Generalized Microfluidic Device Controller

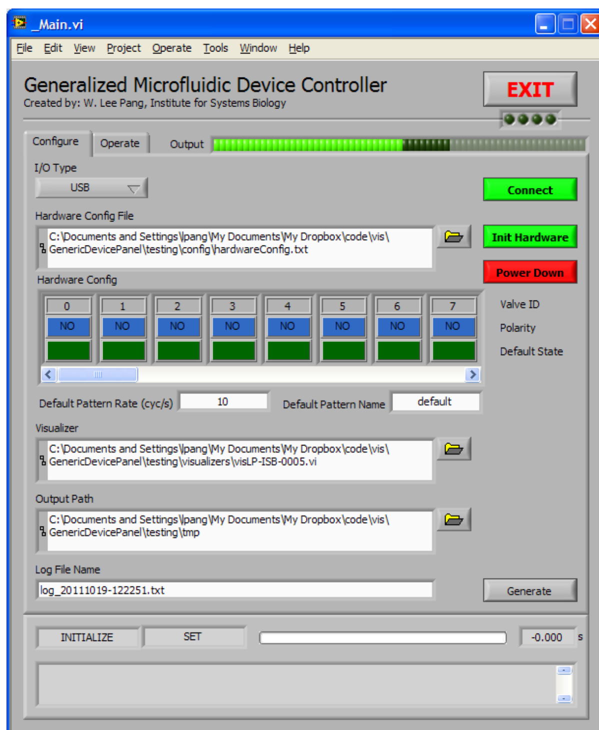
Written by Lee Pang, © 2011

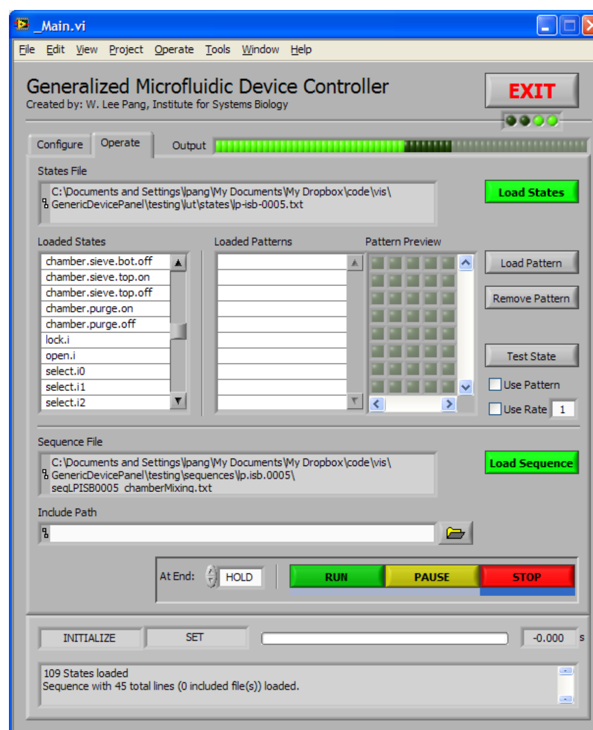
Overview

The Generalized Microfluidic Device Controller (GμDC) is a flexible microfluidic user interface built using LabVIEW. It is both hardware and device design agnostic and can be used to programmatically operate any device with minimal software development. The program does this through the use of easy to generate tab-delimited text files that define:

- Microfluidic device states
- Scripts for programmatic execution of device states
- Hardware configuration - e.g. the number of solenoid valves available and polarity (normally closed or normally open)

Screenshots





Minimum Requirements

- LabVIEW 2010 or later
- LabVIEW Package Manager
  - http://jki.net/vipm
- Text editor

Installation

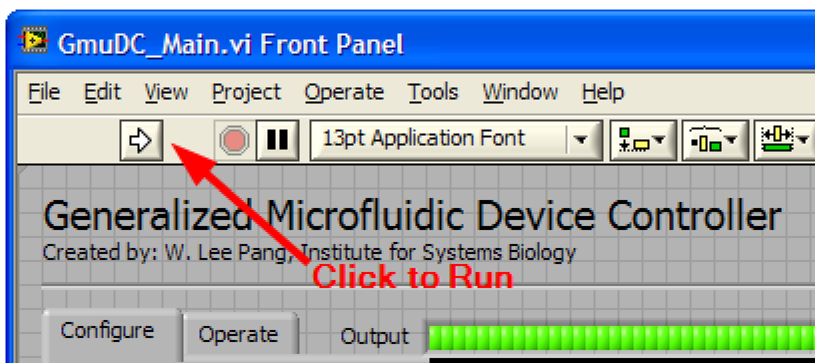
Download the ZIP archive linked at the bottom of the page and extract to your location of choice.

