```
'''
Created on Fri Jun 26 20:01:52 2020

@author: gregorywildes

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

https://www.geeksforgeeks.org/how-to-use-threading-in-pyqt5/
https://www.pythonguis.com/tutorials/multithreading-pyqt-applications-qthreadpool/
'''
import platform

import stepper_Commands
uname = platform.uname()
import os
import numpy as np
import traceback
import sys
import time

import csv
import random
import pickle
import gc
import signal
# Import resource only on Unix-like systems
if platform.system() != 'Windows':
    import resource
else:
    # On Windows, we'll use psutil as an alternative
    import psutil
import datetime
from copy import copy,deepcopy
from os import path
from time import strftime, localtime, gmtime
from pympler import muppy,summary
#from mem_top import mem_top
from PyQt5 import uic,QtGui
from PyQt5 import QtWidgets
from PyQt5 import QtCore
# ------------------------------------------------------------------
# Early QApplication instance to provide Qt context during module import
_early_qapp = QtWidgets.QApplication.instance()
if _early_qapp is None:
    _early_qapp = QtWidgets.QApplication(sys.argv)
# ------------------------------------------------------------------
# from PyQt5.QtCore import QTextBlock, QTextCursor, qRegisterMetaType
# # Register the QTextBlock and QTextCursor types
# qRegisterMetaType(QTextBlock)
# qRegisterMetaType(QTextCursor)
import pandas as pd
from dict_tools import makeitemlist,makevaluelist,makekeylist,namesfix
from command_help import print_command_help
import config
# from threader import myThread
from functools import partial
from PyQt5.QtCore import QTimer
import threading
import queue
from db_writer import event_q, DbWriter, ensure_table
from db_connect import db_init
import mysql.connector as sql
from typing import List
from types import SimpleNamespace
import subprocess
import stepper_index as sidx
import serial
from instrument import Instrument, X_Stepper, Z_Stepper, Tuner_Stepper
from instrument import Stepper

from decimal import Decimal, ROUND_HALF_UP
import pg_steppers  # new Postgres stepper logger

def round_normal(x):
```

```python
    return int(Decimal(x).to_integral_value(rounding=ROUND_HALF_UP))


# Decorator to add execution messages for commands that don't have their own output
def command_logger(func):
    """Decorator that adds an execution message for commands that don't have their own output"""
    def wrapper(self, *args, **kwargs):
        # Get the command name from the function name (remove 'change_' prefix)
        command_name = func.__name__.replace('change_', '')

        # Call the original function
        result = func(self, *args, **kwargs)

        # Check if the function added any message content
        # If not, add a simple execution message
        if not hasattr(self, 'message') or not self.message or self.message.strip() == '':
            self.message += f'\nExecuting: {command_name}'

        return result
    return wrapper

# Initialize globals used by background threads BEFORE they start
CURR_DIR = os.getcwd()
make_change = SimpleNamespace(message='')  # temporary placeholder
new_headers = True  # CSV header flag required by write_csv before init()

# Initialize global runtime state so GUI can reference them safely before
# -------------------------------------------------------------------
# Runtime state arrays
# -------------------------------------------------------------------
# Build *main-board* positions array sized only to the highest index that
# actually lives on the carriage Arduino.  This avoids the earlier mistake
# of allocating space for tuner indices as well.

_main_steps = [s for s in config.ins.all_steps if s and s.arduino == config.ard]
_max_main_index = max((s.index for s in _main_steps), default=-1)

# The old logic for sizing the `positions` and `tuner_positions` arrays
# here was flawed. The correct approach is to have two separate arrays,
# one for each potential Arduino board, sized independently. This is now
# handled correctly in the `init()` function.
x_position: List[int] = []
z_positions: List[int] = []
tuner_positions: List[int] = []

timestamp = None  # Last update time from Arduino(s)
report = None     # Raw (command, data, ts) tuple from last Arduino report

# Add global variable to track if exit is requested
exit_requested = False
_shutdown_in_progress = False

# Add signal handler for clean exit
def signal_handler(sig, frame):
    global exit_requested, _shutdown_in_progress
    if _shutdown_in_progress:
        return  # Already shutting down, ignore
    _shutdown_in_progress = True
    print("\nCtrl+C detected. Cleaning up and exiting...")
    exit_requested = True
    # If we have a TextInputCommands instance, set its exit flag
    if 'make_change' in globals():
        make_change.change_quit(True)
    else:
        # No make_change yet, exit directly
        QtWidgets.QApplication.quit()

def to_args(mess):
    args = list()
    for i in mess[1:]: #.split(',')[1:]:
        try:
            val = int(i)
        except ValueError:
            try:
                val = float(i)
            except ValueError:
                val = i
        args.append(val)
    return args

def str_to_num(inp):
    try:
        out = float(inp)
        if out.is_integer():
            out = int(out)
    except ValueError:
        pass
    return out

def str_to_nums(data):
    result = []
    for inp in data:
        # Default None to 0 to keep column alignment and avoid TypeError
        if inp is None:
            result.append(0)
            continue
        # Preserve non-numeric strings as-is (e.g., timestamp, command)
        if isinstance(inp, str) and not inp.isdigit():
            result.append(inp)
            continue
        # Numbers pass through; floats coerced to int when integral
        try:
            value = float(inp)
            if value.is_integer():
                result.append(int(value))
            else:
                result.append(value)
        except (ValueError, TypeError):
            # As a last resort, coerce to 0 to maintain value count
            result.append(0)
    return result


def ensure_persist_monitor():
    """Launch persist.sh once so the Rust backend supervisor is active."""
    persist_script = os.path.join(os.path.dirname(__file__), 'rust_driver', 'persist.sh')
    if not os.path.isfile(persist_script):
        make_change.message += "\n[persist] Script missing; expected at %s" % persist_script
        return
    if not os.access(persist_script, os.X_OK):
        make_change.message += "\n[persist] Script not executable: %s" % persist_script
        return

    try:
        probe = subprocess.run(
            ['pgrep', '-f', 'Persist Monitor'],
            check=False,
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL
        )
        if probe.returncode == 0:
            make_change.message += "\n[persist] Monitor already running; skipping launch."
            return
    except FileNotFoundError:
        make_change.message += "\n[persist] pgrep unavailable; launching persist.sh without pre-check."

    try:
        subprocess.Popen([persist_script], env=os.environ.copy())
        make_change.message += "\n[persist] Launching persist monitor."
    except Exception as exc:
        make_change.message += "\n[persist] Failed to launch persist: %s" % exc

class TextInputCommands:
    def __init__(self,stor_size=100):
        # if config.db.preset_file is None:
        #     config.db.preset_file = input('preset storage filename: ')
        # if config.db.data_file is None:
        #     config.db.data_file = input('data storage filename: ')
        self.stor_size = stor_size
        self.storlist = []
        self.set_storage = [None,None,None,None,None,None]*stor_size
        self.current_set = 0
        self.exit = False
        self.message = ''
        self.bump_retry_counts = {}

    def box_dispatch(self,newval):
        """Robustly parse and dispatch commands from the GUI text box."""
        try:
            # The input `newval` is a list from `to_args`.
            if not isinstance(newval, list) or not newval:
                self.message += f"\nInvalid command format: {newval}"
                return

            command = str(newval[0])
            args = newval[1:]
            args = str_to_nums(args)

            # The first argument might be a stepper index, which needs to be
            # converted to a Stepper object for modern dispatch.
            # Skip probing for single-arg position/delta commands where the
            # first argument is a target value, not a stepper index.
            if not (command in ('abs_reposition', 'rel_reposition') and len(args) == 1):
                if len(args) > 0:
                    try:
                        stepper_index = int(args[0])
                        stepper_obj = _find_stepper(stepper_index)
                        if stepper_obj:
                            args[0] = stepper_obj
                    except (ValueError, TypeError):
                        # Not a stepper index, leave it as is.
                        pass

            self.dispatch(command, *args)
        except Exception as e:
            self.message += f"\nError dispatching command '{newval}': {e}"
        update()


    def dispatch(self,*args):
        global command
        # Convert stepper objects to their index numbers for cleaner command logging
        processed_args = []
        for arg in args:
            if (hasattr(arg, '__class__') and not isinstance(arg, str) and
                hasattr(arg, 'arduino') and
                hasattr(arg, 'index') and
                not callable(getattr(arg, 'index', None)) and
                isinstance(arg.index, int)):
                # This is likely a stepper object, use its index
                class_name = arg.__class__.__name__
                processed_args.append(f"<{class_name} with index {arg.index}>")
            else:
                processed_args.append(str(arg))

        command = str(tuple(processed_args))
        # Truncate command to fit within database varchar(60) limit
        if len(command) > 60:
            command = command[:57] + "..."
        if len(args) != 0:
            method_name = 'change_' + args[0]
            try:
                method = getattr(self,method_name)
                try:
                    # Same: avoid stray per-command timestamps – rely on print_report().
                    # Suppress raw tuple debug output – rely on print_report() for clean logging.
                    # make_change.message += '\n'+str(args)
                    if len(args) == 1:
                        method()
                    else:
                        method(*args[1:])
                except Exception:
                    traceback.print_exc()
                    make_change.message += '\nUnexpected error: '+str(sys.exc_info())
            except AttributeError:
                make_change.message += '\n'+method_name + ' command not found,type "help" for command list'
            update()

    def change_exit(self,val=True):
        # Only set the exit flag to break out of loops
        self.exit = val

    def change_quit(self,val=True):
        # Exit the entire application (idempotent - only run once)
        global exit_requested, _shutdown_in_progress
        if _shutdown_in_progress:
            return  # Already shutting down
        _shutdown_in_progress = True
        exit_requested = True
        print("\nShutting down gracefully...")
