Table of Contents

1 Klipper Dynamic Macros

Never restart Klipper again for simple macros.

Klipper Dynamic Macros is an unofficial way to update macros without restarting Klipper, so you can update macros mid-print and see their results live. It also supports extra features that normal GCode Macros don't have.

1.1 Features

- Recursion
- Receiving Variables
- Utility Functions
- Variables
- Python
- Delayed GCode

1.2 How Normal Macros Work

Your macros are written in a `.cfg` file, then included into your `printer.cfg`. When Klipper restarts, it parses these files and saves the macros internally (you can't change them without restarting Klipper). When a macro is called, the cached code is interpreted and run.

1.3 How Dynamic Macros Work

Your macros are written in a `.cfg` file, then the relative path to that file is configured in a `[dynamicmacros]` config section. The config files are read and parsed every time you run the macro, allowing you to update macros without restarting Klipper.

1.4 Get Started

Follow Setup to get started with Dynamic Macros.

1.5 Planned Features

See Development Status for the currently available features, and planned features.

1.6 Examples

See Example Macros for examples of Dynamic Macros.

1.7 More Projects

If you like this project, don't forget to give it a star! Also, check out the 3MS, a modular multimaterial system for Klipper!

2 Setup

Follow this guide to setup and confiure Dynamic Macros.

2.1 Install

Run in your terminal:

```
cd ~
git clone https://github.com/3DCoded/DynamicMacros
cd DynamicMacros
sh install.sh
sudo service klipper restart
```

Add to your `moonraker.conf`:

moonraker.conf

```
# DynamicMacros Update Manager
[update_manager DynamicMacros]
type: git_repo
path: ~/DynamicMacros
origin: https://github.com/3DCoded/DynamicMacros.git
primary_branch: main
is_system_service: False
install_script: install.sh
```

2.2 Updating

First, update via Moonraker's update manager. Then run in your terminal:

```
cd ~/DynamicMacros
sh install.sh
sudo service klipper restart
```

2.3 Configuration

To configure Dynamic Macros, put in your `printer.cfg` (or a file included in it):

printer.cfg

```
[dynamicmacros]
config_path: ~/printer_data/config # If you left your printer configuration path as the default, you don't need to
specify this
configs: dynamic.cfg
```

Create a new file in the same folder as your `printer.cfg` called `dynamic.cfg`. In it, configure some macros like you normally would. Example:

dynamic.cfg

```
[gcode_macro MYTEST]
gcode:
    M117 Hello world!

[gcode_macro HEAT_HOTEND]
gcode:
    {% set temp = params.TEMP|int %}
    M104 S{temp}
```

Restart Klipper.

> **Info**
>
> Updating the `[dynamicmacros]` config section requires a Klipper restart. The files listed in the `macros:` parameter must be present when Klipper restarts.

2.4 Testing

If you run the command `MYTEST`, the output should be:

```
Hello world!
```

If you runthe command `HEAT_HOTEND TEMP=200`, the hotend should start heating up.

Next, edit the `MYTEST` macro to output something else, like `Test successful!`

dynamic.cfg

```
[gcode_macro MYTEST]
gcode:
    M117 Test successful!
```

Save the file, but **do not** restart Klipper.

Run `MYTEST` again, and the output should be:

```
Test successful!
```

2.5 Features

- Recursion
- Receiving Variable Updates
- Utility Functions
- Variables

2.6 Tutorial

Follow the Tutorial.

2.7 When to Restart Klipper or Reload Dynamic Macros

Dynamic Macros provides a utility `DYNAMIC_MACRO` command to run macros manually, and to refresh the macros. Usage examples (assuming M900 is defined as a Dynamic Macro):

```
DYNAMIC_MACRO MACRO=M900 K=0.035
```

is the same as:

```
M900 K0.035
```

To refresh Dynamic Macros, just run `DYNAMIC_MACRO` with no parameters.

A Klipper restart is required if:

- You changed the `description`
- You changed the `initial_duration`
- You changed the `repeat`

A `DYNAMIC_MACRO` refresh is required if:

- You created a new macro
- You renamed an existing macro
- You changed the contents of a macro
- You deleted an existing macro

I. Tutorial

3 Table of Contents

Follow this tutorial to learn how to create Dynamic Macros and to use its powerful features.

3.1 Standard Macros

Before writing macros, it's a good idea to learn how to write standard `gcode_macro`s. Here is an amazing guide on how to write standard macros.

3.2 Table of Contents

4 Converting to Dynamic Macros

Follow this tutorial to convert your standard `gcode_macro` to a Dynamic Macro.

4.1 printer.cfg

First, remove the `include` line in your `printer.cfg` referencing the file your macros are in. For example, if your macros are in `macros.cfg`, remove the line:

```
[include macros.cfg]
```

from your `printer.cfg`.

4.2 Dynamic Macros Configuration

Next, if you don't already have one, create a `[dynamicmacros]` config section in your `printer.cfg` and add your macro configuration to it:

printer.cfg

```
[dynamicmacros]
config_path: ~/printer_data/config # If you left your printer configuration path as the default, you don't need to
specify this
configs: macros.cfg # You can add more files to this list, separated by commas.
```

KlipperScreen

If you convert your `LOAD_FILAMENT` and `UNLOAD_FILAMENT` macros to be dynamic, KlipperScreen may not recognize them and report an error. To fix this, add blank macros to your `printer.cfg`, before your `[dynamicmacros]` section. Example:

printer.cfg

```
[gcode_macro LOAD_FILAMENT]
gcode:
    M117 LOAD
[gcode_macro UNLOAD_FILAMENT]
gcode:
    M117 UNLOAD
```

Unknown config object 'gcode_macro'

If you are getting a "Unknown config object 'gcode_macro'" error after converting your macros to Dynamic Macros, move your `[dynamicmacros]` section to be after your `[virtual_sdcard]` section.

