LabVIEW Universal Transcriptor

Lubomir Jagos

somewhere in Universe

2018

“I want thank to my older brother Miroslav who provided some funds for beer at time when this whole project was just idea.”

Content

1 What's this?

Have you every imagined when working in LabVIEW it would be cool have some tool to translate your code for some embedded device? Yes.. this is what you wanted, I know that, because I think about the same for long time until I finally implemented it.

This is transcriptor (on internet also mentioned as transpiler) from LabVIEW to C++ for Arduino, ie. it takes as input VI and returns it's C++ representation which is possible to deploy to embedded devices. It's exactly as it's written. There are some extra steps which has to be done to get right code in the end, so I programme them and test them to create basic customizable tool to translate LabVIEW diagram into requested form.

To show basic principle imagine blinking led diode program below on Illustration 1 translated into some structured C++ code for Arduino.

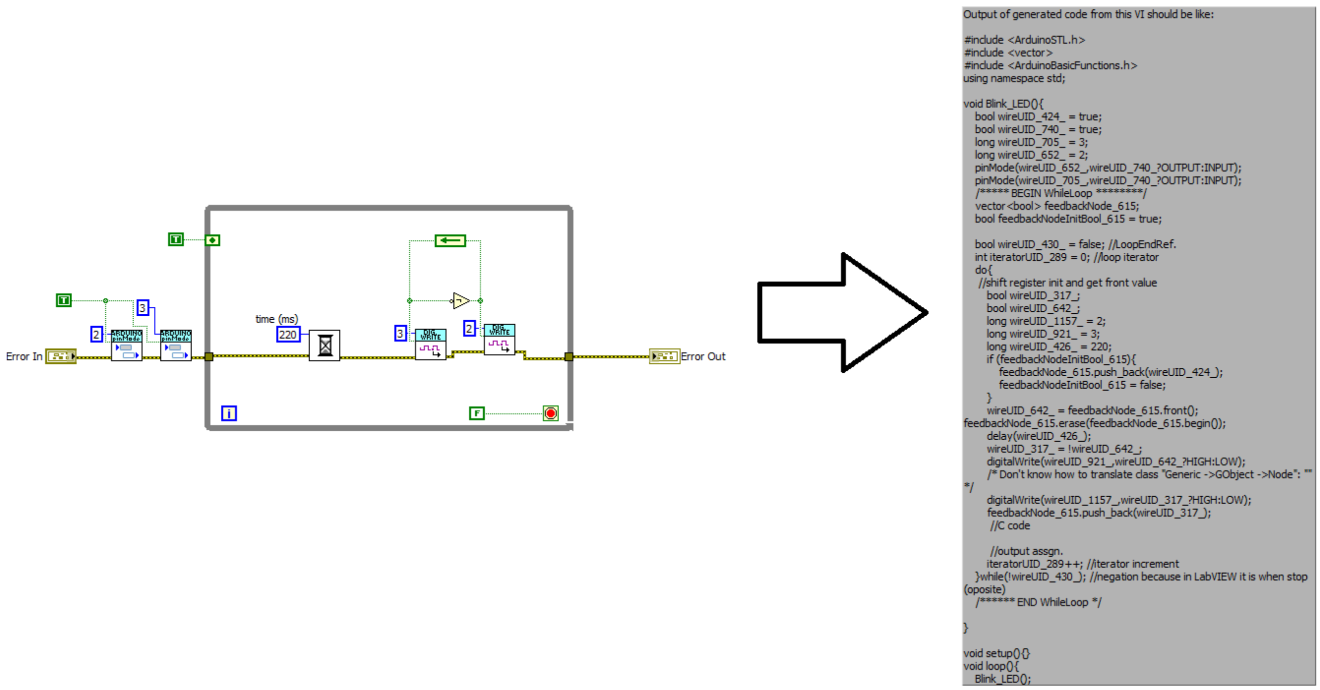


Illustration 1: Simple LED blinking LabVIEW program transcripting into C++ arduino code.

So if you implement some basic structure transcription process to translate diagram into code plus add some needed small function which handles tunnels, shift register you can somewho generates any code from LabVIEW. It sounds quite easy it's not rocket science :D but still there are some mistfalls, luckily they are shallow….. long story short, I made this to you to avoid them and show you how to do that and provide you some tool hopefully useful for you.

1.1 Why “Universal”

Because you can scale it into width and also into deep :D Simple said you can add you own libraries and extende its functionality or modified it’s source to transcritp into another target platform language. All dis up to you.

There are some chapters about creating own transcriptior dictionary, I tried to make easy for you, so you have to just create in folder with your Vis translator VI and that’s all.

I also implemented some special features, killer features, you can create your own classes and use own color for wires! And there is more…. (at least my faviourite test program on Illustration 2)

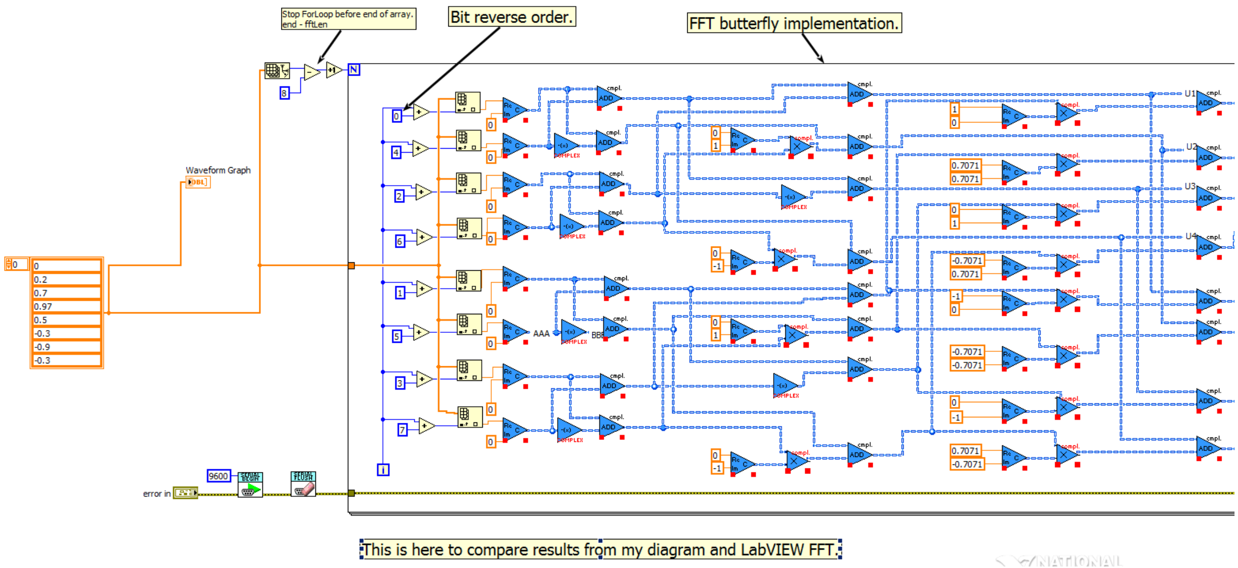


Illustration 2: LabVIEW FFT implementation for MCU.

As written above, you can define your own libraries, but still, you can modify whole transcriptor (I prefer way to add folder into Transcriptors/ folder number 3,4 or anything else, copy basic frame and starts to modify) and target it to any language, as proof of concept my option is C++ for arduino to show how to translates diagram for it.

1.2 There is performance issue, transcripted code cannot be faster than hand written

True and false at same time. Always depends on programme application. I wanted to show you that’s it’s possible to transcript LabVIEW diagram into C++ (but that we all know, because NI is doing this for years), but mostly to create xample application.

Generated code has one advatage and that’s it has same structure over whole application which means it’s really easy to do some analysis and optimalization afterwards. So at first stage I concerd mostly for functionality ignore performance question, these should be handled by some code optiamlizators after transcriptor proces.

2 Implemented functionality

Let's starts with something positive :) To gain your attention I would like to introduce thorought this whole document conecpt and implementation of my transcriptor for LabVIEW VI into target platform code, for beginning chosen C++ because of Arduino.

Really, I'm not lying, this is it and it's here to show you how to done this. So quick review what implement this transcriptor, in other words what could be seamlessly transcripted into C++ code:

- structures
  - while loop
  - for loop
  - case structure
  - flat sequence structures
  - feedback node
- 1D arrays
  - initialize array
  - insert into array
  - delete from array
  - index array
  - array size
- basic string functions (length, subset, replace, conversion functions from numeric to string)
- basic numeric functions (add, multiply, divide, increment, decrement, conversion functions between different numeric datatypes – integer, byte, word, …)
- basic boolean functions (not, and, or, xor, not xor)
- cluster functions
  - not named cluster - bundle cluster, unbundle
  - named cluster – bundle, unbundle cluster
- comparison functions – equal, greater, less, ...
- system serial line functions – availability, print, println, read, …
- digital output – read, write and set mode
- analog output – read, write, analog reference

- interrupts – attach, detach, no interrupts !since now this wasn’t completely tested, but some basic example is running

Then there is pallete ”Translate tools” which holds result this whole effort, there is located whole transcriptor written into one sub vi, calling it and passing it reference to VI which we want to transcript is all what we have to do.

Last pa section in pallete called “Scripting” contains Vis used to build this whole transcriptor, they can be used in users Vis to improve this whole tool.

This is basic functionality which is making this transcriptor usable to transcript at least VI “Hello world” and prove concept functionality. Transcriptor can be also extended by user and I interfaced into this transcriptor some another arduino libraries to make it useful at least for playing, to hold your attnetion they are:

- wifi implementation (yes, it’s what you think, you can programme ESP modules using this transcriptor)
- DAC (more specifically MPC4725)
- accelerometer & gyroscope module (sure, it’s MPU6050)
- complex numbers functions
- advanced math functions (trigonometry, hyperboli functions, …)
- DDS sinewave generator (for radioamateurs well known AD9850)
- SD card functionality
- Servo motor control (yeah, the blue one, cheap one)
- basci matrix math functions (invert matrxi and print matrix into serial output, this is usuable, it’s done in little creepy way, but it’s running)
- there is also library for LCD display, I haven’t test it
- there is also library for TFT display, this will be one day tested, it’s possible that’s is already running without any change, but I haven’t time to check it, feel free to try it and report me :)

3 LabVIEW Under Hood

source:http://www.ni.com/tutorial/11472/en/

One time during my internship in NI through summer 2013 I found on internet this document describing LabVIEW. I thought it's quite short and there should be written more about language because it's not possible to have whole language specification so short. Most informative picture from that document is on Illustration 1.

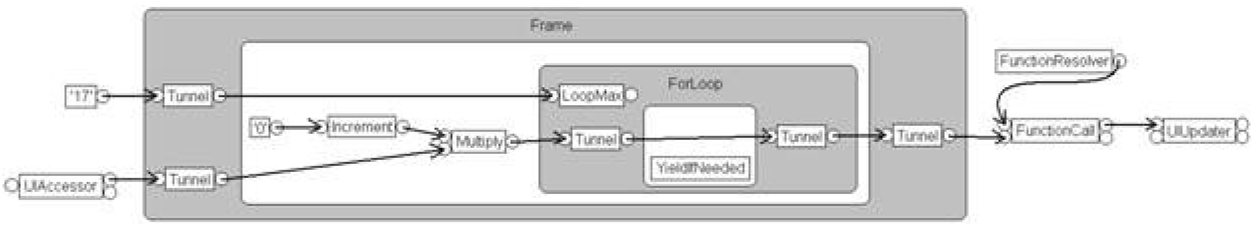


Illustration 3: LabVIEW under hood, DFIR principle

Since I started worked on transcriptor I was looking for VI Scripting library and was discovering true nature of LabVIEW I figure out, that document is really true :) (how surprise) and that LabVIEW program representation is just DFIR (Dataflow Intermediate Representation).

From DFIR is later produced machine code using LLVM compiler. I had no clue about LLVM compiler so I started to read some informations on wikipedia.

4 The Principle

At the beginning there was thought how to simply transcript diagram into C code. I was thinking about each SubVI as separate function waiting for its inputs, then process them and provide some outputs. So I started with simple transcription process described on next image Illustration 4.

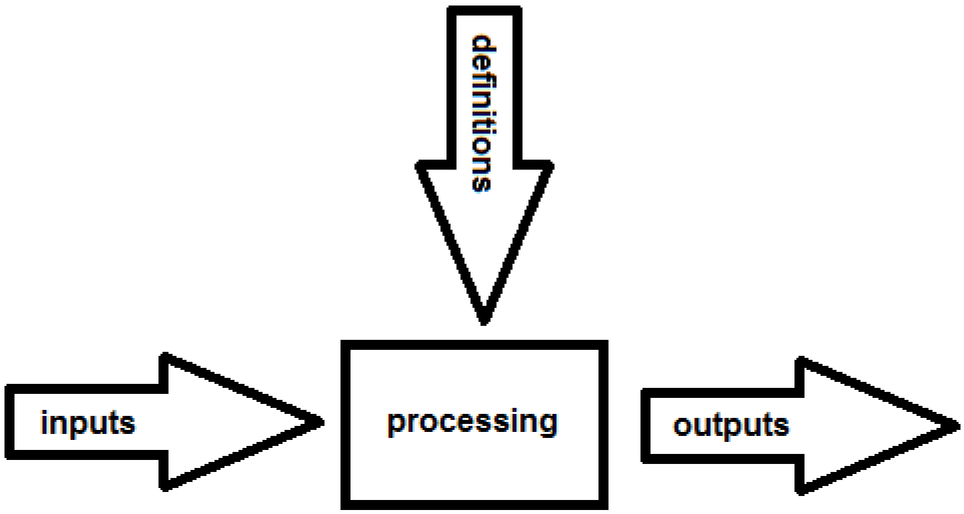


Illustration 4: SubVI processing

Each process on image is separate, doesn't matter on order step, just one rule has to be fulfilled, that to process step all its inputs has to be know at the beginning (that's dataflow programming principle).

So after I looked at image I decided to split transcription process into these steps:

0. creating definitions for variables which are not essential to target language (structures, user typedefinitions)
1. creating known constants and values (variables, ...)
2. create mechanism to pass values (constants and input params) into nodes
3. getting input values for particular node and process data
4. outputs data

These steps creates base stone for whole transcriptor and whole application is written around them. I named this chapter four step process because as shown later, steps 1. and 2. could be combined becuasue relevant values are just values which will be passed to nodes for processing (or somewhere elese for processing), so I combined them in result transcriptor into step 1. and 2. (and I think about them as one step :)) and they are implemented in one VI.

After I defined basic rules for transcriptor there is need to handle all special stuff, because transcripting LabVIEW to some target code is not just about translating each SubVI into function and then call function, but we have to transcript all elements which was used to draw block diagram which includes for basic programs tunnels, while loops, for loops, other loops, shift registers, controls, constants on diagram and basically all stuff which we want to be supported by our transcriptor.

When we look at Illustration 3 we can see, that as simple example block diagram can be represented as on Illustration 5 which shows how LabVIEW interprets program in memory. This model of program in memory is accessible in LabVIEW by “VI Scripting”, this pallete is located, see Illustration 6, and using property method for block diagram elements.

VI Scripting is in default turned off, so we have to activate it to see LabVIEW system object properties when using property nodes and invoke nodes, enable Vi Scripting is done enabling options in “Tools → Options” and then in tab “VI Server” both options should be checked, Illustration 7.