Usage

Navigate to the directory in which you extracted the application during installation.

Double click on the file: GmuDC_Main.vi This will start LabVIEW and bring the main application front panel into view.

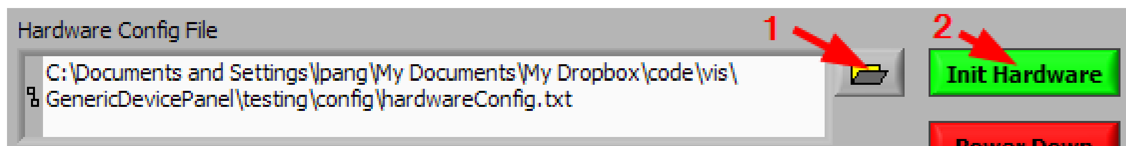
To start the application, click on the Run arrow on the toolbar or hit Ctrl-R on the keyboard.



Initializing

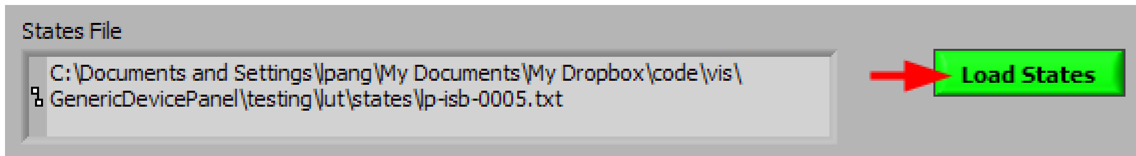
Prior to driving devices you must at minimum do the following: On the "Configure" tab:

1. Connect to your hardware by selecting the appropriate "I/O Type" value and clicking on the "Connect" button.
2. Locate your hardware configuration file (see below) and click on the "Init Hardware" button.



On the "Operate" tab:

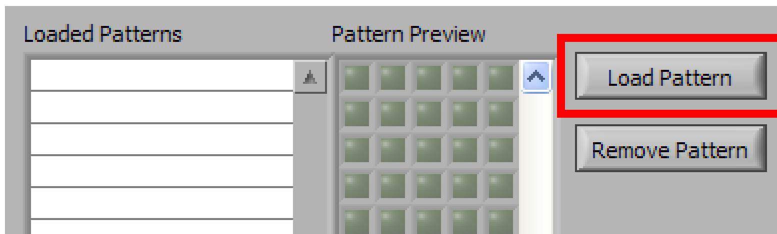
1. Click on the "Load States" button and select a state definition file (see below).



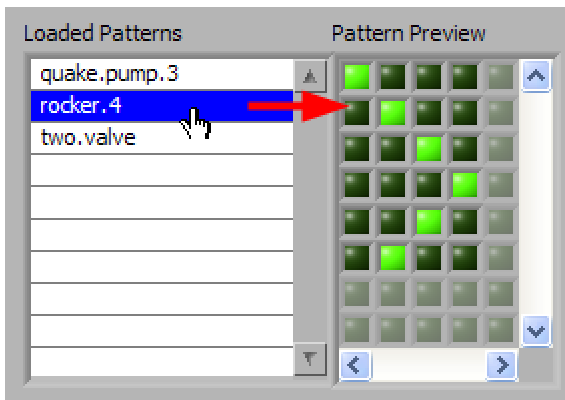
Adding user defined patterns:

On the "Operate" tab:

