IDDS API Builder

By David Parker

**Introduction**

The IDDS API Builder is a tool for converting Instrument Driver Excel API specifications into compatible Instrument Driver Development Studio (IDDS) files. This allows the developer to work more efficiently by not having to re-translate the API into the IDDS and ultimately convert his/her API directly into LabVIEW code. By following the structure and grammar outlined in this file, one will be able to compile an Excel spreadsheet with the API Builder tool.

**Setup**

Before one can get started, there are few things that (s)he must have.

1) The latest copy of the LabVIEW Instrument Driver Development Driver Studio (IDDS). This can be downloaded for free at <http://www.ni.com/gate/gb/GB_EVALTLKTLVIDDS/US>.

2) A copy of LabVIEW 8.0 or newer.

3) Familiarity with the guidelines for creating Instrument Drivers, the guidelines can be found at <http://www.ni.com/devzone/idnet/library/instrument_driver_guidelines.htm>.

4) Understanding of the structure rules defined by this document (they are really simple, I promise!).

5) Java Runtime Environment 7.0 or newer.

**Usage**

The API Builder program comes in two forms, a GUI Java Applet that allows you to select your excel file and compile it.

It also includes a command line version of the same program in the bin folder. To run the command line program simply type:

java –jar apibuilder.jar path/to/excelsheet.xlsx [–t]

The –t option will disable the automatic creation of Template Vis which is enabled by default.

The spec file will be created in the API Builder directory and will be called spec.driver. Copy and save this file to the location and name of your choosing, but note that spec.driver in the API Builder directory will get replaced with every compile. You can now open this spec file from IDDS.

**Table Of Contents**

1. Template API Sheet

2. Instrument Information

3. Folders

i. Creating Folders

ii. Nesting Folders

4. Vis

i. Creating Vis

ii. Multi-Line Vis

iii. Naming Vis

5. Controls

i. Creating Controls

ii. Control Types

6. Commands

i. Adding Commands

ii. Multi-Control Commands

7. Comments

8. Compile Errors

i. Instrument Information Errors

ii. Folder Errors

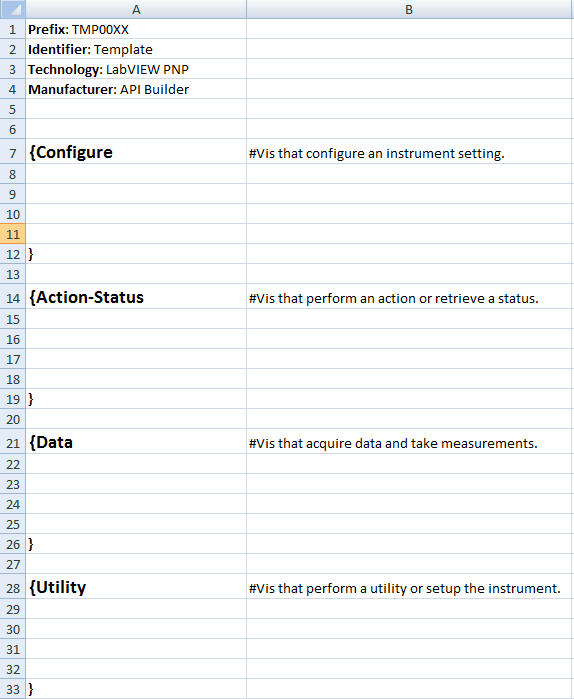
iii. Vi Errors

iv. Control Errors

v. Command Errors

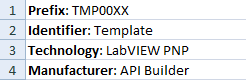
1) **Template API Sheet**

Included in the API Builder directory is a template excel API spreadsheet named template.xlsx. This file outlines all the basics for creating a LabVIEW Instrument Driver API. Try compiling and opening it in IDDS.



2) **Instrument Information**

As seen in the top left corner of the picture above, the first thing that the API compiler will look for are four keywords. “Prefix”, “Identifier”, “Technology”, and “Manufacturer”. These keywords may be placed anywhere in the spreadsheet as long as they are not contained inside a folder hierarchy. The names are also non-case sensitive, however it is recommended that these four keywords be placed in the top left corner of the spreadsheet like in the template.



3) **Folders**

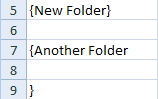
Folders are an important way of categorizing and grouping your Vis. They also represent a hierarchical view of the structure of your API.

i. Creating Folders

To create a new folder simply use the open curly brace followed by a name. All Excel formatting information like color, size, boldness etc. are all ignored.