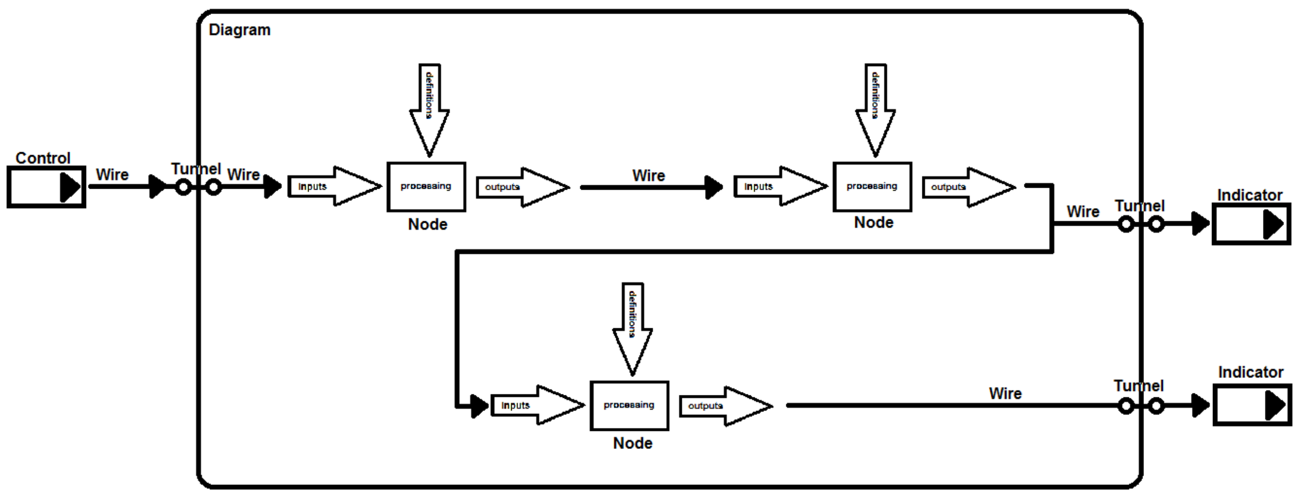


Illustration 5: DFIR program representation in LabVIEW memory

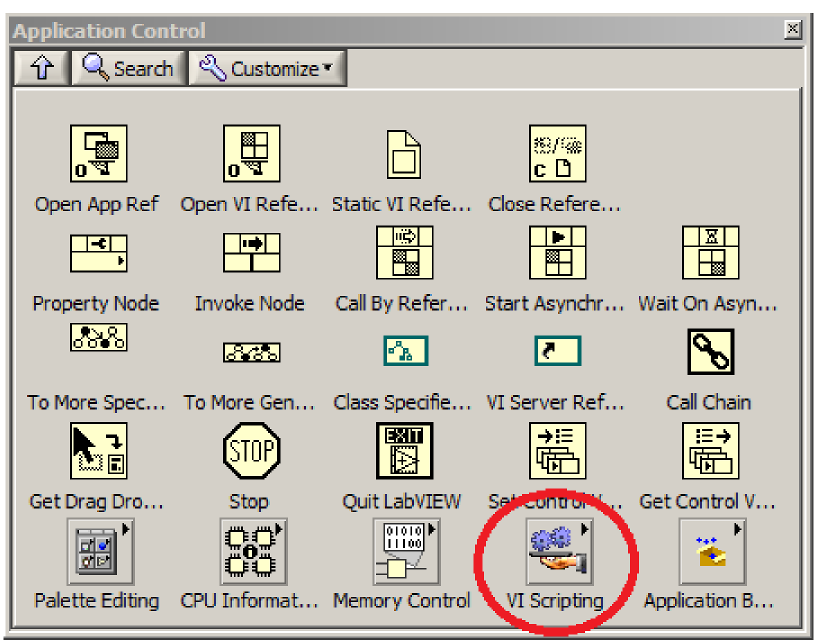


Illustration 6: VI Scripting pallete

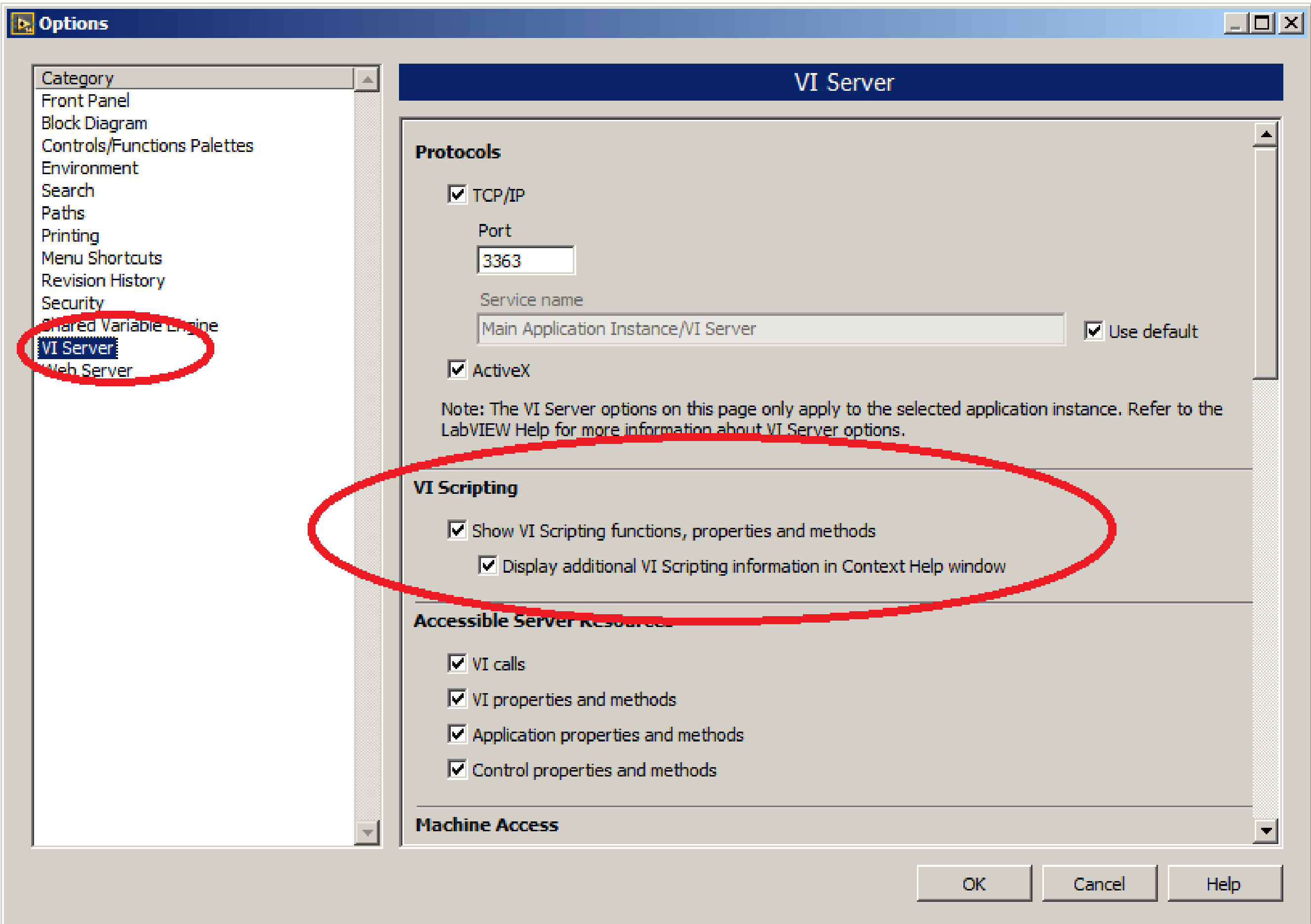


Illustration 7: Enabling VI Scripting in LabVIEW environment, also enabling see additional system classes and their properties when using ie. "To More Specific Class" or when using property nodes with LabVIEW system objects

For beginning I set goal to at least make LED blink and communicate through UART. This makes tasks for implementation more clear:

1. Loops – need to implement basic loops as while loop, for loop
2. conditional – means implement if{}else{}end statement and switch(){case: …} statement
3. SubVI translation
4. shift registers and feedback nodes
5. structure implementation (includes clusters)
6. tunnels implementation

These things I defined as priorities for implementation and started on them still having four step transcription process in mind.

Most of time I'm working in transcriptor code with these elements reading their values (I'm transcripting LabVIEW into C++ so 95% of time I'm just reading properties of elements, only in few rare cases I’m creating new objects) using VI “To More Specific Class” and then access their properties.

Hope that you are little familiar with object oriented programming (known as OOP), because DFIR is using classes and each element in LabVIEW is part of LabVIEW internal class hierarchy.

Example of using “To More Specific Class” and how child classes are extending parent class is shown on Illustration 8.

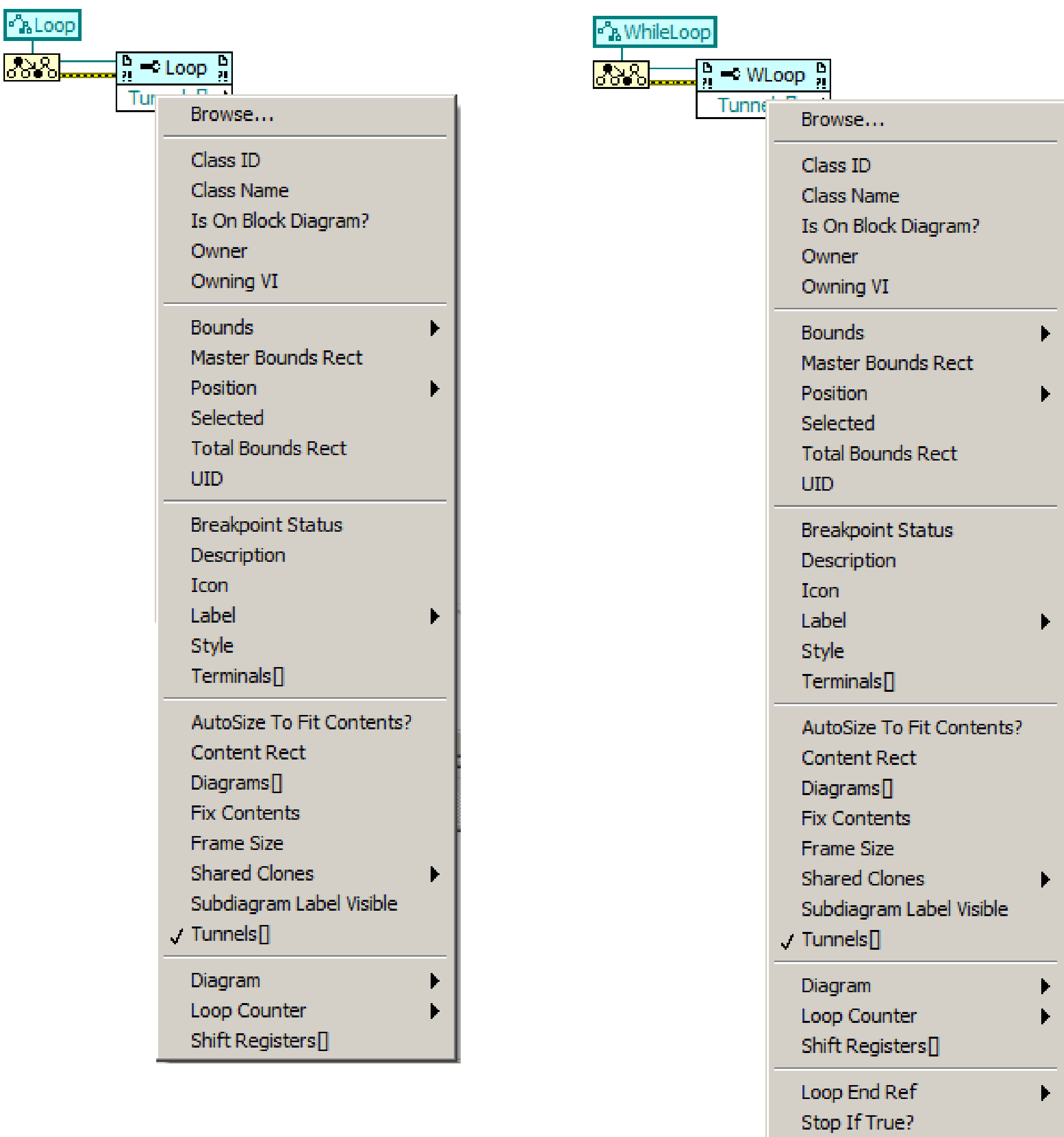


Illustration 8: Example how to use "To More Specific Class" and example of new properties of child class WhileLoop (parent class is Loop)

5 Target code platform

First we need to think about target language. This whole idea appears when I saw first few times LabVIEW and starts working with it, I was always looking at it and was thinking how cool it will be to have some possibility to convert it into embedded device. It interested me more and morse since after I started to play with arduino. So I choosed C++ as target language for LabVIEW transcripting, because I want to be able program arduino using LabVIEW.

At some places on internet there is always mentioned that arduino is using arduino code, but this is not true, arduino environment is using C++ with some macros plus environment itself is adding needed libraries to code and defines few macros, but still using C++ programming language.

Arduino is great platform, it's open and it's targeting a lot of low cost electronic HW and modules and what's best is that arduino itself is using C++. This way when transcripting LabVIEW into arduino compatibile C++ I can reach lot of HW platforms as on Illustration 9.

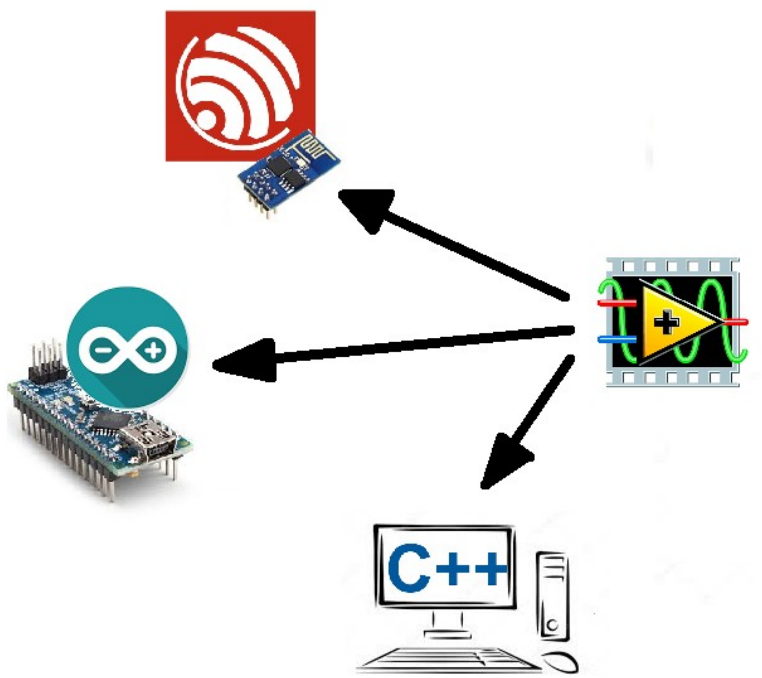


Illustration 9: LabVIEW to another platform using arduino HW platform

Some functionalities are programmatically not optimalized (almost nothing is :D) beacause this is more demonstration and try if it's somehow possible (and looks more and more that it is), for everything else there is somewhere some optimalization hopefully, but still you can optimalize generating code process in LabVIEW, or you can write some text processor which will optimalize result generated code. You have freedom and you can choose, result will be similar, that's point of that all.

Since I’m working in company I like working with GIMP and creating images, so here is one just for fun, because page full of text is really boring (I started to like comics style of creation pictures, it could be also nice done in LabVIEW)

Illustration 10: LabVIEW hero

6 LabVIEW datatypes, their conversion

LabVIEW native datatypes have inside LabVIEW different names as datatypes in C++. To translate datatype names correctly I put on every place in transcriptor where generating some datatype names prefix specified in “types\LabVIEW Datatype Prefix.vi” which output is my prefix.

At the end of transcription process there is used string function for replacing text to replace all match datatype names which contains my prefix as on Illustration 30.

