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 document, 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.

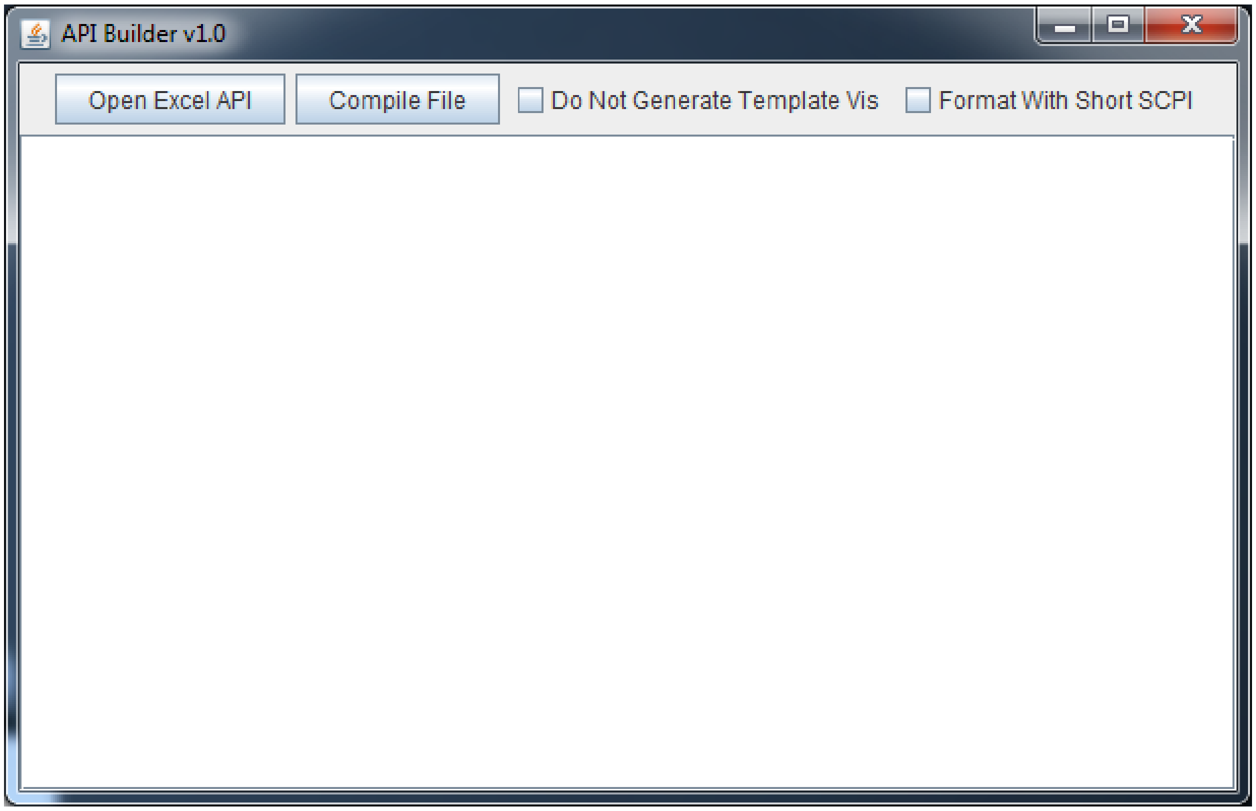
It also includes a command line version of the same program in the bin folder. To run the command line program simply type:

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

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

The –s option will enable the formatting of Short SCPI commands.

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.



1) If you choose to check the "Do Not Generate Template Vis" option, IDDS will require that you recreate all of those Vis before you can generate LabVIEW code. To do this, be sure that all 7 Vis exist in their respective directories and by changing their configuration types to "Template" under the User VI Type. This option is most useful to developers who wish to create custom template Vis, otherwise leave the option unchecked.

