VisLang: A graphical programming language

Bryant Eisenbach

June 12, 2015

1 Introduction

VisLang is a block diagram language designed to allow fast and easy prototyping of programs for embedded processors. The language is created with a graphical editor in mind, and the core language is designed to be extensible so that any graphical editor can add additional elements or attributes for graphical display or other features.

2 Key Language Features

The language itself is based on the idea of blocks: small parts that can be grouped together into ever larger blocks and re-used across a program or programs. A small group of fundamental (or atomic) blocks will be defined that will be understood by the compiler for this language. Other blocks will be constructed as configurations of these atomic blocks. A standard library of useful functions will be constructed from these atomic blocks containing common parts such as timers, latches, etc. However, this standard library is not required, and the ability to include parts libraries and other blocks is a standard feature of the language.

Side effects in the produced code will be minimized by the combination of a strong type system and bounded code execution. The type system of VisLang supports common datatypes such as boolean, single, double, signed and unsigned integers, as well as static arrays and structures of these simple datatypes. Bounded code execution is guarenteed through the restriction of array operations to use a static size. This works well with embedded processors as programs should only need to parse through large buffers such as those implemented for I/O using digital busses, whose buffers are typically defined as being a static size. The language will contain implementations of the popular methods map, filter, and reduce to enable operating on arrays. This will allow VisLang to have methods for working with digital message structures and parsing those structures into the signals that can used by the program. Some standard implementations for parsing both packet-based (e.g. Ethernet) and word-based (e.g. RS232) buffer data will be provided by the standard library.

Lastly, time variance will be something provided fundamentally by the language. The time-step between subsequent iterations will be maintained in every VisLang program and provided to the user as needed as a atomic part. Most of the atomic parts will be time-invariant as they are pure functions, but this language feature will provide users with the ability to create dynamic parts that will care about time as a measured quantity to create parts such as digital filters, derivatives, integrators, etc.

3 Syntax

The syntax of VisLang leverages standard XML, giving the language a well-formed and machine readable backbone. As noted previously, the point of leveraging XML is so that 3rd party programs can manipulate the file format in an easy way, and so that those programs can add additional elements (e.g. visual comment blocks) and attributes (e.g. location information) to the existing set of elements and attributes defined by the language. Those tags (including XML comment tags) not included in the list of recognized elements/attributes will be ignored by the compiler, however this decision should have minimal impact on ambigious errors when compiling programs due to the restriction that all parts require their necessary attributes, and that all connections require the source to exist. This creates a natural flow to interpreting the language, such that any errors should be raised by the compiler during compilation.

Below is the list of standard elements supported by the language, and their attributes:

element	input(s)	outputs	attributes [optional]
INPUT	none	provides 'name'	scope, size, name, type, [address]
OUTPUT	provides 'name'	none	scope, size, name, type, [address]
CONSTANT	none	provides 'name'	size, name, type, value
SIGNAL	provides 'name'	provides 'name'	scope, size, name, type
PROGRAM	device inputs	device, global outputs	name
BLOCK	as defined	as defined	name, [reference]
CONNECTION	none	none	to, from
CAST	input	output	name, type
MEM	current	stored	name, initial_condition
DT	none	dt	name
NOT	input	output	name
AND	input# (# > 1)	output	name
OR	input# (# > 1)	output	name
NOR	input# (# > 1)	output	name
NAND	input# (# > 1)	output	name
XOR	input# (# > 1)	output	name
BITWISE	input# (# > 0)	output	name, operation, [mask]
IF	control, true, false	output	name
COMPARE	lhs, rhs	output	name, operation
SUM	input# (# > 1)	output	name
PROD	input# (# > 1)	output	name
GAIN	input	output	name, value
INV	input	output	name
MUX	input# (# > 1)	provides 'array'	name, type, [address]
DEMUX	array	output# (# > 1)	name, type, [address]
STRUCT	none	provides 'struct'	name, member# (# > 0)
CONSTRUCT	by definition	provides 'name'	name, definition
DESTRUCT	struct	by definition	name, definition
MAP	input (array)	output (array)	name, func (SISO block ref)
FILTER	input (array)	output	name, size, func (SIBO block ref)
REDUCE	input (array)	output	name, initial_value, func (MISO block ref)

The first group of elements (INPUT, OUTPUT, CONSTANT, NAME) provide named memory locations for use in the application. All of these elements have a type attribute where the datatype of the name is specified. INPUT, OUTPUT, and SIGNAL have a scope attribute, which defines whether the name is local scope (block-wide), global scope (program wide), or device scope (only Inputs and Outputs can be device scope). When device scope is chosen, an IO address must be chosen to specify where to read or write the I/O data to the interface with the target hardware. The CONSTANT element describes a named constant, which is initialized to the value provided and will reside in memory and read each pass during program execution. All of these elements have a size attribute, which means they can refer to array quantities as well. A structure can be used as a datatype.