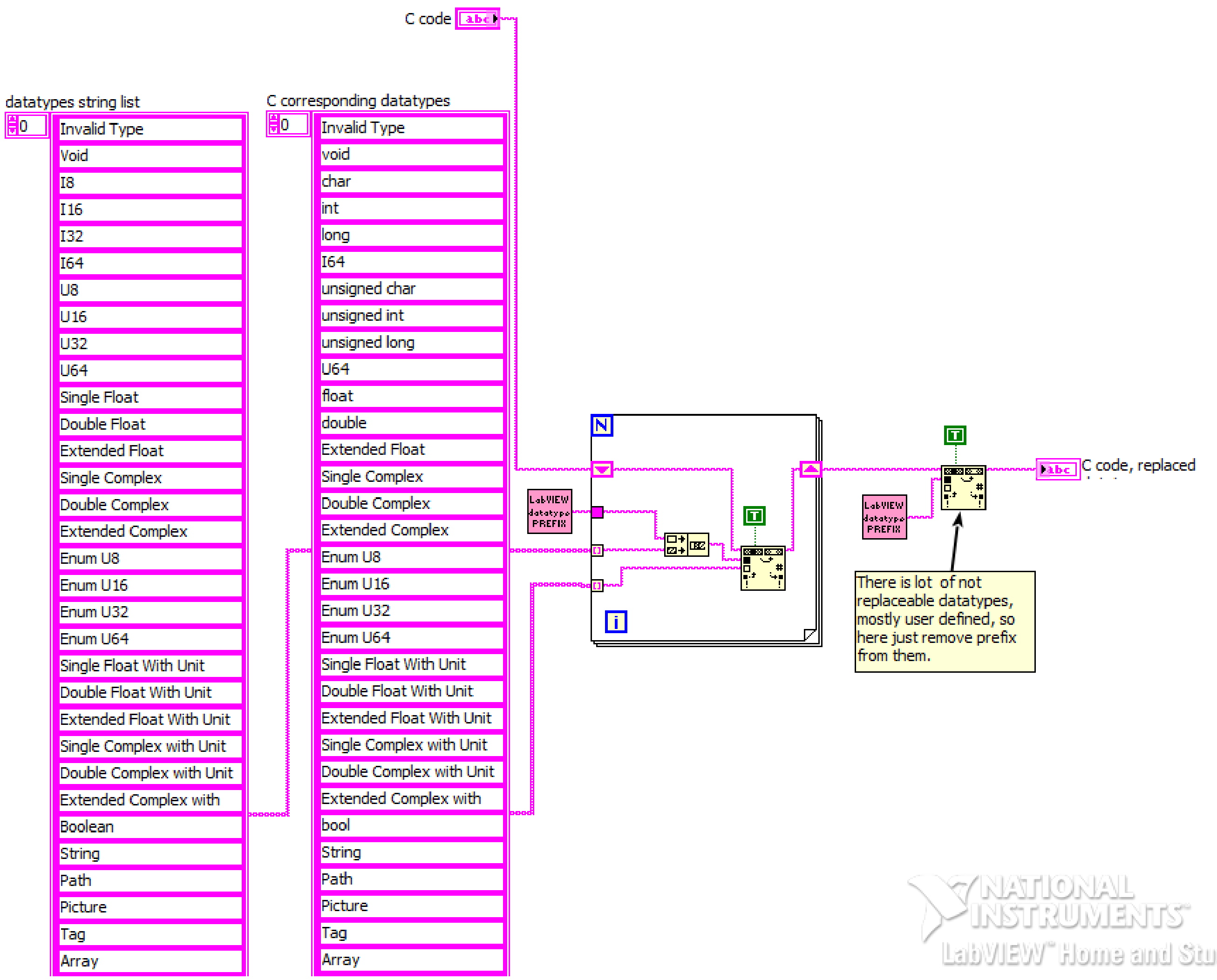


Illustration 11: Replacing LabVIEW datatypes with C++ datatypes, for this purpose was implemented "String Replace LabVIEW Data Types With C types.vi"

7 Converting Variant to right datatype

Most of time during transcription process objects returns their value as Variant, what in LabVIEW can represent any value and we need to get right value, but for this we need to know what was inserted into Variant. There is no VI which is able to do this in normal palletes.

During inventing way how to solve this problem, I found most easiest solution. NI engineers must be using for this some function, so I was looking in *vi.lib* directory and found inside directory *"vi.lib\Utility\VariantDataType"* lots of function dealing with Variants and one of them is *"Get Type Info.vi"* which tells you datatype of specified Variant.

On Illustration 30 you can see block diagram of transcriptor VI for obtaining Variant datatype. Case on illustration is for normal native LabVIEW datatypes, for some specials (array, cluster, error cluster) there has to be done few more steps, so there is some logic how to deal with these datatypes to translate them right.

Datatypes for Variant are returned in array for case there is needed to provdie more values at output, this is use ie. for error, it's returned string "error" and next element is "cluster", because error in LabVIEW is ordinary cluster.

(I'm using error wire and hopefully LabVIEW also for execution order, so I'm always excluding it from everything as wire creation, ...)

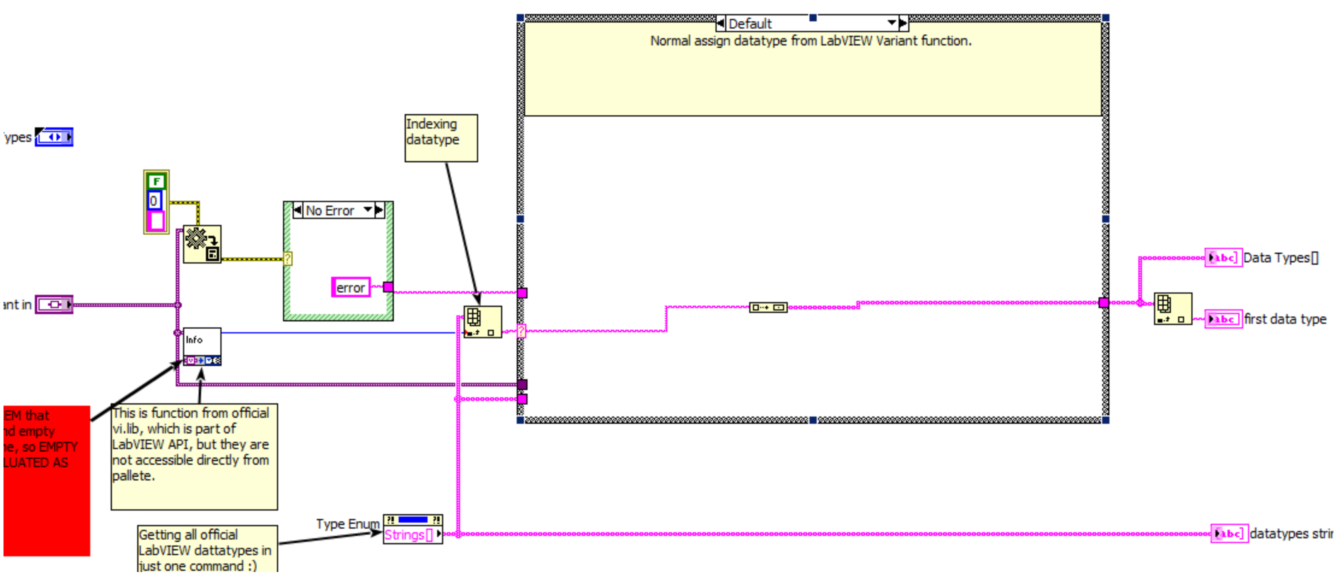


Illustration 12: Obtaining Variant datatype using "hidden" LabVIEW system library for Variants

8 Array Implementation

Arrays in LabVIEW behave identically to vectors in C++ (they are not static size). For this I have to find some library which implements vectors for arduino. I found library ~~*StandardCplusplus* and add it into transcriptor to be able use vectors~~ , but because it's not full implementation some things weren't running always correctly due to it's non-complete implementation , so in the end I used ArduinoSTL. If you see somewhere StandardCplusplus library, please remove it from there, it could casue some problems (ie. shift registers are using vectors, so there could be some unpredictble behaviour because of this library).

In code then for array datatype I generate code snippets like "vector<String>" or "vector<int>".

~~Multidimensional arrays are now problem, I haven't test if shift register and other stuff is OK with them, if program with them not just compile but also run smoothly.~~

Array functions were implemented for now in mind 1D arrays. Still syntax should be correct, so program which uses 2D arrays will be syntactically correct, array are handled as vector<vector<[datatype]>> so theirs definition is alright, indexing in 2D array using just one dimension is also ok (ie. "twoDimArray[42] is valid) but I haven't test it, so in some little more complex programs it doesn't have to necesssaraily be correctly generated.

Anyway transcriptro dictionary for arrays is located inside "Transcriptors/2/Transcriptor LabVIEW to C code Translator Native Functions.vi" so feel free to improve this.

Implemented functions are on Illustration 13, they are running for 1D arrays now, still waiting for right testing.

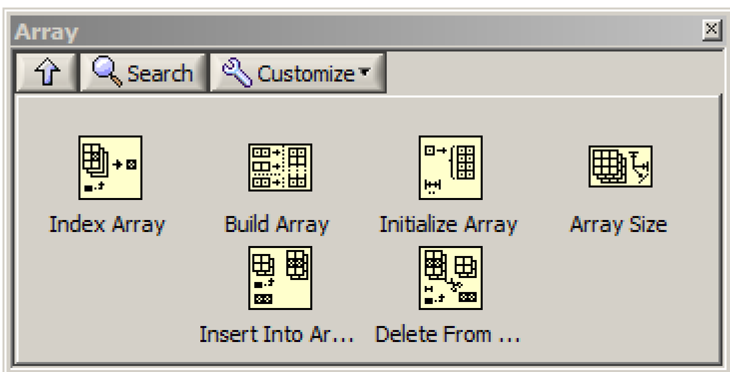


Illustration 13: Array, briefly imeplemented functions, for now for 1D arrays, not tested for 2D arrays

9 Four step translation processing

This is handled in particular transcriptor engine. On next picture you can see block diagram of “Tanscriptor LabVIEW to C code.vi”

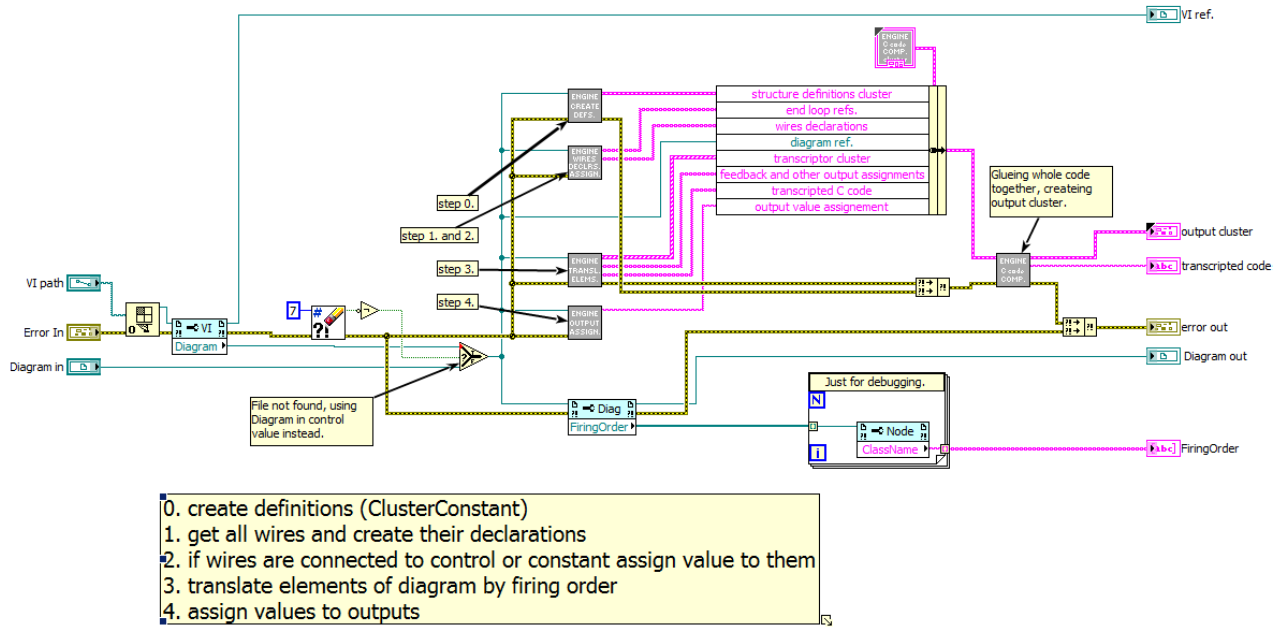


Illustration 14: Four step translating process

Next all part will be discussed more specific.

As it was already mention in chapter 4 and what is showing at Illustration 5 every diagram is just network which can be traversed and transcript into some textual form. As form for beginning I choosed C++ fo arduino but it can be any language in basic.

As was already written in chapter 4, thae process consis from 4 steps – creating definitions (for variable datatypes), creating wires variables in C++ declarations and assigining values, translating elements (SubVI, FeedbackNode, Loops, …) into C++, assigning output values.

9.1 Step 0 – Creating definitions

Every programming language contains natural datatypes as numbers of different kinds (integer, real, float point, fixed-point numbers, …) text (string, ...) and some others (files, …) but except these there used to be possibility for user to define its own datatypes which can compose from this already well known and compiler/interpreter implemented datatypes.

In LabVIEW user is allowed create new variables datatype this way using cluster or class. These two are in C++ represented by structure and class, so there just need to define how to translate each of them into C++ representation.

At both elements (cluster and class) I had to do some decision so here are more specifically described.

9.1.1 Creating Cluster definitions

Cluster in LabVIEW is equivalent of structure in C++, so creating definition for cluster should be pretty straightforward. At start I look what all is related in LabVIEW with clusters, to know how many things I should implement.

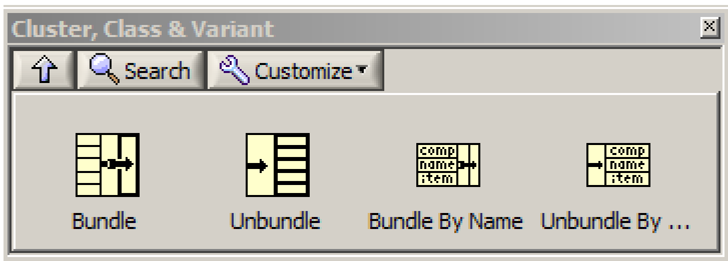


Illustration 15: LabVIEW Cluster functions

For cluster was needed to implement four functions above. The main difference between all of the functions is, that Bundler and Unbundler are working with cluster items without specifying their names and Bundle By Name and Unbundle by Name are using items names to access them.

At this point I was little confused, because I wasn't sure, if I'm able to get cluster item names everywhere in diagram in LabVIEW so I tried next experiment.

I created in LabVIEW case structure with two case true and false, doesn't matter, inside each of them I put cluster with same datatypes in same order but different names and lead wire out of case structure for indicator.

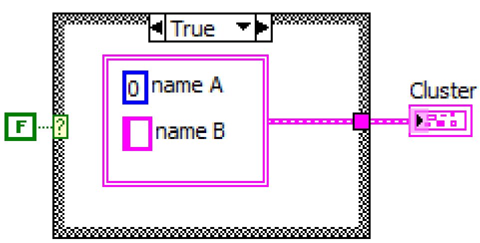


Illustration 16: Case structure case true

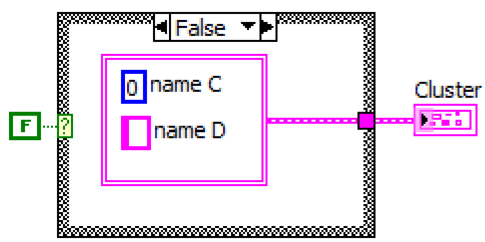


Illustration 17: Case structure, case false cluster with same datatypes as before, but different names

After then I clicked right mouse button on wire outside case structure and create constant, I got cluster with items as you can see on next image.

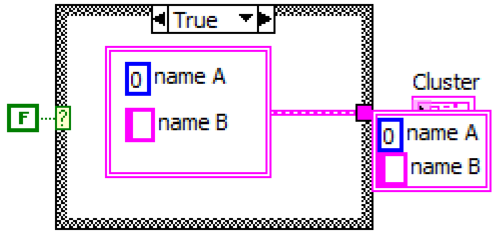


Illustration 18: Created constant using right mouse button on wire outside case structure, pay attention for items names

Then I clicked on cluster wire inside false case and created constant, I got another cluster as you can see below.