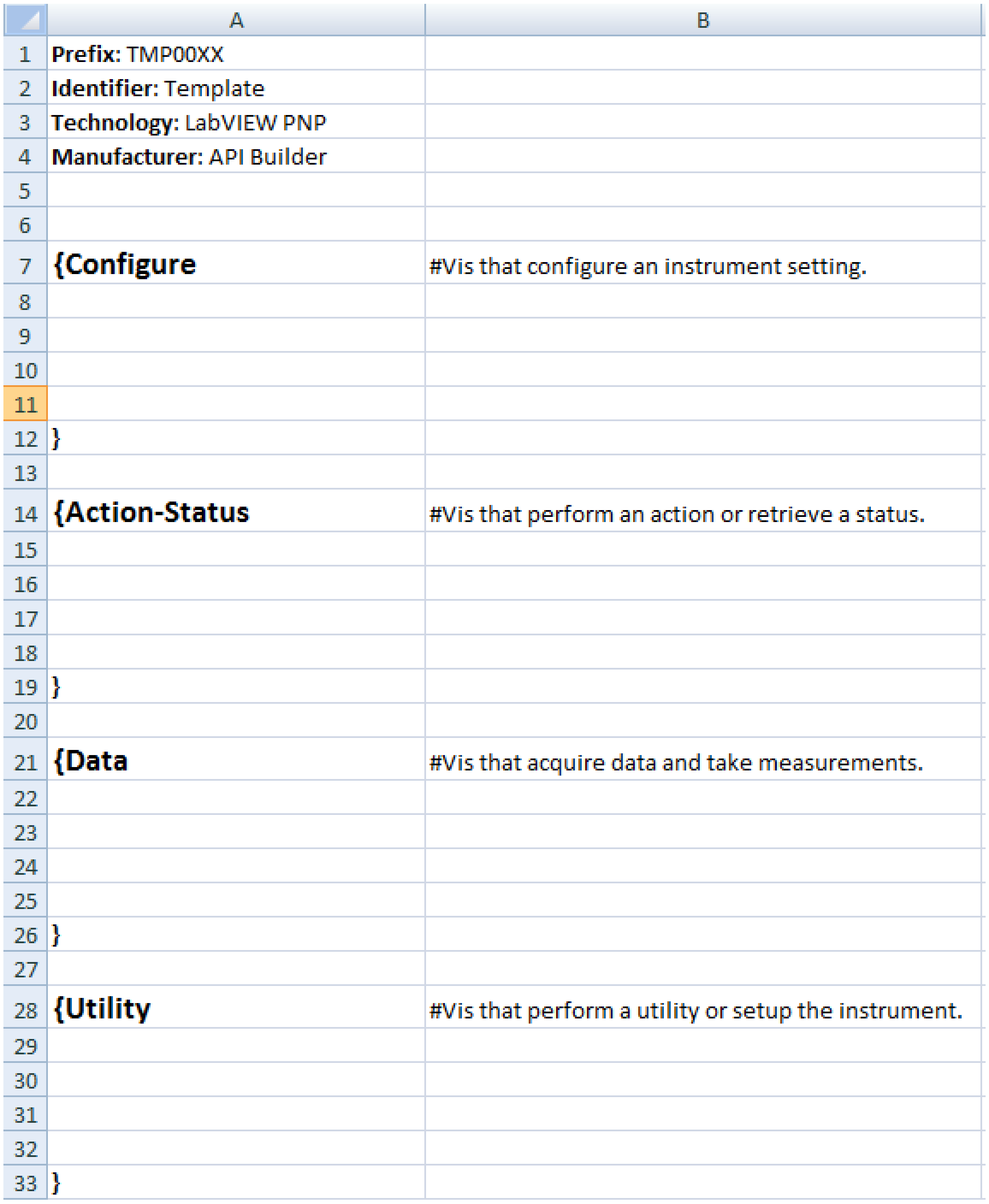
2) The "Format With Short SCPI" option will convert all commands into their condensed SCPI forms. Be sure that your instrument supports this behavior before compiling with this option. For example the command MIT:Test:Command(i) would be changed to MIT:T:C(i).

The rest of this document outlines how to structure an excel based API so that it may be compiled into IDDS code and subsequently LabVIEW code.

Table Of Contents

1) **Template API Sheet**

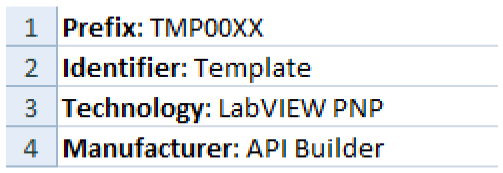
Included in the API Builder directory is a template excel API spreadsheet titled template.xlsx. This file outlines all the basics for creating a LabVIEW Instrument Driver API. Try compiling it with the API Builder and then opening it in IDDS.



	A	B
1	**Prefix:** TMP00XX	
2	**Identifier:** Template	
3	**Technology:** LabVIEW PNP	
4	**Manufacturer:** API Builder	
5		
6		
7	**{Configure**	#Vis that configure an instrument setting.
8		
9		
10		
11		
12	}	
13		
14	**{Action-Status**	#Vis that perform an action or retrieve a status.
15		
16		
17		
18		
19	}	
20		
21	**{Data**	#Vis that acquire data and take measurements.
22		
23		
24		
25		
26	}	
27		
28	**{Utility**	#Vis that perform a utility or setup the instrument.
29		
30		
31		
32		
33	}	

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.



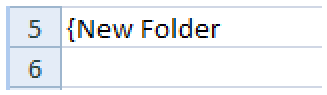
1	**Prefix:** TMP00XX
2	**Identifier:** Template
3	**Technology:** LabVIEW PNP
4	**Manufacturer:** API Builder

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.

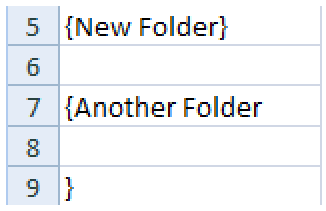
i. Creating Folders

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.



5	{New Folder
6	

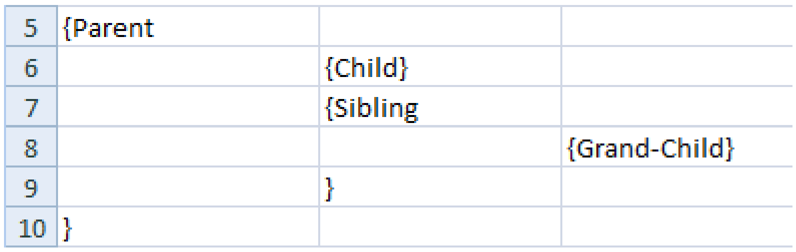
It is necessary to close every new folder with a matching closing brace. Closing braces can be on the same line immediately proceeding the folder name to create an empty folder, or the brace can be found on a new line in order to allow room for sub-folders and Vis.