1. Click on the "Load Pattern" button and select a pattern definition file (see below).



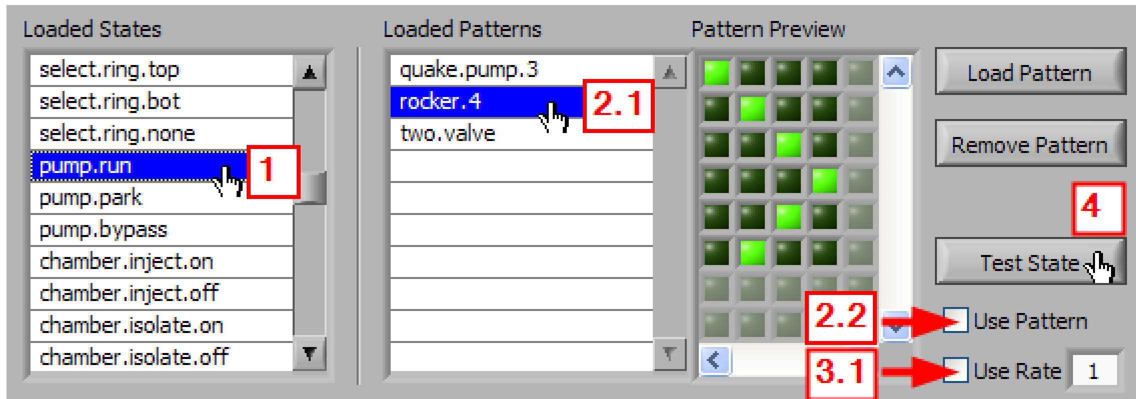
If you load more than one pattern into memory, you can preview each by selecting its entry in the "Loaded Patterns" list.



Testing States

At this point, you may test your predefined states:

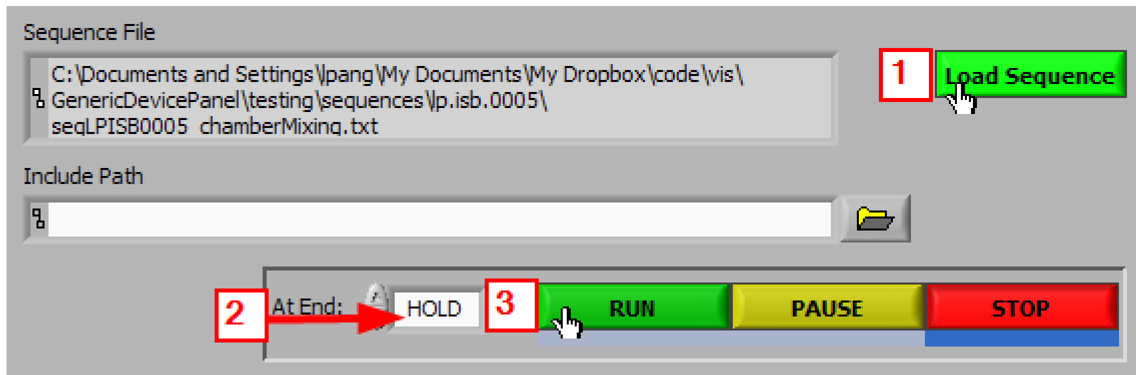
1. Select the state in the "Loaded States" list.
2. To test a state with a user defined pattern:
    1. Select the pattern name in the "Loaded Patterns" list.
    2. Check the "Use Pattern" check box below the "Test State" button
3. To test a state with a user defined rate:
    1. Check the "Use Rate" check box below the "Test State" button
    2. Provide a rate value (in cycles per second) in the numeric field. Negative values run the pattern in reverse.
4. Click the "Test State" button.

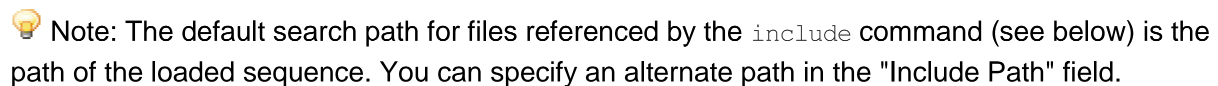


Running Sequences

On the "Operate" tab:

1. Click on the "Load Sequence" button and select a sequence definition file (see below).
2. Select the end of sequence behavior from the "At End" menu:
    1. HOLD: The sequence will self abort when it reaches the end and retain the last applied state
    2. LOOP: The sequence will resume execution from the first step
3. Click on the "Run" button to start the sequence.