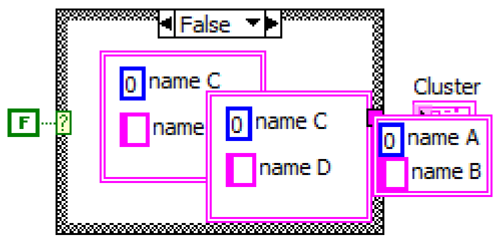


Illustration 19: Cluster constant created inside case structure, look at items names, they are different

So basically we cannot know during traversing diagram what will be cluster items names (this “name uncertainity” is more obvious when creates cluster, connects som cluster to it, then remove wire and use create constant on bundler terminal), instead we have information about cluster items datatypes and their order.

Then I was thinking if cluster are equal, does it means, that their items datatypes are same in same order and doesn’t matter on their names or does it? I couldn’t decided, but then during testing generated code compilation I got error when I was assigning one structure into variable which was defined with other structure which was identical by datatypes, they were differencies just in items names, see Illustration 20.

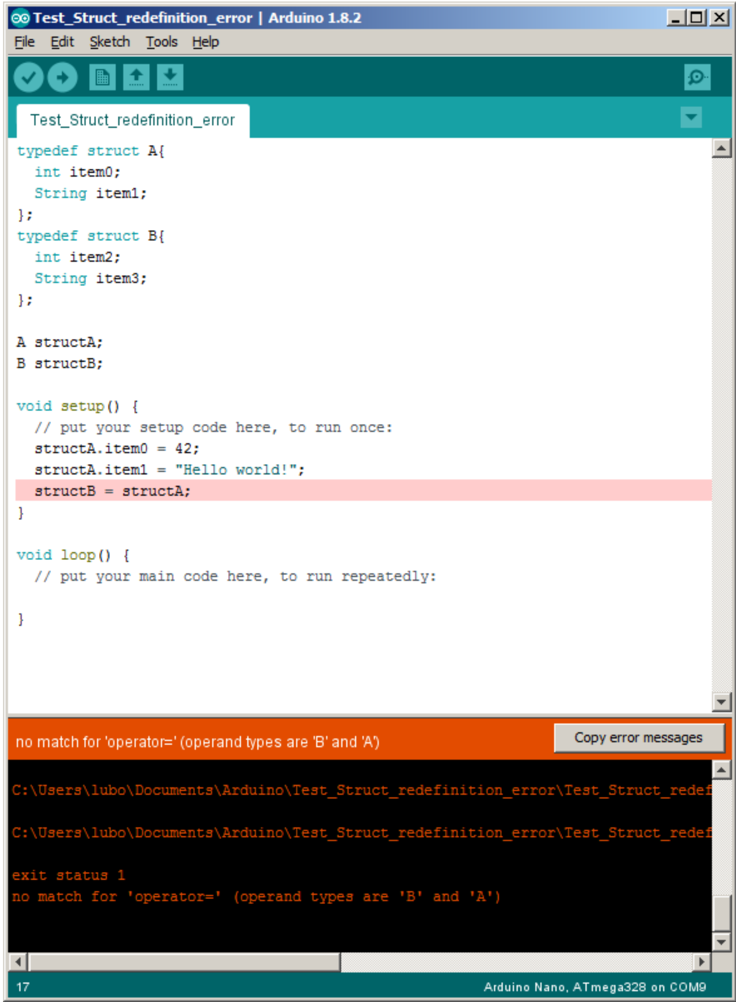


Illustration 20: Arduino compiler showing error because assignement different structures (datatypes are same, but items names inside structure are different)

So there is needed to decide how to generate cluster definitions, how to named their typedefinitions and how to named items.

9.1.2 Cluster typedefinition name, items names

~~This problem is not solved totally until now. I put into code VI “utils\Cluster Generate Datatype Name.vi” which has on its input two arrays one for item names and second for datatypes, for now I’m using just datatypes, so if cluster is not typedef its name is autogenerate from datatypes divided by underscore ie. generated cluster datatype on illustration will Illustration 21 be LVcluster_I32_Double_Float_ClusterTypedef1~~

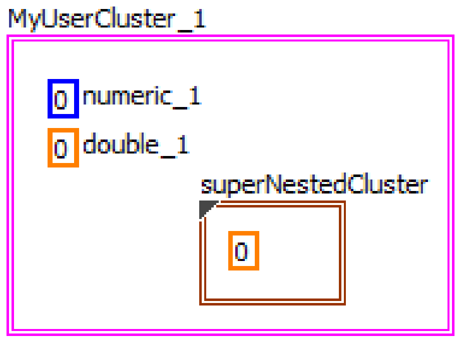


Illustration 21: Example cluster, generated datatype name is LVcluster_I32_Double_Float_ClusterTypedef1

~~Here is quite problem, because same datatype will be generated for cluster on Illustration 22.~~

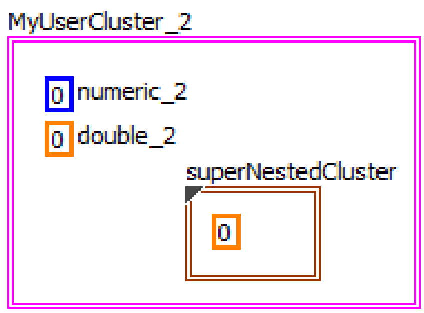


Illustration 22: Another cluster with same generated datatype as cluster before.

~~So until now this problem remains, but code s compilable, it will be fixed later.~~

~~One of possible solution is use item names for cluster datatype name generation and in bundler, unbundler used retyping, LabVIEW IDE takes care, that there are connected just reliable clusters, so retyping will be just to conserve name consistency, it adds some overhead to code but what to do anyway.~~

Cluster typedefinitions are whole transcriptor process create using just their items datatypes and stored in array, they are written int result C++ in the end of process in code composer. So two code chunks above will produce same definition for cluster (they have items of same datatypes) and there will be duplication in cluster defionition list, for addressing cluster items when using named bundler/unbundler is used algortihm which will find items indexes and calculates item reference at time of transcripting bundler/unbundler, so cluster item names are no important. Duplications which were gathered by using cluster containing same datatypes are removed from cluster defitions list in code composer VI at the end of transcription process and everything is correct.

If cluster is tpyedefinition it's datatype structure is still generated by transcriptor, because it's needed to have this definition for bundlers (not for named unbundlers, they are using pointer arithmetic).

There is generated cluster definition with default values:

```
typedef LVcluster_Single_Float_Single_Float_Singl_Float{
        float LVClusterItem_0;
        float LVClusterItem_1;
        float LVClusterItem_2;
}
#define typeDefName  LVcluster_Single_Float_Single_Float_Singl_Float
```

~~Transcriptor then defines typedef as alias of this structure, so whenever in code this typedef when is used it's name is replaced by generated structure name. This definitely need to be tested, because there can be some cases when this screw whole code because duplication or incompatibile types.~~

Transcriptor generates typedefinition for all cluster even for typedef ones. It's because clusters are used in LabVIEW and they are not mechanism to define new datatypes, but to define clusters. This way, there should be no problem if their definition is generated in the beginning of file.

If you want to define own datatype, please use class mechanism described below.

9.1.3 Class as datatype

I implemented translating cluster into C++ structures and then was about using LabVIEW classes, I was using object oriented programming before in other languages, never in LabVIEW, I looked how to use them and it’s same as in C++, but there is another thing which is great for classes in LabVIEW! You can create own wires and give them color and choose from patterns.

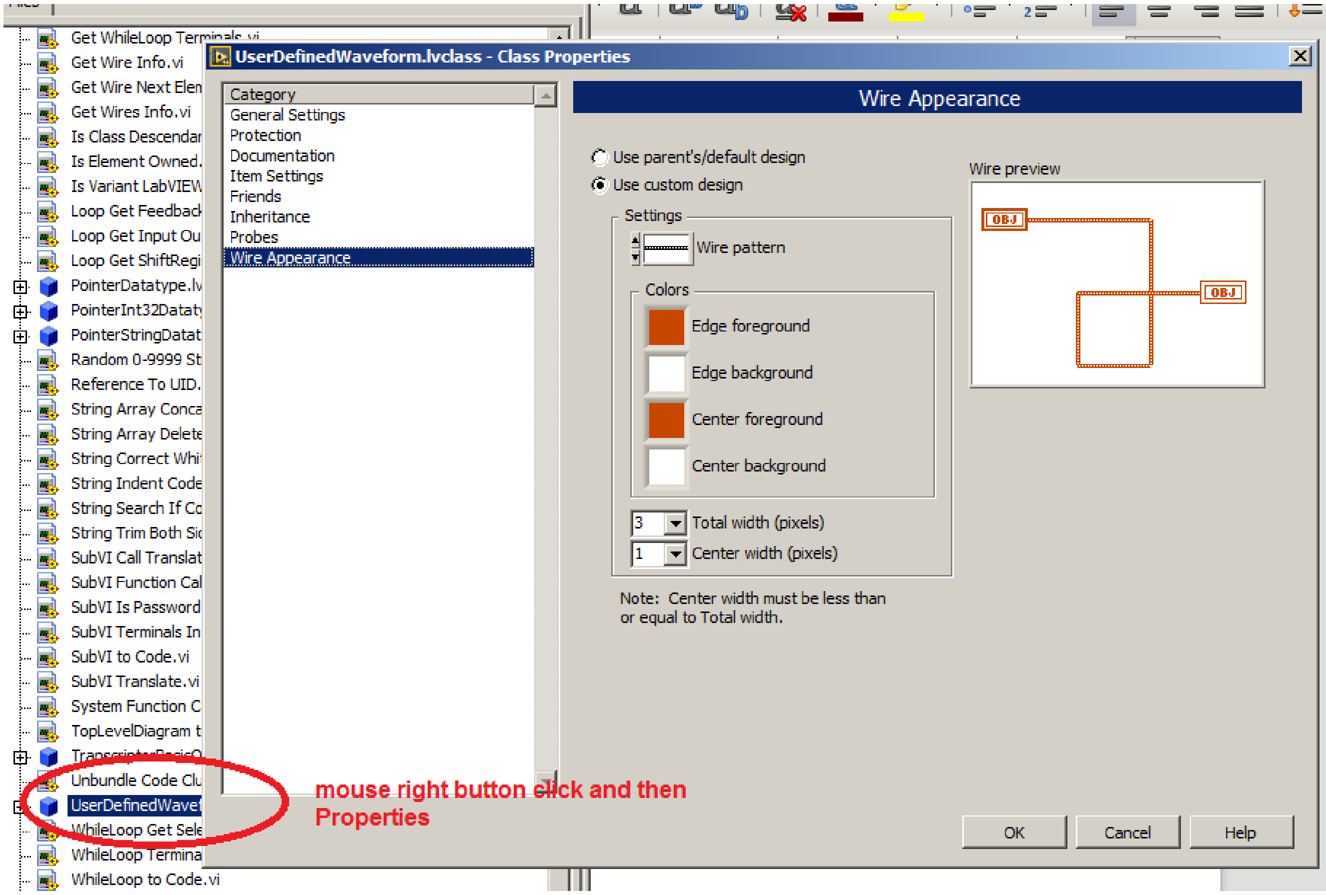


Illustration 23: Changing wire color and pattern for LabVIEW class.

This is like dream of everybody who is working in LabVIEW (but without classes you will get nice pink looking diagrams mostly :)).

Until now I implemented basic root class as I know from Java, there is class Object, so I put some origin and other classes in utils are descendand of this class TranscriptorBasicObject.

This class contains methods which are used in transcriptor ie. when getting datatype for some item (transcriptor took variant, look at if, if class retype it for TranscriptorBasicObject and after call generic method getDatatype() which returns string with datatype of specific class inside).

So I think that class can be used as new datatype, anyway calling their methods is through SubVI, and transcripting SubVI is already implemented so there is nothing what I know for now that can cause some errors.

I haven't time to test if this feature is generating proper code which is able to compile and not even some example so this still wait for evaluation (as whole transcriptor).

In utils you can find some already implemented classes, I was playing trying to figure out during development if it's running or not and still to have some examples how to setup class to provide right result. You can find there now these classes:

- PointerDatatype
- PointerI32Datatype
- PointerStringDatatype
- UserDefinedWaveform (this one is really for fun, waveforms are still not implemented at all :D)

9.1.4 ShiftRegister FeedbackNode definition

These two elements Illustration 24, Illustration 25are almost equal, there is just subtle difference in LabVIEW in initialization (shift register is initiliazed outside loop, feedback node can be also, but if not, then first sample can be undefined – or from last execution). Anyway functionality of these elements is delay data datapropagation by user specified delay. In C++ is this similar to pushing data into queue with exact size and then wait on its other end for data.

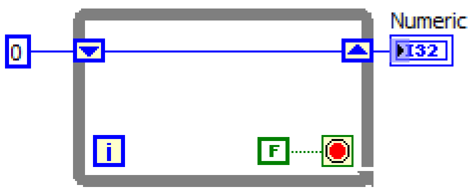


Illustration 24: ShiftRegister on loop

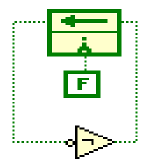


Illustration 25: FeedbackNode example

C++ provides for this behaviour great tool – vectors. Instead of queue I used vector. LabVIEW environment takes care, that shift register is always wired, if not, code is not executable, so in generated code, if I initialize shiftregister (or feedback) and at loop begin I take on element from shift register beginning (or feedback node) and at end of loop push one element at end, this system will run with overflow or underflow. Vectors in C++ behaves like length variable arrays.

So to implement this system I have to imeplement these things:

1. creating shift register/feedback node definition in front of code block (in LabVIEW diagram) where they appears
2. initialize register/feedback node inside their code block
3. take element from vector front and erase them (other elements pushed closer to vector begin after this delete)
4. at loop end push element into vector back

In step 0. (which runs for every diagram) is implemented 1. step of creating register/feedback nodes, their definition is created like on:

```
vector<String> feedbackNode_143;
bool feedbackNodeInitBool_143 = true;
```

Notice there is bool variable which later serves as flag for initialization.

Second step is always generated inside loop (while loop, for loop) and looks like this:

```
if (feedbackNodeInitBool_143){
    feedbackNode_143.push_back(wireUID_289_);
    feedbackNodeInitBool_143 = false;
}
```

After pushing elements into vector, flag bool variable is set to false and this block is not executed anymore.
There is problem that initializer doesn’t has to be necessarily wired, so in that case flag variable is set to false and there is no initialization (no pushing elements into vector) and generated code probably compiled, but there will be runtime error memory access! This will be solved later.

Then inside code block there will be generated somewhere at beginning getting first element from vector and at the end of code block will be pushing element into vector’s end:

```
wireUID_232_ = feedbackNode_143.front(); feedbackNode_143.erase(feedbackNode_143.begin());

feedbackNode_143.push_back(wireUID_202_);
```

Described generated code was taken from testing generating code from “Example Feedbacknode 1.vi”, generated code for shift register is same except word *feedbackNode* is replace by *shiftRegiste.*

Code for shift registers is generated at step 3. at translating loops, because shift register is always placed just on loops (while loop, for loop) there is created special VI for this named “Loop ShiftRegister Get Input Output Assignements.vi”, see Illustration 26.

