API Programming Guide

MOCVD Growth Sequence Editor LabVIEW Programming Documentation

Introduction

The LabVIEW MOCVD Growth Sequence Editor features a native interface for implementing custom commands which can be executed in a growth sequence. This is called the Application Programming Interface (API). All API VIs placed in the "api" directory relative to "Growth_Sequence_Editor.vi" are dynamically loaded by the editor.

There are three fundamental "modes" around which an API function may be designed. A summary of the available modes is given below.

Mode	Description
Discrete	This mode requires two separate commands in the editor, both of which will be inserted automatically: the "Start" command and the "End" command. API functions using this mode will begin execution when the "Start" command is reached and halt execution when the "end" command is reached.
Continuous	This mode requires only one command. API functions using this mode will begin execution and continue until another command of the same denomination is reached.
Instantaneous	This mode requires only one command. API functions using this mode will be executed once and will not continuously update.

The API framework is designed around the idea of *states*. There are five states that may be implemented: *Edit*, *Start*, *Run*, *End*, and *Idle*. A summary of these states is given below.

State	Description
Edit	This state opens the front panel as a popup window, allowing the user to set any necessary parameters, and then close the VI front panel. Any modifications will be saved if the user selects "OK", and will be discarded if the user selects "Cancel".
Start	This state will be the first state executed when the API VI is called during a growth sequence.
Run	This state will be called continuously between the execution of the *Start* and *End* states of an API VI. The total time elapsed is given as a control, allowing the API VI to make time-based calculations. It is only applicable to *Discrete* and *Continuous* mode.
End	This state will be last state executed when the API VI is called during a growth sequence. It is only applicable to *Discrete* mode.
Idle	This state should do nothing but pass the argument list through.

1.1 Front Panel

In the following tutorial, we will create an example command that will oscillate the flow of gallium as a square wave. A cursory knowledge of LabVIEW programming is assumed.

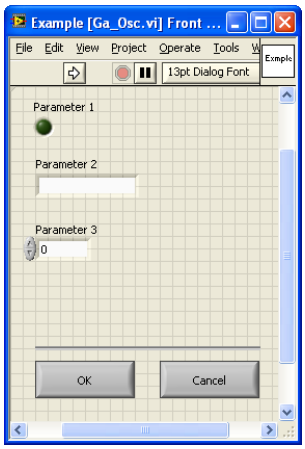

Figure 1

To begin, navigate to the "api" directory and copy Example.vi to a new file. Give the copied VI a new filename, such as "Ga_Osc.vi". Open this file in LabVIEW. Your front panel should match Figure 1.

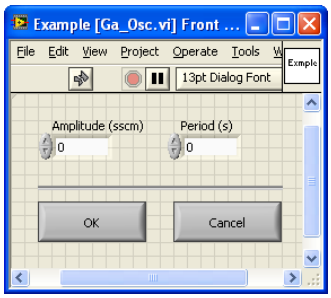

Figure 2

Delete the three controls labeled "Parameter 1", "Parameter 2" and "Parameter 3". Replace them with two numeric controls. Label these "Amplitude (sscm)" and "Period (s)". These will control the amplitude and period of the oscillation. Optionally, you may resize the window and/or move the controls to minimize screen clutter. We are now finished with the front panel.

1.2 API Mode

The first step to writing an API function VI is to set the appropriate API mode. Since oscillating the gallium flow will require time-based calculations, we must use either *Discrete* mode or *Continuous* mode. In this case, we will choose *Discrete* mode because the user will most likely want control over when the oscillation should stop. Most time-based functions should use *Discrete* mode. *Continuous* mode is only applicable to functions for which an additional "halt" command would be inconvenient, such as controlling a temperature set point. *Instantaneous* mode should only be used for simple one-time commands, such as turning a switch on or off.

To open up the block diagram, press Ctrl+E. To switch our API function to *Discrete* mode, locate the "Set API Mode" VI on far left of the block diagram. Select "Discrete" from the enumeration parameter wired to the "Set API Mode" VI. Our API function is now set to *Discrete* mode.

1.3 Argument Lists

On the far left of the block diagram there is a control called "Argument List In". API functions are controlled by the editor via an array of strings called an "Argument List". Arguments can be either a singular text value, such as "SomeArg", or they can contain data in the form of "SomeArg("SomeData")". All data encapsulated in arguments must be stored as strings within the required double quotes. For this reason, routines are proved such as "Number_To_String.vi" (located in the "editor_lib" subdirectory) to easily convert numerical values to strings. "Number_To_String.vi" has similar functionality to LabVIEW's

native typecast functions; however it removes all insignificant leading and trailing zeros from the output string. Other useful routines are also provided in the “editor_lib” directory, relative to the top-level editor VI.

One caveat of this form of data storage is that input strings should be purged of both double quotes and newline characters. A double quote signifies an end of the string to the argument parser, and a newline will cause issues when saving a text file. If either of these characters is necessary, the user is encouraged to use escape sequences, although no internal implementation exists at this time of writing.