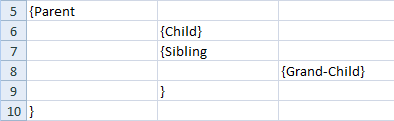


It is necessary to close every new folder with a matching closing brace. Closing braces can be on the same line immediately proceeding the folder name to create an empty folder, or the brace can be found on a new line in order to allow room for sub-folders and Vis.



ii. Nesting Folders

As mentioned above, every folder can contain sub-folders, and those sub-folders can then have sub-folders ad infinitum. To create a sub-folder simply create a folder as you would in the previous step but fit it inside the braces of the parent folder. While one can technically do this on the same column, it is recommended to put sub-folders in the next column to create a visual hierarchy.

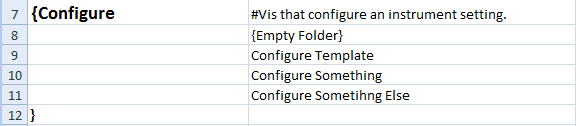


4) **Vis**

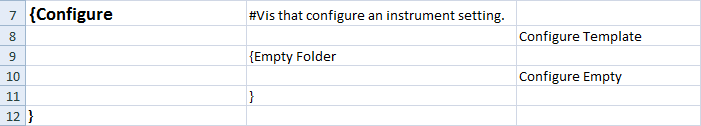
The Vis are what an instrument driver is all about. They are the functional sub-routines that programmatically instruct the instrument to perform an action.

i. Creating Vis

Creating Vis couldn’t be simpler. Anything under a folder that is not another folder, is a Vi.



This creates 3 empty Vis, “Configure Template”, “Configure Something” and “Configure Something Else”. Note that Vis can be created in any and every folder of a folder hierarchy.

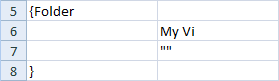


“Configure Template” will be created under Configure and “Configure Empty” will be created under Empty Folder.

It is recommended, but not required that you line up all the Vis in the column adjacent to the column of the last folder hierarchy. In other words, all Vis line up in their own column as shown above to improve readability.

ii. Multi-Line Vis

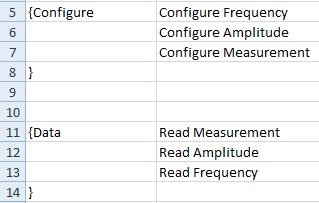
Almost always, a developer will want to group multiple controls into one Vi, this can be done using two double quotes to specify that this control (we’ll get to these in a bit) will be under the same Vi.



iii. Naming Vis

The naming conventions can be found from reading the Instrument Driver Development Guidelines (found above). But as a general convention configuration Vis all start with the word “Configure”, and all data Vis start with the word “Read”.

Duplicate Vi names are not allowed in the IDDS and subsequently not allowed in the API Builder. If a Vi name is duplicated, the API Builder will throw an error.



5) **Controls**

Vis need controls so that users can specify different values to send to the instrument.

i. Creating Controls

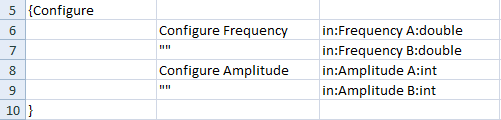
To create a control, simply place it in the column directly right of the Vi that you want the control to be in. The control must be in the format of in/out:name:type. So for example, a frequency input would look like in:Frequency:double and an output called message would look like out:Message:string all of the different types are specified in the next section.



Alternatively, if you don’t want to specify a control and just want to add a command (we’ll get to these next) you can use the none keyword in lieu of the control creation format.



Remember that to add multiple controls in the same Vi you must use double quotes “”.



ii. Control Types

There are six different control types in the API Builder, integers, doubles, strings, booleans, rings and a no-type (none). Note that control types are not case sensitive, “iNt”, “INT” and “inT” all evaluate to “int”. Some control types can even be specified in more than one way, check the complete list below.

|  |  |  |
| --- | --- | --- |
| Control Type | Alias 1 | Alias 2 |
| Integer | Int | Integer |
| Double | Dbl | Double |
| Boolean | Bool | N/A |
| String | String | N/A |
| Ring | Ring | N/A |
| None | none | N/A |

Rings and booleans are special data types. They require an additional parameter that specifies the value enumerations. These are specified by adding : after the control type and giving the enumeration values.

There are many ways that these enumerated values may be formatted. The delimiters {} and () are completely ignored, effectively making {A,B,C} and (A,B,C) equivalent to A,B,C.

The values are tokenized by a space, a comma or a vertical bar.

For example:

in:My Ring:ring: A B C

in:My Ring:ring: A,B,C

in:My Ring:ring: A|B|C

All evaluate to an enumeration list of values A B and C. This allows for a large variety of SCPI commands to be correctly read and parsed. Again adding braces or parenthesis would not change the way that the values are parsed.

In:My Ring:ring:{A|B,C D} is still the same as in:My Ring:ring:A B C D

**NOTE**: Booleans values are formatted the same way as rings but they **MUST** have exactly two values or the compiler will throw an error.



