eye
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4

Helper functions

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

```julia
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
```

ber_check_prbs! (generic function with 1 method)

```julia
function ber_check_prbs!(bist)
    @unpack polynomial, inv, chk_seed, Si_bits = bist
    nbits_rcvd = lastindex(Si_bits)

    # Uncomment if you want to add artifical BER
    # err_loc = rand(Uniform(0,1.0), nbits_rcvd).< 1e-4;
    # Si_bits .= Si_bits .⊻ err_loc

    if bist.chk_lock_status
        ref_bits, chk_seed = bist_prbs_gen(poly=polynomial, inv=inv,
                                           Nsym=nbits_rcvd,seed=chk_seed)

        bist.ber_err_cnt += sum(Si_bits .⊻ ref_bits)
        bist.ber_bit_cnt += nbits_rcvd
    else
        for n = 1:nbits_rcvd
            brcv = Si_bits[n]
            btst = inv
            for p in polynomial
                btst ⊻= chk_seed[p]
            end

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

            chk_seed .= [brcv; chk_seed[1:end-1]]

            if bist.chk_lock_cnt == bist.chk_lock_cnt_threshold
                bist.chk_lock_status = true
                println("prbs locked")
                ref_bits, chk_seed = bist_prbs_gen(poly=polynomial, inv=inv,
                                                   Nsym=nbits_rcvd-n,
seed=chk_seed)
                bist.ber_err_cnt += sum(Si_bits[n+1:end] .⊻ ref_bits)
                bist.ber_bit_cnt += nbits_rcvd-n
                break
            end
        end
    end

    return nothing
end
```

int2bits (generic function with 1 method)

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

Drv

```julia
@kwdef mutable struct Drv #dummy struct for TX driver
    param::Param
end
```

Ch

```julia
@kwdef mutable struct Ch #dummy struct for channel
    param::Param
end
```

Cdr

```julia
@kwdef mutable struct Cdr #dummy struct for CDR
    param::Param
end
```

```julia
using Makie, CairoMakie
```

```julia
using UnPack, DataStructures, Random, DSP
```

```julia
using BenchmarkTools
```