1.3 Initialize Controls

Figure 3 shows part of our block diagram thus far. The “Argument List In” is hidden on the front panel, but it is wired to the appropriate terminal for use by the editor. When we open the front panel to our VI from the editor, we want to load any values that were previously set. Otherwise, our VI would only show default values when opened. Furthermore, the “Argument List” also passes the user’s previously selected parameters to the API function while the growth sequence is running, as the front panel will not be visible.

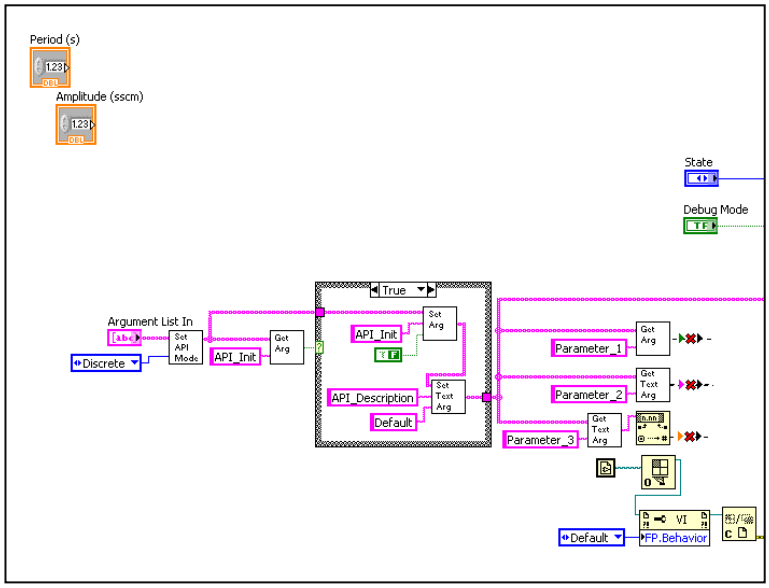

Figure 3

First we will create default values for our two parameters. We can set these inside of the case structure wired to the furthest left instance of the “Get Argument” VI. This VI detects the value of the “API_Init” Boolean parameter, which is set by the editor for all newly inserted commands. When “API_Init” is true, our API function should set any relevant API parameters to suitable default values, then disable “API_Init”.

API Description

A less obvious argument is “API_Description”. This textual argument is used by the editor as a short identifier in the visual commands list. It is not required, but it is convenient under some circumstances. For example, a temperature command may set “API_Description” to the temperature set point, so the user will see “Temperature (500 C)” or “Temperature (300 C)” in the editor. A good default value is “Default”, to signify that the default values of the API function have not yet been modified.

Copy the “Set Text Argument” VI twice, placing the copies inside the case structure. Wire the “Argument List” inputs/outputs to each other as shown in Figure 4. Set the first argument to “Period” with a value of “1”, and set the second argument to

"Amplitude" also with a value of "1". Note that all argument defaults are set via strings, to minimize unnecessary conversions.

Next, we will set the front panel controls to their corresponding values from the argument list. This code must come after we set the default values to ensure that the VI does load any necessary defaults. Delete the "Get Argument" VI's wired for "Parameter 1", "Parameter 2", and "Parameter 3", since those example controls have been deleted. Now create two instances of "Get_Text_Argument.vi", and set their respective argument names to "Period" and "Amplitude". Wire them to the "Argument List" branch coming from the case structure, as shown in Figure 5.

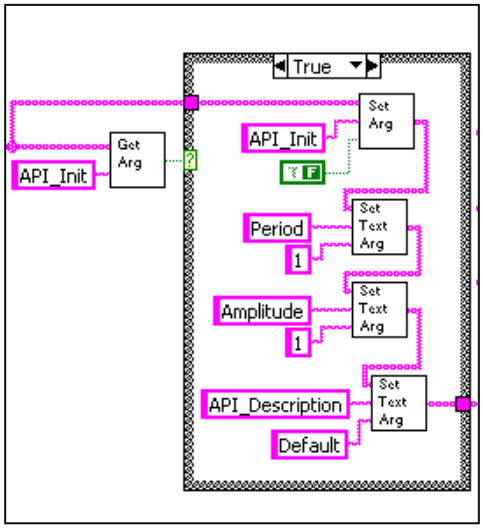

Figure 4

Create two instances of the native LabVIEW function "Fract/Exp String To Number". Wire these to the "Get Text Argument" VIs, as per Figure 5.

Now we need to create two property nodes for the front panel controls. Right click on the "Period (s)" terminal on the block diagram and navigate to Create→Property Node→Value. Place it on the block diagram, then right click the property node and select "Change All To Write". Wire this to the output of the string conversion VI we just created. We now have a property node that will set the value of the "Period" control to whatever its corresponding value is in the "Argument List". Repeat this same procedure for the "Amplitude" control. Your block diagram should now resemble Figure 5.

