Commands templates

This document provides information on implementing G-Code command sequences in gcode_macro (and similar) config sections.

G-Code Macro Naming

Case is not important for the G-Code macro name - MY_MACRO and my_macro will evaluate the same and may be called in either upper or lower case. If any numbers are used in the macro name then they must all be at the end of the name (eg, TEST_MACRO25 is valid, but MACRO25_TEST3 is not).

Formatting of G-Code in the config

Indentation is important when defining a macro in the config file. To specify a multi-line G-Code sequence it is important for each line to have proper indentation. For example:

```
[gcode_macro blink_led]
gcode:
  SET_PIN PIN=my_led VALUE=1
  G4 P2000
  SET_PIN PIN=my_led VALUE=0
```

Note how the `gcode:` config option always starts at the beginning of the line and subsequent lines in the G-Code macro never start at the beginning.

Add a description to your macro

To help identify the functionality a short description can be added. Add `description:` with a short text to describe the functionality. Default is "G-Code macro" if not specified. For example:

```
[gcode_macro blink_led]
description: Blink my_led one time
gcode:
  SET_PIN PIN=my_led VALUE=1
  G4 P2000
  SET_PIN PIN=my_led VALUE=0
```

The terminal will display the description when you use the `HELP` command or the autocomplete function.

Save/Restore state for G-Code moves

Unfortunately, the G-Code command language can be challenging to use. The standard mechanism to move the toolhead is via the `G1` command (the `G0` command is an alias for `G1` and it can be used interchangeably with it). However, this command relies on the "G-Code parsing state" setup by `M82`, `M83`, `G90`, `G91`, `G92`, and previous `G1` commands. When creating a G-Code macro it is a good idea to always explicitly set the G-Code parsing state prior to issuing a `G1` command. (Otherwise, there is a risk the `G1` command will make an undesirable request.)

A common way to accomplish that is to wrap the `G1` moves in `SAVE_GCODE_STATE`, `G91`, and `RESTORE_GCODE_STATE`. For example:

```
[gcode_macro MOVE_UP]
gcode:
  SAVE_GCODE_STATE NAME=my_move_up_state
  G91
  G1 Z10 F300
  RESTORE_GCODE_STATE NAME=my_move_up_state
```

The `G91` command places the G-Code parsing state into "relative move mode" and the `RESTORE_GCODE_STATE` command restores the state to what it was prior to entering the macro. Be sure to specify an explicit speed (via the `F` parameter) on the first `G1` command.

Template expansion

The gcode_macro `gcode:` config section is evaluated using the Jinja2 template language. One can evaluate expressions at run-time by wrapping them in `{ }` characters or use conditional statements wrapped in `{% %}`. See the Jinja2 documentation for further information on the syntax.

An example of a complex macro:

```
[gcode_macro clean_nozzle]
gcode:
  {% set wipe_count = 8 %}
  SAVE_GCODE_STATE NAME=clean_nozzle_state
  G90
  G0 Z15 F300
  {% for wipe in range(wipe_count) %}
    {% for coordinate in [(275, 4),(235, 4)] %}
      G0 X{coordinate[0]} Y{coordinate[1] + 0.25 * wipe} Z9.7 F12000
    {% endfor %}
  {% endfor %}
  RESTORE_GCODE_STATE NAME=clean_nozzle_state
```

Macro parameters

It is often useful to inspect parameters passed to the macro when it is called. These parameters are available via the `params` pseudo-variable. For example, if the macro:

```
[gcode_macro SET_PERCENT]
gcode:
  M117 Now at { params.VALUE|float * 100 }%
```

were invoked as `SET_PERCENT VALUE=.2` it would evaluate to `M117 Now at 20%`. Note that parameter names are always in upper-case when evaluated in the macro and are always passed as strings. If performing math then they must be explicitly converted to integers or floats.

It's common to use the Jinja2 `set` directive to use a default parameter and assign the result to a local name. For example:

```
[gcode_macro SET_BED_TEMPERATURE]
gcode:
  {% set bed_temp = params.TEMPERATURE|default(40)|float %}
  M140 S{bed_temp}
```

The "rawparams" variable

The full unparsed parameters for the running macro can be access via the `rawparams` pseudo-variable.

Note that this will include any comments that were part of the original command.

See the sample-macros.cfg file for an example showing how to override the `M117` command using `rawparams`.

The "printer" Variable

It is possible to inspect (and alter) the current state of the printer via the `printer` pseudo-variable. For example:

```
[gcode_macro slow_fan]
gcode:
  M106 S{ printer.fan.speed * 0.9 * 255}
```

Available fields are defined in the Status Reference document.

Important! Macros are first evaluated in entirety and only then are the resulting commands executed. If a macro issues a command that alters the state of the printer, the results of that state change will not be visible during the evaluation of the macro. This can also result in subtle behavior when a macro generates commands that call other macros, as the called macro is evaluated when it is invoked (which is after the entire evaluation of the calling macro).

By convention, the name immediately following `printer` is the name of a config section. So, for example, `printer.fan` refers to the fan object created by the `[fan]` config section. There are some exceptions to this rule - notably the `gcode_move` and `toolhead` objects. If the config section contains spaces in it, then one can access it via the `[ ]` accessor - for example: `printer["generic_heater my_chamber_heater"].temperature`.