5	{New Folder}
6	
7	{Another Folder
8	
9	}

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.



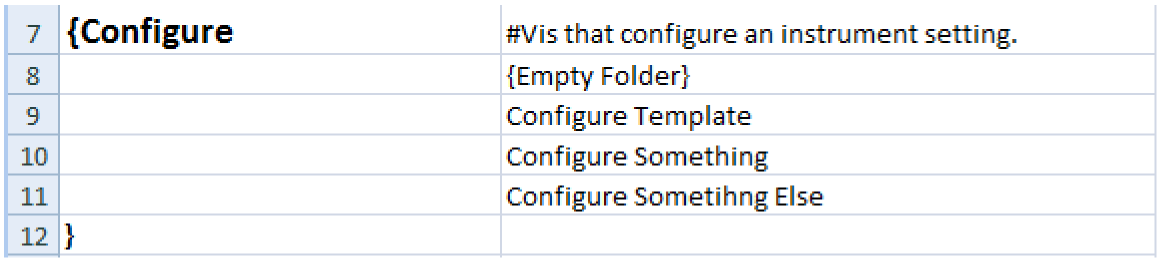
5	{Parent		
6		{Child}	
7		{Sibling	
8			{Grand-Child}
9		}	
10	}		

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.

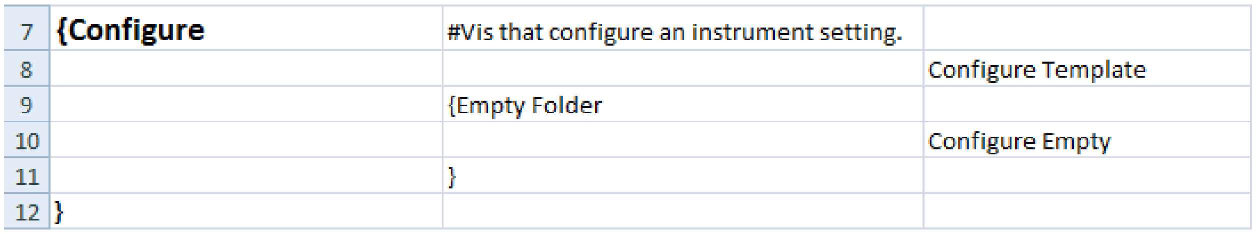
i. Creating Vis

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



7	**{Configure**	#Vis that configure an instrument setting.
8		{Empty Folder}
9		Configure Template
10		Configure Something
11		Configure Sometihng Else
12	}	

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.



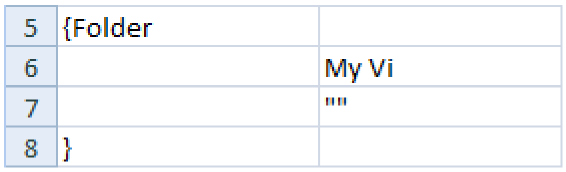
7	**{Configure**	#Vis that configure an instrument setting.	
8			Configure Template
9		{Empty Folder	
10			Configure Empty
11		}	
12	}		

"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.

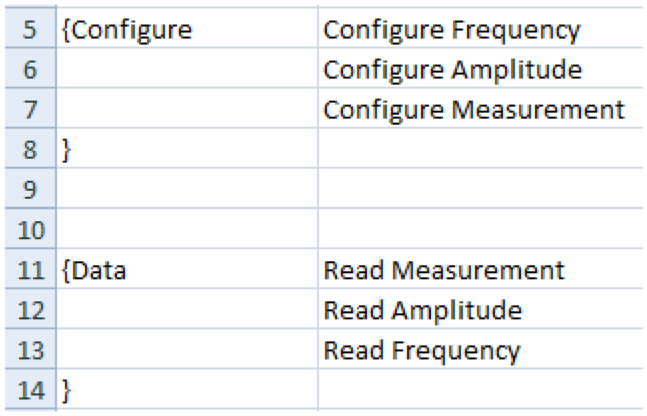


5	{Folder	
6		My Vi
7		""
8	}	

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.



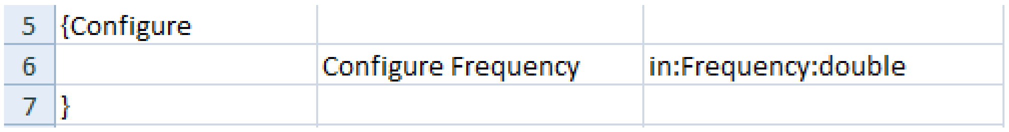
5	{Configure	Configure Frequency
6		Configure Amplitude
7		Configure Measurement
8	}	
9		
10		
11	{Data	Read Measurement
12		Read Amplitude
13		Read Frequency
14	}	

5) **Controls**

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

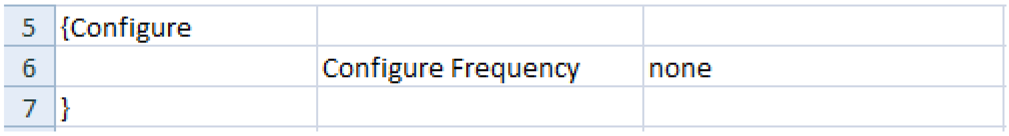
i. Creating Controls

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 control types are specified in the next section.