        # Clean up resources
        if hasattr(config, 'sampler') and hasattr(config.sampler, 's'):
            try:
                config.sampler.s.stop()
            except:
                pass
        # Exit the application (let Qt handle it cleanly, no sys.exit)
        QtWidgets.QApplication.quit()

    def change_help(self):
        print_command_help()

    def change_set_accel(self, *args):
        if len(args) == 2:
            which, amount = int(args[0]), int(args[1])
            stepper = _find_stepper(which)
            if stepper and stepper.arduino:
                msg = stepper.arduino.set_accel(stepper.local_index, amount)
                if msg:
                    self.message += msg
                stepper.accel = amount
            else:
                self.message += f'\nStepper {which} not found or not configured.'
        else:
            self.message += '\nUsage: set_accel <stepper_index> <amount>'

    def change_set_speed(self, *args):
        if len(args) == 2:
            which, amount = int(args[0]), int(args[1])
            stepper = _find_stepper(which)
            if stepper and stepper.arduino:
                msg = stepper.arduino.set_speed(stepper.local_index, amount)
                if msg:
                    self.message += msg
                stepper.speed = amount
            else:
                self.message += f'\nStepper {which} not found or not configured.'
        else:
            self.message += '\nUsage: set_speed <stepper_index> <amount>'

    def change_z_step_size(self,*args):
        # Map firmware code to microstep factor (logical distance per raw step)
        global z_step_size
        code_to_factor = {0: 1.0, 4: 0.5, 2: 0.25, 6: 0.125, 7: 0.0625}
        if len(args) == 1:
            code = int(args[0])
            # Send carriage-wide z_size command (applies to all Z steppers in firmware)
            if config.ard.exist:
                msg = config.ard.z_step_size(code)
                if msg:
                    self.message += msg
            # Update global z_step_size to match
            if code in code_to_factor:
                z_step_size = code_to_factor[code]
                # Refresh GUI so increments change immediately
                gui.gui_sync()
            else:
                self.message += f'\nUnknown z_step_size code: {code}'
        else:
            self.message += '\nUsage: z_step_size <stepper_index>'

    def change_set_width(self, *args):
        if len(args) == 2:
            which, amount = args[0], int(args[1])
            if which == 't':
                for stepper in config.ins.tuner_steps:
                    if stepper.arduino:
                        msg = stepper.arduino.set_width(stepper.local_index, amount)
                        if msg:
                            self.message += msg
            elif which == 'x':
                stepper = config.ins.x_step
                if stepper.arduino:
                    msg = stepper.arduino.set_width(stepper.local_index, amount)
                    if msg:
                        self.message += msg
            elif which == 'z':
                for stepper in config.ins.z_steps:
                    if stepper.arduino:
                        msg = stepper.arduino.set_width(stepper.local_index, amount)
                        if msg:
                            self.message += msg
            else:
                self.message += '\nUsage: set_width <t|x|z> <amount>'
        else:
            self.message += '\nUsage: set_width <t|x|z> <amount>'

    def change_set_min(self,*args):
        if len(args) == 2:
            which, where = int(args[0]), int(args[1])
            stepper = _find_stepper(which)
            if stepper and stepper.arduino:
                msg = stepper.arduino.set_min(stepper.local_index, where)
                if msg:
                    self.message += msg
                stepper.min_pos = where
            else:
                self.message += f'\nStepper {which} not found or not configured.'
        else:
            self.message += '\nUsage: set_min <stepper_index> <position>'

    def change_set_max(self,*args):
        if len(args) == 2:
            which, where = int(args[0]), int(args[1])
            stepper = _find_stepper(which)
            if stepper and stepper.arduino:
                msg = stepper.arduino.set_max(stepper.local_index, where)
                if msg:
                    self.message += msg
                stepper.max_pos = where
            else:
                self.message += f'\nStepper {which} not found or not configured.'
        else:
            self.message += '\nUsage: set_max <stepper_index> <position>'

    def change_abs_move(self, *args):
        """Absolute move."""
        if len(args) != 2:
            self.message += '\nUsage: abs_move <stepper> <target>'
            return
        target, target_pos = args
        from instrument import Stepper as _Stepper
        if isinstance(target, _Stepper):
            stepper = target
        else:
            stepper = _find_stepper(int(target))
        if not stepper or not stepper.arduino:
            self.message += f"\nabs_move failed – bad stepper {target}"
            return

        # Calculate delta for Arduino
        if isinstance(stepper, Z_Stepper):
            delta = round_normal((target_pos - stepper.position) / z_step_size)
        else:
            delta = target_pos - stepper.position

        if delta != 0:
            msg = stepper.arduino.rmove(stepper.local_index, delta)
            if msg:
                self.message += msg

        # Update Python position
        stepper.position = target_pos

        # Reset Arduino position
        stepper.arduino.set_stepper(stepper.local_index, round_normal(stepper.position))

        # Wait for move completion
        if isinstance(stepper, X_Stepper):
            waiter(config.ins.x_rest)
        elif isinstance(stepper, Z_Stepper):
            waiter(config.ins.z_rest)
        elif isinstance(stepper, Tuner_Stepper):
            waiter(config.ins.tune_rest)

    def change_rel_move(self, *args):
        """Relative move."""
        if len(args) != 2:
            self.message += '\nUsage: rel_move <stepper> <delta>'
            return
        target, delta = args
        from instrument import Stepper as _Stepper
        if isinstance(target, _Stepper):
            stepper = target
        else:
            stepper = _find_stepper(int(target))
        if not stepper or not stepper.arduino:
            self.message += f"\nrel_move failed – bad stepper {target}"
            return

        # Send raw delta directly to Arduino
        msg = stepper.arduino.rmove(stepper.local_index, delta)
        if msg:
            self.message += msg

        # Update Python position tracking
        if isinstance(stepper, Z_Stepper):
            stepper.position = stepper.position + delta * z_step_size
        else:
            stepper.position = stepper.position + delta

        # Reset Arduino position
        stepper.arduino.set_stepper(stepper.local_index, round_normal(stepper.position))

        # Wait for move completion
        if isinstance(stepper, X_Stepper):
            waiter(config.ins.x_rest)
        elif isinstance(stepper, Z_Stepper):
            waiter(config.ins.z_rest)
        elif isinstance(stepper, Tuner_Stepper):
            waiter(config.ins.tune_rest)

    def change_x_rand_move(self):
        if self.exit:
            return
        if config.ard.exist:
            self.change_abs_move(config.ins.x_step,random.randint(config.ins.x_step.min_pos,config.ins.x_step.right_limit))
        else:
            self.message += '\nno Arduino'

    def change_slow(self):
        for stepper in config.ins.all_steps:
            if stepper.arduino:
                msg = stepper.arduino.move_slow()
                if msg: self.message += msg
        if not any(stepper.arduino for stepper in config.ins.all_steps):
            self.message += '\nNo Arduinos connected'

    def change_medium(self):
        if config.ard_T.exist:
            for stepper in config.ins.tuner_steps:
                stepper.speed = 1000
                stepper.accel = 10000
            config.ins.x_step.speed = 500
            config.ins.x_step.accel = 10000
            for stepper in config.ins.z_steps:
                stepper.speed = 100
                stepper.accel = 10000
            for stepper in config.ins.z_steps:
```

```
            if stepper.arduino:
                msg = stepper.arduino.set_speed(stepper.local_index, stepper.speed)
                if msg: self.message += msg
                msg = stepper.arduino.set_accel(stepper.local_index, stepper.accel)
                if msg: self.message += msg
        for stepper in config.ins.tuner_steps:
            if stepper.arduino:
                msg = stepper.arduino.set_speed(stepper.local_index, stepper.speed)
                if msg: self.message += msg
                msg = stepper.arduino.set_accel(stepper.local_index, stepper.accel)
                if msg: self.message += msg
