**A Brief Overview of the API Builder IDDS Tool**

Purpose: This is a brief walk through of how the tool is being built and its current state. I will include pictures of Excel sheets and source code for clarity.

What Is it? The goal of this tool is to increase Instrument Driver development efficiency. Instead of defining an API in Excel then copy and pasting that information into IDDS, you could write the Excel API in such a form and structure that would allow you to directly convert it into IDDS code.

What Will It Look Like? The intent was to make the Excel APIs still look like… well Excel APIs. I didn’t want to overdue the structure rules to a point where the API sheets looked like source code. The ultimate priority is clarity and readability. So far I feel like this has been achieved and the current state of the design feels really intuitive (this is just an opinion though, I will need to get feedback from Xudong, Yang, Juanita and any other IDDS developers maybe Jia can help too).

Inspiration: The biggest motivational cause for creating this tool came from when I was converting the API of my Stanford Research SR1 Audio Analyzer API into IDDS. This API is really big, it has 235 Vis and nearly 700 commands that span a total of 1300 Excel rows. Even with multiple small tools that I created to help me copy and paste the information from Excel to IDDS, the process still took me nearly an entire work week of painful and tedious API translation. So when I brought this up with Shawn and Juanita we threw around the idea of “what if we could just have a tool to convert the Excel API to IDDS.” And that’s where I am now.

Process

The IDDS is setup into a three step process.

1) Open an Excel spreadsheet file and convert the text into a 2D array of Strings.

2) Parse this new array and set up all the necessary data structures.

3) Traverse these data structures and output the information into XML code that is readable by the IDDS.

Excel API Structure

The structure for creating compliant Excel APIs is simple.

1) The pre-processor will first look for four keywords, “prefix”,”identifier”,”technology” and “manufacturer”. These keywords can be placed anywhere in the Excel document and are NOT case-sensitive. The Pre-processor will ignore any leading spaces but will give a compile error if doesn’t find all four keywords filled out.

2) Braces {} create new folder hierarchies. Any folder can contain two things, Vis and other folders.

3) Writing a name in a folder without braces will create a new VI, the column directly adjacent to that VI will be its control definition (if any, for no control but still send a command use the keyword none). And the column adjacent to the control will be its command, which is also optional but you do not need to specify none.

4) Anything with # in front of it is a comment and will not be parsed. Also text outside of any folder hierarchy will be treated as a comment.

Compile Errors

The API Builder tool includes an extensive error handling / checking class. When the user does something that is deemed illegal, (s)he will get a compile error specifying what (s)he did wrong and what line the error occurred.

Here is a list of compile errors:

1) Missing closing braces for open folders.

2) Creating folders with the same name under the same scope/layer.

3) Creating folders with empty names.

4) Not defining the 4 instrument header keywords.

5) Specifying a control that is not under any VI.

6) Specifying a control with un-recognized data types.

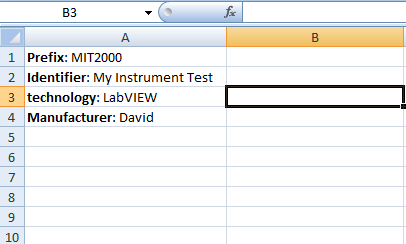
7) Extra closing braces.

If one or more compile error is found, the program will exit after printing the errors that it encountered and no spec file will be created.

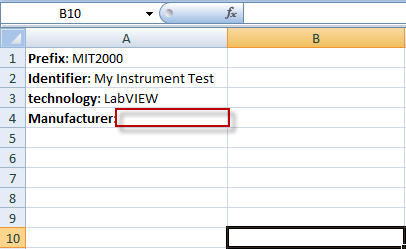
Examples

Here I will give some examples and show how the data is parsed into its hierarchical structure inside Java.

- Setting up the instrument information:



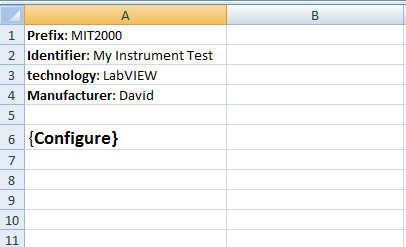
But if you were to not fill out one of the fields then you get this compile error:



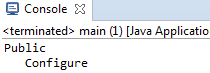


- Defining the folder hierarchy:

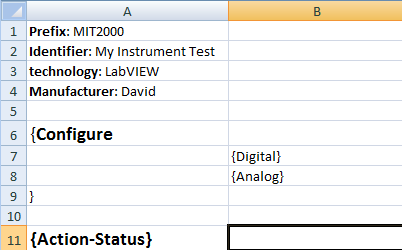
The API Builder ignores formatting things like bold, italics and font size.

