FlowIO()	Class Definition: **/FlowIO/FlowIO.h** Keywords: **/FlowIO/keywords.txt** Class Source: /**FlowIO/FlowIO.cpp**

CONSTRUCTORS:

`FlowIO(Configuration config);`

`config` - the type of pneumatic configuration. Can be one of the following:

`GENERAL`, **`INFLATION_SERIES`**, **`INFLATION_PARALLEL`**, **`VACUUM_SERIES`**, **`VACUUM_PARALLEL`**

NOTE: If you specify no parameters in the constructor, then the GENERAL configuration is assumed, where pump1 is for inflation and pump2 for vacuum.

(You must instantiate the flowIO object outside the setup loop, and you must initialize it at the top of the setup loop, before any Bluetooth commands. If you initialize it after you have set up Bluetooth, then you may run into issues.)

You can check the current configuration or change it at any time using the following methods:

`void setConfig(Configuration config);`

`Configuration getConfig();` returns the configuration as a single byte whose value is between 0 and 4.

Pressure-Sensing API

`bool activateSensor();`

Initializes the I2C communication with the integrated pressure sensor, and checks the status byte sent by the sensor. RETURNS: `true` if the status byte is as expected; otherwise returns `false` and error code 222.

`void setPressureUnit(Unit pUnit);`

Sets the unit that will be used for all pressure readings. If this is not set, This method is optional, and if not used, units will be set to PSI. You can change the units at any point in the code. Possible argument values are **`PSI`**, **`ATM`**, **`HPA`**.

`float getPressure();`

Gets a new absolute pressure reading from the integrated sensor in terms of the unit in `setPressureUnit()`.

`float getPressure(Unit pUnit);`

Gets a new absolute pressure reading from the integrated sensor in terms of the unit specified in the argument. Possible argument values are **`PSI`**, **`ATM`**, **`HPA`**.

NOTE: Sensor must be activated before requesting pressure. Otherwise `getPressure()` will return the value 888.8.

Indicators API

`void blueLED(bool power);` Controls the onboard led. The argument can be either `LOW / 0` or `HIGH / 1`.

`void pixel(uint8_t red, uint8_t green, uint8_t blue);` Controls the onboard neoPixel color and intensity.. For highly-dynamic actions involving the neoPixel LED, avoid using this method, and instead use the methods from the Adafruit_Neopixel library directly, because the efficiency and speed are much better.

`void raiseError(uint8_t error);` Sets the value of the internal error flag.

`uint8_t readError();` Reads the value of the internal error flag. See the error code table on the last page.

`void powerOFF();` Shuts down the FlowIO device by setting pin 16 as HIGH.

`uint16_t getHardwareState();` Returns a 16-bit value, where each bit maps to the state of a hardware component.

`bool getHardwareStateOf(Component ComponentName);` Takes the enum-defined component name,

`bool getHardwareStateOf(uint8_t bitNumber);` Takes the bit number corresponding to a particular component,

Returns `true` if that component is active / on, otherwise it returns `false`.

Bit #	11	10	9	8	7	6	5	4	3	2	1	0
Component	**`SENSOR`**	**`LEDBLUE`**	**`LEDRED`**	**`PUMP2`**	**`PUMP1`**	**`OUTLET`**	**`INLET`**	**`PORT5`**	**`PORT4`**	**`PORT3`**	**`PORT2`**	**`PORT1`**

High-Level Pneumatics API (behavior level). This is an abstraction layer based on the Low-Level Pneumatics API above.

`void stopAction(uint8_t ports)`
Stops all pumps and closes the inlet valve, outlet valve, and the ports specified in the argument.
(E.g. To stop the action at all ports, set the argument to 0xFF or any other byte whose last 5 bits are 11111.)
Supported in all configurations

`void startInflation(uint8_t ports,uint8_t pwmValue=255)`
Stops any ongoing actions at the specified port(s), then starts inflation with all the inflation pumps available in the chosen `configuration`. An optional second parameter enables PWM'ing the pumps for lower pressure.
Supported configurations: GENERAL, INFLATION_SERIES, INFLATION_PARALLEL

`void startVacuum(uint8_t ports, uint8_t pwmValue=255)`
Stops actions at specified port(s), then starts vacuum with all vacuum pumps available in the chosen `configuration`.
Supported configurations: GENERAL, VACUUM_SERIES, VACUUM_PARALLEL