    else:
        self.message += '\nno Arduino'

def change_fast(self):
    for stepper in config.ins.all_steps:
        if stepper.arduino:
            msg = stepper.arduino.move_fast()
            if msg: self.message += msg
    if not any(stepper.arduino for stepper in config.ins.all_steps):
        self.message += '\nNo Arduinos connected'

def change_right_move(self,*args):
    global left_limit, right_limit
    if config.ard.exist:
        if len(args) == 0:
            step_size = 10
        elif len(args) == 1:
            step_size = args[0]
        elif len(args) == 2:
            step_size = args[1]
            left_limit = args[0]
        elif len(args) == 3:
            step_size = args[2]
            left_limit = args[0]
            right_limit = args[1]
        self.change_abs_move(config.ins.x_step,left_limit)
        # self.dispatch('z_calibrate')
        while config.ins.x_step.position < right_limit:
            update()
            if self.exit:
                return
            if config.db.table_enable:
                t_spot,message = config.db.table_check(config.ins.x_step,z_positions)
                self.message += message
                if t_spot is not None:
                    for i,j in enumerate(config.ins.z_steps):
                        self.change_abs_move(j,t_spot[i])
                        #self.dispatch('abs_move',j.index,t_spot[i])
            self.change_z_adjuster_tuner()
            if config.db.table_enable:
                if config.db.table_enable:
                    self.message += config.db.table_update(config.ins.x_step,config.ins.z_steps,z_positions)
            if config.ins.x_step.enable:
                self.change_rel_move(config.ins.x_step,step_size)
    else:
        self.message += '\nno Arduino'

def change_left_move(self,*args):
    global left_limit, right_limit
    if config.ard.exist:
        if len(args) == 0:
            step_size = 10
        elif len(args) == 1:
            step_size = args[0]
        elif len(args) == 2:
            step_size = args[1]
            left_limit = args[0]
        elif len(args) == 3:
            step_size = args[2]
            left_limit = args[0]
            right_limit = args[1]
        self.change_abs_move(config.ins.x_step,right_limit)
        self.change_z_calibrate()
        while config.ins.x_step.position > left_limit:
            update()
            if self.exit:
                return
            if config.db.table_enable:
                t_spot,message = config.db.table_check(config.ins.x_step,z_positions)
                self.message += message
                if t_spot is not None:
                    for i,j in enumerate(config.ins.z_steps):
                        #self.change_abs_move(j.index,t_spot[i])
                        self.change_abs_move(j,t_spot[i])
            self.change_z_adjuster_tuner()
            if config.db.table_enable:
                if config.db.table_enable:
                    self.message += config.db.table_update(config.ins.tablestep,config.ins.z_steps,z_positions)
            if config.ins.x_step.enable:
                self.change_rel_move(config.ins.x_step,-step_size)
    else:
        self.message += '\nno Arduino'

def change_down_step(self,*args):
    if config.ard.exist:
        if len(args) == 0:
            for string in config.ins.strings:
                string.z_in.down_step = -4
                string.z_out.down_step = -4
        elif len(args) == 1:
            for string in config.ins.strings:
                string.z_in.down_step = args[0]
                string.z_out.down_step = args[0]
        elif len(args) == 2:
            if args[0] <= config.ins.string_num:
                config.ins.strings[args[0]].z_in.down_step = args[1]
                config.ins.strings[args[0]].z_out.down_step = args[1]
            else:
                self.message += '\nwhich string,step size?'
        else:
            self.message += '\nwhich string,step size?'
    else:
        self.message += '\nno Arduino'

def change_right_left_move(self, *args):
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return
    while not self.exit:
        self.change_right_move(*args)
        if self.exit:
            break
        self.change_left_move(*args)

def change_left_right_move(self, *args):
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return
    while not self.exit:
        self.change_left_move(*args)
        if self.exit:
            break
        self.change_right_move(*args)

def change_up_step(self,*args):
    if config.ard.exist:
        if len(args) == 0:
            for string in config.ins.strings:
                string.z_in.up_step = 3
                string.z_out.up_step = 3
        elif len(args) == 1:
            for string in config.ins.strings:
                string.z_in.up_step = args[0]
                string.z_out.up_step = args[0]
        elif len(args) == 2:
            if args[0] < config.ins.string_num:
                config.ins.strings[args[0]].z_in.up_step = args[1]
                config.ins.strings[args[0]].z_out.up_step = args[1]
            else:
                self.message += '\nwhich string,step size?'
        else:
            self.message += '\nwhich string,step size?'
    else:
        self.message += '\nno Arduino'

def change_min_voice(self,*args):
    if len(args) == 0:
        for string in config.ins.strings:
            string.min_voice = 0
    elif len(args) == 1:
        if args[0] in range(config.sampler.myfreq.voices):
            for string in config.ins.strings:
                string.min_voice = args[0]
        else:
            self.message += '\nout of range,must be '+str(config.sampler.myfreq.voices)+' or less'
    elif len(args) == 2:
        if args[1] in range(config.sampler.myfreq.voices):
            config.ins.strings[args[0]].min_voice = args[1]
        else:
            self.message += '\nout of range,must be '+str(config.sampler.myfreq.voices)+' or less'

def change_min_thresh(self,*args):
    if len(args) == 0:
        for string in config.ins.strings:
            string.min_thresh = 20
    elif len(args) == 1:
        for string in config.ins.strings:
            string.min_thresh = args[0]
    elif len(args) == 2:
        config.ins.strings[args[0]].min_thresh = args[1]

def change_max_voice(self,*args):
    if len(args) == 0:
        for string in config.ins.strings:
            string.max_voice = config.sampler.myfreq.voices-1
    elif len(args) == 1:
        if args[0] in range(config.sampler.myfreq.voices+1):
            for string in config.ins.strings:
                string.max_voice = args[0]
        else:
            self.message += '\nout of range,must be '+str(config.sampler.myfreq.voices)+' or less'
    elif len(args) == 2:
        if args[1] in range(config.sampler.myfreq.voices+1):
            config.ins.strings[args[0]].max_voice = args[1]
        else:
            self.message += '\nout of range,must be '+str(config.sampler.myfreq.voices)+' or less'

def change_max_thresh(self,*args):
    which = int(args[0])
    where = int(args[1])
    config.ins.strings[which].max_thresh = where

def change_enable(self, *args):
    """Correctly sets the .enable flag on a Stepper object."""
    try:
        if len(args) != 2:
            raise ValueError("Requires exactly two arguments: stepper (object or index) and state.")

        stepper_ref, state = args

        # If it's an int, look up the Stepper object
        if isinstance(stepper_ref, int):
            stepper_obj = _find_stepper(stepper_ref)
            if stepper_obj is None:
                raise ValueError(f"No stepper found with index {stepper_ref}")
        elif isinstance(stepper_ref, Stepper):
            stepper_obj = stepper_ref
        else:
            raise TypeError("First argument must be a Stepper object or index (int).")

        # Ensure state is a boolean (or can be treated as one)
        new_state = bool(state)
        stepper_obj.enable = new_state

        self.message += f"\nStepper {stepper_obj.index} enable set to {new_state}"

    except (ValueError, TypeError, IndexError) as e:
        self.message += f"\nUsage: enable <Stepper object or index> <state>. Error: {e}"

def change_quiet(self):
    self.message = ''

def change_follow(self,*args):
    if len(args) == 2:
        string_idx, follow_idx = int(args[0]), int(args[1])
        if string_idx in range(len(config.ins.strings)):
            config.ins.strings[string_idx].follow = follow_idx
            self.message += f'\nString {string_idx} now follows {follow_idx}'
        else:
            self.message += f'\nString index {string_idx} out of range.'
    elif len(args) == 1:
        follow_idx = int(args[0])
        for i, string in enumerate(config.ins.strings):
            string.follow = follow_idx
        self.message += f'\nAll strings now follow {follow_idx}'
    else: # No args, reset all to follow themselves
        for i, string in enumerate(config.ins.strings):
            string.follow = i
        self.message += f'\nAll strings reset to follow themselves.'