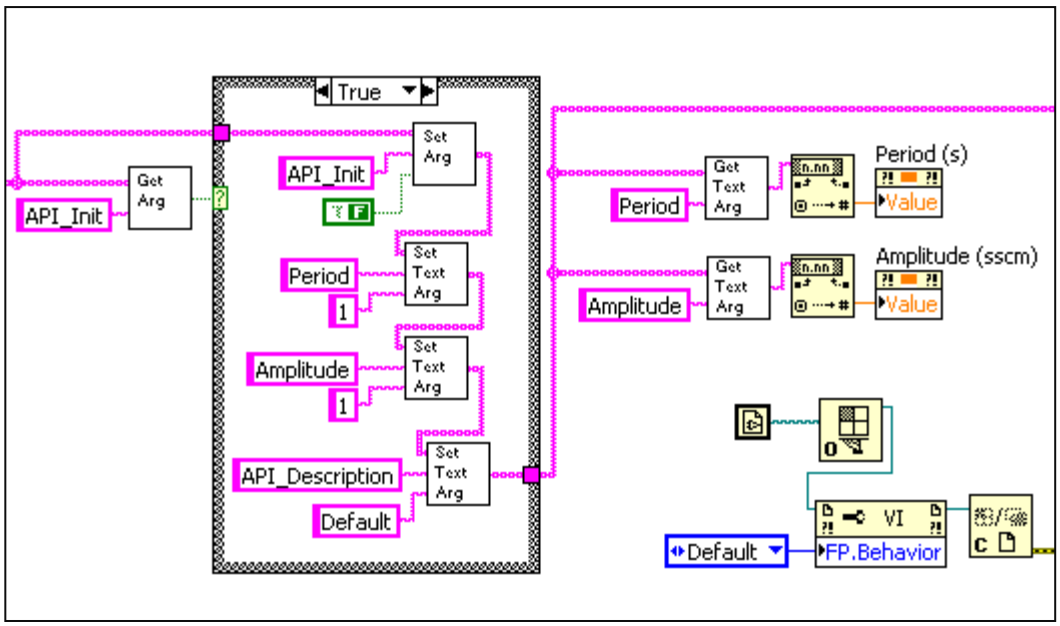

Figure 5

1.4 Save Control Values

After the user has finished modifying the controls on the front panel, the API VI must save those values by returning them to the calling VI through the “Argument String”. Navigate to the large case structure on the block diagram. It should resemble Figure 6.

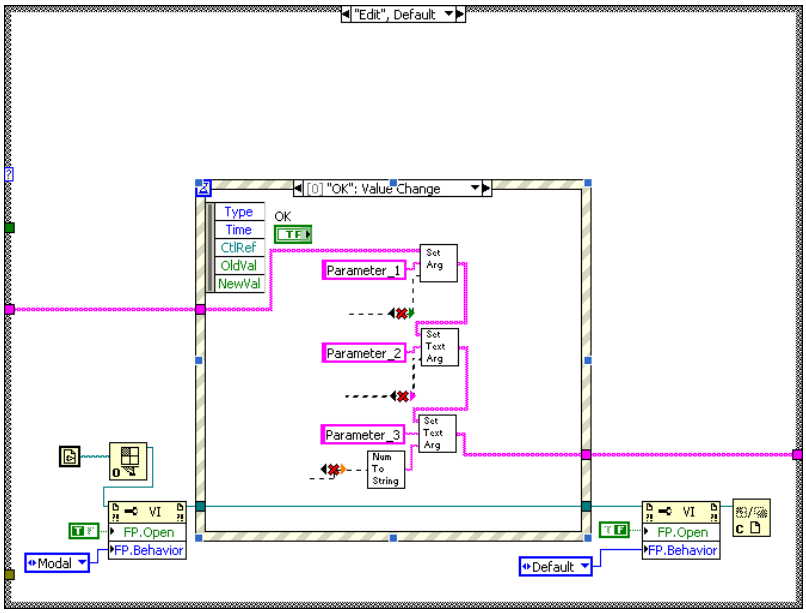

Figure 6

As previously mentioned, there are two types of arguments available: Boolean arguments and text arguments. Boolean arguments store either a “true” or “false” flag, while text arguments store a variable-length string. Use text arguments with the “Number to String” SubVI to store numeric data. The “Number to String” SubVI is similar to the built-in string conversion functions in LabVIEW, but automatically trims off unnecessary zeroes to optimize space and readability.

Find the “Period (s)” and “Amplitude (sscm)” control terminals on the block diagram. Move them into the event structure, and wire them to the “Set Text Argument” terminals as shown in Figure 7. You will need to copy and paste the “Number to String” SubVI. Change the “Argument” parameter on both “Set Text Argument” SubVIs to “Period” and “Amplitude”, respectively.

All functionality related to saving/loading parameters should now be complete. We can now move on to implementing the necessary hardware-level communications.