Nacro Names

Klipper has certain macro names reserved for core functionality. If you are experiencing errors, don't name your Dynamic Macros any of the following:

- `PAUSE`
- `RESUME`
- `CANCEL_PRINT`

If you want to make those macros dynamic, first define them as a **standard** GCode macro:

```
[gcode_macro PAUSE]
gcode:
    DYNAMIC_PAUSE
```

Restart Klipper.

That's it. Your macros are now Dynamic Macros.

5 Using Dynamic Macros Features

Now that your macros are dynamic, you can use the powerful feature set of Dynamic Macros.

5.1 Recursion

Recursion allows a macro to call itself, something standard GCode macros can't do.

When using recursion, it's important to make sure your macro has an "end case". This is a case when the macro won't call itself again. Otherwise the macro will be stuck in an infinite loop and freeze Klipper.

Here are a few examples of recursion:

Counting to 10

```
[gcode_macro COUNT]
gcode:
    {% set num = params.NUM|default(1)|int %}
    {% if num <= 10 %}
        RESPOND MSG={num}
        COUNT NUM={num+1} # Count up 1
    {% else %}
        RESPOND MSG="Done Counting"
    {% endif %}
```

Load to Filament Sensor

```
[gcode_macro LOAD_TO_FSENSOR]
gcode:
    {% set val = printer["filament_sensor fsensor"].filament_detected %}
    {% if val == 0 %}
        M83
        G1 E50 F900 # Move filament 50mm forwards
        RESPOND MSG="Waiting for fsensor"
        LOAD_TO_FSENSOR # Recursion
    {% else %}
        G1 E65 F900 # Move filament to nozzle
        RESPOND MSG="Filament Loaded"
    {% endif %}
```

5.2 Receiving Variable Updates

Receiving variable updates allows Dynamic Macros to update variables without rerunning the macro.

An example of this is getting the printer's position. A standard GCode macro will evaluate all the variables, then run the GCode. However, using three newlines (two blank lines) between code segments in Dynamic Macros will allow each segment to be evaluated at runtime, allowing for variable updates.

5.3 Utility Functions

See the link above (the subtitle) for more information on utility functions.

5.4 Python

Dynamic Macro's most powerful feature allows you to run Python code from within a macro. See the link above (the subtitle) for more information.

5.5 delayed_gcode

Dynamic Macros supports a feature similar to `delayed_gcode`. See the link above (the subtitle) for more information.

6 Experimental Features

> **Warning**
>
> The features on this page are experimental, and untested or only lightly tested. Proceed at your own risk.

II. Features

7 Recursion

Unlike normal `gcode_macro`s, Dynamic Macros supports recursion (allowing a macro to call itself). For more examples, see Examples

8 Receiving Variable Updates

Unlike normal `gcode_macro`s, Dynamic Macros supports receiving variable updates within the same macro. For example, the following macro will show the same output in both `M117`s:

```
[gcode_macro STATIC_MOVE]
gcode:
  G28
  M117 Before: {printer.toolhead.position.z}
  # Above displays position before macro run
  G90
  G1 Z20
  M117 After: {printer.toolhead.position.z}
  # Above displays same position
```

However, this macro will show different outputs based on the current Z position of the toolhead:

```
[gcode_macro DYNAMIC_MOVE]
gcode:
  G28


  M117 Before: {printer.toolhead.position.z}
  # Above displays position after G28
  G90
  G1 Z20


  M117 After: {printer.toolhead.position.z}
  # Above displays position after G1
```

Notice the large whitespaces. Three newline characters (two blank lines) between code segments denotes a variable update. However, some variables won't be preserevd across the whitespace.

```
[gcode_macro VARIABLES]
gcode:
    {% set num = 10 %}
    M117 {num}


    M117 {num}
    # Above line outputs nothing
```

You can use the `update()` function to preserve certain variables across the whitespaces:

```
[gcode_macro VARIABLES]
gcode:
    {% set num = update("num", 10) %}
    M117 {num}


    M117 {num}
    # Above line outputs 10
```

See Examples for examples.

9 Utility Functions

Dynamic Macros provides a few utility functions to make Dynamic Macros easier to write.

9.1 update()

The `update()` function allows to save variables across whitespaces (see Receiving Variables for more information). Example:

```
[gcode_macro MY_MACRO]
gcode:
    {% set a = 10 %}
    {% set b = update("b", 20) %}
    RESPOND MSG="A: {a}" # 10
    RESPOND MSG="B: {b}" # 20


    RESPOND MSG="A: {a}" # ""
    RESPOND MSG="B: {b}" # 20
```

9.2 get_macro_variables()

The `get_macro_variables()` function allows to retrieve all the variables from another macro in one line of code. Example:

```
[gcode_macro MY_SETTINGS]
variable_a: 10
variable_b: 20
gcode:
    RESPOND MSG="settings"

[gcode_macro GET_SETTINGS]
gcode:
    {% set settings = get_macro_variables("MY_SETTINGS") %}
    RESPOND MSG="Settings: {settings}"
    RESPOND MSG="A: {settings.a}"
```

9.3 update_from_dict()

The `update_from_dict()` function allows for saving the output of `get_macro_variables()` (or other dictionaries) across the whitespaces when Receiving Variables. Example:

```
[gcode_macro MY_SETTINGS]
variable_a: 10
variable_b: 20
gcode:
    RESPOND MSG="settings"

[gcode_macro GET_SETTINGS]
gcode:
    {% set settings = get_macro_variables("MY_SETTINGS") %}
    {% set settings = update_from_dict(settings) %}


    RESPOND MSG="A: {a}" # 10
    RESPOND MSG="B: {b}" # 20
```

10 Variables

Dynamic Macro variables work nearly the same way as standard `gcode_macro` variables. This guide covers the differences.