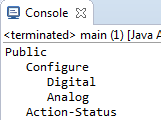


This will create an empty folder called Configure:

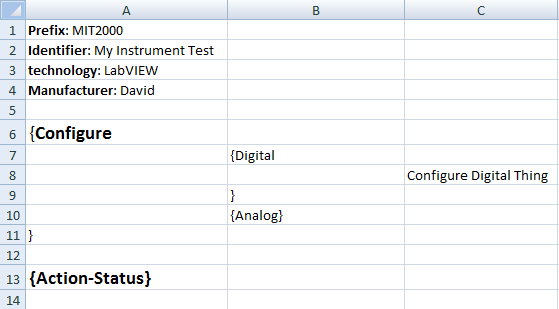


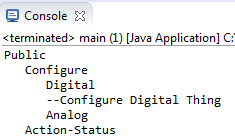
This time we have a larger folder hierarchy:



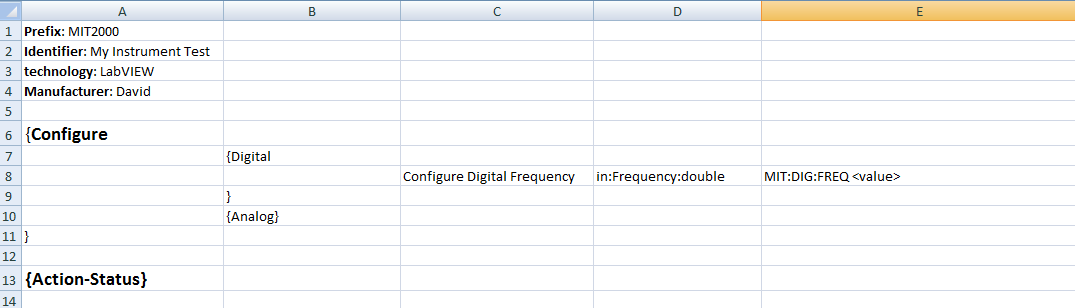


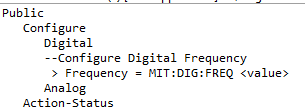
We can also create Vis.



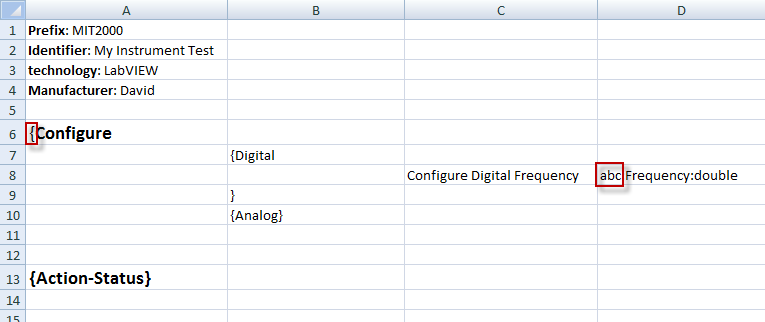


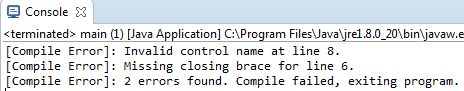
We can give this VI a control and a command, inputs are defined in the format in:name:type and out:name:type, so for a control that sets the frequency, one could do in:Frequency:double