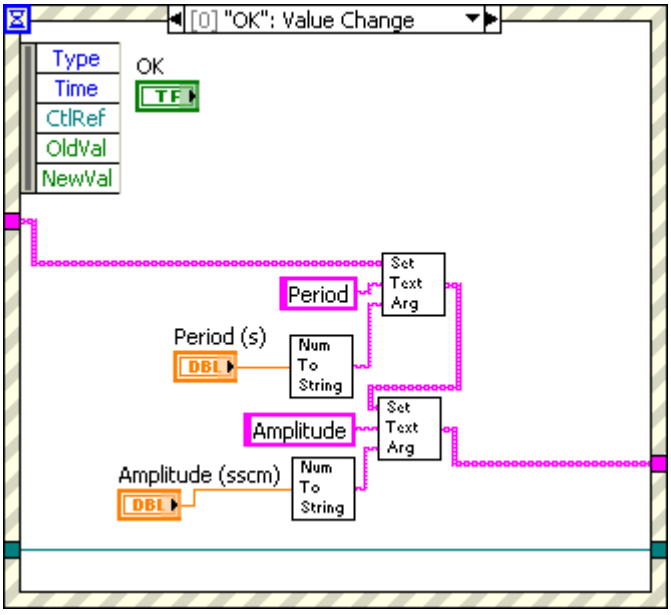

Figure 7

1.5 Start State

In the MOCVD Growth Sequence Editor, "Debug Mode" denotes that no hardware communications should take place. We must keep this in mind when implementing any hardware communications functionality.

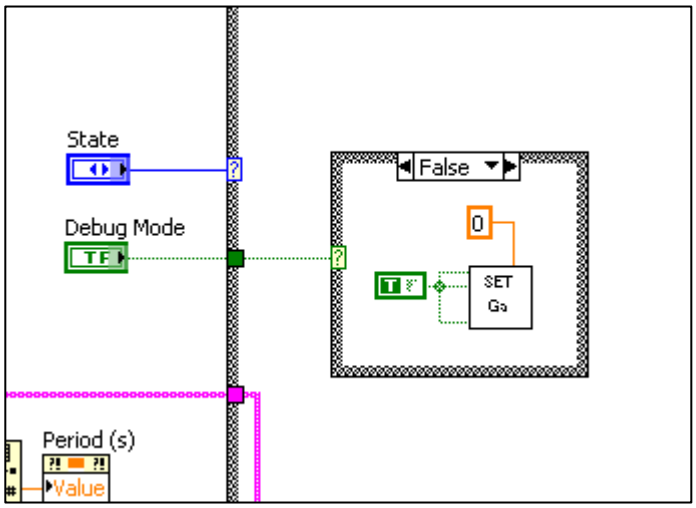

Figure 8

Navigate to the "Start" state of the largest case structure on the block diagram. Create a new case structure and wire its conditional terminal to the "Debug Mode" control. Find "Set_Ga.vi" in the "high_level_proc" subdirectory and drag it into the "False" state of the case structure. Add the appropriate constants to set Gallium to active and initialize its flow to zero, as shown in Figure 8.

1.6 Local Variables

The "Run" state provides us with one input, the "Elapsed Time". This value corresponds to the total time elapsed, in seconds, since the call to the "Start" state. However, we also need to know the time delta to oscillate over a given period. We will create two local variables; one to keep track of the previous elapsed time, and one to count the amount of time since the last period began.

Before we can create local variables, we need first need to create some correlated controls/indicators. Scroll down in the front panel window and add two indicators. Name them "Timer" and "Previous Time". Find the terminals on the block diagram, right-click and select "Hide Indicators". Now resize the front panel window so it appears as it did before the indicators were added. This minimizes screen clutter and hides unnecessary indicators from the user.

1.7 Run State

Navigate to the "Run" state of the large case structure. Since the primary focus of this tutorial is on programming custom commands with the API, the actual implementation of the Gallium oscillation along a square wave has been pre-written. Navigate to the "tutorial subdirectory from the parent "api" directory. Drag "Ga_Osc_Control.vi" onto the block diagram. Open up this SubVI's block diagram. We will only discuss its implementation, not edit its contents. It should look similar to Figure 9.

The run state provides us with only one input: "Elapsed Time". Since we also wish to know the time delta, we must keep track of the elapsed time of the previous call to the function, which we do with a "Previous Time" local variable. We also wish to keep track of the time since the start of the current oscillation's period, which we store in the "Timer" local variable.

The “Ga_Osc_Control.vi” SubVI calculates the time change since the last call, updating the values of “Previous Time” and “Timer”. The “Timer” value is reset to zero once the period length has been reached. During each period, the flow of Gallium is set to zero once if the current time is less than half the period, otherwise the flow is set to “Amplitude”. The Gallium SubVI has the “Remote”, “Active”, “Run”, and “Bubbler” parameters enabled. The current flow rate is output to the “Flow” indicator.