Note: The default search path for files referenced by the `include` command (see below) is the path of the loaded sequence. You can specify an alternate path in the "Include Path" field.

When a sequence is running you can pause execution by clicking on the "Pause" button. Clicking on the "Run" button will resume the sequence at the step it was previously paused at.

To abort a running sequence, click the "Stop" button. This will reset the sequence so that when the "Run" button is pressed again, the sequence will start from the beginning.

Run-time Logging

All states applied by the system, either through manual testing or via scripted sequences, are logged in a time-stamped text file. The output path and specific file name are defined on the "Configure" tab.

To avoid creating overly long log files, it is recommended that a new file be generated for each sequence run. To do this: On the "Configure" tab:

1. Click on the "Generate" button.

This will allocate a new log file with the current time-stamp for recording.

Exiting

When exiting the program:

1. On the "Configure" tab:
   1. (Optional) Click the "Power Down" button to de-energize all hardware solenoids.
   2. Disconnect from the hardware IO by clicking on the "Disconnect" button
2. Click the large "Exit" button
3. Close the VI front panel
4. Exit LabVIEW

Defining States

The file containing state definitions is a simple tab delimited file that is both human readable and easily editable in common spreadsheet programs (MS Excel, Libre Office, etc.). Microfluidic device states are defined in a lookup table with static, inherited, or dynamic valves. State timing is controlled by a sequence script (see below) that is defined independent of states.

A simple static state:

```
        <name>\t<interpreter>\t[01^*]\t...\t#<notes>
```

where:

```
\t              A tab character



name:           Name of the state.

                Must be unique within the state definition list.
```

```
                If not, the name is made unique with appended numbers.



interpreter:    modules or manual (see state generators)



[01^*]\t...     A tab separated list of binary values that correspond to valve states.

                0 = valve is open (fluid flows)

                1 = valve is closed (fluid stops)

                ^ = valve state is inherited (see below)

                * = valve is dynamically driven (see below)



#<notes>        Description of the state.  Must begin with '#'.

                Any lines that begin with '#' or are blank are stripped from
```

State generators:

States can be generated by hand using software that can read / write tab-delimited text files or via custom graphical UI specific to a device's design.

To ease state editing using custom UIs it is important to know if the defined state can be programmatically built based on modules incorporated in the chip design.

The `interpreter` value helps to determine how a state generator will read a previously defined state.

```
        modules          Defined using on chip module classes: e.g. single valve, mux,
pump, etc.

        manual           Defined directly (no different than editing the state using a
text editor)
```

Overall, it is probably easier and more efficient to generate states using a tab-delimited text file editor.

State inheritance:

Valves with a value of `^` in a state definition inherit the valve value in the last executed state.

```
state 0: 0      0       0       0       1       1       1       1
state 1: ^      ^       1       1       0       0       ^       ^
```

At run time, state 1 is:

```
                0       0       1       1       0       0       1       1
```

Inherited states are based on the last output value. useful in cases where there is only a slight change in upstream valves.

```
mux0-sel0
flow-on         (inherits mux valves)
flow-off
mux0-sel1
flow-on         (inherits mux valves)
```

Dynamic states:

Valves with a value of `*` in a state definition are operated dynamically - toggled repeatedly at a defined frequency - until the next state is applied.

```
state 0: 0      0       0       0       1       1       1       1
state 2: 1      ^       ^       *       *       *       1       1
```

At run time, state 2 is:

```
                 1     0     0     *     *     *     1     1
```

where `*` valves are dynamically driven based on a defined pattern. The pattern and speed is set in the state's respective sequence call.

Dynamic valve states can be inherited.

```
     state 0: 0     0     0     0     1     1     1     1
     state 2: 1     ^     ^     *     *     *     1     1
     state 3: 1     1     1     ^     ^     ^     1     1
```

at run time, state 3 is:

```
                 1     1     1     *     *     *     1     1
```

The default dynamic pattern is a simple rotating 1D array. For example for three dynamically driven valves:

```
     ValveID: 0     1     2
     -------------------------------
     step 1:        1     0     0
     step 2:        0     1     0
     step 3:        0     0     1
```

The default rate is initially 10 cycles/sec, but can be set by the user on the configuration tab.