`void startRelease(uint8_t ports)`
Stops all actions at the specified port(s), then opens the inlet and outlet valves, to bring the pressure to ATM. In the GENERAL configuration, this exploits the fact that the pumps act like a diode in their off state, allowing one-way flow.
Supported in all configurations

`void startInflationHalfCapacity(uint8_t ports)`
Executes the *stopAction(ports)* function, then starts inflation with a single pump.
Supported configurations: INFLATION_SERIES, INFLATION_PARALLEL

`void startVacuumHalfCapacity(uint8_t ports, uint8_t pwmValue=255)`
Executes the *stopAction(ports)* function, then starts vacuum with a single pump.
Supported configurations: VACUUM_SERIES, VACUUM_PARALLEL

NOTE: If you are in a configuration that is not supporting a particular function, and you try calling that function anyways, it will simply be ignored and nothing will happen. No error will occur either.

Low-Level Pneumatics API (component level)

`void startPump(uint8_t pumpNumber,uint8_t pwmValue=255);`
`void stopPump(uint8_t pumpNumber);`
`pumpNumber` is either 1 or 2. `pwmValue` is optional and ranges from 0 to 255.

`void openInletValve();` *(right side)*
`void closeInletValve();`
`void openOutletValve();` *(left side)*
`void closeOutletValve();`

`void setPorts(uint8_t ports);`
Ports is a byte whose lowest 5 bits correspond to the 5 Flow IO ports, where bit0 maps to port 1 on the FlowIO, bit1 maps to port 2, etc. Setting a bit to 0 closes the corresponding port, setting a bit to 1 opens it.

`void openPorts(uint8_t ports);`
If the value of a bit is 1, the corresponding port is opened. If a bit is 0, the corresponding port remains unchanged.

`void closePorts(uint8_t ports);`
If the value of a bit is 1, the corresponding port is closed. If a bit is 0, the corresponding port remains unchanged.

Command Control API. Controlling the device using 2-byte or 3-byte commands.

```
float command(uint8_t action, uint8_t ports, uint8_t pwmValue=255)
```

Byte1 = what action to perform. Specified as a char and can only be '+', '-', '!', 'p', 'n', '^', 'o', 'c', 'f', 'r', 'b', or '?'
Byte2 = On which port(s) or location to perform the action.. Bit 0 of 'ports' corresponds to port 1, bit 1 to port 2, etc.
Byte3 = optional PWM value for the pumps. Defaults to 255 (max) if not specified.
RETURNS: Pressure value. If the returned value is 888.8 or 999.9 it is not an applicable value.

Command	Correspondence	Returns
'!'*	`stopAction(ports).`	999.9
'+'**	`startInflation(ports,pwmValue)`	999.9
'-'**	`startVacuum(ports,pwmValue)`	999.9
'p'**	`startInflationHalfCapacity(ports,pwmValue)`	999.9
'n'**	`startVacuumHalfCapacity(ports,pwmValue)`	999.9
'^'*	`startRelease(ports)`	999.9
'o'*	`openPorts(ports)`	999.9
'c'*	`closePorts(ports)`	999.9
'ff'	`powerOFF()`	999.9
'r'*	`redLED(bool)`	999.9
'b'*	`blueLED(bool)`	999.9
'??'	`getPressure()` at the current state	Pressure or 888.8 if error

'?'*	`stopAction(0xff);`
	`setPorts(ports);`
	`delay(10);`
	`getPressure();`
		Pressure or 888.8 if error (if sensor not initialized)

Char	'!'	'+'	'-'	'p'	'n'	'^'	'o'	'c'	'f'	'r'	'b'	'?'
ASCII hex	0x21	0x2b	0x2d	0x70	0x6e	0x5e	0x6f	0x63	0x66	0x72	0x62	0x3f

Error Code	Description
0	no error
222	activateSensor() error
223	getPressure() error
2	invalid value written to chrPin02
102	notify02flag set to true in unsupported state

TODO: Add a 4th byte to the communication protocol, so that you can specify the PWM values separately for each pump. This is especially important in the alternative configurations, where you may wish one pump to be at max pwm and another one to be at a different PWM. This will be a disruptive change, so before emparking on it, ensure that everything else has been done in the JavaScript with the PWM control and works with the current protocol.