6) **Commands**

Adding commands to controls will allow the IDDS to automatically generate block diagram code with the right formatting to send command information to the instrument.

i. Adding Commands

To add a command to a control simply paste the command to the column directly to the right of the control.



To add a command without specifying any controls use the none keyword. This will add a command to the Vi without a corresponding control.



ii. Multi-Line Commands

Commands specify how many controls they require by the number of spaced input tokens. The first example shows one because of the <value> after the command. If a command looked like this

CMD A B C

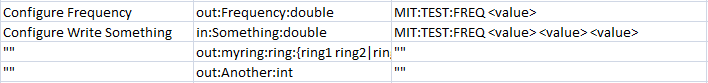
or

CMD: <val> <val> <val>

Then it is required that the user specify three controls for that command, one for each token that is delimited by a space. If your command doesn’t have spaces to delimit inputs, you must add spaces in between each one if you want to make use of multi-line commands. For example,

CMD: <val><val><val> would be the same as CMD: <val>

The syntax for multi-line commands is the same as multi-line Vis, two double quotes “”.



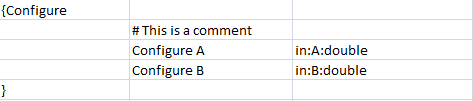
Note that the none keyword cannot be used in multi-line commands, it can only be used by the first command to specify that this command, regardless of the number of input parameters has no controls at all.

Using multi-line commands is tricky at first, but once you get the hang of it, they are easy to implement and they give the IDDS all the information it needs to generate the corresponding LabVIEW code.

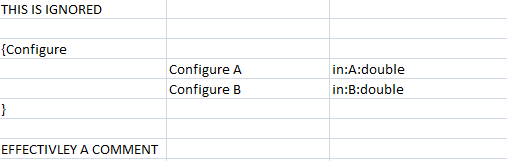
7) **Comments**

Comments let the user write notes and descriptions that won’t have any effect on the syntax of the API.

As you may have already noticed, creating a comment can be done by adding a # as the first character in a cell. That cell will then be treated effectively as if it were empty.



Any text that is not inside a folder structure or is not the four instrument information keywords, is treated as a comment.

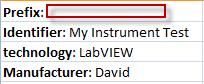


8) **Compile Errors**

If an API Excel sheet compiles and generates an XML file, the API Builder must guarantee that what has been generated is a valid IDDS specification file. Which is why improper use of the syntax and structure described in this document will cause a variety of compile errors.

i. Instrument Information Errors

1 - Missing any of the four instrument information keywords, or having any of the four text fields empty will cause an error.

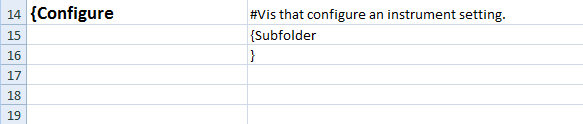


Will throw



ii. Folder Errors

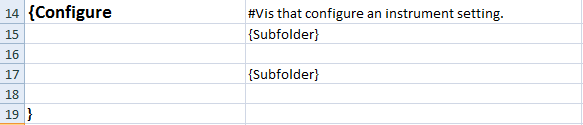
1 - Forgetting to match every open folder brace with a closing brace will throw an error.



Will throw



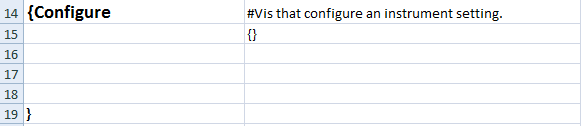
2 - Creating folders on the same hierarchical layer will cause a duplicate name error.



Will throw



3 - Trying to create folders with empty names will also cause an error.

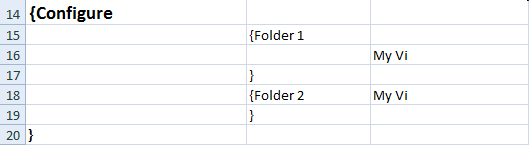


Will throw



iii. Vi Errors

1 – Duplicate Vi names regardless of what folder scope they are in are not allowed and will cause an error.

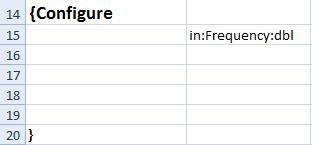


Will throw



iv. Control Errors

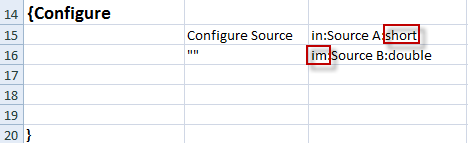
1 – Specifying a control that is not under a Vi will cause an error.