10.1 SET_DYNAMIC_VARIABLE

Instead of using `SET_GCODE_VARIABLE`, Dynamic Macros use `SET_DYNAMIC_VARIABLE` to update Dynamic Macro variables. Example (assuming `VAR_TEST` is defined from Utilities):

```
1  SET_DYNAMIC_VARIABLE MACRO=var_test VARIABLE=a VALUE=15
```

11 Python

Dynamic Macros's most powerful feature allows you to run Python code directly from within a macro.

11.1 Running Python from within a Macro

Disclaimer

This functionality allows Dynamic Macros to gain significant control over your printer and Klipper host. I am not responsible for whatever happens if you download a malicious macro.

There are three main reasons why this could be helpful:

1. Allowing for deeper control of Klipper and the Klipper host
2. A learning bridge for creating Klipper plugins/extras
3. A tool to help develop Klipper plugins/extras without restarting Klipper

To run Python from within a Dynamic Macro, use either the `python()` utility function, or the `python_file()` utility function. The `python()` function accepts python code as a multiline string, and the `python_file()` function accepts a filename (relative to your `printer.cfg` folder).

Tip

When using the `python()` utiltiy function, Jinja2 (which converts the macro to GCode) may throw errors during parsing. If you are getting errors, it is recommended to switch to `python_file()`.

Here are a few examples:

11.1.1 Python Math

macros.cfg

```
[gcode_macro MATH]
gcode:
    {% set value = python("""
    a = kwargs['a']
    b = kwargs['b']
    c = a + b
    output(c)
    """, a=1, b=2) %}
    RESPOND MSG={value}
```

11.1.2 Python File Running GCode

macros.cfg

```
[gcode_macro PYFILE]
gcode:
    {% set value = python_file("test.py") %}
    RESPOND MSG={value}
```

test.py

```
print("Hello from Python!")
gcode("G28\nG1 X100 Y100 Z100 F1200")
output("GCode Executed")
```

12 Delayed GCode

Dynamic Macros supports a feature similar to `delayed_gcode`.

12.1 Configuration

To configure a DynamicMacro to be run repeatedly/after a delay, update your macro as follows:

Before

```
[gcode_macro test]
gcode:
    RESPOND MSG="test"
```

After

```
[gcode_macro test]
initial_duration: 2 # Wait two seconds after Klipper starts to run this, and two seconds between repeats (if enabled)
repeat: true # true means repeat, false means don't repeat
gcode:
    RESPOND MSG="test"
```

12.2 Parameters

`initial_duration` specifies the delay after Klipper starts that the macro will be run, and the duration between repeats, if `repeat` is true.

`repeat` specifies whether or not to repeat the macro.

III. Klippy Extras

13 Start Writing Klippy Extras

While DynamicMacros makes Klipper macros much more powerful, sometimes a Klippy extra is required for more functionality. In this tutorial, you will learn how to develop a Klippy extra and test it using DynamicMacros.

Info

To write a Klipper extra, you should be **fluent** in Python.

Info

This guide is still under construction

Klipper vs Klippy

To limit confusion, I'm going to explain this here. "Klipper" is the name of the firmware running on your 3D printer. "Klippy" is the name of the software running on your Klipper Host (usually a Raspberry Pi). Klipper is written in C code, and Klippy is mostly written in Python.

Start writing your first Klippy extra:

Start {...}

14 Example 1: Greeter (Structure)

> **Info**
>
> This page is under construction.

Before you start to write a Klippy extra, it is important to understand the structure of a Klippy extra.

In Python, a Klippy extra is defined as a class, then referenced in a function at the end of the file. Example:

greeter.py

```
class Greeter:
...

def load_config(config):
    return Greeter(config)
```

There are two functions that can be used at the end of a file:

- `load_config_prefix`, allows for configurations like `[greeter mygreeter]`
- `load_config`, like in this example, allows for configurations like `[greeter]`

14.1 Initializer

In the last line of the above code block, you can see that a `config` object is passed as a parameter to the `Greeter` object. The initializer of the `Greeter` class is shown below in sections:

Get printer objects and configuration options

```
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')

        self.message = config.get('message', 'Welcome to Klipper!')
```

This section gets the `printer` object with `config.get_printer()`.

It then gets the `gcode` object with `self.printer.lookup_object('gcode')`.

Then, it gets the `reactor` object with `self.printer.get_reactor()`.

After that, it reads the configuration to get the `message` parameter, specifying `'Welcome to Klipper!'` as the default, using `config.get('message', 'Welcome to Klipper!')`.

The next part of the initializer:

Register GCode command and event handler

```
        self.gcode.register_command(
            'GREET', self.cmd_GREET, desc=self.cmd_GREET_help)
        self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)
```

The `self.gcode` object is used here to register the `GREET` command:

`self.gcode.register_command('GREET', self.cmd_GREET, desc=self.cmd_GRRET_help)`

The `self.printer` object is then used to register an event handler to run when Klippy starts:

`self.printer.register_event_handler('klippy:ready', self._ready_handler)`

Event handlers

Klipper supports the following event handlers for extras to use:

- `"klippy:ready"`
- `"klippy:firmware_restart"`
- `"klippy:disconnect"`

The combined initializer is:

greeter.py

```
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')

        self.message = config.get('message', 'Welcome to Klipper!')

        self.gcode.register_command(
            'GREET', self.cmd_GREET, desc=self.cmd_GREET_help)
        self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)
```

14.2 Functions

The next part of this Klippy extra is the class functions. This class has three functions aside from the initializer, two of which were mentioned in the `__init__` function:

- `cmd_GREET`
- `_ready_handler`

This example will start with the `_ready_handler` function:

_ready_handler

```
    def _ready_handler(self):
        waketime = self.reactor.monotonic() + 1
        self.reactor.register_timer(self._greet, waketime)
```

This event handler sets a timer for one second after Klippy starts to run `_greet()`. `self.reactor.monotonic()` represents the current time, and `+ 1` adds one second. `self.reactor.register_timer` registers `_greet()` to run when `waketime` occurs.