def change_trans(self,*args):
    if len(args) == 0:
        # for i in config.sampler.mysines:
        #     i.trans = 1
        for i in config.mm.mymidis:
            i.trans = 1
    elif len(args) == 1:
        # for i in config.sampler.mysines:
        #     i.trans = args[0]
        for i in config.mm.mymidis:
            i.trans = args[0]
    elif len(args) == 2:
        # config.sampler.mysines[args[0]].trans = args[1]
        config.mm.mymidis[args[0]].trans = args[1]

def change_midi_trans(self,*args):
    if len(args) == 0:
        for i in config.mm.mymidis:
            i.trans = 1
    elif len(args) == 1:
        for i in config.mm.mymidis:
            i.trans = args[0]
    elif len(args) == 2:
        config.mm.mymidis[args[0]].trans = args[1]
def change_midi_show_notes(self,*args):
    if len(args) == 0:
        for i in config.mm.mymidis:
            i.show_notes = 1
    elif len(args) == 1:
        for i in config.mm.mymidis:
            i.show_notes = args[0]
    elif len(args) == 2:
        config.mm.mymidis[args[0]].show_notes = args[1]

def change_midi_program(self,*args):
    if len(args) == 0:
        for i in config.mm.mymidis:
            i.program_change(0)
    elif len(args) == 1:
        for i in config.mm.mymidis:
            i.program_change(args[0])
    elif len(args) == 2:
        config.mm.mymidis[args[0]].program_change(args[1])

def change_midi_pan(self,*args):
    if len(args) == 0:
        for i in config.mm.mymidis:
            i.pan(0)
    elif len(args) == 1:
        for i in config.mm.mymidis:
            i.pan(args[0])
    elif len(args) == 2:
        config.mm.mymidis[args[0]].pan(args[1])

def change_min_pos(self,*args):
    try:
        stepper_obj, value = args
        if not isinstance(stepper_obj, Stepper):
            raise TypeError("First argument must be a Stepper object.")

        value = int(value)
        stepper_obj.min_pos = value
        self.message += f"\nStepper {stepper_obj.index} min_pos set to {value}"

    except (ValueError, TypeError, IndexError) as e:
        self.message += f"\nUsage: min_pos <Stepper object> <value>. Error: {e}"

def change_max_pos(self,*args):
    try:
        stepper_obj, value = args
        if not isinstance(stepper_obj, Stepper):
            raise TypeError("First argument must be a Stepper object.")

        value = int(value)
        stepper_obj.max_pos = value
        self.message += f"\nStepper {stepper_obj.index} max_pos set to {value}"

    except (ValueError, TypeError, IndexError) as e:
        self.message += f"\nUsage: max_pos <Stepper object> <value>. Error: {e}"

def change_lap_rest(self,val=4):
    config.ins.lap_rest = val

def change_z_rest(self,val=5):
    if config.ard.exist:
        config.ins.z_rest = val
    else:
        self.message += '\nno Arduino'

def change_tune_rest(self,val=10):
    if config.ard.exist:
        config.ins.tune_rest = val
    else:
        self.message += '\nno Arduino'

def change_x_rest(self,val=10):
    if config.ard.exist:
        config.ins.x_rest = val
    else:
        self.message += '\nno Arduino'

def change_encoder_rest(self,val=1):
    if config.gpio.exist:
        config.gpio.encoder_rest = val
    else:
        self.message += '\nno GPIO'

def change_get_x_encoder_steps(self):
    if config.gpio.exist:
        if config.gpio.encoder:
            encoder_steps = config.gpio.get_encoder_steps()
            self.message += f"\nEncoder Steps: {encoder_steps}"
            return encoder_steps
        else:
            self.message += '\nno encoder'
    else:
        self.message += '\nno GPIO'

def change_get_x_distance(self):
    if config.gpio.exist:
        if config.gpio.distance_sensor:
            distance = config.gpio.get_distance()
            self.message += f"\nX distance: {distance}"
            return distance
        else:
            self.message += '\nno distance sensor'
    else:
        self.message += '\nno GPIO'

def change_adjustment_level(self,val=4):
    if config.ard.exist:
        config.ins.adjustment_level = val
    else:
        self.message += '\nno Arduino'

def change_retry_threshold(self,val=50):
    if config.ard.exist:
        config.ins.retry_threshold = val
    else:
        self.message += '\nno Arduino'

def change_z_variance_threshold(self,val=50):
    if config.ard.exist:
        config.ins.z_variance_threshold = val
    else:
        self.message += '\nno Arduino'

def change_delta_threshold(self,val=50):
    if config.ard.exist:
        config.ins.delta_threshold = val
    else:
        self.message += '\nno Arduino'

def change_input_select(self,val='carriage'):
    if config.ard.exist:
        config.ins.log.input_select = val
    else:
        self.message += '\nno Arduino'

def change_carriage_coil_phase_0(self,val=0):
    if config.ard.exist:
        config.ins.log.carriage_coil_phase_0 = val
    else:
        self.message += '\nno Arduino'

def change_carriage_coil_phase_1(self,val=0):
    if config.ard.exist:
        config.ins.log.carriage_coil_phase_1 = val
    else:
        self.message += '\nno Arduino'

def change_fixed_coil_phase_0(self,val=0):
    if config.ard.exist:
        config.ins.log.fixed_coil_phase_0 = val
    else:
        self.message += '\nno Arduino'

def change_fixed_coil_phase_1(self,val=0):
    if config.ard.exist:
        config.ins.log.fixed_coil_phase_1 = val
    else:
        self.message += '\nno Arduino'

def change_piezo_left_level(self,val=0):
    if config.ard.exist:
        config.ins.log.piezo_left_level = val
    else:
        self.message += '\nno Arduino'

def change_piezo_right_level(self,val=0):
    if config.ard.exist:
        config.ins.log.piezo_right_level = val
    else:
        self.message += '\nno Arduino'

def change_carriage_coil_level(self,val=0):
    if config.ard.exist:
        config.ins.log.carriage_coil_level = val
    else:
        self.message += '\nno Arduino'

def change_fixed_coil_level(self,val=0):
    if config.ard.exist:
        config.ins.log.fixed_coil_level = val
    else:
        self.message += '\nno Arduino'

def change_midi_gain(self,*args):
    if len(args) == 0:
        for i in config.mm.mymidis:
            i.gain = 1
    elif len(args) == 1:
        for i in config.mm.mymidis:
            i.gain = args[0]
    elif len(args) == 2:
        config.mm.mymidis[args[0]].gain = args[1]

def change_save_csv(self,val=1):
    global new_headers
    config.db.save_csv = val
    if val:
        new_headers = True

def change_print_report_enable(self,val=1):
    config.db.print_report_enable = val

def change_z_calibrate_enable(self,val=1):
    config.ins.z_calibrate_enable = val

def change_bump_check_enable(self,val=1):
    if config.ard.exist:
        config.ins.bump_check_enable = val

def change_bump_check_repeat(self,val=10):
    config.ins.bump_check_repeat = val

def change_table_fill(self):
    if path.isfile(CURR_DIR+'/data/'+config.db.data_file):
        self.message += config.db.table_fill(config.ins.x_step,config.ins.z_steps,config.db.data_file)
    else:
        self.message += '\nno data file'

def change_table_enable(self,val=1):
    if config.ard.exist:
        config.db.table_enable = val
    else:
        self.message += '\nno Arduino'

def change_predict_enable(self,val=1):
    config.db.predict_enable = val
    #if config.db.predict_enable:
     #   self.dispatch('predict_enable',0)

def change_table_clear(self):
    config.db.table = [None]*3000

def change_table_print(self):
    self.message += config.db.table_print()

def change_preset_file(self,val='preset_file.txt'):
    config.db.preset_file = val
    self.dispatch('load')

def change_data_file(self,val='data_file.csv'):
    global new_headers
    new_headers = True
    config.db.data_file = val
    if config.db.table_enable:
        if path.isfile(CURR_DIR+'/data/'+config.db.data_file):
            self.message += config.db.table_fill(config.ins.x_step,config.ins.z_steps,config.db.data_file)

def change_predictors(self,val=''):
    config.wiz.load_predictors(tag=val)

def change_save_db(self,val=1):
    config.db.save_db = val

def change_play_enable(self,val=1):
    config.db.play_enable = val
    if val:
        self.dispatch('db_play')

def change_loop_enable(self,val=1):
    config.db.loop_enable = val

def change_real_time_enable(self,val=1):
    config.db.real_time_enable = val

def change_start_point(self,val=0):
    config.db.start_point = min(config.db.df.shape[0],val)
    config.db.steps = min(config.db.df.shape[0]-config.db.start_point,config.db.steps)

def change_steps(self,val=1):
    config.db.steps = min(config.db.df.shape[0]-config.db.start_point,val)


def change_reset(self, *args):
    """Reset either all steppers or a single Stepper instance.

    • No arguments  → reset ALL connected boards
    • (stepper_obj, pos) → reset that stepper to *pos*
    • (idx:int, pos)    → index-based path kept for compatibility; will raise once fully migrated
    """

    # 0 args → board-wide reset
    if len(args) == 0:
        if config.ard.exist:
            msg = config.ard.reset_all()
            if msg:
                self.message += msg
        if config.ard_T.exist:
            msg = config.ard_T.reset_all()
            if msg:
                self.message += msg
        # Sync Python-tracked positions to zero for all steppers
        for s in config.ins.all_steps:
            if s is not None:
                s.position = 0
        if config.gpio and config.gpio.encoder:
            config.gpio.set_encoder_steps(0)
        return

    if len(args) == 2:
        target, where = args

        # New-style: first arg is a Stepper object
        from instrument import Stepper as _Stepper
        if isinstance(target, _Stepper):
            stepper = target
        else:
            # Legacy int path (will be removed soon)
            stepper = _find_stepper(int(target))

        if not stepper or not stepper.arduino:
            self.message += f"\nStepper reset failed – bad stepper {target}"
            return

        where_int = int(where)
        msg = stepper.arduino.set_stepper(stepper.local_index, where_int)
        if msg:
            self.message += msg
        stepper.position = where_int
        from instrument import Stepper as _Stepper
        if isinstance(stepper, X_Stepper):
            if config.gpio and config.gpio.encoder:
                config.gpio.set_encoder_steps(where_int*2)
        return

    self.message += '\nUsage: reset [stepper] [where]'

def change_abs_reposition(self,*args):
    """Queue absolute moves for the selected Z-steppers.