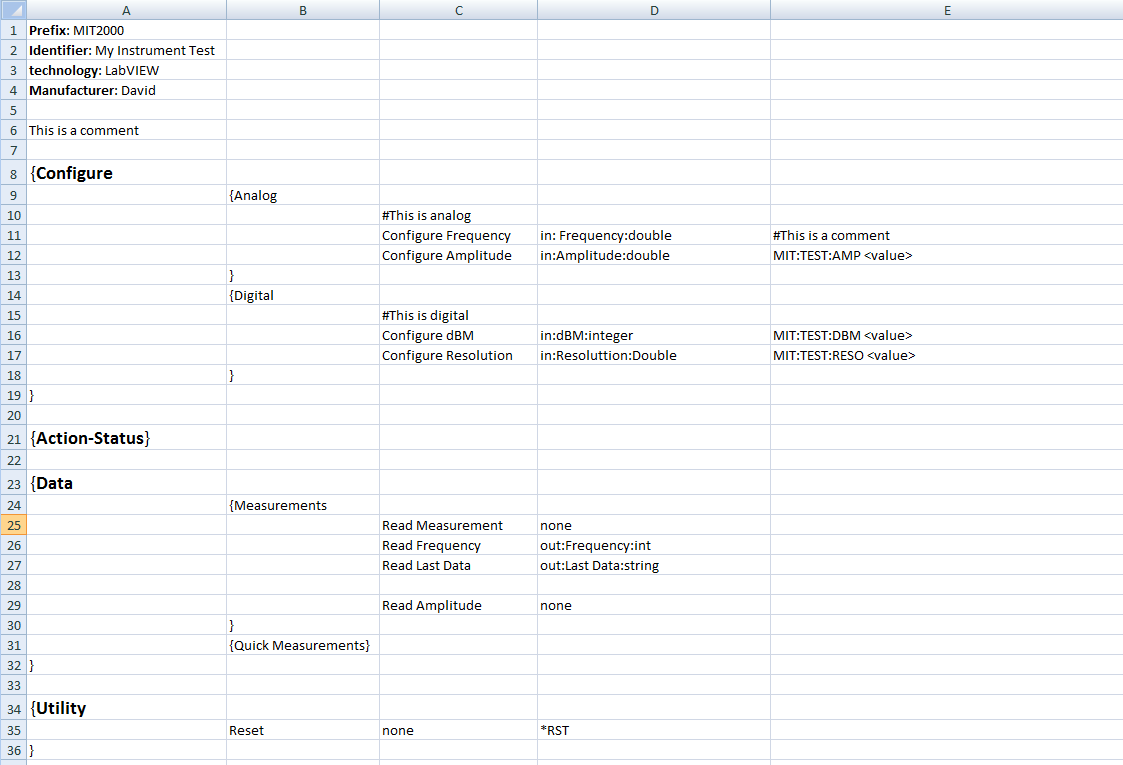


Here’s an example of some compile errors:

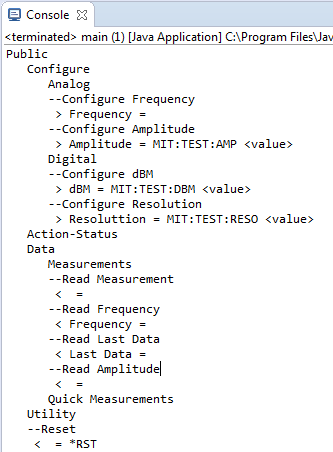




By following this grammar and structure you will be able to create APIs that look like this:

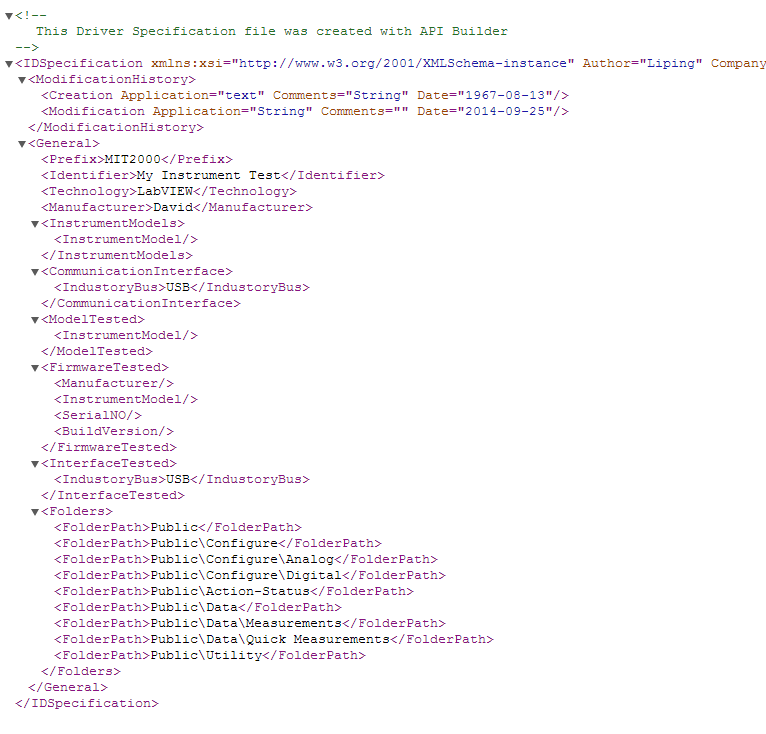


And its tree structure will look like this in Java:



Outputting as IDDS XML Code

Now that the grammar is well defined and the compiler can parse it, I need to export it to IDDS compliant XML code. Currently I have it setting up the general instrument info and the Folder paths:



This is all generated by the XMLWriter class using the information from the SheetParser class that put all the Excel information into an arbitrary sized tree structure. Getting the commands and controls parsed into their corresponding XML format is a little bit trickier and that is where I am currently struggling and researching.

Here is a link to the source code repository.

https://github.com/David-Parker/Tool-Bin/tree/master/Java/API%20Builder