The second group of elements (PROGRAM, BLOCK, CONNECTION) describes block constructs. This is the basis of how the program works. The PROGRAM element is a container for the entirety of the program.

The PROGRAM element has a name attribute which describes what the compiled program should be called by the compiler. The BLOCK element is also a container, but used throughout the program to encapsulate functionality or define helper parts in referenced files. BLOCK has an optional reference property which describes the location of a block in a file using the "/path/to/file.vs|path|to|block" syntax, where the vertical bars describe the heirarchy required to discover the referenced block. Any referenced block will have to have connections to all of the defined inputs in the referenced block. The outputs are not required.

The CONNECTION element is how blocks and parts are connected together. The "to" attribute describes where the connection is headed, and cannot have a different CONNECTION element with a matching "to" attribute name. The "from" attribute describes where the connection came from, and does not have the same restriction. Connections can either be made to named signals such as INPUTS, OUTPUTS, CONSTANTS, or SIGNAL, or they can be made implicitly to blocks themselves using "block|input" or "block|output" syntax.

The third group is the atomic parts. The Language Reference Manual will go into more detail here but all of these parts should form the bare minimum required by the base language to compose all other required functions from. All of the atomic parts operate on specific types and have a specific defined function present in the compiler. All of the blocks have one or more inputs (except DT) and have exactly 1 output. All of the blocks are time invariant, ,e.g. given the same input they produce the same output, except the DT and MEM blocks. The MEM block will store the value of something until the next pass. The DT block outputs the delta time between each pass. Since all blocks are simple and operate on a defined amount of inputs and outputs, they can be used with arrays to define parallel operations, e.g. applying the OR gate operation against two boolean arrays of the same size to produce a third boolean array of that same size. Structures will not work as inputs to these parts.

All the logic gates besides the NOT gate (AND, OR, etc.) as well as the BITWISE, SUMMER, and PROD elements are defined as having two or more inputs (BITWISE can be one if a mask is set). The way this works in practice is that each successive input increments the number after the word "input" e.g. "input1", "input2", "input3", etc. when making connections to these parts.

The fourth group describes array and structure utilities. MUX creates arrays from a group of inputs with the same type, and similarly DEMUX will deconstruct an array into a group of outputs of the same type. Similar to the atomic gate parts, these parts allow on the relevant inputs/outputs a variable amount of elements to be specified using the input# syntax on the connection elements. The size of arrays will be static during compilation, and array size will be a quantity traced between blocks to verify size matching between usages.

CONSTRUCT and DESCTRUCT create and decompose STRUCT elements respectively. The way this works is that a STRUCT element is defined seperately, with member attributes defining the name and type of each member of the struct. The CONSTRUCT element will allow a set of inputs matching the type of each of the members of a struct to create a new named quantity of that structure, and similarly DESTRUCT deconstructs a structure into signals that match each of those members. The STRUCT element can be defined or referenced through a file, allowing a seperate definition file to be reused over a project. Additionally, a STRUCT can be created from or written to an array of certain types, using the exact amount of memory required with buffer space as necessary to represent it as a chunk of memory. This will allow message definitions to be encoded or decoded to/from a hardware buffer. Lastly, an array can be composed of struct elements, as long as all elements match that structure definition.

The last group of elements is the array functions. These functions operate on arrays per element of the array, and require a referenced function of a certain number and type of inputs/outputs to translate the input array into the output. MAP will apply the referenced function of a single input to a single output (that may or may not matches the input's type). What is important is that the input array's type and the type of the input to the function match, as well as the same criteria for the output.

FILTER is similar to MAP in that it has a referenced function, however the output is required to be a boolean conditional. What FILTER will do is use that conditional output to filter the input array into an output array. Since there is a design decision not to allow variable array sizes, FILTER requires a size attribute for the output, and will not update the output if the size does not match what the attribute requires. This can be useful for parsing a buffer of word-based definitions, as we can filter the array of words to find a single UID that is a part of the word definition and pass that as the output to a parsing function.

The REDUCE element is similar to MAP and FILTER in that it takes an array argument and use the function reference, however this function will take each element of the array and apply it to a function that takes two inputs: the first being the same type as each array element, and the second matching the output type of the function. REDUCE will start with the first element and the initial value for the output and recursively run through each element of the array, feeding back the output as the inital value input for the next iteration. When REDUCE has exhausted all of the elements of the array, the final output of the function is returned back as the output of the REDUCE element.