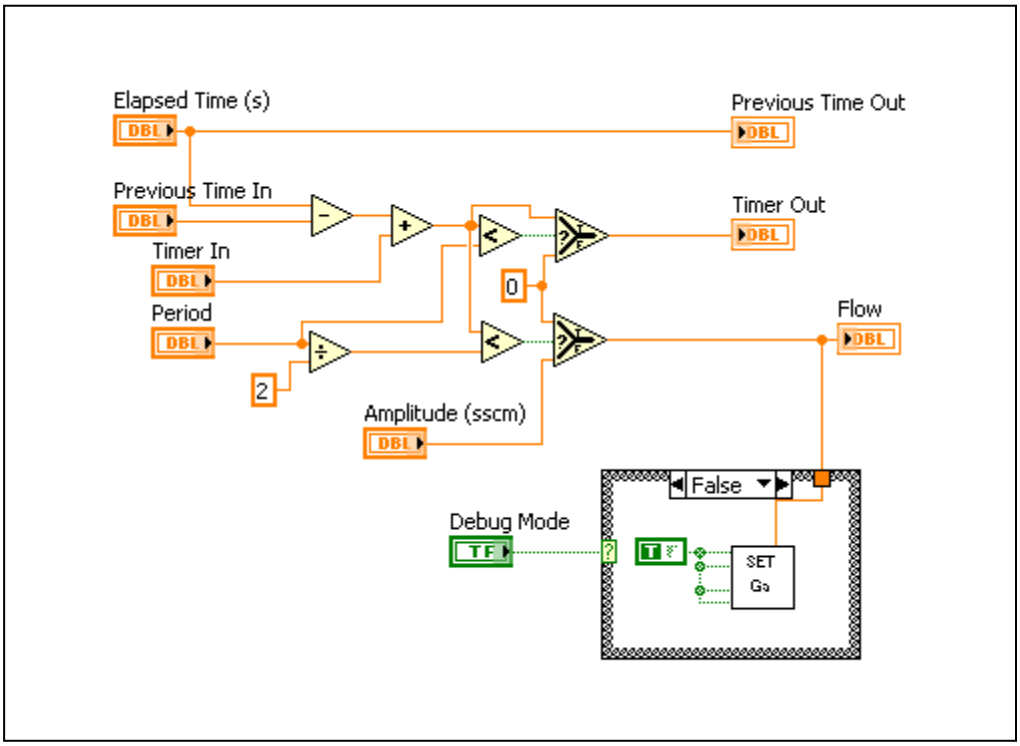

Figure 9

It bears mentioning that a number of options exist for controlling MOCVD flow rates in software. The simplest option is to use the high-level hardware communication functions located in the “high_level_proc” subdirectory. Alternatively, the programmer may make use of the wrapper functions used by the MOCVD Growth Sequence Editor, located in the “editor_lib” subdirectory. These SubVIs include “Flow_Commands.vi” and “Gas_Commands.vi”. The latter two options are more suitable for use in polymorphic design patterns;

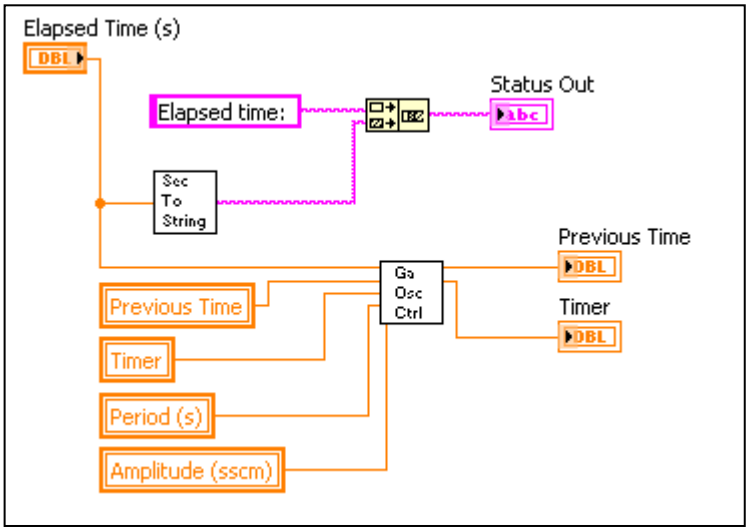

Figure 10

e.g., if the programmer wishes to make a custom command that works for any gas source.

On the primary block diagram, create four new local variables: Previous Time, Timer, Period, and Amplitude. A local variable can either be created from the functions palette, or by right-clicking on the corresponding indicator's terminal on the block diagram and navigating to Create→Local Variable. If the local variables have input terminals instead of output terminals, right-click on them and select "Change to Read". Note that a local variable cannot exist independently; it must correlate to an existing indicator or control. Wire the appropriate terminals to the "Ga_Osc_Ctrl.vi" terminal on the block diagram, as shown in Figure 10.