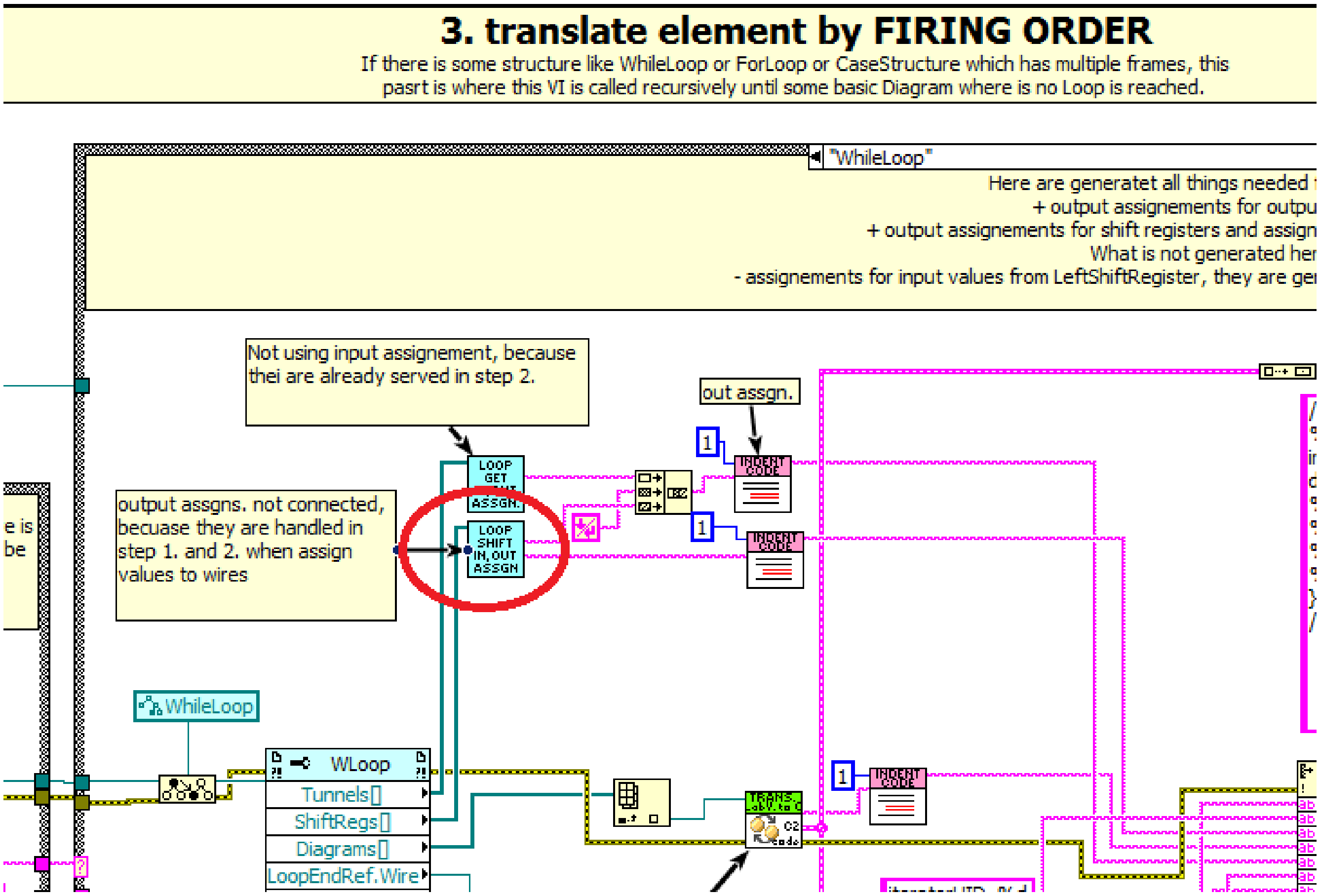


Illustration 26: Generating code for shift registers inside step 3.

These VI implements generateing code described before, on Illustration 27 you can see its block diagram, there are used just basic functions, so it looks like there is some complicated logic implemented but it just looks so.

Handling translating FeedbackNode into C++ code is done as separate case in step 3. because this node is separate element (child class of Node), while shift register is child class of Tunnel (Tunnel and Node are not in any relation). So feedback node is handled in separate case, but there is one implementation differecne compare to shift register. Because shift register is always part of some loop which has own diagram inside, its output assignement are done adding these assignements for shift register at the end of loop, see Illustration 28.

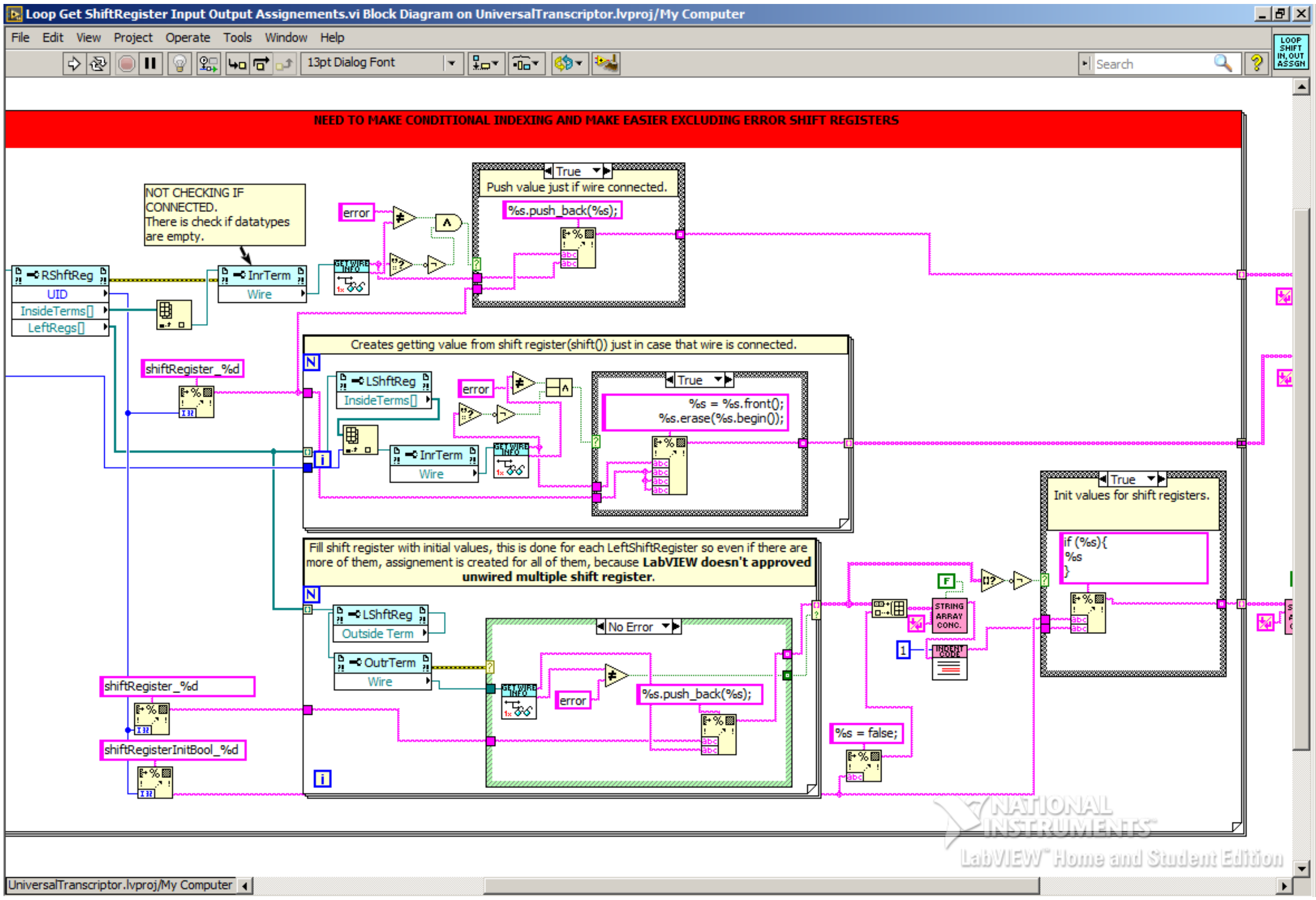


Illustration 27: Generating code for shift registers, initialization, obtaining elements on beginning of vector and pushing elements into vector

Feedback node is different in this, it can be as SubVI part of loop diagram or part of VI diagram or part of any other diagram, so pushing elements into it’s end must be done similar to shift register somewhere at the end of generated code for diagram. This is achieved by output code for pushing elements into feedback node from step 3. as item of cluster which holds informations about som code chunks during transcription process, it’s typedefinition (strict typedef. To change its definition at all places in my transcriptor code if I add in future in case of need some new items) named *“types\ LabVIEW Universal Transcriptor C code Composer Input Cluster.ctl”*.

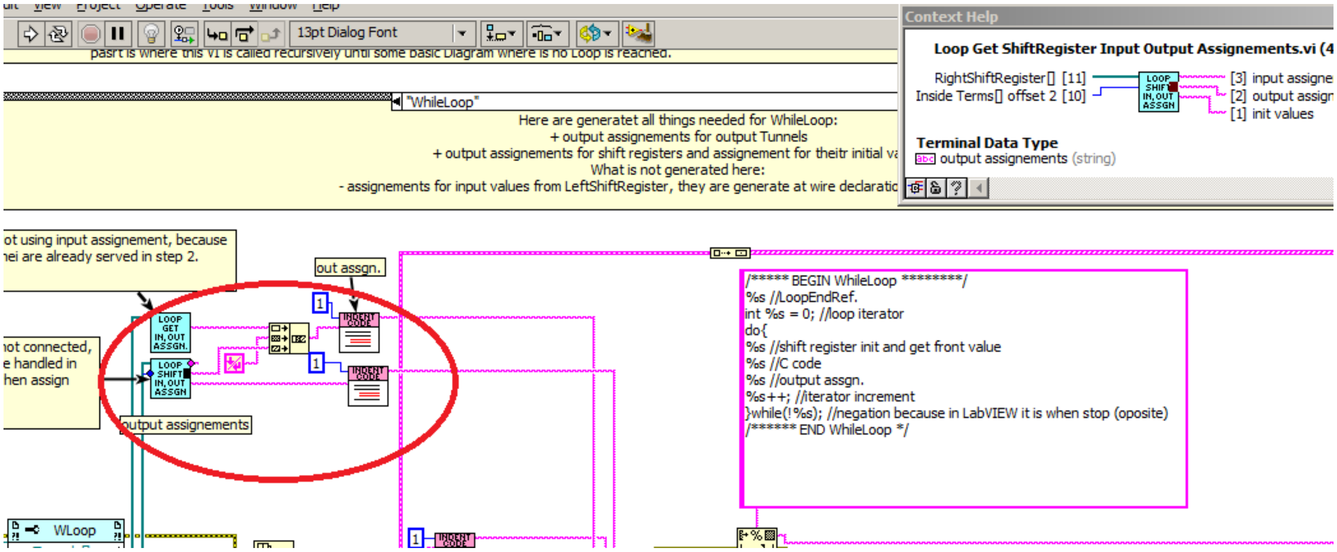


Illustration 28: Creating C++ code to push elements into shift register (implemented as vector) and aput this code at the end of generated loop code.

9.2 Step 1 and 2 Wires Declarations and Assignment

In chapter 9 was specified four step translation process. As second step there are two steps from original process merged together – creating constants and other values and transfer them to nodes.

For this purpose serve in LabVIEW wires. When we are debugging some code and turn on highglihg code execution (bulb next to run button) we can see dots moving along wires representing dataflow during execution Illustration 29.

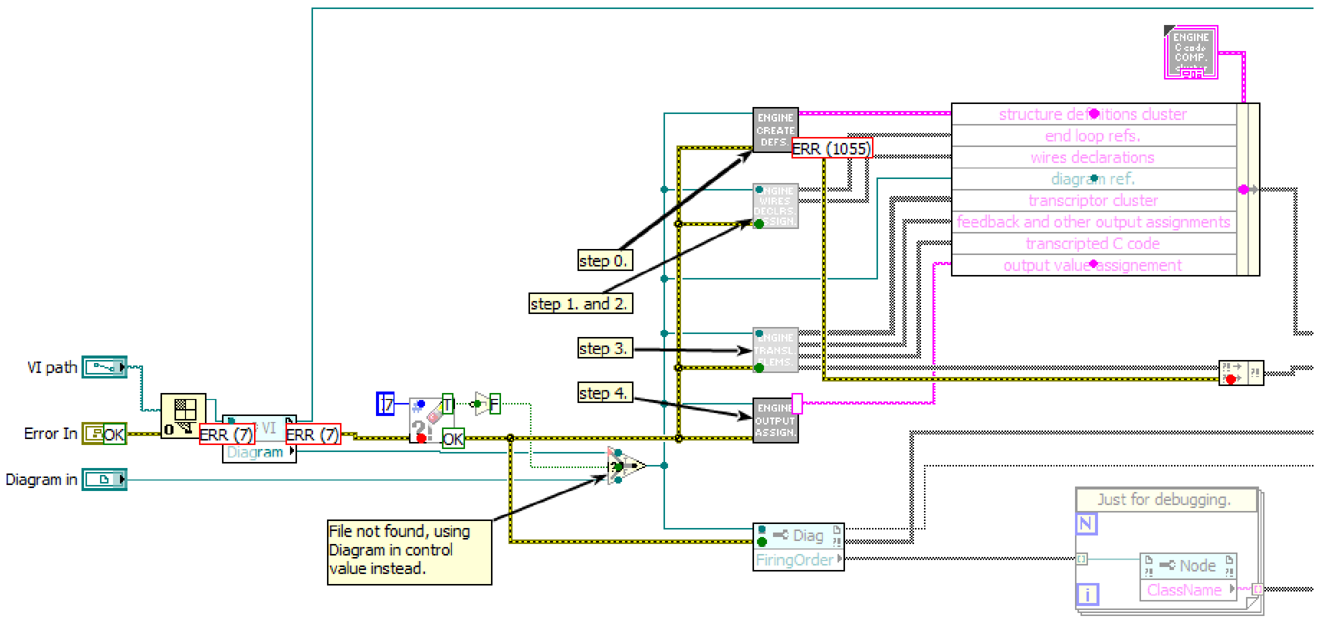


Illustration 29: Execution highlight, dots represents dataflow

So the mechanism for passing data to nodes is served by wires. I achieved this functionality by declaring variable for each wire (variable has same datatype as wire) and this way transfer mechanism is done by assigning value to variable for wire input terminal and reading value from variable for output terminal of wire.

Step 1. and 2. reduced then just looking for all wires on diagram, declaring variables for them and if wire is connected to some constant assign this value to the wire as part of its definition. This way all connected constants would be assigned as input value for some wire and constant which are not connected wouldn't be translated.

This task has some subtask, because to be sucesful we must be able:

1. generate wire unique name – this is done generating wire name as

   wireUID_xxx_[label if exists], see Illustration 30

2. read constant value, it could be anything, for this I created in transcriptor “Get Variant Value.vi” because LabVIEW system onbjects which contains some value usually have property “Value” which return their values as Variant and we have to decode this Variant

3. generate right assignement to right object because wire can be connected to different things include Tunnel, LeftShiftRegister, RightShiftRegister, FlatSequenceInnerTunnel, Control, Indicator, ….