_greet()

```
    def _greet(self, eventtime=None):
        self.gcode.respond_info(self.message)
        return self.reactor.NEVER
```

This function uses the `self.gcode` object declared in the initializer. Here, the `respond_info` command is used (similar to running `RESPOND MSG=""`) to display `self.message`.

What is `eventtime`?

You may have noticed that `eventtime` is passed to the `_greet()` function. This is because when Klippy runs `_greet()` from the previous `register_timer`, it passes `eventtime` as a parameter. This is useful if you want a function to repeat itself after a specified interval. In this case, we only want it to run once, so `eventtime` is unused.

The `self.gcode` object

The `self.gcode` object has a few useful functions:

- `register_command` (used in the initializer)
- `register_mux_command` (explained later)
- `respond_info` (used here)
- `run_script_from_command` (runs the provided string as GCode. The provided string can be multiple lines long)

Finally, the `cmd_GREET` function:

cmd_GREET

```
    cmd_GREET_help = "Greet the user"
    def cmd_GREET(self, gcmd):
        self._greet()
```

Here, you can see the `cmd_GREET_help` is set to a string. Next, the `cmd_GREET` function takes in a `gcmd` parameter (unused). Finally, the `cmd_GREET` function calls `_greet()`.

The `gcmd` parameter

The `gcmd` parameter allows functions to receive parameters. For example, if `GREET REPEAT=3` was called, the `REPEAT` parameter could be read as follows:

```
repeat = gcmd.get_int('REPEAT', 1)
```

There are a few `get_` functions that can be used with the `gcmd` parameter:

- `get` returns a `str`
- `getint` returns an `int`
- `getfloat` returns a `float`

`gcmd` also has a `respond_info` function, similar to the `self.gcode.respond_info` function.

The `repeat` variable can then be used:

```
for _ in range(repeat):
    self._greet()
```

14.3 Install

To install a Klippy extra, it has to be placed in the `~/klipper/klippy/extras/` folder. Here's a simple command to install `greeter.py`:

```
cp greeter.py ~/klipper/klippy/extras/greeter.py
```

You can also create an install script that uses the `ln` command to create a link to the file, rather than a copy:

install.sh

```
ln -f greeter.py ~/klippy/klippy/extras/greeter.py
```

14.4 Other Things

A few things that are good to know before moving on:

- If your Klippy extra uses `load_config_prefix`, instead of `load_config`, you can get the name of the configuration section (e.g. `[greeter first]` is named `first`) by using:

```
config.get_name().split()[1]
```

- If you want to learn more about additional capabilities of Klippy extras, check out the built-in Klippy extras.
- For further explanation of topics on this page, open the dropdowns.

Next example:

Example 2: BetterGreeter

15 Example 2: BetterGreeter

Info

This page is under construction.

For this tutorial, we are going to improve on the greeter code used in Structure.

For reference, the entire original greeter code is below:

Original Greeter

```
class Greeter:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')

        self.message = config.get('message', 'Welcome to Klipper!')

        self.gcode.register_command(
            'GREET', self.cmd_GREET, desc=self.cmd_GREET_help)
        self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)

    def _ready_handler(self):
        waketime = self.reactor.monotonic() + 1
        self.reactor.register_timer(self._greet, waketime)

    def _greet(self, eventtime=None):
        self.gcode.respond_info(self.message)
        return self.reactor.NEVER

    cmd_GREET_help = "Greet the user"
    def cmd_GREET(self, gcmd):
        self._greet()

def load_config(config):
    return Greeter(config)
```

15.1 Goals

For the `BetterGreeter`, we want the following features:

- Ability to have multiple different greetings
- Allow the user to choose if they want their greeting to display after Klipper starts, and if so, to set the delay
- Allow the user to set the message of the greeting

Here's an example configuration:

better_greeter.cfg

```
[greeting welcome]
message: Welcome to Klipper!
delay: 1

[greeting gcode]
message: Upload some GCode!
delay: 2

[greeting print_done]
message: Print completed!
```

Here's the desired behavior (anything on the line of a `>` is a user-typed command):

```
> RESTART
Welcome to Klipper! (1)
Upload some GCode! (2)
> GREETING NAME=print_done
Print completed!
```

(1) One second after `RESTART`

(2) Two seconds after `RESTART`

15.2 Creating the Base Class

The first step of creating a Klippy extra is to make the base class and the config function:

```
class Greeting:
    def __init__(self, config):
        pass

(1)
def load_config_prefix(config):
    return Greeting(config)
```

(1) `load_config_prefix` is used here instead of `load_config` because there will be **multiple** `greeter` configuration sections.

15.3 Reading the Configuration

The next step of our Klippy extra is to setup the class variables and read the parameters:

```
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)
```

Here, the `printer`, `reactor`, `gcode`, and `message` variables are the same as in the previous Klippy extra. However, in this case, there are a couple new variables:

- `name` is explained in the last section of Structure.
- `delay` is read as an `int` from the `config` object, with default value `0`. The default value of `0` indicates it will not be run when Klippy starts.

15.4 GCode Commands and Event Handler

After reading the configuration variables, we need to setup the GCode commands and event handler:

```
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)

        (1)
        self.gcode.register_mux_command(
            'GREETING',
            'NAME',
            self.name,
            self.cmd_GREETING,
            desc=self.cmd_GREETING_help
        )

        if self.delay > 0:
            self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)
```

(1) `register_mux_command` is used here because there are **multiple** `greeting` configuration sections, and each should be called separately.

Here, a few parts are similar to the example in Structure, but not identical. Let's start with the GCode command.