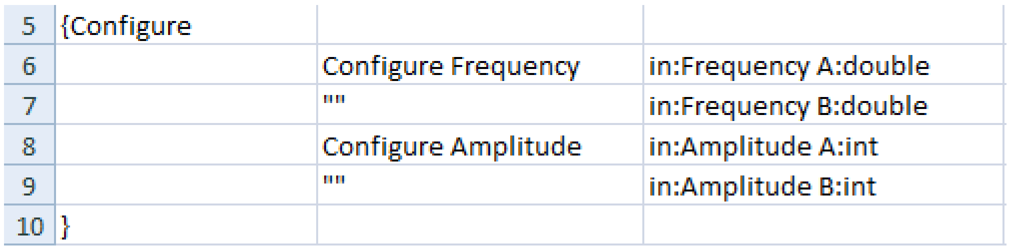
5	{Configure		
6		Configure Frequency	in:Frequency:double
7	}		

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.



5	{Configure		
6		Configure Frequency	none
7	}		

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



5	{Configure		
6		Configure Frequency	in:Frequency A:double
7		""	in:Frequency B:double
8		Configure Amplitude	in:Amplitude A:int
9		""	in:Amplitude B:int
10	}		

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	Boolean
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

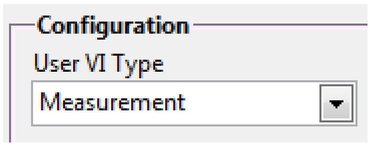
NOTE: Booleans values are formatted the same way as rings but they **MUST** have exactly two values or the compiler will throw an error.

{Analog	Start Measurement	in:Trigger:bool:(Enabled, Disabled)
	""	in:Measurement:ring:meas1 meas2 meas3
}		

iii. Inputs vs. Outputs

Inputs are used for sending user-specified values to the instrument. Outputs are for receiving data sent from the instrument.

Controls that start with “in:” are inputs and controls that start with “out:” are outputs. The API Builder and IDDS treat these two types of controls differently. As explained in the next section, inputs can have commands directly attached to them, I.E the API Builder will automatically format all of the inputs for the given command. Outputs however cannot be automatically formatted by the API Builder nor by IDDS. Therefore, outputs will not have a command and will have to be re-arranged in LabVIEW. Fortunately, IDDS will automatically add a VISA Read VI by selecting Measurement for the Vi’s “User VI Type” field in IDDS.


Configuration
User VI Type
Measurement

To create a simple Read Vi for a query command, one could do this:

{Data			
	Read Stuff	none	MIT:QUERY?
	""	out:Output:string	
}			

6) **Commands**

Adding commands to controls will allow the IDDS to automatically generate block diagram code with the right formatting for sending 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.

Configure Mode	in:Operation:int	MIT:TEST:OPERATION <value>

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

Read Whatever	none	MIT:READ:WHATEV <value>

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>, because there isn't a space in between each <val>.

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

Configure Frequency	out:Frequency:double	MIT:TEST:FREQ <value>
Configure Write Something	in:Something:double	MIT:TEST:FREQ <value> <value> <value>
""	out:myring:ring:{ring1 ring2|rin	""
""	out:Another:int	""

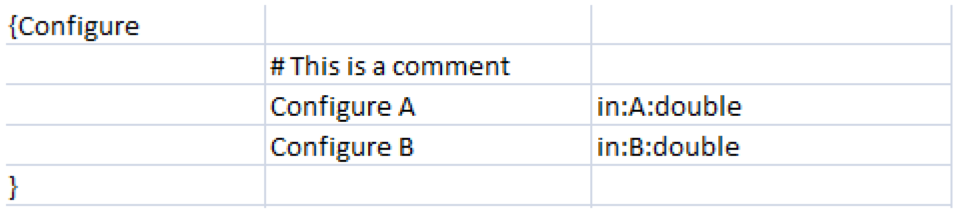
Note that the none keyword cannot be used in multi-line commands, it can only be used by the first line to specify that this command, regardless of the number of input parameters has no controls at all.

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**

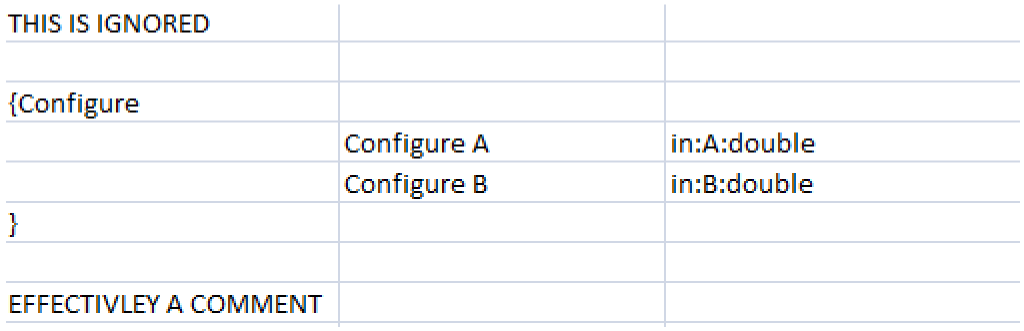
Comments let the user write notes and descriptions that won't have any effect on the syntax of the API.

