**Overview**

The **PrintManager** class is designed to manage print jobs for a 3D printing setup that operates on well plates. It does the following:

* Maintains a dynamic queue of print jobs, where each job contains:
  + A well identifier (e.g., "A1")
  + Target coordinates (X, Y, Z) for that well
  + An associated print file (which contains detailed printing instructions)
* Uses a global “fast-move” Z value to quickly lift the stage before any XY movements to avoid collisions.
* Processes print job files that load a list of jobs into the queue and offers the ability to save the current queue to a file.
* Supports runtime commands to start, pause, resume, or stop print job processing.
* Provides commands to move (GOTO) and calibrate the stage based on well locations.

**Initialization**

To use the PrintManager, you must instantiate it with:

* **processor:** An instance of your command processor that handles command registration and dispatch.
* **stage\_handler:** An instance that controls stage movements (for both XY and Z).

Example:

python

Copy

print\_manager = PrintManager(processor, stage\_handler)

**Features & Capabilities**

**1. Global Fast-Move Z Setting**

* **Purpose:**  
  The fast-move Z is a safety parameter that lifts the Z-axis to a predefined height before any XY movement is executed.
* **How to Set It:**  
  Use the "set\_fastmove\_z" command (or call set\_fastmove\_z(z\_value) directly) to change the global fast-move height.  
  *Example:*

python

Copy

processor.add\_command("set\_fastmove\_z", 15.0)

This sets the fast-move Z to 15.0 (units as defined by your system).

**2. Dynamic Well Queue Management**

* **Job Structure:**  
  Each print job in the queue is a dictionary with these keys:
  + "well\_id": The identifier for the well (e.g., "A1").
  + "target": A tuple of coordinates (target\_x, target\_y, target\_z).
  + "print\_file": The filename containing printing instructions for that job.
* **Listing Jobs:**  
  The "list\_print\_jobs" command displays and returns a list of pending jobs as tuples of (well\_id, print\_file).  
  *Example:*

python

Copy

processor.add\_command("list\_print\_jobs")

* **Retrieving the Next Job:**  
  The method get\_next\_job() pops the next job from the queue. This is used when the system processes a "GOTO: NEXT" command.

**3. Loading and Saving Print Jobs**

**Loading a Print Job File**

* **File Format:**  
  The file should be a CSV file where each line is formatted as:

Copy

well\_id, target\_x, target\_y, target\_z, print\_file

* **Behavior:**  
  When you issue the "load\_print\_job" command, the PrintManager:
  + Clears the existing job queue.
  + Reads the file line by line.
  + Parses each valid line and adds it to the queue.
* **Example Usage:**

python

Copy

processor.add\_command("load\_print\_job", "print\_jobs.csv")

**Saving the Current Print Job Queue**

* **Purpose:**  
  Save the current queue into a CSV file in the same format so it can be reloaded later.
* **Example Usage:**

python

Copy

processor.add\_command("save\_print\_job", "saved\_jobs.csv")

**4. Print Job Processing**

* **Starting the Print Job:**  
  The "start\_print" command begins processing the queue. If a file path is provided, it will load that job file (clearing the existing queue first).  
  *Example:*

python

Copy

processor.add\_command("start\_print", "print\_jobs.csv")

* **Processing Steps:**  
  For each job, the class:
  1. **GOTO Movement:**
     + Moves Z to the global fast-move height.
     + Moves XY to the job’s target X and Y coordinates.
     + Lowers Z to the job’s target Z coordinate.
  2. **Executing the Print File:**  
     If a print file is specified for the job, the class reads that file and processes each instruction.
     + **Print File Format:** Each line in the print file should be a CSV line with the following fields:

css

Copy

x, y, z, p1, p2, p3, time

* + - For every instruction:
      * The system first moves Z to the fast-move height.
      * Then moves XY to the specified x and y.
      * Finally moves Z to the given z value.
      * Additional parameters (p1, p2, p3) are processed as needed.
      * The system waits for the specified time (to allow for dwell or processing).
  1. **Respecting Control Flags:**
     + **Pause:** If the print process is paused via the "pause\_print" command, the system halts until "resume\_print" is issued.
     + **Stop:** If "stop\_print" is issued, the processing stops and resets.

**5. Movement Commands**

**GOTO Command**

* **Usage:**  
  The "goto" command moves the stage to a specified well location.
* **Parameters:**
  + If you pass "NEXT", it automatically dequeues and processes the next job in the queue.
  + Otherwise, it searches for a job matching the given well identifier (without removing it from the queue) and moves to that location.
* **Example Usage:**

python

Copy

processor.add\_command("goto", "NEXT")

# or

processor.add\_command("goto", "A1")

**CALIBRATE Command**

* **Usage:**  
  The "calibrate" command uses the same movement logic as "goto" to move to a target well location and then waits for an external calibration signal.
* **Example Usage:**

python

Copy

processor.add\_command("calibrate", "A1")

# or

processor.add\_command("calibrate", "NEXT")

* **Note:**  
  Calibration waits for an external event (simulated with a timeout in this implementation).

**6. Job Control Commands**

* **Pause Print:**  
  Use "pause\_print" to temporarily pause the job processing. The current state is maintained so that processing can resume from the same point.

python

Copy

processor.add\_command("pause\_print")

* **Resume Print:**  
  Use "resume\_print" to resume processing after a pause.

python

Copy

processor.add\_command("resume\_print")

* **Stop Print:**  
  Use "stop\_print" to terminate the current processing. This resets the job pointer and stops any ongoing print file processing.

python

Copy

processor.add\_command("stop\_print")

**Example Workflow**

1. **Set the Fast-Move Z Value:**

python

Copy

processor.add\_command("set\_fastmove\_z", 15.0)

1. **Load Print Jobs from a File:**  
   This clears the current queue and loads new jobs.

python

Copy

processor.add\_command("load\_print\_job", "print\_jobs.csv")

1. **List Pending Print Jobs:**  
   This will display all jobs with their well IDs and associated print files.

python

Copy

processor.add\_command("list\_print\_jobs")

1. **Start Processing Print Jobs:**  
   Optionally, you can pass the file path to reload jobs.

python

Copy

processor.add\_command("start\_print")

1. **Move to a Specific Well (GOTO):**  
   To move to the next job or a specific well:

python

Copy

processor.add\_command("goto", "NEXT") # Dequeues and moves to the next job.

# or

processor.add\_command("goto", "A1") # Moves to well A1 if found in the queue.

1. **Calibrate the Stage at a Specific Well:**

python

Copy

processor.add\_command("calibrate", "A1")

1. **Pause and Resume Processing:**

python

Copy

processor.add\_command("pause\_print")

# ...when ready...

processor.add\_command("resume\_print")

1. **Stop Processing the Print Job:**

python

Copy

processor.add\_command("stop\_print")

1. **Save the Current Job Queue:**  
   Save the jobs to a file for later use.

python

Copy

processor.add\_command("save\_print\_job", "saved\_jobs.csv")

**Conclusion**

The **PrintManager** class provides a modular and dynamic framework to manage well-based print jobs by:

* Allowing dynamic updating and reordering of well locations.
* Enabling the loading and saving of print jobs from/to CSV files.
* Integrating with stage movement commands to perform safe fast-move operations.
* Offering real-time controls to start, pause, resume, or stop printing.

By using the registered commands through your central Processor, you can easily integrate this class into your overall application workflow, achieving flexible and controlled 3D print operations on well plates.