In the Structure example, the GCode command was registered with:

`self.gcode.register_command('GREET', self.cmd_GREET, desc=self.cmd_GREET_help)`

In this new example, the GCode command is registered with:

```
self.gcode.register_mux_command(
    'GREETING',
    'NAME',
    self.name,
    self.cmd_GREETING,
    desc=self.cmd_GREETING_help
)
```

The difference here is that there can be **multiple** `greeting` configuration sections, and as a result, multiple `Greeting` objects. To call each one separately, `register_mux_command` is used, passing the following parameters:

- Macro name: `"GREETING"`
- Parameter name: `"NAME"`
- Parameter value: `self.name`
- Function: `self.cmd_GREETING`
- Description `self.cmd_GREETING_help`

Next, the `register_event_handler` is nearly identical to the Structure example, except in this case, it is run only `if self.delay > 0`.

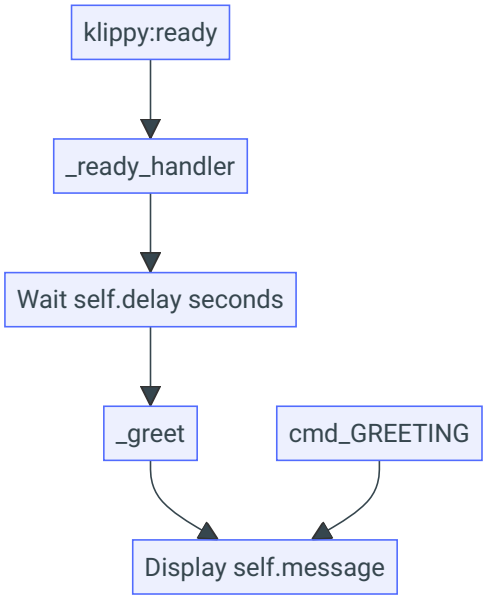
15.5 Functions

The next part to creating this Klippy extra is the functions.

There are three functions in the `Greeting` class:

- `ready_handler`
- `_greet`
- `cmd_GREETING`

Flowchart



The first function, `_ready_handler`:

greeting.py

```
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)
        self.gcode.register_mux_command(
            'GREETING',
            'NAME',
            self.name,
            self.cmd_GREETING,
            desc=self.cmd_GREETING_help
        )

        if self.delay > 0:
            self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)

    def _ready_handler(self):
        waketime = self.reactor.monotonic() + self.delay
        self.reactor.register_timer(self._greet, waketime)
```

This function takes no parameters, and runs when Klippy reports `"klippy:ready"`.

Inside, the `waketime` variable is declared as `self.reactor.monotonic() + self.delay`. Let's break this declaration down:

- `self.reactor.monotonic()` This is the current Klippy time
- `+ self.delay` This adds `self.delay` to the current time.

The result is that `waketime` contains the Klippy time for `self.delay` seconds in the future. Finally, we use `self.reactor.register_timer` to register this time, telling it to run `self._greet()` when the time occurs.

The next function, `_greet`:

greeting.py

```
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)
        self.gcode.register_mux_command(
            'GREETING',
            'NAME',
            self.name,
            self.cmd_GREETING,
            desc=self.cmd_GREETING_help
        )

        if self.delay > 0:
            self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)

    def _ready_handler(self):
        waketime = self.reactor.monotonic() + self.delay
        self.reactor.register_timer(self._greet, waketime)

    def _greet(self, eventtime=None):
        self.gcode.respond_info(self.message)
        return self.reactor.NEVER
```

This function takes an optional `eventtime` parameter (unused) (1), and prints out `self.message` to the Klipper console, using `self.gcode.respond_info`.

(1) This is passed by Klippy when it calls `_greet()` after the timer occurs.

The final function in this class is the `cmd_GREETING` function:

greeting.py

```
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)
        self.gcode.register_mux_command(
            'GREETING',
            'NAME',
            self.name,
            self.cmd_GREETING,
            desc=self.cmd_GREETING_help
        )

        if self.delay > 0:
            self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)

    def _ready_handler(self):
        waketime = self.reactor.monotonic() + self.delay
        self.reactor.register_timer(self._greet, waketime)

    def _greet(self, eventtime=None):
        self.gcode.respond_info(self.message)
        return self.reactor.NEVER

    cmd_GREETING_help = "Greet the user"
    def cmd_GREETING(self, gcmd):
        self._greet()
```

This function simply calls `self._greet()`, which displays `self.message` to the Klipper terminal.

15.6 Full Code

The full code of this Klippy extra is:

greeting.py

```python
class Greeting:
    def __init__(self, config):
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.message = config.get('message', 'Welcome to Klipper!')
        self.delay = config.getint('delay', 0)

        self.gcode.register_mux_command(
            'GREETING',
            'NAME',
            self.name,
            self.cmd_GREETING,
            desc=self.cmd_GREETING_help
        )

        if self.delay > 0:
            self.printer.register_event_handler(
            'klippy:ready', self._ready_handler)

    def _ready_handler(self):
        waketime = self.reactor.monotonic() + self.delay
        self.reactor.register_timer(self._greet, waketime)

    def _greet(self, eventtime=None):
        self.gcode.respond_info(self.message)
        return self.reactor.NEVER

    cmd_GREETING_help = "Greet the user"
    def cmd_GREETING(self, gcmd):
        self._greet()

def load_config_prefix(config):
    return Greeting(config)
```

You can install it following these instructions, replacing `greeter.py` with `greeting.py`.

Last example (so far):

Example 3: KlipperMaintenance

16 Example 3: KlipperMaintenance

This last example of the Klippy extra development tutorial is actually one of my other Klipper projects: KlipperMaintenance, a maintenance reminder system for Klipper.

This example will go through how the KlipperMaintenance code works to teach a few more important things about developing Klippy extras.

16.1 Full Code

For this tutorial, we'll start with the full code, then break it down and explain what each section does. If you can learn to read existing Klippy extras, you can read the builtin Klippy extras when developing your own extras.