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.



{Configure		
	# This is a comment	
	Configure A	in:A:double
	Configure B	in:B:double
}		

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



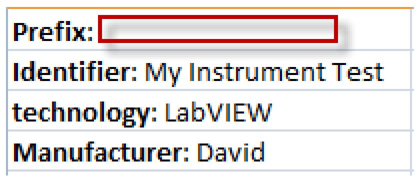
THIS IS IGNORED		
{Configure		
	Configure A	in:A:double
	Configure B	in:B:double
}		
EFFECTIVLEY A COMMENT		

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.

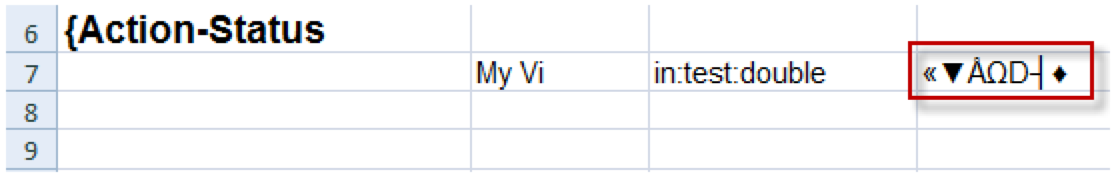


Prefix:
Identifier: My Instrument Test
technology: LabVIEW
Manufacturer: David

Will throw

```
[Compile Error]: Instrument Prefix has not been set.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

2 – The use of non-ASCII characters in any part of the Excel document except comments will cause an error.



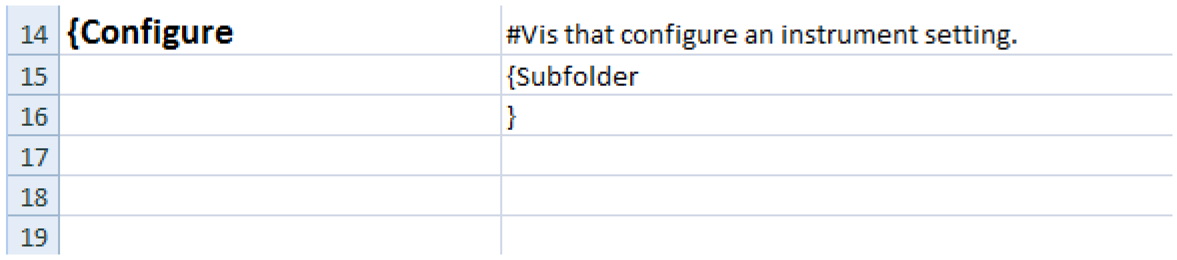
6	**{Action-Status**			
7		My Vi	in:test:double	«▼ÅΩD┤♦
8				
9				

Will throw

```
[Compile Error]: A non-ASCII character was found at line 7.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

ii. Folder Errors

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

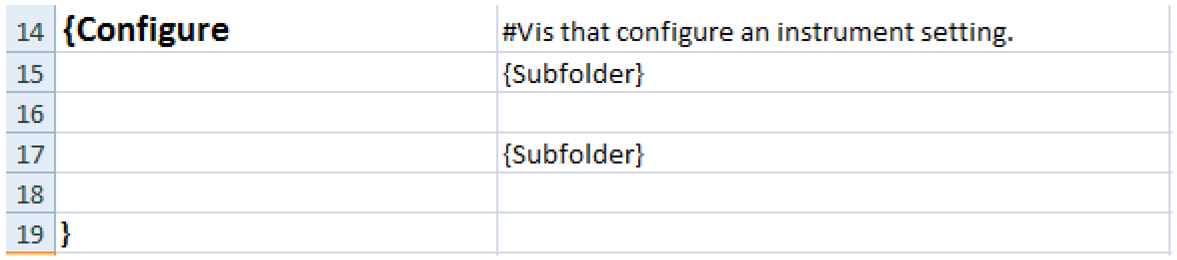


14	**{Configure**	#Vis that configure an instrument setting.
15		{Subfolder
16		}
17		
18		
19		

Will throw

```
[Compile Error]: Missing closing brace for line 14.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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

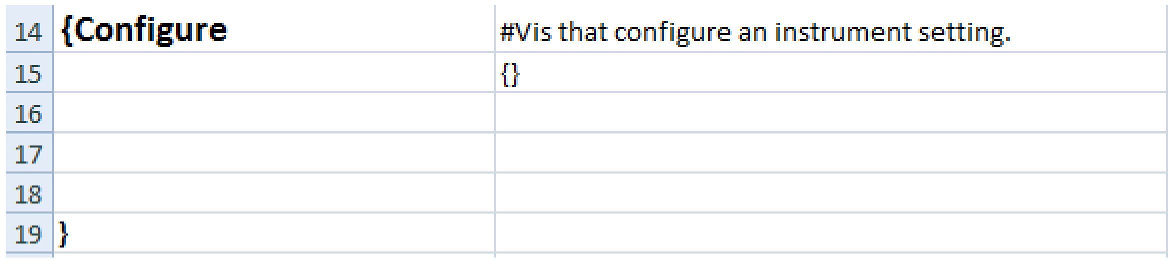


14	**{Configure**	#Vis that configure an instrument setting.
15		{Subfolder}
16		
17		{Subfolder}
18		
19	}	

Will throw