Most attributes for elements are defined with a string. In some cases, it may be preferable to use a literal definition for a value instead of creating a reference or using a named quantity. Every datatype in VisLang has a way of creating a literal. Examples of literals for all VisLang types are below:

datatype	example literal(s)
boolean	false, true
single, double	1.000, 1e-6, 100
integer	100, 0x20, 2x1011, 8x671, -120
address	0x1A56, 0x10A8|14
arrays	$[1, 2, 3, 4]$
structures	{1, {2, 3}}

Booleans can only be false or true. Floating point quantities (single, double) can be a decimal quantity (with or without significant digits after the decimal point), or specified using scientific notation e.g. 10e6. Floating point literals can be specified with a negative sign as well e.g. -1.234e-6. Integer quantities (e.g. uint8, uint16, uint32, uint64, int8, int16, int32, int64) can be specified as a decimal quantity (cannot have a decimal point), or using hex (e.g. 0x1A), binary (e.g.2x1010), or octal (e.g. 8x2462) representation. Only signed integers can have a negative sign in front. Additionally, for address literals, specifying a bit of and address can be done with the | operator against a hex representation of the address (e.g. 0x1235|7). Address literals can only be in hex. Array literals are created with the bracket operators (e.g. $[true, true, false]$), and structure literals can be created using the braces operator (e.g. {0x10, {1, false}}.

Any attribute that accepts a literal can also take a reference to a named value, in a similar way that block references are defined (e.g. "/path/to/file.vs|path|to|value").

4 Example Program

The following program illustrates some of the major features of the language. The program itself takes a Digital Input on the target device, reads it, and starts a timer when the input is enabled. When the timer counts up to the target time, it will set a Digital Output true and reset the timer, creating a fast-blinking light with a period of 2 seconds.

Listing 1: ../example/timed-blinking-light/timed-blinking-light.vs

```
<?xml version="1.0" encoding="UTF-8"?>
<PROGRAM name="timed-blinking-light.bin">
<!-- This block denotes the contents of a program. Everything
     contained within (including file references would be compiled
     as a single binary. -->
<INPUT scope="device" name="digital_input_1" size="1"
     type="boolean" address="0x0123|0"/>
<!-- Hardware address 123 bit 0 is DI_1 for ATMega328 -->
<!-- TODO: Find real hardware address for this DI -->
<!-- Literal constants for unsigned integers can be hex (0x20), binary
     (2x110101), octal (8x266), or decimal (268). Literal constants for
     signed intergers must have a sign prefixed (e.g. +0x26A8 or -268).
     Neither int nor uint can have a decimal point. -->
<SIGNAL scope="global" name="count_expired" size="1" type="boolean"/>
<!-- It is good practice to specify Inputs, Signals, and Constants at the
     top of a document, although not strictly necessary -->
<NOT name="not_di_1"/>
<!-- The | operator on a name denotes an available connection -->
<CONNECTION to="not_di_1|input" from="digital_input_1">
    <!-- A GUI Program could specify the shape of the connection
         here. Not relevant for the compiler -->
</CONNECTION>
<!-- literal constants for booleans are "true" and "false"-->
<MEM name ="count_expired_lp" initial_condition="false"/>
<!-- Memory block would store the state each pass of the variable
     specified by current_pass_value at the end of execution
     such that the last_pass_value can be used in the local scope
     without suffering from algebraic loops -->
<CONNECTION to="count_expired_lp|current" from="count_expired"/>
<OR name="reset_blink"/>
<!-- OR, AND, etc. Gates can specify any number of inputs via
     incrementing the input specifiers "input1", "input2",
     "input3", etc. -->
<CONNECTION to="reset_blink|input1" from="not_di_1|output"/>
<!-- The "block/output_name" syntax is how to create a connection
     directly without an intermediate signal -->
<CONNECTION to="reset_blink|input2" from="count_expired_lp|stored"/>
<!-- The | operator on a reference denotes a member of the referenced
     file or library -->
<BLOCK name="timer_instance_1" reference="./timer.vs|timer">
    <!-- Subsystem block can either be locally defined (no reference
         attribute) or reference an external file (with reference
         attribute). File location is referenced relatively, or via
         include statements for library parts folders -->
    <!-- If a subsystem block were locally defined, everything inside
         the tag would be considered a part of that subsystem  -->
</BLOCK>
<CONNECTION to="timer_instance_1|start" from="digital_input_1"/>
<!-- Input and output connections to blocks are partially ambigious.
     However for a Connection to work, one and only one of "to" or
     "from" attributes must be an input/output of the part. -->
<CONNECTION to="timer_instance_1|reset" from="reset_blink|output"/>
<!-- You can plug in hardcoded constants as inputs to blocks
     directly. Literal Constants for singles have decimal points
     (e.g. 2.000). Literal Constants for doubles have "L" as a
     suffix (e.g. 2.00L) -->
<CONNECTION to="timer_instance_1|time" from="2.000"/>
<CONNECTION to="count_expired" from="timer_instance_1|count_expired"/>
<!-- An output can be used in a from statement as many times as desired.
     However, a name can only be assigned to a to statement once. -->
<CONNECTION to="digital_output_1" from="timer_instance_1|count_expired"/>
<!-- Any un-attached outputs to a block are optimized out, e.g.
     elapsed_time. All inputs are required -->
<OUTPUT scope="device" name="digital_output_1"
     size="1" type="boolean" address="0x0456|0"/>
<!-- It is good practice to define outputs at the bottom of a document -->
<!-- Hardware address 456 bit 0 is DO_1 for ATMega328 -->
<!-- TODO: Find real hardware address for this DO -->
</PROGRAM>
```