Note that the Jinja2 `set` directive can assign a local name to an object in the `printer` hierarchy. This can make macros more readable and reduce typing. For example:

```
[gcode_macro QUERY_HTU21D]
gcode:
    {% set sensor = printer["htu21d my_sensor"] %}
    M117 Temp:{sensor.temperature} Humidity:{sensor.humidity}
```

Actions

There are some commands available that can alter the state of the printer. For example, `{ action_emergency_stop() }` would cause the printer to go into a shutdown state. Note that these actions are taken at the time that the macro is evaluated, which may be a significant amount of time before the generated g-code commands are executed.

Available "action" commands:

- `action_respond_info(msg)`: Write the given `msg` to the /tmp/printer pseudo-terminal. Each line of `msg` will be sent with a "// " prefix.
- `action_raise_error(msg)`: Abort the current macro (and any calling macros) and write the given `msg` to the /tmp/printer pseudo-terminal. The first line of `msg` will be sent with a "!! " prefix and subsequent lines will have a "// " prefix.
- `action_emergency_stop(msg)`: Transition the printer to a shutdown state. The `msg` parameter is optional, it may be useful to describe the reason for the shutdown.
- `action_call_remote_method(method_name)`: Calls a method registered by a remote client. If the method takes parameters they should be provided via keyword arguments, ie: `action_call_remote_method("print_stuff", my_arg="hello_world")`

Variables

The SET_GCODE_VARIABLE command may be useful for saving state between macro calls. Variable names may not contain any upper case characters. For example:

```
[gcode_macro start_probe]
variable_bed_temp: 0
gcode:
  # Save target temperature to bed_temp variable
  SET_GCODE_VARIABLE MACRO=start_probe VARIABLE=bed_temp VALUE=
{printer.heater_bed.target}
  # Disable bed heater
  M140
  # Perform probe
  PROBE
  # Call finish_probe macro at completion of probe
  finish_probe

[gcode_macro finish_probe]
gcode:
  # Restore temperature
  M140 S{printer["gcode_macro start_probe"].bed_temp}
```

Be sure to take the timing of macro evaluation and command execution into account when using SET_GCODE_VARIABLE.

Delayed Gcodes

The [delayed_gcode] configuration option can be used to execute a delayed gcode sequence:

```
[delayed_gcode clear_display]
gcode:
  M117

[gcode_macro load_filament]
gcode:
 G91
 G1 E50
 G90
 M400
 M117 Load Complete!
 UPDATE_DELAYED_GCODE ID=clear_display DURATION=10
```

When the `load_filament` macro above executes, it will display a "Load Complete!" message after the extrusion is finished. The last line of gcode enables the "clear_display" delayed_gcode, set to execute in 10 seconds.

The `initial_duration` config option can be set to execute the delayed_gcode on printer startup. The countdown begins when the printer enters the "ready" state. For example, the below delayed_gcode will execute 5 seconds after the printer is ready, initializing the display with a "Welcome!" message:

```
[delayed_gcode welcome]
initial_duration: 5.
gcode:
  M117 Welcome!
```

Its possible for a delayed gcode to repeat by updating itself in the gcode option:

```
[delayed_gcode report_temp]
initial_duration: 2.
gcode:
  {action_respond_info("Extruder Temp: %.1f" %
(printer.extruder0.temperature))}
  UPDATE_DELAYED_GCODE ID=report_temp DURATION=2
```

The above delayed_gcode will send "// Extruder Temp: [ex0_temp]" to Octoprint every 2 seconds. This can be canceled with the following gcode:

```
UPDATE_DELAYED_GCODE ID=report_temp DURATION=0
```

Menu templates

If a display config section is enabled, then it is possible to customize the menu with menu config sections.

The following read-only attributes are available in menu templates:

- `menu.width` - element width (number of display columns)
- `menu.ns` - element namespace
- `menu.event` - name of the event that triggered the script
- `menu.input` - input value, only available in input script context

The following actions are available in menu templates:

- `menu.back(force, update)`: will execute menu back command, optional boolean parameters `<force>` and `<update>`.
  - When `<force>` is set True then it will also stop editing. Default value is False.
  - When `<update>` is set False then parent container items are not updated. Default value is True.
- `menu.exit(force)` - will execute menu exit command, optional boolean parameter `<force>` default value False.
  - When `<force>` is set True then it will also stop editing. Default value is False.

Save Variables to disk

If a save_variables config section has been enabled, `SAVE_VARIABLE VARIABLE=<name> VALUE=<value>` can be used to save the variable to disk so that it can be used across restarts. All stored variables are loaded into the `printer.save_variables.variables` dict at startup and can be used in gcode macros. to avoid overly long lines you can add the following at the top of the macro:

```
{% set svv = printer.save_variables.variables %}
```

As an example, it could be used to save the state of 2-in-1-out hotend and when starting a print ensure that the active extruder is used, instead of T0:

```
[gcode_macro T1]
gcode:
  ACTIVATE_EXTRUDER extruder=extruder1
  SAVE_VARIABLE VARIABLE=currentextruder VALUE='"extruder1"'

[gcode_macro T0]
gcode:
  ACTIVATE_EXTRUDER extruder=extruder
  SAVE_VARIABLE VARIABLE=currentextruder VALUE='"extruder"'
```

```
[gcode_macro START_GCODE]
gcode:
  {% set svv = printer.save_variables.variables %}
  ACTIVATE_EXTRUDER extruder={svv.currentextruder}
```