```
[Compile Error]: Duplicate folder name in the same scope found at line 17.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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



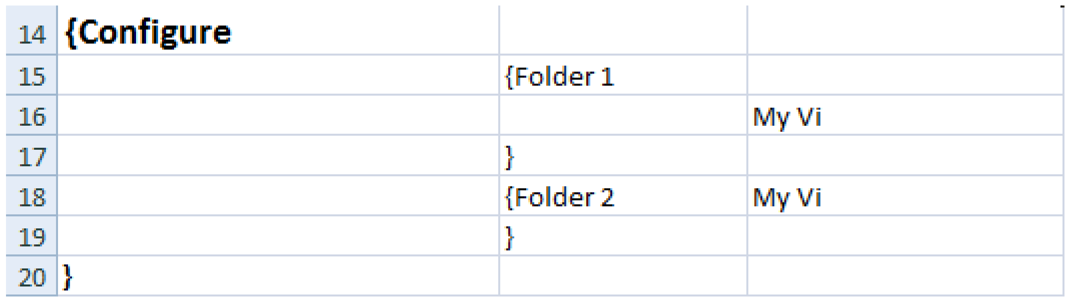
14	**{Configure**	#Vis that configure an instrument setting.
15		{}
16		
17		
18		
19	}	

Will throw

```
[Compile Error]: Folders cannot have empty names at line 15.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

iii. Vi Errors

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



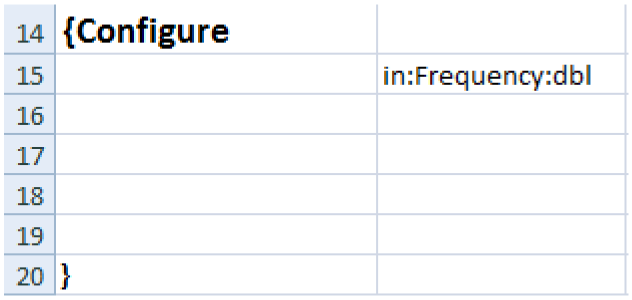
14	**{Configure**		
15		{Folder 1	
16			My Vi
17		}	
18		{Folder 2	My Vi
19		}	
20	}		

Will throw

```
[Compile Error]: Duplicate Vi name at line 18.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

iv. Control Errors

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

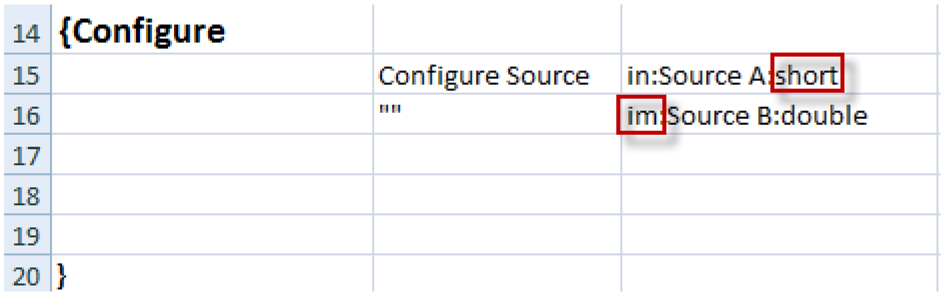


14	**{Configure**	
15		in:Frequency:dbl
16		
17		
18		
19		
20	}	

Will throw

```
[Compile Error]: Control found without a Vi at line 15.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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

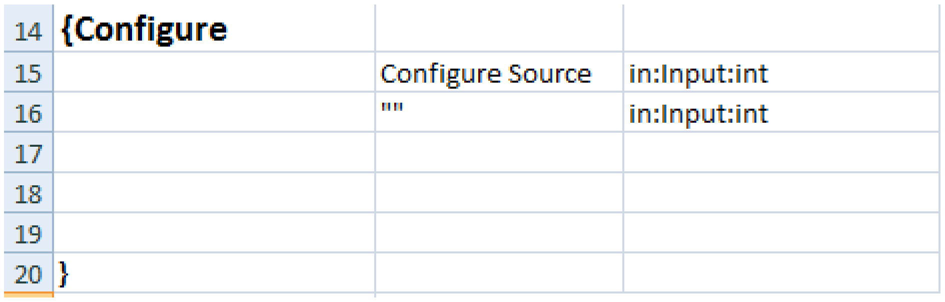


14	**{Configure**		
15		Configure Source	in:Source A:short
16		""	im:Source B:double
17			
18			
19			
20	}		

Will throw