    No GUI or state shortcutting is performed – the *only* source of
    positional truth is the Arduino report that arrives through
    `HardwareWorker.positionsChanged`.  This keeps the system strictly
    closed-loop.

    • No args        → move every enabled Z-stepper to its `max_pos`.
    • One arg (pos)  → move every enabled Z-stepper to <pos>.
    • Two args (idx,pos) → move stepper <idx> to <pos>.
    """

    if not config.ard.exist:
        self.message += '\nno Arduino'
        return

    # Helper to queue a single absolute move safely.
    def _queue_move(step_index: int, target: int):
        """Queue the move – no local cache tweaks, hardware is the authority."""
        stepper = _find_stepper(step_index)
        if stepper and stepper.enable:
            hw_amove(stepper, target)

    # No arguments – use each stepper's max_pos
    if len(args) == 0:
        for z_step in config.ins.z_steps:
            self.change_abs_move(z_step, z_step.max_pos)
            # _queue_move(z_step.index, z_step.max_pos)
            # waiter(config.ins.z_rest)

    # One argument – common target for every enabled Z-stepper
    elif len(args) == 1:
        target = int(args[0])
        for z_step in config.ins.z_steps:
            self.change_abs_move(z_step, target)
            # _queue_move(z_step.index, target)
            # waiter(config.ins.z_rest)

    # Two arguments – explicit (index, target)
    elif len(args) == 2:
        step_index = int(args[0])
```

```
            target = int(args[1])
            self.change_abs_move(step_index, target)
            # _queue_move(step_index, target)

        else:
            self.message += '\nInvalid abs_reposition arguments'

    def change_rel_reposition(self,*args):
        """Re-implement relative repositioning without GUI-blocking loops.

        0 args → move every enabled Z-stepper to its max_pos in **one** step.
        1 arg  → add <delta> to every enabled Z-stepper.
        2 args → add <delta> to the specified stepper only.

        All hardware I/O is delegated to the HardwareWorker; once the worker
        finishes the real absolute positions will be broadcast through the
        regular positionsChanged signal, keeping the GUI in sync with the
        *actual* Arduino state.
        """

        if not config.ard.exist:
            self.message += '\nno Arduino'
            return

        def _queue_rmove(step_index: int, delta: int):
            """Helper that validates enable state and schedules the move."""
            stepper = _find_stepper(step_index)
            if stepper and stepper.enable and delta != 0:
                hw_rmove(stepper, delta)

        # 0-arg form → go to max_pos for every enabled Z-stepper.
        if len(args) == 0:
            for z_step in config.ins.z_steps:
                if z_step.enable and z_step.local_index is not None :
                    current_position = z_step.position
                    delta = int(z_step.max_pos) - current_position
                    self.change_rel_move(z_step, delta)
                    # _queue_rmove(z_step.index, delta)
                    # waiter(config.ins.z_rest)

        # 1-arg form → same delta for all enabled Z-steppers.
        elif len(args) == 1:
            # Provided delta is raw counts; keep as-is
            delta = int(args[0]*z_step_size)
            for z_step in config.ins.z_steps:
                if z_step.enable:
                    # z_step.position+= delta
                    self.change_rel_move(z_step, args[0])
                    # _queue_rmove(z_step.index, args[0])
                    # waiter(config.ins.z_rest)

        # 2-arg form → explicit index and delta.
        elif len(args) == 2:
            step_index = int(args[0])
            delta = int(args[1]*z_step_size)
            stepper_obj = _find_stepper(step_index)
            if stepper_obj and stepper_obj.enable:
                self.change_rel_move(stepper_obj, args[1])
                # stepper_obj.position+=delta
                # _queue_rmove(step_index, args[1])
        else:
            self.message += '\nInvalid rel_reposition arguments'



    def change_initialize(self):
        self.dispatch('abs_reposition',20)
        self.dispatch('recalibrate')
        self.dispatch('bump_check')

        # self.dispatch('load')

    def change_do_predict(self):
        message =  ''
        data = pd.DataFrame(np.array([data_row]), columns=makeheaders())
        print('data shape:',data.shape)
        predictions = config.wiz.make_predictions(data)
        for i,prediction in enumerate(predictions):
            # print(prediction.target)
            message += '\nThe prediction for {stepper} is {prediction} with {conf:.2f} confidence \n'.format(stepper=list(prediction.target)
            #self.change_abs_move(config.ins.z_steps[i].index,int(list(prediction.prediction)[0]))
            self.change_abs_move(config.ins.z_steps[i],int(list(prediction.prediction)[0]))
        self.message += message

    def change_x_adjust(self):
        global retry
        if not config.ard.exist:
            self.message += '\nno Arduino'
            return
        adjusted = [False]*config.ins.string_num
        while not all(adjusted):
            if self.exit:
```

```
            return
        self.message += f'\nretrying x_adjustment: {retry}'
        if retry > config.ins.retry_threshold:
            retry = 0
            config.ins.level = 0
            self.change_recalibrate()
        for i,string in enumerate(config.ins.strings):
            if self.exit:
                return
            # self.message += '\nstring_'+str(i)+', amp_sum: '+str(current[i])
            # self.message += '\nstring_'+str(i)+', voice_count: '+str(v_num[i])
            if not string.min_thresh <= current[i] <= string.max_thresh or not string.min_voice <= v_num[i] <= string.max_voice:
                if current[i] > string.max_thresh or v_num[i] > string.max_voice: #too loud
                    if delta[i] < -config.ins.delta_threshold:
                        self.message += '\namplitude falling\nno x_adjustment on string: '+str(i)
                    else:
                        self.message += '\n'+str(i)+' too loud'
                        if config.ins.x_step.position >= 1300: #past halfway
                            self.change_rel_move(config.ins.x_step,10) #move closer to end
                            # self.dispatch('rel_move',config.ins.x_step.index,10) #move closer to end
                        else:
                            self.change_rel_move(config.ins.x_step,-10) #move closer to beginning
                            # self.dispatch('rel_move',config.ins.x_step.index,-10) #move closer to beginning
                elif current[i] < string.min_thresh or v_num[i] < string.min_voice: #too quiet
                    if delta[i] > config.ins.delta_threshold:
                        self.message += '\namplitude rising\nno x_adjustment on string: '+str(i)
                    else:
                        self.message += '\n'+str(i)+' too quiet'
                        if config.ins.x_step.position >= 1300: #past halfway
                            self.change_rel_move(config.ins.x_step,-10) #move closer to middle
                        else:
                            self.change_rel_move(config.ins.x_step,10) #move closer to middle
                adjusted[i] = False
                config.ins.level = 0
            else:
                # self.message += '\nstring: '+str(i)+' x_adjusted'
                adjusted[i] = True
        # self.message += f'\nadjusted: {[int(b) for b in adjusted]}'
        self.message += f'\nadjusted: {adjusted}'
        update()
        if not all(adjusted):
            retry += 1
        else:
            self.message += '\nall x_adjusted\n'
            return


def determine_index(self, string, positions, closer=True):
  if not string.z_in.enable:
      return string.z_out
  if not string.z_out.enable:
      return string.z_in
  if closer:
      return string.z_in if string.z_in.position <= string.z_out.position else string.z_out
  return string.z_in if string.z_in.position >= string.z_out.position else string.z_out


def change_z_adjust(self):
    global retry
    if 'retry' not in globals():
        retry = 0
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return

    adjusted = [False] * config.ins.string_num
    while not all(adjusted):
        if self.exit:
            return

        self.message += f'\nretrying z_adjustment: {retry}'
        if retry > config.ins.retry_threshold:
            retry = 0
            config.ins.level = 0
            self.change_recalibrate()

        # ■■■■■■■■■ gather sensor state ■■■■■■■■■■
        # Only consider **enabled** Z-steppers and use
        # sum(|pos - mean|) as the variance metric requested by the user.
        enabled_pos = [s.position for s in config.ins.z_steps if s and s.enable]
        if enabled_pos:
            mean_pos = np.mean(enabled_pos)
            z_variance = float(np.sum(np.abs(np.subtract(enabled_pos, mean_pos))))
        else:
            # If no stepper is enabled, treat variance as 0 to avoid spurious recalibrations.
            z_variance = 0.0

        self.message += f'\nz_variance: {int(z_variance)}'
        if z_variance > config.ins.z_variance_threshold:
            retry = 0
            config.ins.level = 0
            self.change_recalibrate()
        # ■■■■■■■■■■ per-string adjustment ■■■■■■■■■■■
        for i, string in enumerate(config.ins.strings):
            if self.exit:
                return

            # skip disabled strings
            if not (string.z_in.enable or string.z_out.enable):
                self.message += f'\nSkipping disabled string: {i}'
                adjusted[i] = True
                continue

            too_close = current[i] > string.max_thresh or v_num[i] > string.max_voice
            too_far  = current[i] < string.min_thresh or v_num[i] < string.min_voice
            needs_adj = too_close or too_far

            if needs_adj:
                if too_close:
                    if delta_p[i] <= 1 - (config.ins.delta_threshold / 100):
                        self.message += f'\namplitude falling\nno z_adjustment on string: {i}'
                    else:
                        stepper = self.determine_index(string, z_positions)
                        self.message += f'\n{stepper.index} too close'
                        self.change_rel_move(stepper, stepper.up_step)
                else:  # too_far
                    if delta_p[i] >= 1 + (config.ins.delta_threshold / 100):
                        self.message += f'\namplitude rising\nno z_adjustment on string: {i}'
                    else:
                        stepper = self.determine_index(string, z_positions, closer=False)
                        self.message += f'\n{stepper.index} too far'
                        self.change_rel_move(stepper, stepper.down_step)

                adjusted[i] = False
                config.ins.level = 0
            else:
                adjusted[i] = True

        # ■■■■■■■■■■ end of pass ■■■■■■■■■■■
        self.message += f'\nadjusted: {adjusted}'
        update()  # keep GUI responsive

        if not all(adjusted):
            retry += 1
        else:
            self.message += '\nall z_adjusted\n'
            return

def change_root(self,*args):
    if config.ard.exist:
        if len(args) == 0:
            for string in config.ins.strings:
                string.root = 55
        elif len(args) == 1:
            for string in config.ins.strings:
                string.root = args[0]
        elif len(args) == 2:
            if args[0] in range(config.ins.string_num):
                config.ins.strings[args[0]].root = args[1]
            else:
                self.message += '\nout of range'
    else:
        self.message += '\nno Arduino'

def change_tolerance(self,*args):
    if config.ard.exist:
        if len(args) == 0:
            for string in config.ins.strings:
                string.tolerance = .5
        elif len(args) == 1:
            for string in config.ins.strings:
                string.tolerance = args[0]
        elif len(args) == 2:
            if args[0] in range(config.ins.string_num):
                config.ins.strings[args[0]].tolerance = args[1]
            else:
                self.message += '\nout of range'
    else:
        self.message += '\nno Arduino'

def change_fun(self,*args):
    if config.ard.exist:
        if len(args) == 0:
            for string in config.ins.strings:
                string.fun = 1
        elif len(args) == 1:
            for string in config.ins.strings:
                string.fun = args[0]
        elif len(args) == 2:
            if args[0] in range(config.ins.string_num):
                config.ins.strings[args[0]].fun = args[1]
            else:
                self.message += '\nout of range'
    else:
        self.message += '\nno Arduino'

def change_turns(self,*args):
    if config.ard_T.exist:
        if len(args) == 0:
            for i in config.ins.tuner_steps:
                i.turns = 8000
        if len(args) == 1:
            for i in config.ins.tuner_steps:
                i.turns = args[0]
        elif len(args) == 2:
            if args[0] in range(config.ins.string_num):
                config.ins.tuner_steps[args[0]].turns = args[1]
            else:
                self.message += '\nstring_num out of range'
    else:
        self.message += '\nno Tuner Arduino'

def change_tune(self):
    if not config.ard_T.exist and not config.ard.exist:
        self.message += '\nno Arduino'
        return

    enabled_tuners = [s.tuner for s in config.ins.strings if s.tuner and s.tuner.enable]
    if not enabled_tuners:
        self.message += '\nNo tuners enabled.'
        return

    self.message += f'\nTuning {len(enabled_tuners)} steppers...'
    for i, string in enumerate(config.ins.strings):
        if self.exit:
            return

        if string.tuner.enable:
            make_change.change_bump_check()
            update()
            tuned, message = string.tune(peaks)
            self.message += message
            if not tuned:
                self.message += f'\nString {i} still needs tuning...'
            else:
                self.message += f'\nString {i} is tuned.'

    # After one pass, let the main loop continue
    self.message += '\nTuning cycle complete.'

def change_string_adjust(self,*args):
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return
    if self.exit:
        return
    string = config.ins.strings[args[0]]
    make_change.change_bump_check()
    update()
    if string.z_in.enable or string.z_out.enable:
        if not string.min_thresh <= current[args[0]] <= string.max_thresh or not string.min_voice <= v_num[args[0]] <= string.max_voice:
            if current[args[0]] > string.max_thresh or v_num[args[0]] > string.max_voice: #too close
                if delta_p[args[0]] <= 1 - (config.ins.delta_threshold/100) : #testing, may switch to pct.chg and repurpose delta_thresh
                    self.message += '\namplitude falling\nno z_adjustment on string: '+str(args[0])
                else:
                    if not string.z_in.enable:
                        stepper_to_move = string.z_out
                    elif not string.z_out.enable:
                        stepper_to_move = string.z_in
                    elif string.z_in.enable and string.z_out.enable:
                        if string.z_in.position <= string.z_out.position:
                            stepper_to_move = string.z_in
                        else:
                            stepper_to_move = string.z_out
                    self.message += '\n'+str(stepper_to_move.index)+' too close'
                    if stepper_to_move is not None:
                        self.change_rel_move(stepper_to_move, getattr(stepper_to_move, 'up_step', 0))
                    else:
                        self.message += f'\nWARN: Stepper not found – no move'
            elif current[args[0]] < string.min_thresh or v_num[args[0]] < string.min_voice: #too far
                if delta_p[args[0]] >= 1 + (config.ins.delta_threshold/100): #testing, may switch to pct.chg and repurpose delta_thresho
                    self.message += '\namplitude rising\nno z_adjustment on string: '+str(args[0])
                else:
                    if not string.z_in.enable:
                        stepper_to_move = string.z_out
                    elif not string.z_out.enable:
                        stepper_to_move = string.z_in
                    elif string.z_in.enable and string.z_out.enable:
                        if string.z_in.position >= string.z_out.position:
                            stepper_to_move = string.z_in
                        else:
                            stepper_to_move = string.z_out
                    self.message += '\n'+str(stepper_to_move.index)+' too far'
                    if stepper_to_move is not None:
                        self.change_rel_move(stepper_to_move, getattr(stepper_to_move, 'down_step', 0))
                    else:
                        self.message += f'\nWARN: Stepper not found – no move'
        else:
            self.message += '\nstring_'+str(args[0])+': z_adjusted'

    else:
        self.message += '\nstring: '+str(args[0])+' z_adjustment disabled'


def change_string_tune(self,*args):
    if not config.ard_T.exist:
        self.message += '\nno Tuner Arduino'
        return
    make_change.change_bump_check()
    update()
    tuned,message = config.ins.strings[args[0]].tune(peaks)
    self.message += message
    if not tuned:
        self.message += '\nstring_'+str(args[0])+' still needs tuning...'
    else:
        self.message += '\nstring_'+str(args[0])+' tuned'
        return

def change_z_adjuster_tuner(self):
  global retry
  retry = 0
  config.ins.level = 0
  if not config.ard.exist:
      self.message += '\nno Arduino'
      return

  # Filter for only the enabled steppers before starting the loop
  enabled_z_steps = [s for s in config.ins.z_steps if s and s.enable]
  if not enabled_z_steps:
      self.message += '\nAll Z-steppers are disabled; skipping adjustment.'
      return

  # self.dispatch('recalibrate')
  if config.db.predict_enable:
      self.dispatch('do_predict')
  while config.ins.level < config.ins.adjustment_level:
      if self.exit:
          return
      self.dispatch('z_adjust')
      if config.ard_T.exist:
          self.dispatch('tune')
      config.ins.level += 1
      self.message += f'\nlevel: {config.ins.level}'

def change_x_adjuster_tuner(self):
    global retry
    retry = 0
    config.ins.level = 0
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return

    if not config.ins.x_step.enable:
        self.message += '\nX-stepper is disabled; skipping adjustment.'
        return

    # self.dispatch('recalibrate')
    if config.db.predict_enable:
        self.dispatch('do_predict')
    while config.ins.level < config.ins.adjustment_level:
        if self.exit:
            return
        self.dispatch('x_adjust')
        # self.message += '\nx adjusted'
        if config.ard_T.exist:
            self.dispatch('tune')
            # self.message += '\nall tuned'
        config.ins.level += 1
        self.message += f'\nlevel: {config.ins.level}'

def change_wait(self,*args):
    if len(args) == 0:
        waiter(60)
    if len(args) == 1:
        waiter(args[0])
    else:
        self.message += '\nhow long?'