Will throw



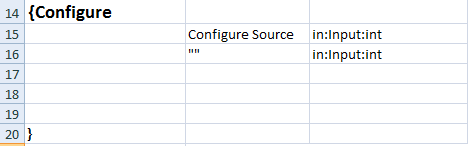
2 – Giving a non-recognized control will cause an error.



Will throw



3- Duplicate control names under the same Vi will cause an error.

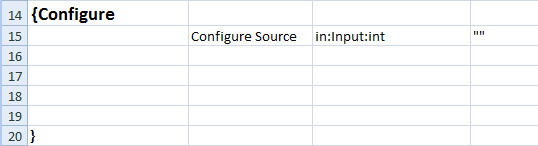


Will throw



v. Command Errors

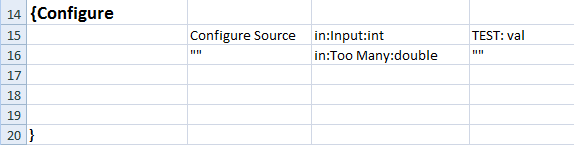
1 – Specifying a multi-line command before any command was ever specified will cause an error.



Will throw



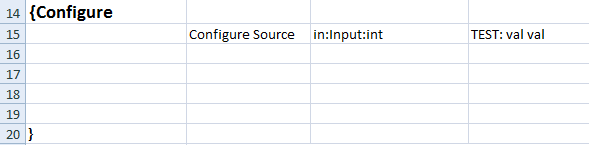
2 – Adding more controls on a multi-line command than the number of tokens will cause an error.



Will throw



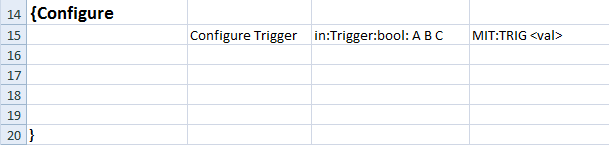
3 – Not adding all the controls that are specified by the number of tokens in a multi-line command will cause an error.



Will throw



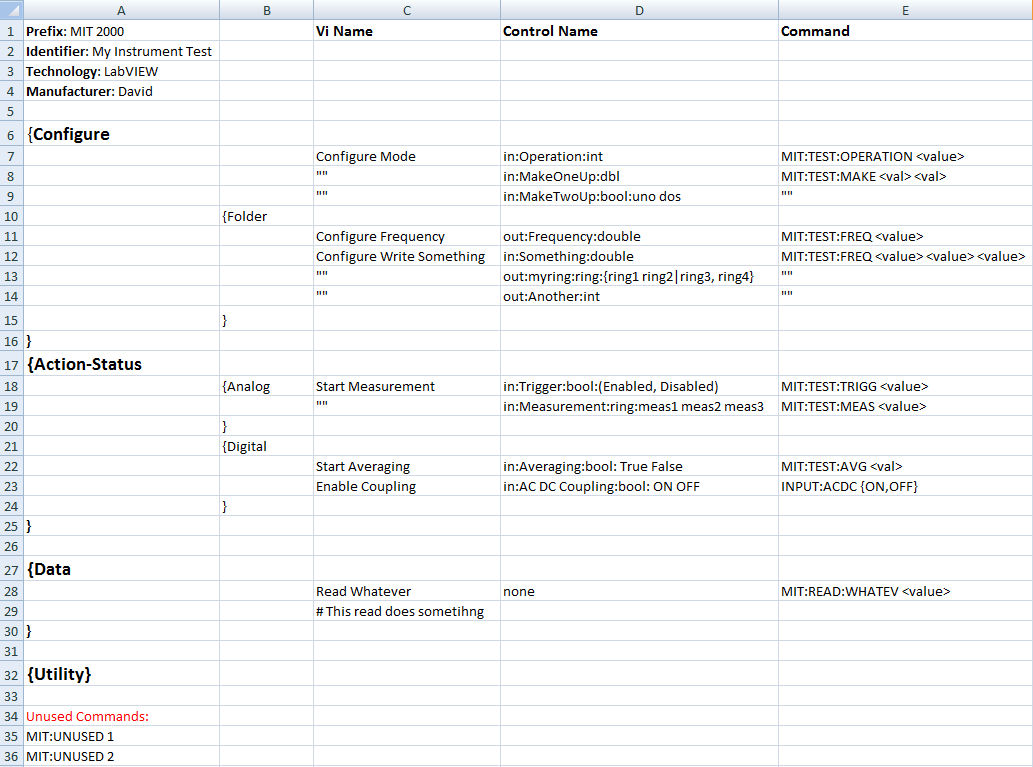
4 – Giving a boolean more than two values will cause an error.



Will throw



That’s it for the API Builder! When all is said in done, you should create API documents that look like this:



Once you have loaded your compiled spec file into the IDDS, you can make modifications and changes before generating the LabVIEW code.

Questions and comments should be directed to david.parker@ni.com