The dynamic pattern can also be user defined as an MxN? 2D array of booleans where M is the number of steps in the pattern and N are the number of valves addressed.

- if N does not match the number of `*` valves (D) in the state:
  - N > D: use only the first D valves specified in the pattern
  - N < D: operate the first N valves in D. remaining D-N valves are inherited from the prior state.

- if a dynamic state is inherited:
    - if a pattern is specified use that pattern
    - if a pattern is not specified use the default

Patterns are defined similarly to states but with no inherited or dynamic valve specification. Each pattern definition is a single tab-delimited file. The first line is the name of the pattern. The first column is the step name. Remaining columns are valve states where:

```
        0 = valve is open (fluid flows)

        1 = valve is closed (flow stopped)
```

Blank lines and comments are handled the same as in State definition files.

An example pattern definition file:

```
quake.pump.3

# step	1	2	3

1	0	1	1

2	0	0	1

3	1	0	1

4	1	0	0

5	1	1	0
```

Defining Sequences:

Sequences are simple scripts that define the operation of a chip.

A simple sequence would be:

```
        set		state0

        waittime	10

        set		state1
```

```
waittime 10
set             state0
set             state2 pattern0,5
waittime 30
```

Recognized commands:

```
Command    Parameters
--------------------------------------------------------------
set        <state name> [<pattern name>[,<rate (cycles/sec)>]]
waittime   <time (sec)>
include    <filepath>
```

There must be whitespace (tab, space) between the command and its parameters to be recognized by the sequence parser. Note: `include` is a treated as a preprocessing directive - i.e. they are evaluated when the sequence is loaded into memory.

Execution control:

The sequence processor supports simple for-loops.

```
for    <interator>,<start value>[,<step value>],<stop value>
   ... commands ...
end    <iterator>
```

Iterator values can be used within for-loops:

```
$<iterator>      is replaced with current iterator value
```

Example, the following:

```
for                 i,1,3
        set         state$i
end                 i
```

is equivalent to:

```
set                 state1
set                 state2
set                 state3
```

Arithmetic operations

Simple mathematical formulas can be evaluated based on substituted variable values. The most common place use for this is in loop iterator start, step, and stop values and in defining the duration of `waittime` commands.

Example: The following sequence progressively fills 24 chambers in banks of 4 (addressed via a demultiplexer)

```
set                 select.i0
for     k,1,6
        set                 ring.isolate.off
        set                 select.ring.bot
        set                 pump.bypass
        for     j,$k,6
                for     i,4*$j,4*$j+3
                        set                     select.c$i
                        waittime 1
                end i
```

```
        end     j
        set                     ring.isolate.on
        set                     select.ring.all
        set                     pump.run default,5
        waittime 10
end k
```

Comments

As with state definition files, comments and blank lines are stripped from sequences. Anything following a `#` character to the end of a line is considered a comment.

Example:

```
# comment line
```

In addition, block (multi-line) comments can be defined as follows:

```
#{
        block comment
#}
```

Hardware abstraction:

State and sequence files are defined independent of the specific hardware used. Any state to be applied during runtime is parsed by a function that checks a hardware configuration file specified by the user.

Hardware configurations are defined with a simple tab-delimited table:

```
        ValveID          Polarity DefaultState
        -------          -------- ------------
        [0-9]*           NC|NO           [01]
```

where:

```
        ValveID: ID number of the hardware valve.  Must start at 0.


        Polarity:        Valves are either Normally Closed (NC) or Normally Opened
(NO), that is in their unpowered state valve output is either linked to exhaust or
linked to supply, respectively.


        DefaultState:    When the UI is initialized valves are placed in this state
                0        unpowered
                1        powered
```

State definitions may specify fewer valves than are physically available. Undefined valves in state definitions will default to the hardware configured DefaultState?. Alternatively, if a state definition specifies more valves than are physically available, the extra valves will be ignored.

Latched based (indirect) valve operation is delegated to sequences of states that drive directly controlled valves.