   Most of these elements are child classes of Tunnel class, but we have to do some special operation, ie. for LeftShiftRegister we are not assign values as normal in C++ using operator “=” but we have to push value into vector (shift register transcription is using vector)

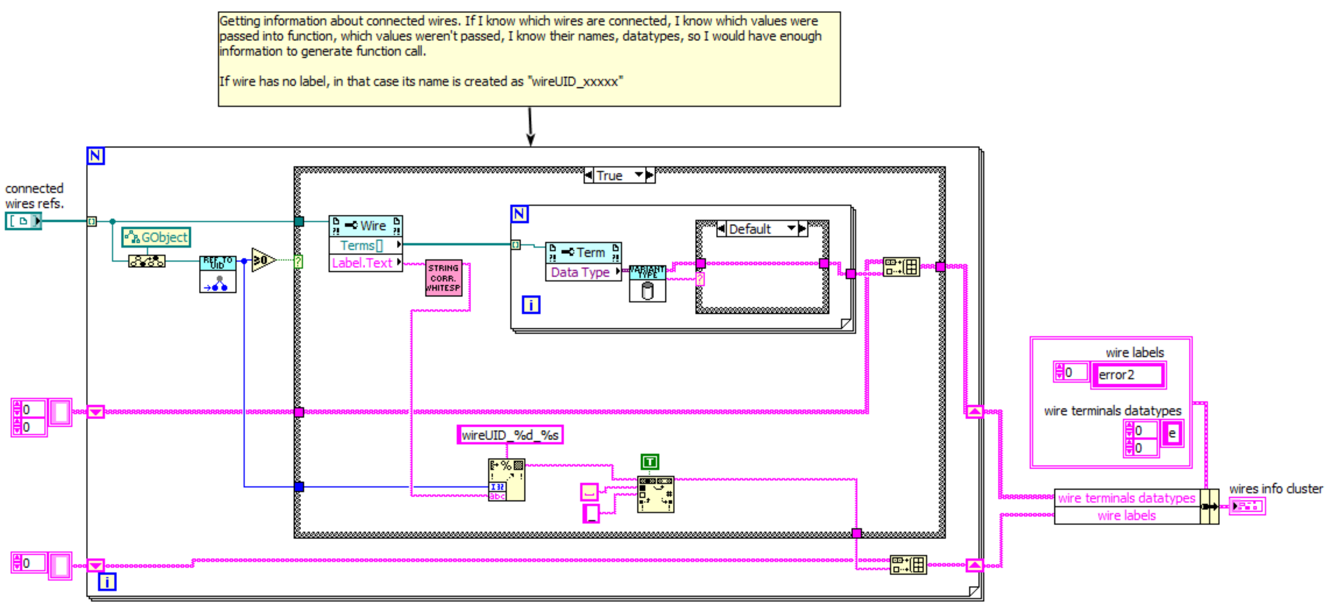


Illustration 30: Generating wire unique name using its UID, source XYZ

9.3 Step 3 - Translate Elements

This is place where most music is played :D Takes care about actual work and that is translate element to its C code representation. Processing of each element is different, transcriptor differs between elements based on their ClassName property (this property is defined in GObject, maybe even in higher object than GObject).

Input object for this stage is Diagram reference, diagram object containts property called *"Firing order"* which is most important for transcriptor, it returns referencies to objects ordered by their execution (calculation of firing order is done by LabVIEW environment, so transcriptor doesn;t has to calculate it).

So in this step transcriptor looks at firing order of provided diagram object and then iterate through all objects and transcript them into equivalent C++ code.

Most of time objects in firing order are type of Node, but there is one special case I observed during transcriptor implementation and that is FlatSequenceStructure is not even child object of Gobject, it's higher object, so there is in block diagram case structure in step 3. which handles FlatSequenceStructure element (there is some retyping of object to Generic).

I started with idea to be able translate from LabVIEW into arduino led blinking example and some programs which would be albe to write int serial bus. At beginning I told myself that it would be enough implement while loop, for loop, case structure, transcripting VI, feedback node and shift register and bundler, unbundler for cluster. Here is where it is implementaed.

It was written briefly before, here is comprehesive list, transcriptior of what is implemented in step 3:

- SubVI
- Polymorphic VI
- WhileLoop
- ForLoop
- Bundler
- Unbundler
- FlatSequenceFrame
- NamedBundler
- NamedUnbundler
- StaticVIReference
- Case Structure

- StaticVIReference (this is implemented for interruptions, need to re-check if it’s runnign properly)

9.3.1 Reentrant SubVI list, how to resolve infinite circular trnscripting

Because in LabVIEW user can create recursive code, transcriptor can enter some SubVI, starts transcripting process, then go through another 3 SubVIs and in 4th suddenly find some SubVI which lays higher, it means, that transcriptor is nested inside some SubVI somehwere deep and now is called to start transcript

process of SubVI higher what is itsel and here can transcriptor stacked in loop.

LabVIEW forbids you to create recursive call without setting Vis reentrant in their properties. So if VI already have information has flag to mark if it’s reentrant ie., if it’s possible to use it in recursive functions, I have to just chek it and add it into internal list when transcriptor passed through it, Illustration 31. Next time when transcriptor find this VI just put there it’s function call and NOT GOING DEEPER, JUST PASSED TO NEXT ITEM. This solution Is OK and is reason why there is shift register on while loop inside “Transcriptor LabVIEW to C code Step 3 Translate Elements.vi”.

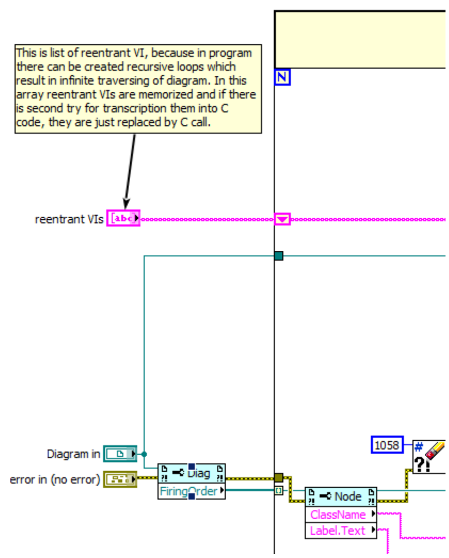


Illustration 31: Reentrant VIs list to avoid infinity circular transcripting in step 3.

9.4 Step 4 – create and assign output

Last step in my four process transcriptor. This is in LabVIEW really simple. Each SubVI has it’s indicators which shows value and just connect them with other objects.

In arduino world which is written in C++, there is possible to return just one value (normaly it’s possible return more values using pointers, arrays, but because of HW implementation on which C++ is running standard is that C++ returns just one value, so we heave to respect this fact), so when transcripti9ng SubVI into C++, in this part is created return statement as you can see on Illustration 32.

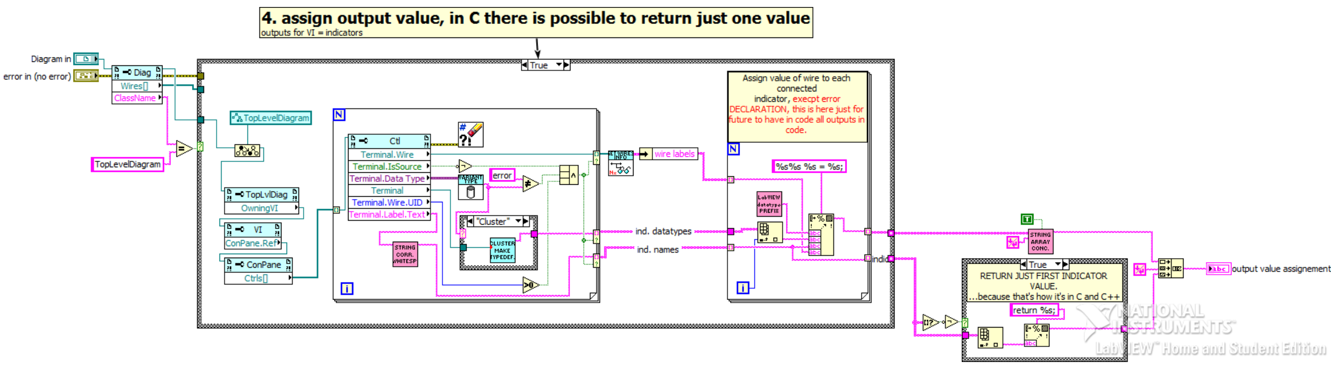


Illustration 32: Creating return statement from function

This code is just looking thorugh all terminals of connector pane of icon of diagram owner and returns all inidcators, at the end just first item is picked up.

10 Glueing whole code together

After each of four step translation process is finished in LabVIEW to C code Engine, they have to be put together, their output are arrays which are stired iinside transcriptor internal cluster, see Illustration 33.

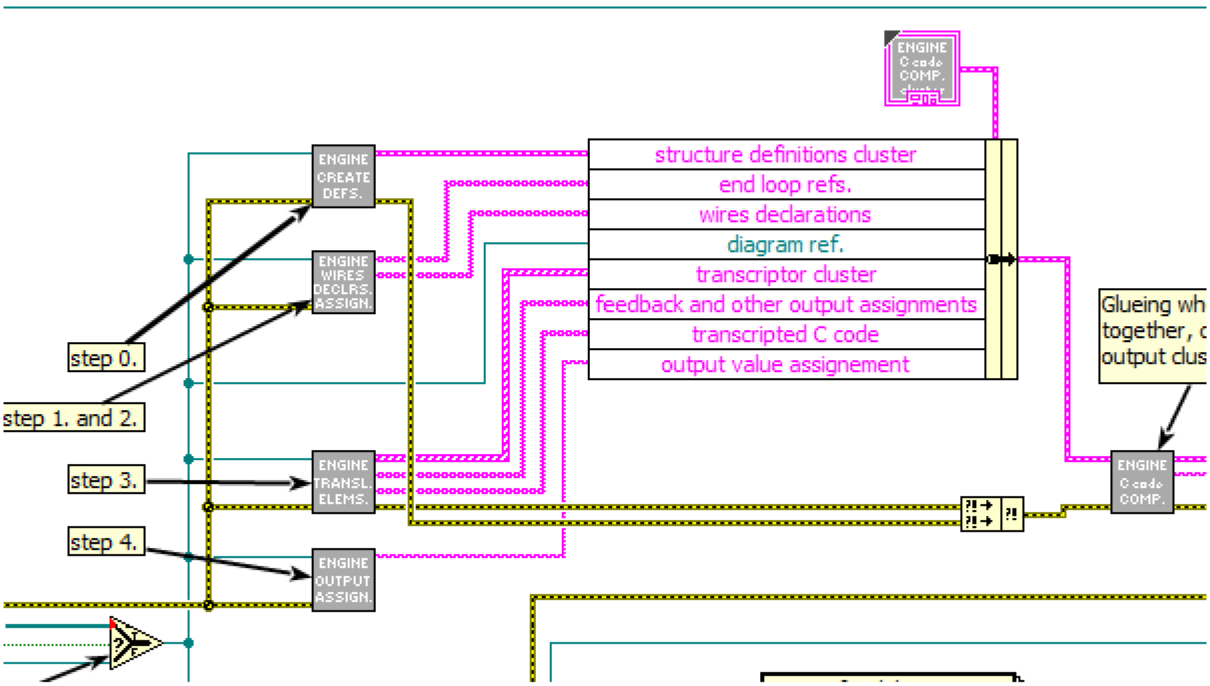


Illustration 33: Composing all code chunks (definitions, wire declars and init values assignements, translated elements and function return info) together to create resul arduino sketch

For this purpose there is sitting at the output VI called “Transcriptor LabVIEW to C code Composer.vi” which takes internal transcriptor cluster and used each item to write result code. More or less it just put each item of cluster and put into if three categories (includes, definitions, transcripted code).

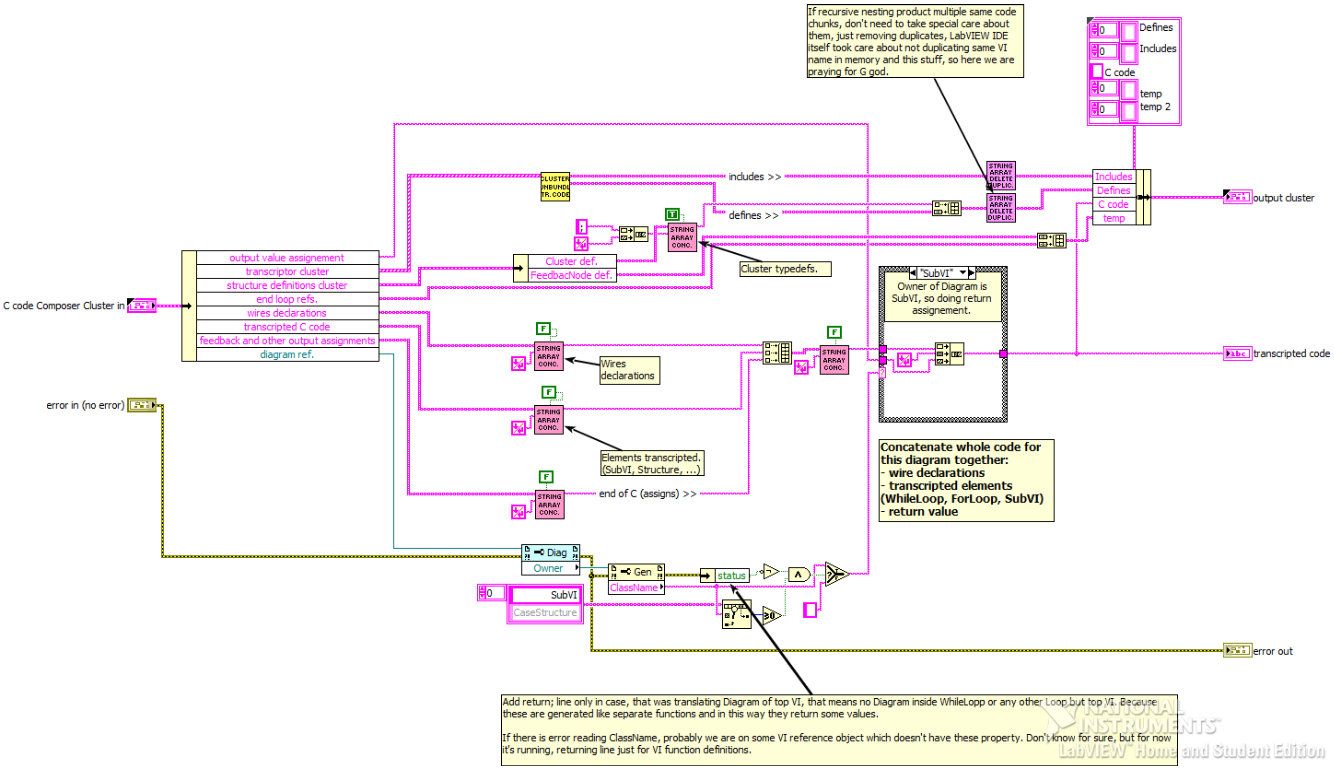


Illustration 34: Transcriptor LabVIEW to C code Composer.vi, putting whole code together

What I can write to this part is that inside 3. step there is output assignement inside case structure of transcripted code into indicator. And there is also transcriptor cluster containing 3 items like definitions, includes and transcripted code. Transcripted code should be wired to both transcripted code indicator tunnel and cluster also, but this is not mandatory, because item from cluster (transcripted code) is not used in the end. I sometimes forget to wire it there during development. Have to correct this mistake.

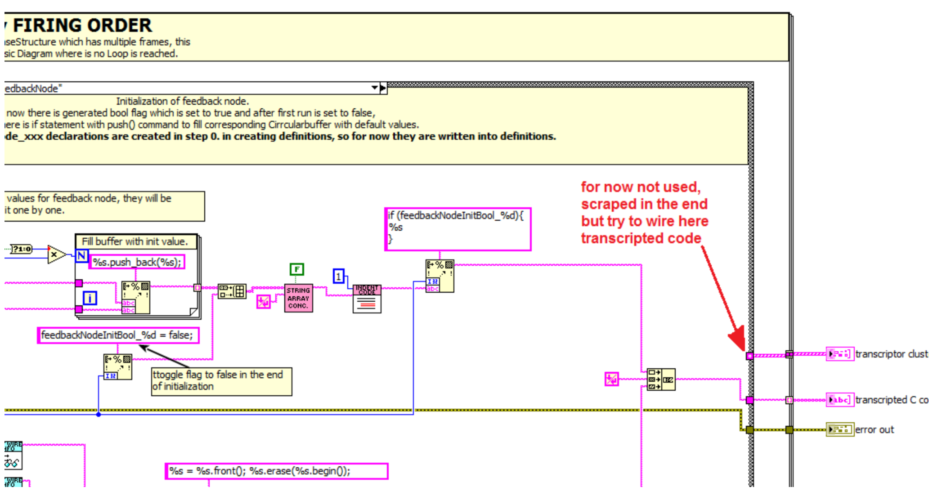


Illustration 35: Step 3. translating elemented, string tunnel for transcripted code, this will be for every case in future wired

11 Transcriptor usage

If you install transcriptor package it should appear at your pallete in block diagram under “LabVIEW Universal Transcriptor”, see Illustration 36, I installed also all of my packages from my repository (https://github.com/LubomirJagos/LabVIEW-for-Arduino-Libraries-Packages) so you can see there all of them, they installation destination is specified as this pallete.