Local Variables and Property Nodes

As you may have noticed, "Value" property nodes and local variables appear to serve the same function. The difference is that "Value" property nodes require LabVIEW to update the front panel instantly, whereas local variables do not. Consequently, local variables are significantly faster. Use "Value" property nodes when you absolutely need the front panel to be updated and computation speed is unimportant, and use local variables everywhere else.

The "Elapsed Time" parameter is controlled by the calling function, and will thus be reset appropriately. However, the values of "Previous Time" and "Timer" are only controlled locally, and need to be reset during the "Start" state, or else the values will accumulate the next time the function is called, resulting in incorrect timing calculations. In other words, we need to guarantee that the initial values of "Previous Time" and "Timer" are set to zero. To do this, create new local variable instances for "Previous Time" and "Timer", and if necessary, right-click and select "Change to Write". Wire them both to a numeric constant of zero, as shown in Figure 11.

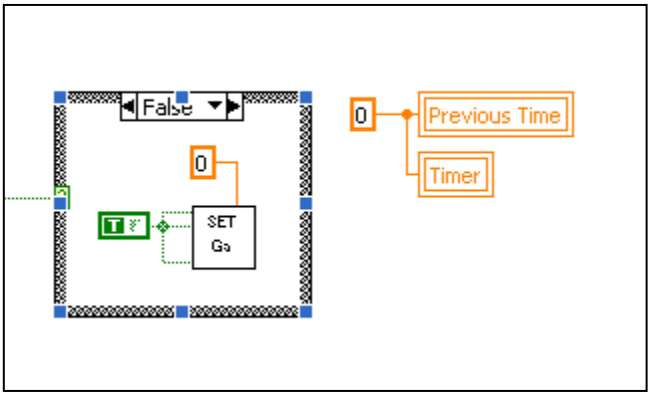

Figure 11

1.8 Status Output

The "Status" of a command should be a short line of text that gives the Growth Sequence Operator some useful information regarding the current state of the function's operation. The example that we started with simply output the elapsed time as the status. Since this isn't much use to us, we will change it to output the current flow rate, which should be equal to either zero or the amplitude parameter. First, delete the "Seconds_To_String.vi" terminal from the block diagram, and replace it with "Number_To_String.vi". Both of these SubVIs are located in the "editor_lib" subdirectory.

Wire the “Flow” output of the “Ga_Osc_Control.vi” terminal to the “Number_To_String.vi” terminal, as shown in Figure 12. Change the preceding string constant to “Current Flow: ”.

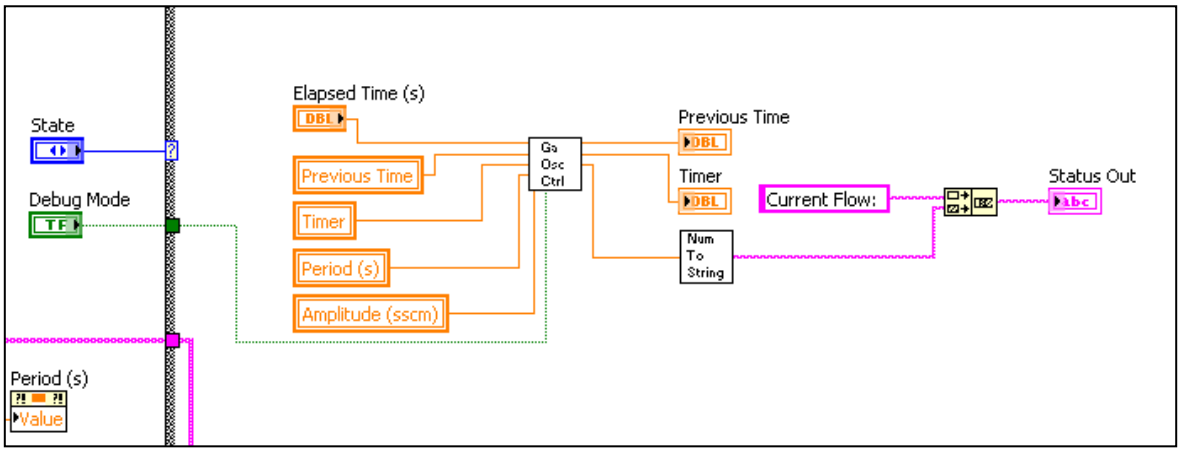

Figure 12

1.9 End State

As it is now, the gallium would continue flowing at whatever the last flow rate was, even after our command has finished running. To turn off the gallium flow, create a new case structure inside the “End” state on the block diagram, as shown in Figure 13. Add “set_Ga.vi” to the “False” case. Set the “Flow 1” parameter to zero, and set the “Remote”, “Active”, “Run”, and “Bubbler” parameters to false. This will turn off gallium once the “End” command is reached.