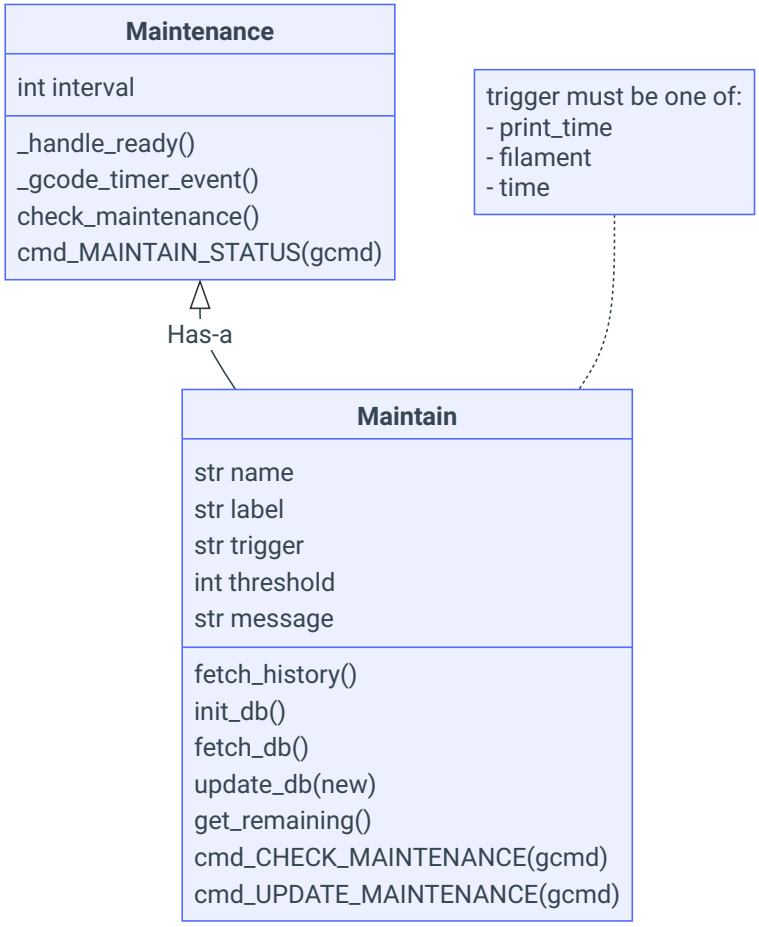
Full Code

```
import json
import os
import time
import urllib.request as requests

API_URL = 'http://localhost:7125/server/history/totals'
HOME_DIR = os.path.expanduser('~')

class Maintenance:
    def __init__(self, config):
        self.config = config
        self.printer = config.get_printer()
        self.reactor = self.printer.get_reactor()
        self.gcode = self.printer.lookup_object('gcode')

        self.interval = config.getint('interval', 60)

        self.timer_handler = None
        self.inside_timer = self.repeat = False
        self.printer.register_event_handler("klippy:ready", self._handle_ready)

        self.gcode.register_command('MAINTAIN_STATUS', self.cmd_MAINTAIN_STATUS, desc=self.cmd_MAINTAIN_STATUS_help)

    def _handle_ready(self):
        waketime = self.reactor.monotonic() + self.interval
        self.timer_handler = self.reactor.register_timer(
            self._gcode_timer_event, waketime)

    def _gcode_timer_event(self, eventtime):
        self.inside_timer = True
        self.check_maintenance()
        nextwake = eventtime + self.interval
        self.inside_timer = self.repeat = False
        return nextwake

    def check_maintenance(self):
        objs = self.printer.lookup_objects('maintain')
        for obj in objs:
            obj = obj[1]
            if not isinstance(obj, Maintain):
                continue
            if obj.get_remaining() < 0:
                self.gcode.respond_info(f'Maintenance "{obj.label}" Expired!\n{obj.message}')
                self.gcode.run_script_from_command('M117 Maintenance Expired!')

    cmd_MAINTAIN_STATUS_help = 'Check status of maintenance'
    def cmd_MAINTAIN_STATUS(self, gcmd):
        objs = self.printer.lookup_objects('maintain')
        for obj in objs:
            obj = obj[1]
            if not isinstance(obj, Maintain):
                continue
            remain = obj.get_remaining()
            if remain < 0:
                self.gcode.respond_info(f'Maintenance "{obj.label}" Expired!\n{obj.message}')
            self.gcode.respond_info(f'{obj.label}: {obj.get_remaining()}{obj.units} remaining')

class Maintain:
    def __init__(self, config):
        self.config = config
        self.printer = config.get_printer()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1]

        self.label = config.get('label')

        self.trigger = config.getchoice('trigger', ['print_time', 'filament', 'time'])
        if self.trigger == 'print_time':
            self.units = 'h'
        elif self.trigger == 'filament':
            self.units = 'm'
        elif self.trigger == 'time':
            self.units = 'h'

        self.threshold = config.getint('threshold')
        self.message = config.get('message')

        self.init_db()
```

```
79
80          # register GCode commands
81          self.gcode.register_mux_command('CHECK_MAINTENANCE', 'NAME', self.name, self.cmd_CHECK_MAINTENANCE,
82  desc=self.cmd_CHECK_MAINTENANCE_help)
83          self.gcode.register_mux_command('UPDATE_MAINTENANCE', 'NAME', self.name, self.cmd_UPDATE_MAINTENANCE,
84  desc=self.cmd_UPDATE_MAINTENANCE_help)
85
86      def fetch_history(self):
87          resp = requests.urlopen(API_URL) # fetch data from Moonraker History API
88          try:
89              json_data = json.loads(resp.read())
90          except Exception:
91              self.gcode.respond_info(f'Data {resp.read()}')
92              return {
93                  'print_time': 0,
94                  'filament': 0,
95                  'time': time.time()/3600
96              }
97
98          job_totals = json_data['result']['job_totals'] # get job totals from JSON response
99          return {
100             'print_time': job_totals['total_time']/3600,
101             'filament': job_totals['total_filament_used']/1000,
102             'time': time.time()/3600
103         }
104
105     def init_db(self):
106         data = self.fetch_db()
107         if data is None:
108             data = self.fetch_history()
109             self.update_db(data)
110
111     def fetch_db(self):
112         path = os.path.join(HOME_DIR, f'maintain-db/{self.name}')
113         if os.path.exists(path):
114             with open(path, 'r') as file:
115                 try:
116                     data = json.load(file)
117                 except:
118                     data = {'print_time': 0, 'filament': 0, 'time': time.time()/3600}
119                 return data
120
121     def update_db(self, new):
122         path = os.path.join(HOME_DIR, f'maintain-db/{self.name}')
123         os.makedirs(os.path.dirname(path), exist_ok=True)
124         with open(path, 'w+') as file:
125             try:
126                 data = json.load(file)
127             except:
128                 data = {'print_time': 0, 'filament': 0, 'time': time.time()/3600}
129             data.update(new)
130             json.dump(data, file)
131         return data
132
133     def get_remaining(self):
134         last = self.fetch_db()[self.trigger]
135         now = self.fetch_history()[self.trigger]
136         return round(self.threshold - (now - last), 2)
137
138     cmd_CHECK_MAINTENANCE_help = 'Check maintenance'
139     def cmd_CHECK_MAINTENANCE(self, gcmd):
140         gcmd.respond_info(f'''
141 Maintenance {self.label} Status:
142 Next maintenance in {self.get_remaining()}{self.units}
143 Maintenance message: {self.message}
144         '''.strip())
145
146     cmd_UPDATE_MAINTENANCE_help = 'Update maintenance'
147     def cmd_UPDATE_MAINTENANCE(self, gcmd):
148         self.update_db(self.fetch_history())
149
150 def load_config(config):
151     return Maintenance(config)
152
    def load_config_prefix(config):
        return Maintain(config)
```