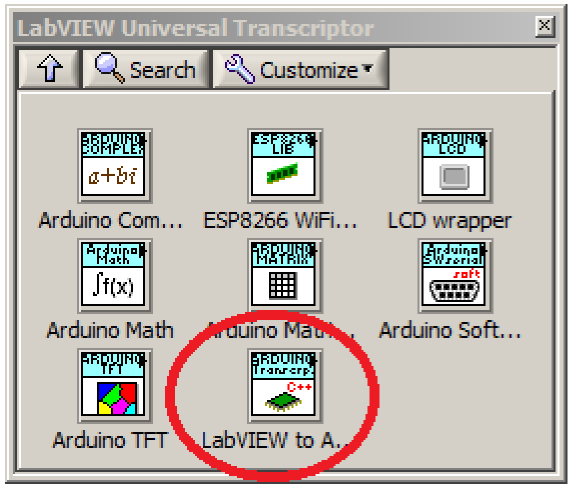


Illustration 36: LabVIEW transcriptor to Arduino pallete

After you click on “LabVIEW to Arduino” you will get to base functions as on Illustration 37 which are allowable to use and transcriptor is able to translate them into C++ code.

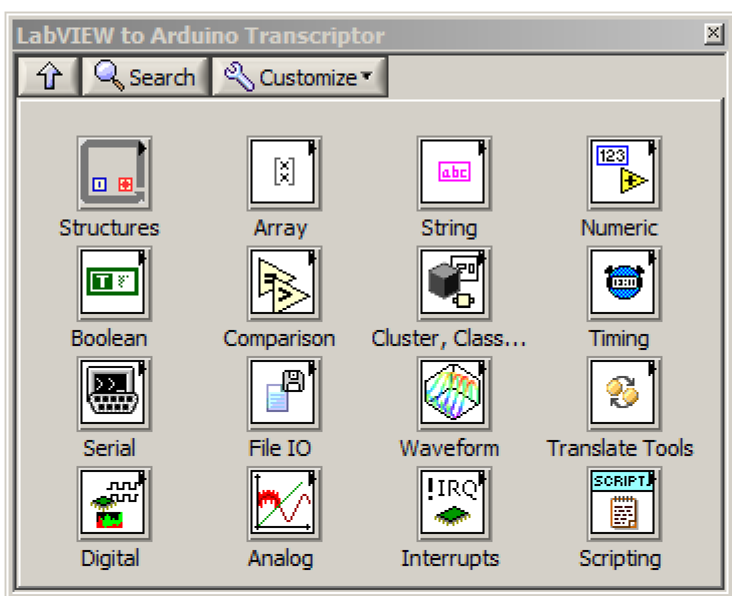


Illustration 37: Transcriptor base functions

Items on pallete Structures, Array, String, Numeric, Booelan, Comparison, Cluster, Class & Variant contains native elements of LabVIEW (String constant, Build Array, Add, …) when transcriptor find these elements in diagram it calls in step 3. “System Function Call Translator If Exists.vi” which calls in the end “Transcriptor LabVIEW to C code Translator Native Functions.vi” and there are implemented LabVIEW rules how to translate LabVIEW native functions, see Illustration 38.

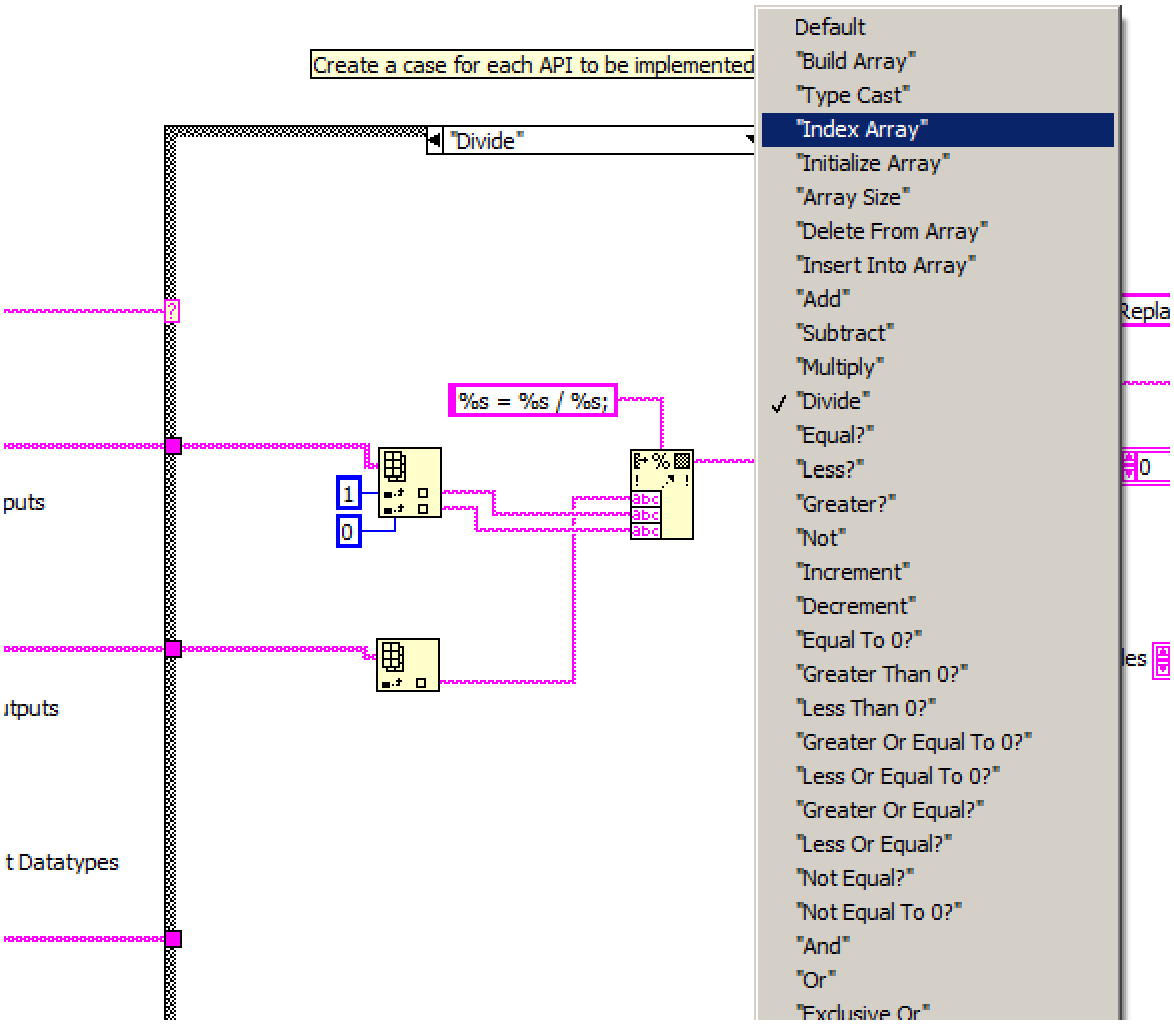


Illustration 38: Transcriptor LabVIEW to C code Translator Native Functions.vi

There are also another aplletes with additional functions characteristic for HW platform arduino which are needed to see in real life if our application is doing something, so for now I implemented few functions to have touch with world and they are Timing, Serial (arduino UART on pins 0 and 1), Digital (for digital pins), Analog (analog pins), Interrupt (adding and removing interrupt function, haven’t tested this yet)

There are two another palletes which I haven’t mention Scripting and Translate Tools:

- Scripting – VI which I’m using to transcript LabVIEW code to arduino, they are working mostly on object referencies to return another referencies to terminals, or return info about variants or doing something else really important for me

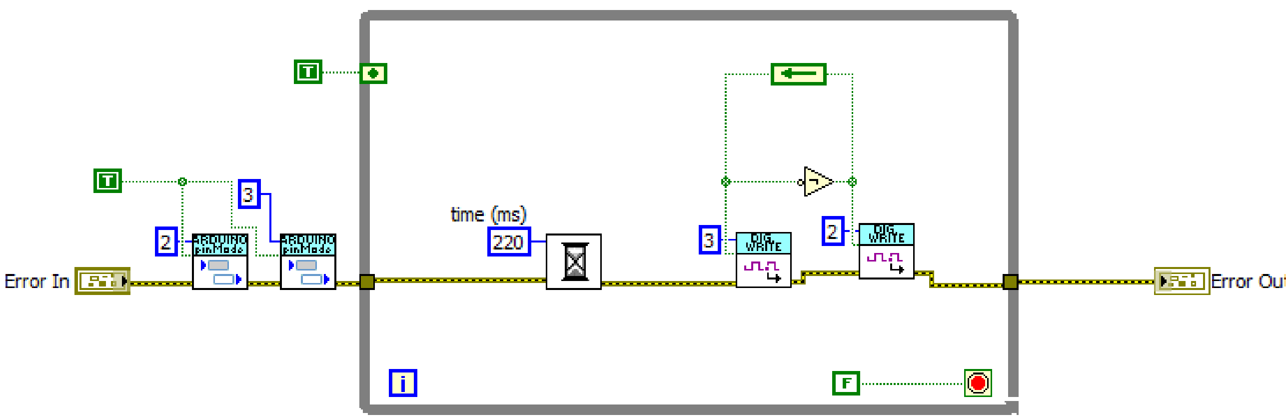
- Translate Tools – this is what it is all about

11.1 Translate Tool

These pallete is all you need to transcript your SubVI into arduino code, it contains just two items (and bascially they are same, just one has no error terminals :D) look at Illustration 39.

Illustration 39: Translate Tool pallete

So if you finally created your VI and let's say it has to blink with LED, you named it "Blink.vi" and looks like:



And now you want to translate it, so you can create simple application (anyway, this application looks simple, but is powerful like hell):

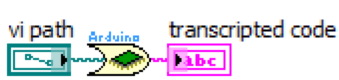


Illustration 40: Simple way how to transcript some VI to arduino code

On fornt panel this looks like on Illustration 41.

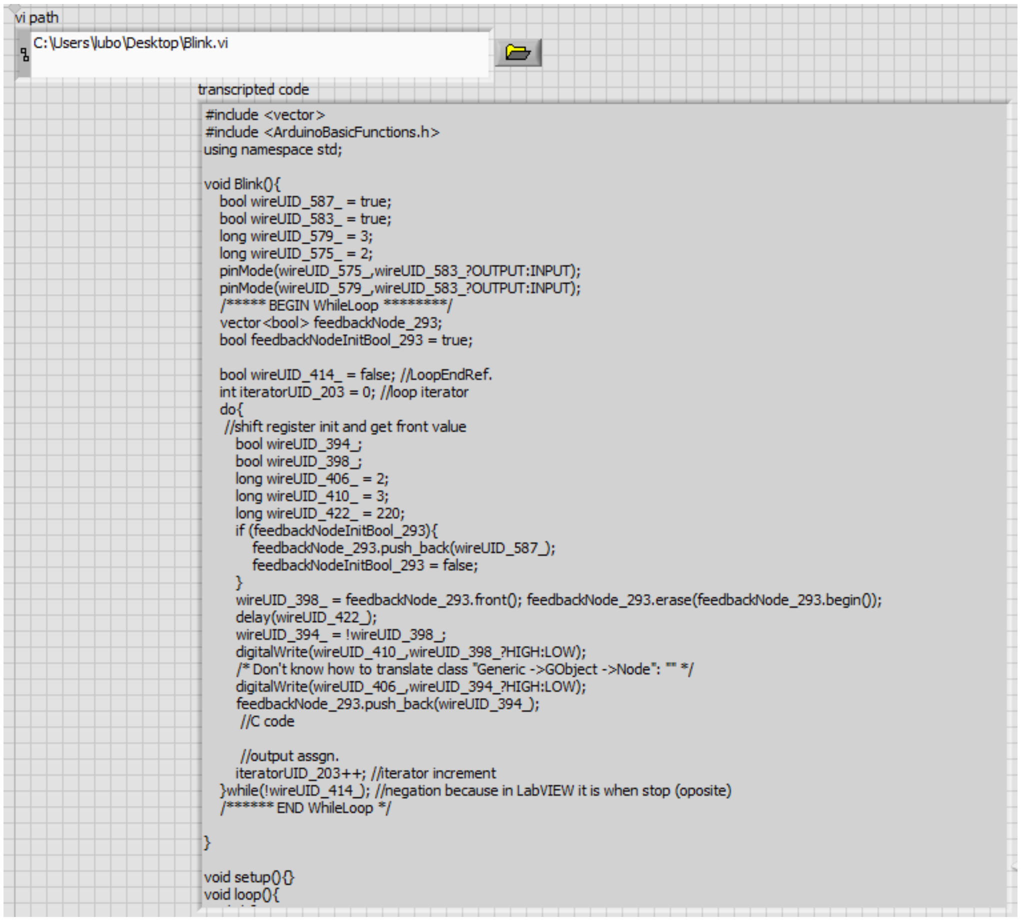


Illustration 41: Frontentd panel application for transcription

Instead of this VI there is another one VI for translating SbuVI into C++ for arduino, but this one has also error terminals (that before is using “Transcriptor LabVIEW to C code Frontend.vi” either but not implement its error terminals). So here is another way how to create previous transcription application.

Similarly we can create same application as before using this second VI (this one is nested in previous so it doesn’t matter at the end), look at Illustration 42.

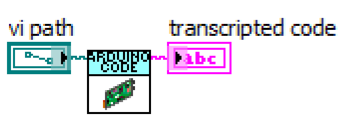


Illustration 42: LabVIEW to C++ for arduino simple transcription application

To make thing easier I put into transcriptor 2 examples for arduino which is enough to run, they will transcript VI and show you code. These are hopefully out of box examples, for now blinking with LED and writing to serial using clusters.

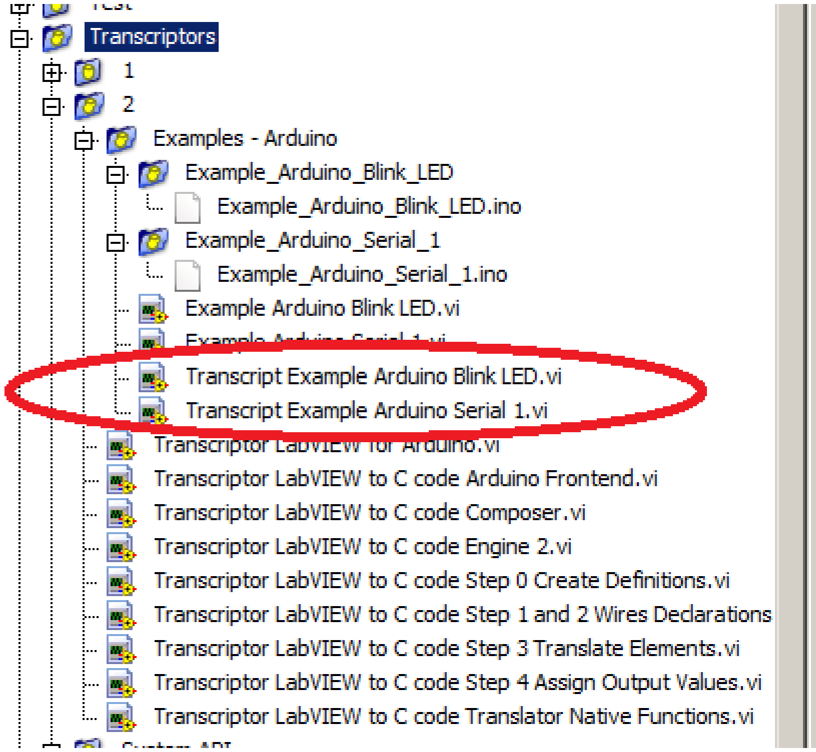


Illustration 43: Out of box examples for LED and serial communication, shows how to use transcriptor to get some results, at least on my side

12 Project directory structure (or at least effort for it)

I was coding and conding and soon realize that for this project just few directories wouldn't be enough, but I can't decided how big it will be in the end (I want to have reasonably enough directories and not hundreads of them) so I choosed middle way and split project like most of normal project between directories for utilities what means small atomic functions, they are independent from rest of program and can be used separately and some other direcotires. You can see whole dir structure on next Illustration 44.