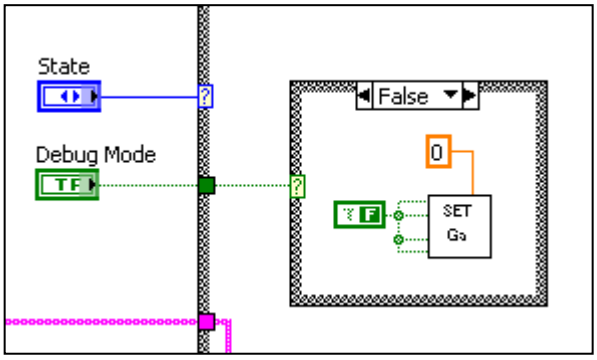

Figure 13

1.10 VI Name

The final necessary step is to change the VI window title to something more suitable for our command. Since we started with an example command, the current window title is “Example”. Navigate to File→VI Properties→Window Appearance. Change the title to something more descriptive, such as “Gallium Oscillation”.

As long as the “Ga_Osc.vi” file is in the “api” subdirectory, the MOCVD Growth Sequence Editor will automatically detect and load it. However, the name of the command in the “Insert” dialog will be the loaded from the “Window Title” property of a given command VI. A descriptive label such as “Gallium Oscillation” is more convenient than “Ga_Osc.vi”.

1.11 Conclusion

To test the VI, launch the MOCVD Growth Sequence Editor and create a new sequence. Insert the “Gallium Oscillation” command into a blank sequence. The editor will create a start and end command, since the API mode of this command is set to “Discrete”. Select “Open External Front Panel” on the “Start” command selection. Change the period to something small, such as three seconds. Press “OK” to save the values.