def change_sequence(self,*args):
    self.message += '\nseq_args: '+str(args)
    if config.ard.exist:
        for arg in args:
            if self.exit:
                return
            self.message += '\nseq_arg: '+str(arg)
            self.dispatch('root',float(arg))
            self.dispatch('z_adjuster_tuner')
    else:
        self.message += '\nno Arduino'

def change_sequence_pairs(self,*args):
    self.message += '\nseq_args: '+str(args)
    if config.ard.exist:
        if (len(args) % 2) == 0:
            for i in range(0,len(args),2):
                if self.exit:
                    return
                self.message += '\nseq_arg_0: '+str(args[i])+', seq_arg_1: '+str(args[i+1])
                self.dispatch('root',0,float(args[i]))
                self.dispatch('root',1,float(args[i+1]))
                self.dispatch('z_adjuster_tuner')
        else:
            self.message += '\nfrequencies must be in pairs'
    else:
        self.message += '\nno Arduino'

def change_range_tune(self,*args):
    #self.message += '\nargs: '+str(args))
    if len(args) != 3:
        self.message += '\nenter integer start_freq, end_freq, step_size (negative for downtune)'
        return
    for arg in args:
        if not isinstance(arg, int):
            self.message += '\nintegers only:'+str(arg)
            return
    for i in range(args[0],args[1],args[2]):
        if self.exit:
            return
        self.message += '\ntuning to : '+str(i)+'Hz'
        self.dispatch('root',i)
        self.dispatch('z_adjuster_tuner')

@command_logger
def change_z_seeker(self):
    global retry
    retry = 0
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return
    while not self.exit:
        update()
        # move carriage a random amount each lap
        if config.ins.x_step.enable:
            self.change_x_rand_move()

        self.change_z_adjuster_tuner()
        if config.db.table_enable:
            self.dispatch('write')
        waiter(config.ins.lap_rest)

@command_logger
def change_x_seeker(self):
    global retry
    retry = 0
    if not config.ard.exist:
        self.message += '\nno Arduino'
        return
    while not self.exit:
        update()

        if config.ins.x_step.enable:
            self.change_x_rand_move()

        self.change_x_adjuster_tuner()
        if config.db.table_enable:
            self.dispatch('write')
        waiter(config.ins.lap_rest)

def change_bonker(self,*args):
    #until complete repeat or exit
    if len(args) == 0: #repeat once, all z_steps
        repeat = 1
        height = 50
        z_range = range(len(config.ins.z_steps))
    elif len(args) == 1: #set repeat, all z_steps
        repeat = args[0]
        height = 20
        z_range = range(len(config.ins.z_steps))
    elif len(args) == 2: #set repeat, set height, all z_steps
        repeat = args[0]
        height = args[1]
        z_range = range(len(config.ins.z_steps))
    elif len(args) == 3: #set repeat, set height, set z_step
        repeat = args[0]
        height = args[1]
        z_range = range(args[2],args[2]+1)
    for i in range(repeat):
        if self.exit:
            return
        for j in z_range:
            self.change_abs_move(config.ins.z_steps[j],0)
        waiter(config.ins.lap_rest)
        for j in z_range:
            self.change_abs_move(config.ins.z_steps[j],height)
        waiter(config.ins.lap_rest)
    return

def change_x_calibrate(self):
    # This function now uses robust stepper routing via x_step.arduino.
    stepper = config.ins.x_step
    if not stepper or not stepper.enable:
        self.message += '\nx is not enabled'
        return
    if not stepper.arduino or not stepper.arduino.exist:
        self.message += '\nNo Arduino connected for X stepper'
        return
    cur_pos = stepper.position
    self.change_abs_move(stepper, 0)
    while not config.gpio.x_home_check():
        stepper.arduino.set_stepper(stepper.local_index,500)
        self.change_rel_move(stepper.index, -500)
        #update()  # Allow GUI to process events
    self.change_reset(stepper.index, 0)
    self.message += '\nx is home!'
    self.change_abs_move(stepper, cur_pos)

def change_z_calibrate(self, *args):
    # Respect the user setting – skip when z_calibrate_enable is false.
    if not config.ins.z_calibrate_enable:
        return
    # This function is critical and now uses the robust stepper routing.
    if not (config.gpio.exist and any(s.arduino for s in config.ins.z_steps)):
        self.message += '\nZ-Calibration requires GPIO and at least one configured Z-stepper.'
        return

    # Temporarily disable bump-check, but restore the user's setting afterward.
    # Also force z_step_size = 1 for consistent raw-step calibration, restoring afterwards.
    global z_step_size
    _saved_z_step_size = z_step_size
    z_step_size = 1.0
    original_bump_check_state = config.ins.bump_check_enable
    config.ins.bump_check_enable = 0

    try:
        def touch_stepper(z_step: Z_Stepper):
            if not z_step.arduino:
                self.message += f'\nSkipping Z-Stepper {z_step.index} (no Arduino configured).'
                return False

            # Use the 1-based GPIO index, which corresponds to the physical wiring.
            try:
                from instrument import host_config as _hc
                base_z_index = _hc.get('Z_FIRST_INDEX', config.ins.z_steps[0].index if config.ins.z_steps else 1)
            except Exception:
                base_z_index = config.ins.z_steps[0].index if config.ins.z_steps else 1
            gpio_index = z_step.index - base_z_index

            # Reset to max position to ensure it's clear of the sensor
            # Hardware must know it's clear of the sensor, but we *must not* mutate
            # the Python-side positional cache here – that gets finalised at 0 once
            # calibration is done.  We therefore talk to the Arduino directly and
            # deliberately skip any local position bookkeeping.
            z_step.arduino.set_stepper(z_step.local_index, z_step.max_pos)
            # Fast direct calibration loop – bypass GUI & waiter for speed
            is_touching = config.gpio.press_check(gpio_index)
            # Track position locally to detect bottom-out without updating global state
            pos_local = z_step.max_pos
            steps_moved = 0  # accumulate absolute steps commanded
            while not is_touching:
                if self.exit:
                    return False, steps_moved
                # Detect bottom-out based on local tracker
                if pos_local <= z_step.min_pos:
                    self.message += f'\nStepper {z_step.index} bottomed out during calibration.'
                    gui.gui_message()
                    z_step.arduino.set_stepper(z_step.local_index, 0)
                    z_step.position = 0
                    return False, steps_moved
                # Perform raw relative move directly on the Arduino
                z_step.arduino.rmove(z_step.local_index, z_step.down_step)
                waiter(config.ins.z_rest)
                pos_local += z_step.down_step
                steps_moved += abs(z_step.down_step)
                # Immediately re-check the sensor after brief wait for stepper to move
                is_touching = config.gpio.press_check(gpio_index)

            # Sensor contacted successfully; reset both hardware and software to 0
            z_step.arduino.set_stepper(z_step.local_index, 0)
            z_step.position = 0
            self.message += f'\nZ_Stepper {z_step.index} touched sensor.'
            gui.gui_message()
            return True, steps_moved

        if args:
            # Calibrate a single, specified Z-stepper
            stepper_idx = int(args[0])
            stepper_to_calibrate = _find_stepper(stepper_idx)
            if isinstance(stepper_to_calibrate, Z_Stepper):
                pre_pos = stepper_to_calibrate.position
                ok, moved = touch_stepper(stepper_to_calibrate)
                self.message += f'\nZ Calibration results: {[bool(ok)]}'
                self.message += f'\nZ Errors: {[pre_pos - moved]}'
            else:
                self.message += f"\nError: Stepper {stepper_idx} is not a Z-Stepper."