As noted, the above file contained a reference to another part called timer defined in timer.vs in the same directory. Any references must take place on a relative path to that file, and that reference must contain the same number of inputs specified by the target file inside the file referencing that part. The number of outputs need not match, but any outputs specified in the file referencing that part must also match what is available from the target file or an exception will be thrown. All other unused outputs will be disregarded.

The following file displays the target file, complete with the relevant inputs and outputs as specified/required by the previous file.

Listing 2: ../example/timed-blinking-light/timer.vs

```
<?xml version="1.0" encoding="UTF-8"?>
<BLOCK name="timer">
<!-- The BLOCK element denotes a subsystem of parts -->
<!-- All "parts" added by the user can use Inputs and/or
     Outputs for utilization elsewhere in project. The
     reference will search the path for that file -->
<!-- All Inputs do not have to be used and will be optimized out -->
<INPUT scope="local" name="start" size="1" type="boolean"/>
<INPUT scope="local" name="reset" size="1" type="boolean"/>
<INPUT scope="local" name="time" size="1" type="single"/>
<!-- Constants can be defined as a seperate block as well -->
<CONSTANT name="zero_constant" size="1" type=single value="0.000"/>
<!-- The DT block puts out the difference in time between
     successive passes of program. In a Soft RTOS, this
     would be a variable number. In a Hard RTOS, this
     would be a constant number. -->
<DT name="time_since_last_pass"/>
<NOT name="count_not_expired"/>
<CONNECTION to="count_not_expired|input" from="count_expired_lp|stored"/>
<AND name="start_enb"/>
<CONNECTION to="start_enb|input1" from="start"/>
<CONNECTION to="start_enb|input2" from="count_not_expired|output"/>
<IF name="increment_value"/>
<!-- Control flow IF switch: If Control is true, execute
     True assignment, else execute False assignment -->
<CONNECTION to="increment_value|control" from="start_enb|output"/>
<CONNECTION to="increment_value|true_input"
     from="time_since_last_pass|dt"/>
<CONNECTION to="increment_value|false_input" from="zero_constant"/>
<SUM name="summer"/>
<!-- The summer will add all the inputs together. If you want
     add a negative number, use the NEG part to negate the
     signal before connecting to this part. -->
<!-- Additionally, the PROD part exists for taking the PI
     product of a set of inputs, and the INV command for taking
     the recipicral of a number (divide by zero runtime error
     possible) -->
<Connection to="summer|input1" from="increment_value|output"/>
<Connection to="summer|input2" from="elapsed_time_lp|stored"/>
<IF name="reset_switch"/>
<CONNECTION to="reset_switch|control" from="reset"/>
<CONNECTION to="reset_switch|true" from="zero_constant"/>
<CONNECTION to="reset_switch|false" from="summer|output"/>
<!-- Outputs of a subsystem need to have a connection specified -->
<CONNECTION to="elapsed_time" from="reset_switch|output"/>
<COMPARE name="is_count_expired" operation=">="/>
<CONNECTION to="is_count_expired|lhs" from="elapsed_time"/>
<CONNECTION to="is_count_expired|rhs" from="time"/>
<CONNECTION to="count_expired" from="is_count_expired|output"/>
<MEM name="elapsed_time_lp" initial_condition="0.000"/>
<CONNECTION to="elapsed_time_lp|current" from="elapsed_time"/>
<MEM name="count_expired_lp" initial_condition="false"/>
<CONNECTION to="count_expired_lp|current" from="count_expired"/>
<!-- All Outputs need to have a connection in the part,
     at least to a constant -->
<OUTPUT scope="local" name="count_expired" size="1" type="boolean"/>
<OUTPUT scope="local" name="elapsed_time" size="1" type="single"/>
</BLOCK>
```