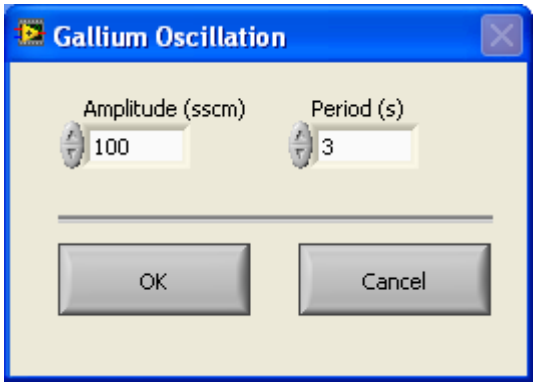

Figure 14

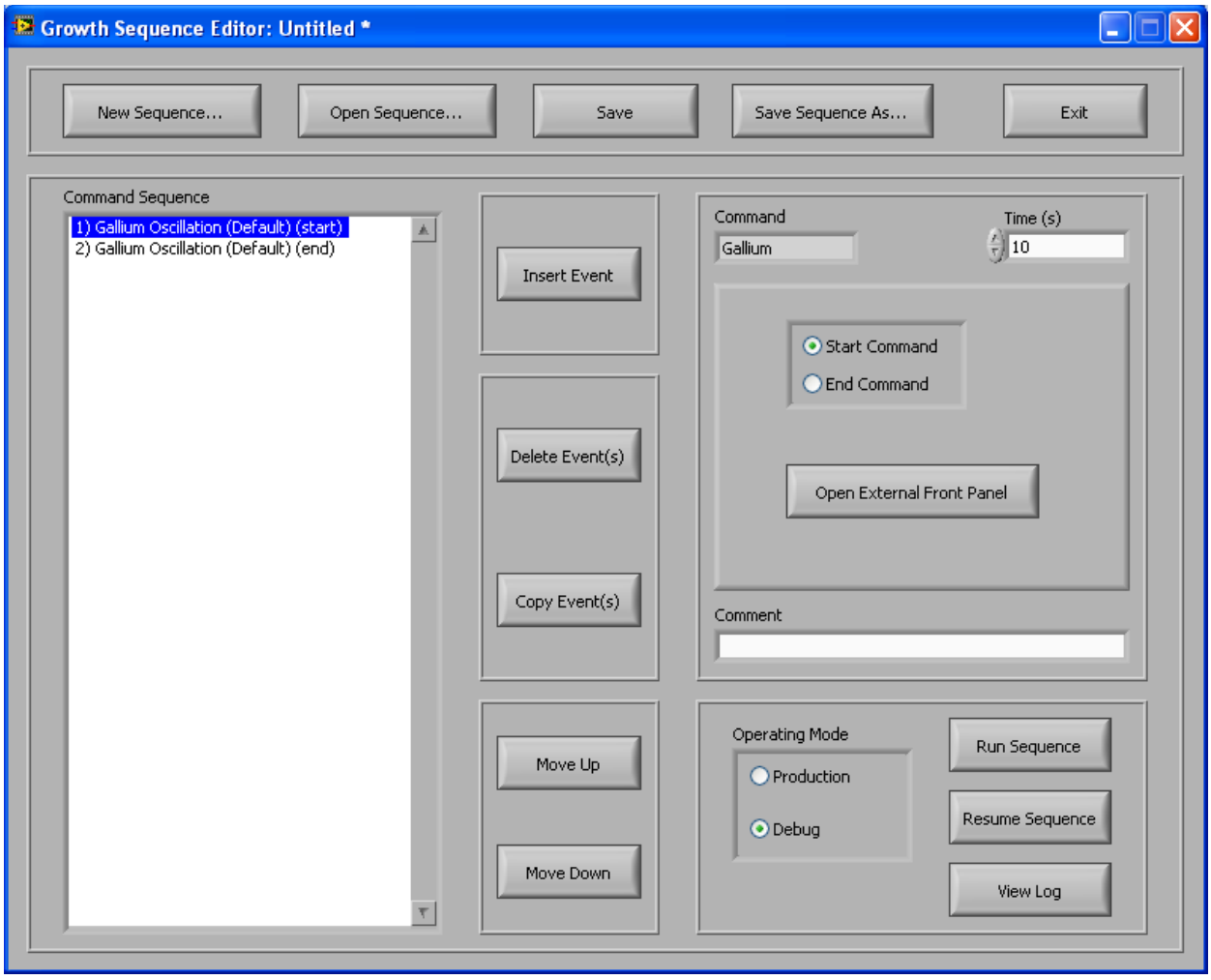

Figure 15

As it is now, the sequence would end instantly. To run the gallium oscillation for longer, change the "Time" control for the currently selected command to something like 10 seconds. If an additional command was added between the gallium oscillation "Start" and "End" commands, the gallium oscillation would continue to run for 10 seconds, and then continue to run through the next command, only stopping when the "End" command is reached.

Since our command calculates the flow rate even in debug mode, we can test our sequence with a debug run. Ensure that the operating mode is set to "Debug" in the editor and run the sequence. The flow rate should oscillate between zero and the given amplitude parameter as expected.

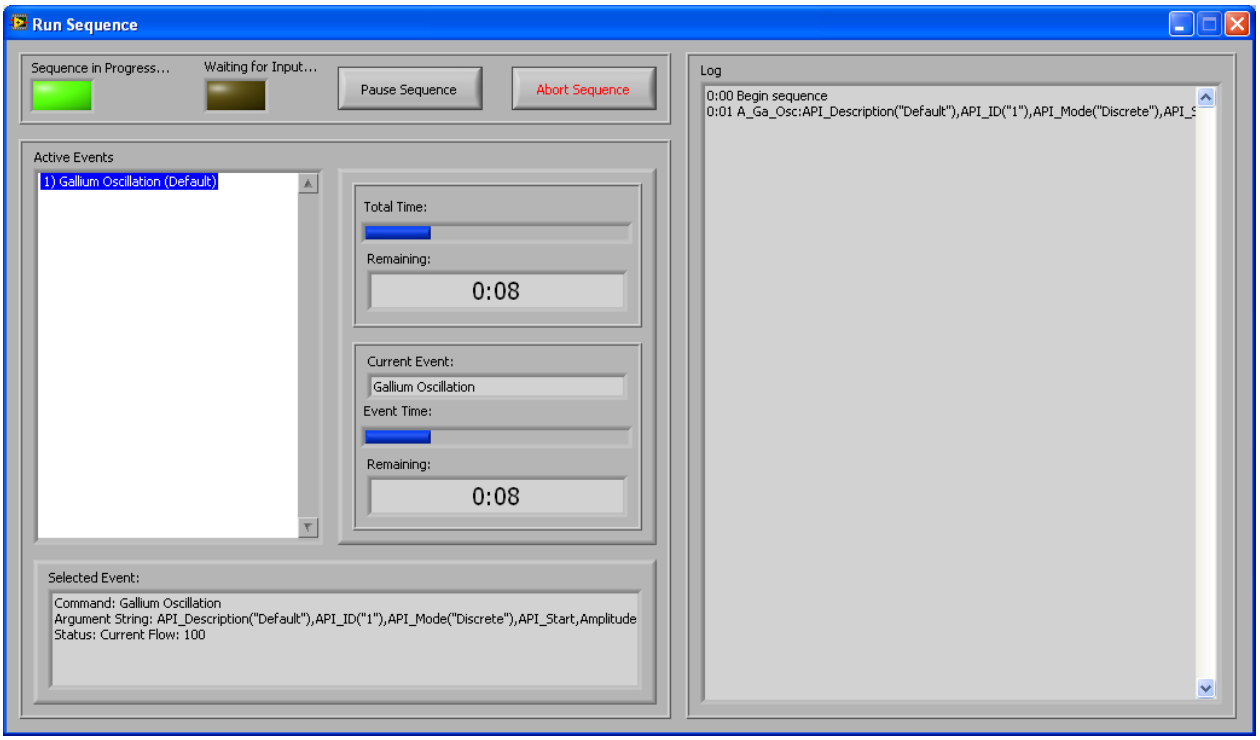

Figure 16