```
[Compile Error]: Invalid control specification at line 15.
[Compile Error]: Invalid control specification at line 16.
[Compile Error]: 2 errors found. Compile failed, exiting program.
```

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



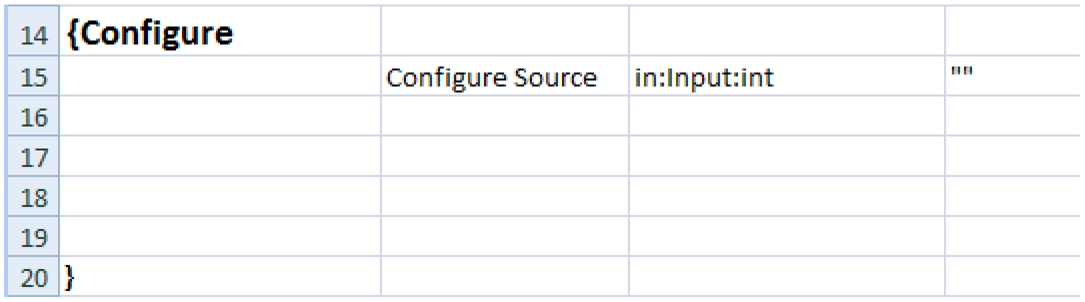
14	**{Configure**		
15		Configure Source	in:Input:int
16		""	in:Input:int
17			
18			
19			
20	}		

Will throw

```
[Compile Error]: Duplicate control name in the same Vi at line 16.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

v. Command Errors

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

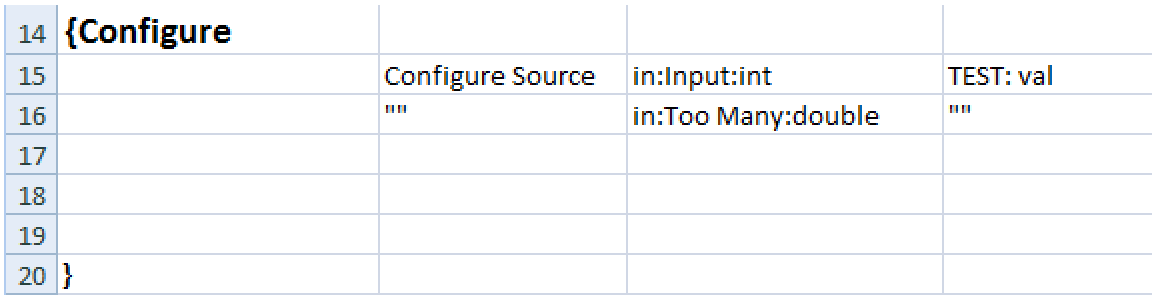


14	**{Configure**			
15		Configure Source	in:Input:int	""
16				
17				
18				
19				
20	}			

Will throw

```
[Compile Error]: No command specified at line 15.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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

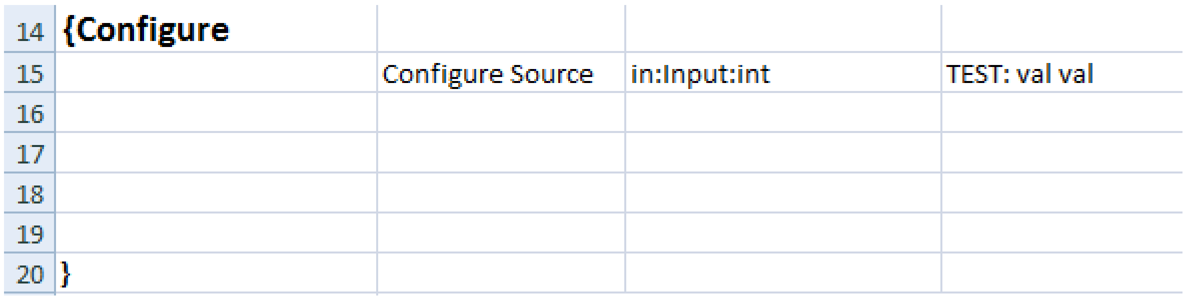


14	**{Configure**			
15		Configure Source	in:Input:int	TEST: val
16		""	in:Too Many:double	""
17				
18				
19				
20	}			

Will throw

```
[Compile Error]: The number of controls found exceeded the command's specifications at line 16.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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

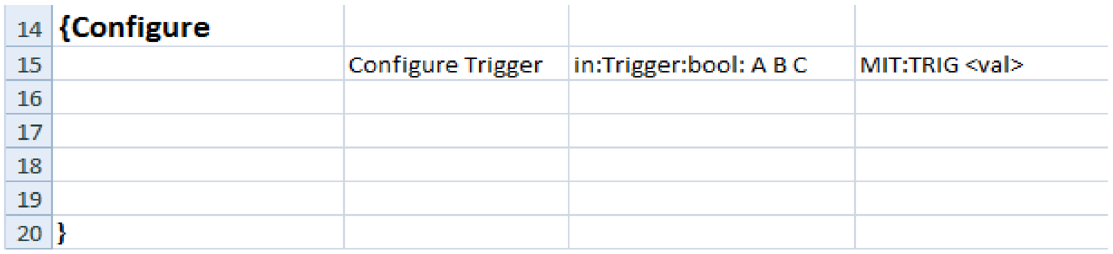


14	**{Configure**			
15		Configure Source	in:Input:int	TEST: val val
16				
17				
18				
19				
20	}			