16.2 Structure

The class structure of this Klippy extra is shown in this flowchart:



This diagram has a lot of information, but the key points are:

- Multiple `Maintain` objects are managed by one `Maintainence` object.
- The `Maintain` class handles maintenance reminders, the `CHECK_MAINTENANCE` command, and the `UPDATE_MAINTENANCE` command.
- The `Maintenance` class handles the `MAINTAIN_STATUS` command.
- A `[maintain name]` section corresponds to the `Maintain` class (multiple).
- A `[maintain]` section corresponds to the `Maintenance` class (one).

The base code for this is:

maintain.py

```
import json
import os
import time
import urllib.request as requests

API_URL = 'http://localhost:7125/server/history/totals'
HOME_DIR = os.path.expanduser('~')

class Maintenance:
    pass

class Maintain:
    pass

(1)
def load_config(config):
    return Maintenance(config)

(2)
def load_config_prefix(config):
    return Maintain(config)
```

(1) This function corresponds to a `[maintain]` config section.

(2) This function corresponds to a `[maintain name]` config section.

The highlighted section in the code contains the classes, and the configuration loading, the parts relevant to this first section. The code above the highlighted section includes the relevant module imports and constant declarations (both explained later).

Much of the content in this example will be embedded in the code (usually the highlighted section) as a plus sign, like this: (1)

(1) Click some more plus marks in the code to learn more about the code.

16.3 `Maintain` class

This next section of the example will focus on the `Maintain` class.

16.3.1 Initializer

When `load_config_prefix` creates a `Maintain` object, it starts in the initializer.

Here's the full initializer, with explanations embedded inside with plus signs. Below is a more general breakdown of the initializer for clarity.

maintain.py

```
class Maintain:
    def __init__(self, config):
        self.config = config
        self.printer = config.get_printer()
        self.gcode = self.printer.lookup_object('gcode')
        self.name = config.get_name().split()[1] (1)

        self.label = config.get('label')

        (2)
        self.trigger = config.getchoice('trigger', ['print_time', 'filament', 'time'])
        if self.trigger == 'print_time': (3)
            self.units = 'h'
        elif self.trigger == 'filament':
            self.units = 'm'
        elif self.trigger == 'time':
            self.units = 'h'

        self.threshold = config.getint('threshold')
        self.message = config.get('message')

        (4)
        self.init_db()

        self.gcode.register_mux_command('CHECK_MAINTENANCE', 'NAME', self.name, self.cmd_CHECK_MAINTENANCE,
desc=self.cmd_CHECK_MAINTENANCE_help)
        self.gcode.register_mux_command('UPDATE_MAINTENANCE', 'NAME', self.name, self.cmd_UPDATE_MAINTENANCE,
desc=self.cmd_UPDATE_MAINTENANCE_help)
```

(1) `config.get_name()` returns the full config name, like `"maintain name"`. The `.split()[1]` splits the name by spaces and gets the last "word".

(2) `config.getchoice()` allows you to only accept values in a certain list.

(3) This part of the initializer chooses the units based on the `trigger` type:

(4) `"print_time"`: `"h"`

(5) `"filament"`: `"m"`

(6) `"time"`: `"h"`

(7) This is explained later, in functions.

The major breakdown of this initializer is:

- **First highlighted section:** Load basic objects
- **Second highlighted section:** Read configuration options
- **Third highlighted section:** Register GCode commands

The line `self.init_db()` will be explained later in functions.