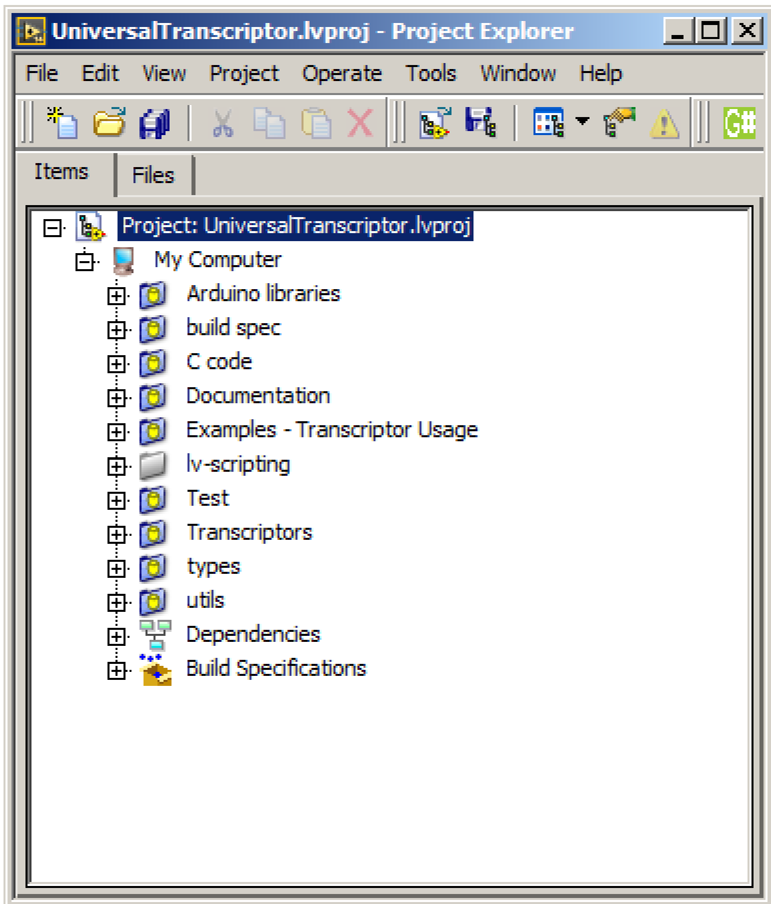


Illustration 44: Directory structure

Here is explanation what contains these dirs:

- Arduino libraries – mostly external projects from github which implement functionality which I want to interface for transcriptor
- build spec – this is where thing to create nice looking LabVIEW package are stored
- C code – generated arduino files, I put them here to have some place where I can copy and paste generated code from my test examples and see if it's able compile, this is my working arduino heap
- Documentation – this file is there :)

- Examples – Transcriptor Usage – here should be examples how to use some VI for reading diagram but probably now are there just som temporary files
- lv-scripting – extern module, this is stupaq project on github (https://github.com/stupaq/lv-scripting) at the moment it’s not used, but at beginning I found this project and think that it will be useful for me, but still there was no need for it
- Test – here are stored VI to check how looks generated code from transcriptor, directly in this folder are test VI, what means, that I use them to run some of my test during devlopment of this transcriptor to see what I can do and what no

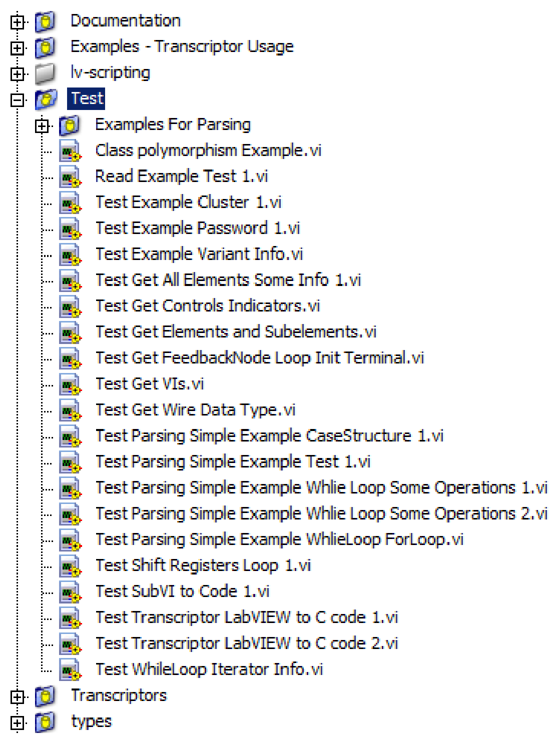


Illustration 45: Test directory

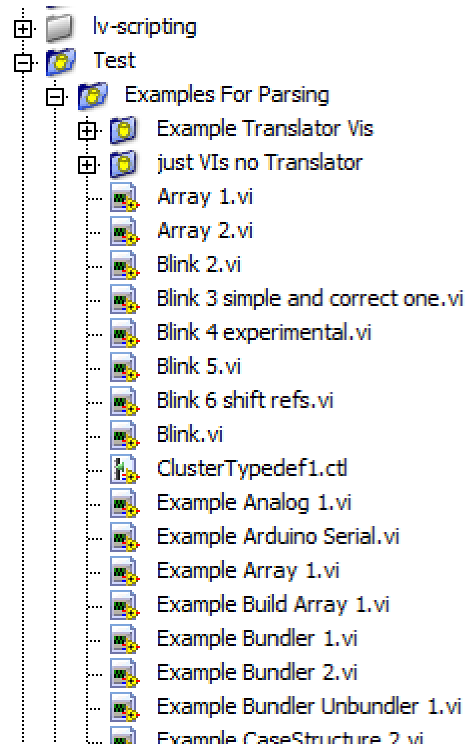


Illustration 46: Examples for parsing in "Test/Examples for Parsing"

- Transcriptors – here are stored transcriptors, in directories named by numbers, I started to develop transcriptor 1, at that time I wasn‘t aware that there is property “Firing order” of Diagram class and was traversing Diagram stepping along error wire, I got SubVI transcripting working, that realize that I need something like firing order and finally found it, I didn’t want to remove whole transcriptor 1 so I started to work on transcriptor 2 and used most of already created functions and created this simple system to separate different transcriptors
    - subdirectory “System API” containts VI and Translator.vi for some arduino API functions (digitalWrite, digitalRead, analogWrite, …), this was mean as arduino system API interface

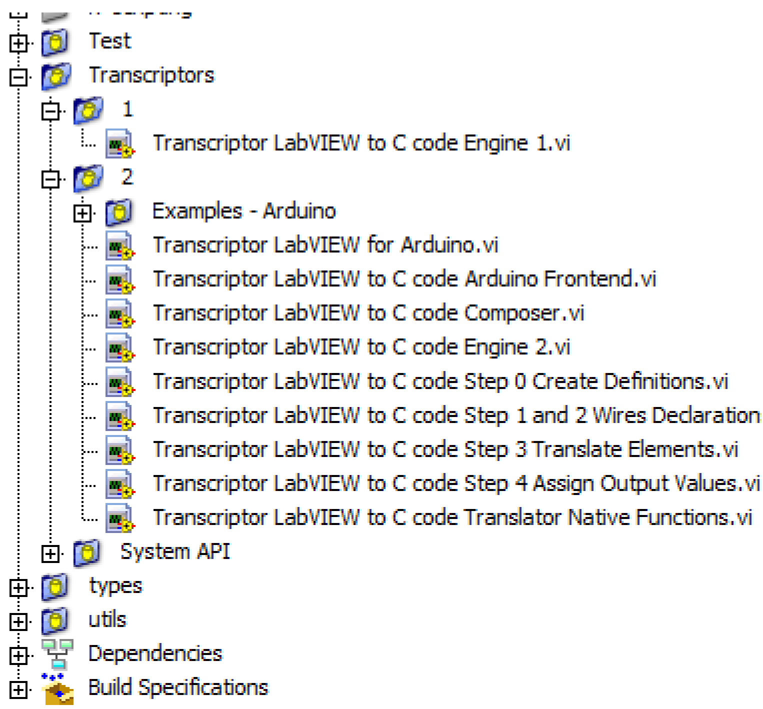


Illustration 47: Transcriptors directory

- types – contains typedefinitions, prefixies and this kind of stuff (may I call it dictionary or lexical style of this transcriptor?)

13 Creating custom SubVI transcriptor rules

This whole project is about implementing transcriptor machine for LabVIEW to some textual representation, there is for sure also for user to implement own transcription rules.

This rules now can user defined just for SubVI, for everything else you have to edit transcriptor itself (but this is ok, it's on github :D).

There is only few rules:

- each SubVI which you want to be transcripted you have to locked by password

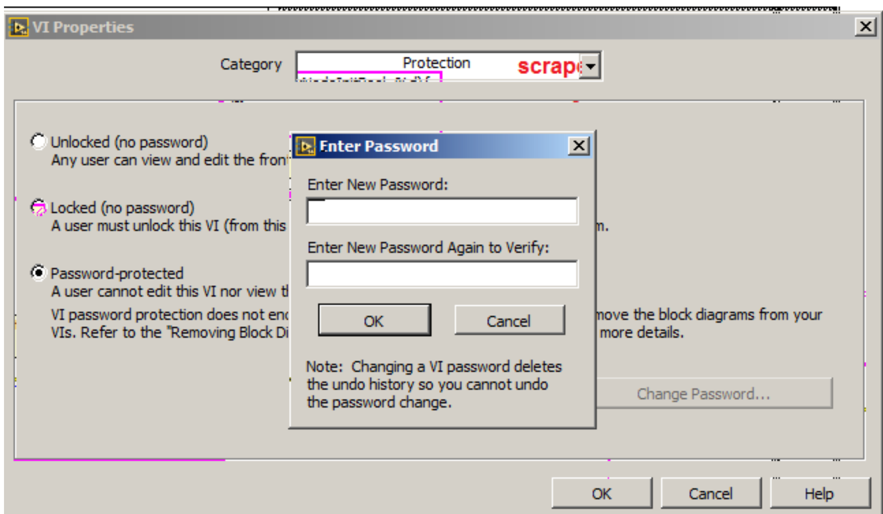


Illustration 48: Locking VI to be use for transcripting into C++ for arduino

- inside directory TOGETHER with with VI there has to be VI with name containing in it “Translator” which has to be done exactly as this one on Illustration 49.

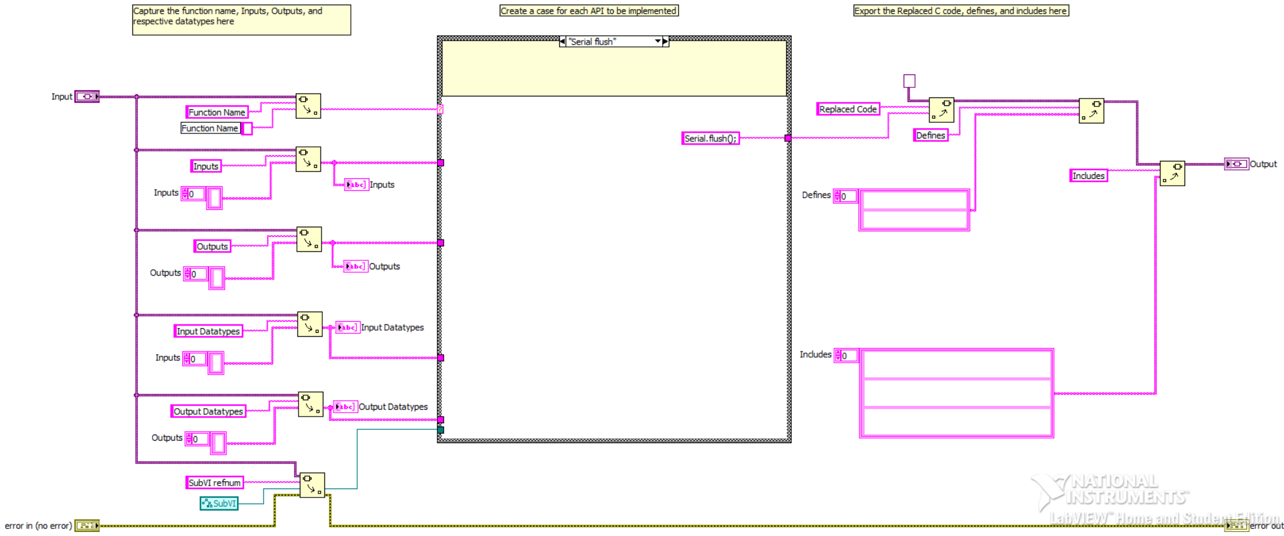


Illustration 49: Translator VI, template for all translators, it's asynchronously run from transcriptor when password locked VI is found, it contains case for names of locked SubVI and some info about inputs, outputs, datatypes and my enhancement reference to that SubVI

If you want to define your own transcriptor just take inspiration from my libraries which I created, for now I haven't test them with my transcriptor and lot of things is not running, because I just started whole system integration (and that's really like oh this is not running? And that there also not? :D).

Until now I tried to transcript simple programs for arduino which I was able directly test, they are about using cluster, writing into serial bus, blinking with leds, setting PWM on digital pin.

14 Killer features

This transcriptor implements transcripting LabVIEW into C++ as you expect it should be, so for loop is translated as C++ for loop, sub VIs calls are translated as functions and so on.

.

.

.

.

.

WRITE MORE!

.

.

.

14.1 Polymorphic VI

Because this transcriptor generates code from LabVIEW it can read all params which are accessible through VI scripting. LabVIEW has quite lot extension when speaking about syntax, one of them is polymorphic VI mechanism. You can group more VIs under one and then by selector choose which implementation has to be used. (you can set that VI will change automatically, but when I tried this it wasn't running correctly with transcriptor)

PICTURE OF PRINTLN POLYMORPHIC VI

14.2 Classes with own wires coloures

Lorem Ipsum.

14.3 Simulation able VI

Transcriptor knows if he has to transcript SubVI into C++ code by checking if it's passowrd locked. If no it's traversing it, if yes it uses translator VI to determine result code and it's not discovering VI inside.

We can use this feature that transcriptor is not discovering password locked VI and put inside some LabVIEW VI. This way this VI will be runnable on normal desktop PC by specified algorithm created inside password locked VIs and it will have som C++ representation defined inside translator VI.

So if algorithm inside these locked VI is identical with normal behaviour on MCU, we can compose our program on desktop PC using LabVIEW, run it, obtain results and after compile it and deploy to embeddded device. This is how most of basic VIs are done (when it was possible and it made sense like readin file or SD card VIs or math VI) except ones which are tight to HW like accelerometer VI or WiFi VI where it's not efficient to simulate their behaviour (better program devlopment is check on PC algorithm behaviour and HW behaviour done directly on HW with compiled code).

15 FAQ

15.1 Why is even possible to transcript diagram into code

Because every finite state machine could have representation of diagram, textual representation or can be written as matrix operation (multiplication of matrix with state vectors, ….). That's why.

16 To the End

Thanks for reading this manual. I tried to briefly explained how I made my transcriptor. I tried as possible to made it consistent and hopefully easy to understand, but like with all projects at the beginning everything runssmoothly, then when more and more things are done you can see some imperctions and the most work is in the end to closed whole code and make it running.

From my test I'm in the finish state to make everything running, then there is … testing :D For me proof of functionality is when there will be created lots of programm which this transcriptor will be able to transcript and succesfully compile and they will be runable without problems as expected.

This is dream because this is embedded. It's changing by rocket speed and still it will be even faster in future :D So at least I done something if not useful at least inspirative.

Enjoy.

LabVIEW super enthusiast

Lubomir Jagos