Will throw

```
[Compile Error]: The number of controls found was less than the command's specifications at line 15.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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

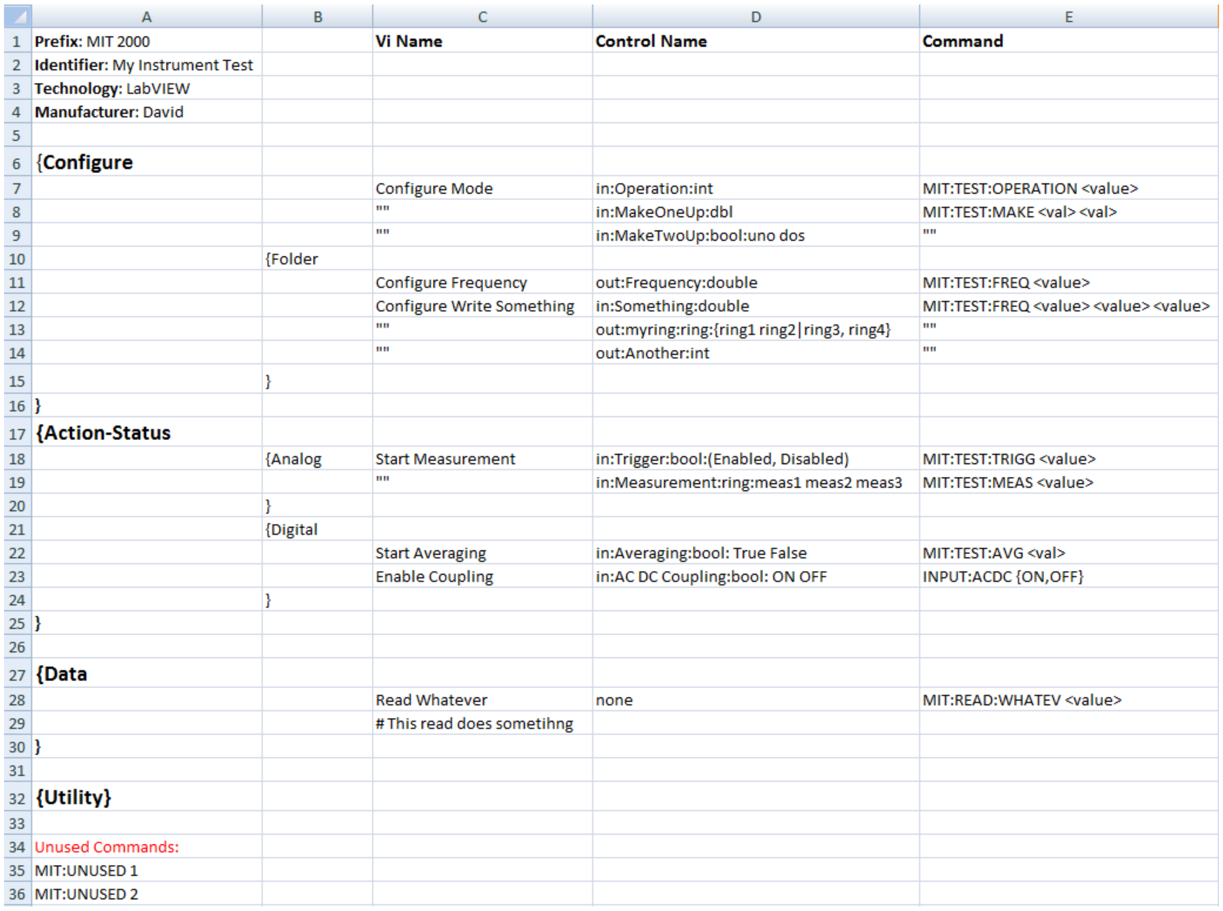


14	**{Configure**			
15		Configure Trigger	in:Trigger:bool: A B C	MIT:TRIG <val>
16				
17				
18				
19				
20	}			

Will throw

```
[Compile Error]: A boolean control does not have the specified two values at line 15.
[Compile Error]: 1 errors found. Compile failed, exiting program.
```

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



	A	B	C	D	E
1	**Prefix:** MIT 2000		**Vi Name**	**Control Name**	**Command**
2	**Identifier:** My Instrument Test				
3	**Technology:** LabVIEW				
4	**Manufacturer:** David				
5					
6	{**Configure**				
7			Configure Mode	in:Operation:int	MIT:TEST:OPERATION <value>
8			""	in:MakeOneUp:dbl	MIT:TEST:MAKE <val> <val>
9			""	in:MakeTwoUp:bool:uno dos	""
10		{Folder			
11			Configure Frequency	out:Frequency:double	MIT:TEST:FREQ <value>
12			Configure Write Something	in:Something:double	MIT:TEST:FREQ <value> <value> <value>
13			""	out:myring:ring:{ring1 ring2|ring3, ring4}	""
14			""	out:Another:int	""
15		}			
16	}				
17	**{Action-Status**				
18		{Analog	Start Measurement	in:Trigger:bool:(Enabled, Disabled)	MIT:TEST:TRIGG <value>
19			""	in:Measurement:ring:meas1 meas2 meas3	MIT:TEST:MEAS <value>
20		}			
21		{Digital			
22			Start Averaging	in:Averaging:bool: True False	MIT:TEST:AVG <val>
23			Enable Coupling	in:AC DC Coupling:bool: ON OFF	INPUT:ACDC {ON,OFF}
24		}			
25	}				
26					
27	**{Data**				
28			Read Whatever	none	MIT:READ:WHATEV <value>
29			# This read does sometihng		
30	}				
31					
32	**{Utility}**				
33					
34	Unused Commands:				
35	MIT:UNUSED 1				
36	MIT:UNUSED 2				

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