16.3.2 Functions

The next part of the `Maintain` class is its functions. The functions in this class are, broken down into two sections:

Database:

- `fetch_history()`
- `init_db()`
- `fetch_db()`
- `update_db(new)`

GCode:

- `get_remaining()`
- `cmd_CHECK_MAINTENANCE(gcmd)`

- `cmd_UPDATE_MAINTENANCE(gcmd)`

First, the database functions (again, plus signs throughout the code explain in more detail what each part does):

maintain.py

```
class Maintain:
    ...
    def fetch_history(self):
        resp = requests.urlopen(API_URL) # fetch data from Moonraker History API
        try:
            json_data = json.loads(resp.read())
        except Exception:
            self.gcode.respond_info(f'Data {resp.read()}')
            return {
                'print_time': 0,
                'filament': 0,
                'time': time.time()/3600
            } (1)

        job_totals = json_data['result']['job_totals'] # get job totals from JSON response
        return {
            'print_time': job_totals['total_time']/3600,
            'filament': job_totals['total_filament_used']/1000,
            'time': time.time()/3600
        } (2)

    def init_db(self):
        data = self.fetch_db() # Load the database
        if data is None:
            data = self.fetch_history()
            self.update_db(data) # Update the database with history data

    def fetch_db(self):
        path = os.path.join(HOME_DIR, f'maintain-db/{self.name}') (3)
        if os.path.exists(path):
            with open(path, 'r') as file:
                try:
                    data = json.load(file) # JSON parse the file contents
                except:
                    data = {'print_time': 0, 'filament': 0, 'time': time.time()/3600}
                return data # Return parsed data

    def update_db(self, new):
        path = os.path.join(HOME_DIR, f'maintain-db/{self.name}') (3)
        os.makedirs(os.path.dirname(path), exist_ok=True) (4)
        with open(path, 'w+') as file: # The "w+" file operator allows reading and writing
            try:
                data = json.load(file) # JSON parse the file contents
            except:
                data = {'print_time': 0, 'filament': 0, 'time': time.time()/3600}
            data.update(new) # Update the file contents with the new data
            json.dump(data, file) # JSON write the new data to the file
        return data # Return the new updated data
```

(1) In case of an error, empty placeholder data is returned.

(2) If history fetch was successful, return the data read from the Moonraker History API.

(3) The `path` is (if the username is `pi` and `self.name` is `"lubricate"`) `"/home/pi/maintain-db/lubricate"`. Even though it has no file extension, it stores JSON data.

(4) The first time running this, the `maintain-db` folder won't exist. `os.makedirs` creates the folder, and `exist_ok=True` doesn't throw an error if it already exists.

The flow of information in this Klippy extra is:

- `init_db()` is called when Klippy starts
- `fetch_db()` is called to read the stored database
- If the data returned by `fetch_db()` is `None` (database is empty)

- `fetch_history()` is called to fetch the history stored by Moonraker
- `update_db()` is called to update the database with the newly fetched data.

Calling `update_db()` will erase the current maintenance state (resetting the timer/filament counter).

The next section of functions in the `Maintain` class are the GCode commands. There are three functions in this section:

- `get_remaining()`
- `cmd_CHECK_MAINTENANCE(gcmd)`
- `cmd_UPDATE_MAINTENANCE(gcmd)`

17 Example Macros

This page will hold several different Dynamic Macro examples. Note that most of the examples here are specific to Dynamic Macros only.

17.1 M900

Normal Macro

In Marlin, M900 K is used to set pressure/linear advance. Now, you can use it in Klipper too:

```
[gcode_macro M900]
description: Set Pressure Advance
gcode:
  {% set k = params.K|default(0)|float %}
  {% if k < 1 %}
    SET_PRESSURE_ADVANCE ADVANCE={k}
  {% endif %}
```

17.2 Recursion

Dynamic Macro

These are a few example Dynamic Macros to demonstrate the recursive functionality of Dynamic Macros.

Counting to 10

```
[gcode_macro COUNT]
gcode:
    {% set num = params.NUM|default(1)|int %}
    {% if num <= 10 %}
        RESPOND MSG={num}
        COUNT NUM={num+1} # Count up 1
    {% else %}
        RESPOND MSG="Done Counting"
    {% endif %}
```

Load to Filament Sensor

```
[gcode_macro LOAD_TO_FSENSOR]
gcode:
    {% set val = printer["filament_sensor fsensor"].filament_detected %}
    {% if val == 0 %}
        M83
        G1 E50 F900 # Move filament 50mm forwards
        RESPOND MSG="Waiting for fsensor"
        LOAD_TO_FSENSOR # Recursion
    {% else %}
        G1 E65 F900 # Move filament to nozzle
        RESPOND MSG="Filament Loaded"
    {% endif %}
```

17.3 Receiving Position Updates

Dynamic Macro

This is an example Dynamic Macro to demonstrate the ability to receive position updates from within the same macro.

```
[gcode_macro DYNAMIC_MOVE]
gcode:
  G28


  M117 Before: {printer.toolhead.position.z}
  # Above displays position after G28
  G90
  G1 Z20


  M117 After: {printer.toolhead.position.z}
  # Above displays position after G1
```

17.4 Preserving Variables

✓ **Dynamic Macro**

This is an example of how to preserve variables across triple-newlines in Dynamic Macros.

```
[gcode_macro VARIABLES]
gcode:
    {% set num = update("num", 10) %}
    M117 {num}


    M117 {num}
    # Above line outputs 10
```

18 Development Status

18.1 Features

- Editing macros without restarting Klipper
- Accessing printer information from within Dynamic Macros
- Accessing parameters from within Dynamic Macros
- Adding new Dynamic Macros without restarting Klipper
- Removing existing Dynamic Macros without restarting Klipper
- Renaming Dynamic Macros without restarting Klipper
- Dynamic Macro descriptions
- Dynamic Macro variables
- Retrieving variables from other macros
- Support for `rename_existing`
- Running Python from within a Dynamic Macro
- Dynamic `delayed_gcode` implementation

18.2 Planned Features

A checkmark indicates a feature is experimental.

- ✅ Allow configuring multiple `dynamicmacros` config sections as clusters
  - ✅ Disable Python per-cluster
  - ✅ Disable `printer` object per-cluster
- Better error handling
- Klippy extras tutorial