        else:
            # Calibrate all enabled Z-steppers
            touched_results = []
            pre_positions = []
            for z_step in config.ins.z_steps:
                if z_step and z_step.enable:
                    pre_positions.append(z_step.position)
                    touched_results.append(touch_stepper(z_step))
            calibrated_flags = [int(ok) for (ok, _moved) in touched_results]
            errors = [pre - moved for ((ok, moved), pre) in zip(touched_results, pre_positions)]
            deltas = [moved for (ok, moved) in touched_results]
            self.message += f'\nCalibration results:\n {calibrated_flags}'
            self.message += f'\nError:\n {errors}'

    finally:
        # Restore original runtime settings
        config.ins.bump_check_enable = original_bump_check_state
        z_step_size = _saved_z_step_size

def change_recalibrate(self):
    self.change_x_calibrate()
    if config.ins.z_calibrate_enable:
        self.change_z_calibrate()

def change_z_end_calibrate(self):
    if self.exit:
        return
    if config.ins.x_step.enable:
        x_pos = max(0,min(config.ins.x_step.right_limit,config.ins.x_step.position))
        if x_pos <= 1300:
            self.change_x_home()
        else:
            self.change_x_away()
    self.change_z_calibrate()
    if config.ins.x_step.enable:
        self.change_abs_move(config.ins.x_step,x_pos) #*step_scale)

def change_bump_check(self, *args):
    # When bump-check is disabled we just exit early without logging –
    # avoids cluttering the message pane every cycle.
    if not config.ins.bump_check_enable:
        return

    if not (config.ard.exist and config.gpio.exist):
        self.message += '\nno Arduino or no GPIO'
        return

    # Probe status for ALL Z-steppers; recovery acts only on enabled ones.
    all_z_steps = [s for s in config.ins.z_steps if s]
    enabled_z_steps = [s for s in all_z_steps if s.enable]
    if not all_z_steps:
        return

    # Determine mapping from Z stepper to its GPIO sensor index using the same
    # convention as z_calibrate: gpio_index = z_step.index - Z_FIRST_INDEX.
    try:
        from instrument import host_config as _hc
        base_z_index = _hc.get('Z_FIRST_INDEX', config.ins.z_steps[0].index if config.ins.z_steps else 1)
    except Exception:
        base_z_index = config.ins.z_steps[0].index if config.ins.z_steps else 1

    # Build the list of steppers to probe: either all, or one specified (1-based)
    steppers_to_check = all_z_steps if not args else [all_z_steps[args[0] - 1]]

    # Initialize the bump_statuses array to track the status for each stepper
    bump_statuses = [False] * len(steppers_to_check)

    for _ in range(config.ins.bump_check_repeat):
        if self.exit:
            return

        # Call press_check with the correct GPIO index for each stepper
        for idx, z_step in enumerate(steppers_to_check):
            gpio_index = z_step.index - base_z_index
            is_bumping = config.gpio.press_check(button_index=gpio_index)
            bump_statuses[idx] = is_bumping
            # self.message += f'\nDEBUG: press_check(button_index={gpio_index}) -> {is_bumping}'
        # Compact display with '-' for disabled, '1' for bumping, '0' otherwise
        bit_chars = []
        bumping_indices = []
        for i, z_step in enumerate(steppers_to_check):
            if not z_step.enable:
                bit_chars.append('-')
            else:
                if bump_statuses[i]:
                    bit_chars.append('1')
                    bumping_indices.append(z_step.index)
                else:
                    bit_chars.append('0')
        self.message += f"\nbump statuses: [{', '.join(bit_chars)}]"
        if bumping_indices:
            self.message += f'\nbumping: {bumping_indices}'

        # Track which steppers we issue rel_moves to in this pass
        moves_issued = []

        for idx, bump in enumerate(bump_statuses):
            z_step = steppers_to_check[idx]
            stepper_index = z_step.index

            if not bump:
                # If not bumping, it's successfully recovered. Reset the counter.
                self.bump_retry_counts[stepper_index] = 0
                continue

            # If we get here, it's bumping.
            # Increment the counter for this stepper.
            self.bump_retry_counts[stepper_index] = self.bump_retry_counts.get(stepper_index, 0) + 1
            # self.message += f'\nDEBUG: Bump detected on stepper {stepper_index}. Total attempts: {self.bump_retry_counts[stepper_index

            # Define a limit for retries. After 5 consecutive failed checks, disable it.
            BUMP_RETRY_LIMIT = 5

            if self.bump_retry_counts[stepper_index] > BUMP_RETRY_LIMIT:
                self.message += f'\nCRITICAL: Stepper {stepper_index} is stuck after {BUMP_RETRY_LIMIT} attempts. DISABLING.'
                z_step.enable = False
                self.bump_retry_counts[stepper_index] = 0 # Reset counter
                continue

            # If we are under the limit, perform the existing recovery logic.
            current_pos = z_step.position
            max_pos = z_step.max_pos
            # self.message += f'\nDEBUG: BUMP DETECTED on stepper {stepper_index}. Position={current_pos}, Max={max_pos}. Attempting rec

            if z_step.enable:
                if current_pos >= max_pos:
                    self.change_abs_move(z_step.index, z_step.max_pos)
                    z_step.enable = False
                    self.message += f'\nCRITICAL: DISABLING stepper {stepper_index}. Reason: Bumping at max position.'
                    self.bump_retry_counts[stepper_index] = 0 # Also reset here
                else:
                    # On first detection, zero; on subsequent retries, accumulate upward moves
                    if self.bump_retry_counts[stepper_index] == 1:
                        self.message += f'\nStepper {stepper_index} is bumping.  Resetting to 0 and moving up_step.'
                        self.change_reset(z_step, 0)
                    else:
                        self.message += f'\nStepper {stepper_index} still bumping; moving up_step again.'
                    self.change_rel_move(z_step, z_step.up_step)  # use working code, not hw crap
                    moves_issued.append(stepper_index)

        if moves_issued:
            self.message += f"\nbump_check rel_move indices: {moves_issued}"

def change_home(self): #return x_step and z_steps to 0
    if config.ard.exist:
        self.change_x_home()
        self.change_z_calibrate()
    else:
        self.message += '\nno Arduino'

#@command_logger
def change_x_home(self): #return x_step and z_steps to 0
    if config.ard.exist:
        stepper = config.ins.x_step
        if not stepper or not stepper.enable:
            self.message += '\nx is not enabled'
            return
        if not stepper.arduino or not stepper.arduino.exist:
            self.message += '\nNo Arduino connected for X stepper'
            return
        self.change_abs_move(stepper,0)

        while not config.gpio.x_home_check():
            stepper.arduino.set_stepper(stepper.local_index,500)
            # Move hardware and update Python-tracked position via wrapper
            stepper.arduino.rmove(stepper.local_index,-500)
        self.change_reset(stepper,0)
        self.message += '\nx is home!'
    else:
        self.message += '\nno Arduino'
```

```python
#@command_logger
def change_x_away(self): #return x_step and z_steps to max
    if config.ard.exist:
        stepper = config.ins.x_step
        if not stepper or not stepper.enable:
            self.message += '\nx is not enabled'
            return
        if not stepper.arduino or not stepper.arduino.exist:
            self.message += '\nNo Arduino connected for X stepper'
            return
        self.change_abs_move(stepper,stepper.max_pos)
        # Directional detection: for single-pin systems the same pin is used
        # for both ends. We must observe a release while moving right, then
        # a re-press which indicates the far (away) contact.
        single_pin = (
            hasattr(config.gpio, 'x_home_line') and hasattr(config.gpio, 'x_away_line')
            and config.gpio.x_home_line is not None and config.gpio.x_away_line is not None
            and config.gpio.x_home_line == config.gpio.x_away_line
        )

        if single_pin:
            seen_release = False
            while True:
                stepper.arduino.set_stepper(stepper.local_index,stepper.max_pos - 500)
                stepper.arduino.rmove(stepper.local_index,500)
                pressed = config.gpio.x_away_check()
                if not seen_release:
                    if not pressed:
                        seen_release = True
                else:
                    if pressed:
                        break
                if hasattr(stepper, 'position') and hasattr(stepper, 'max_pos') and stepper.position >= stepper.max_pos:
                    break
        else:
            while not config.gpio.x_away_check():
                stepper.arduino.set_stepper(stepper.local_index,stepper.max_pos - 500)
                # Move hardware and update Python-tracked position via wrapper
                stepper.arduino.rmove(stepper.local_index,500)
                if hasattr(stepper, 'position') and hasattr(stepper, 'max_pos') and stepper.position >= stepper.max_pos:
                    break
        self.change_reset(stepper,stepper.max_pos)
        self.message += '\nx is away!'
    else:
        self.message += '\nno Arduino'

def change_x_home_check(self):
    self.message += '\nx_home is: '+str(config.gpio.x_home_check())

def change_x_away_check(self):
    self.message += '\nx_away is: '+str(config.gpio.x_away_check())

def change_shut_down(self):
    """Legacy shutdown command - delegates to change_quit for clean exit."""
    self.message += '\nInitiating shutdown...'
    self.change_quit()

def change_auto_write(self,val=1):
    config.db.auto_write = val

def change_write(self):
    with open(uname.node+config.db.preset_file,'wb') as preset_file:
        pickle.dump(self.set_storage,preset_file)

def change_load(self):
    if path.isfile(uname.node+config.db.preset_file):
        with open (uname.node+config.db.preset_file,'rb') as preset_file:
            self.set_storage = pickle.load(preset_file)
    else:
        self.message += '\nno saved presets!'
    self.get_storlist()
    gui.form.recall_preset.clear()
    for i in self.storlist:
        gui.form.recall_preset.addItem(QtWidgets.QListWidgetItem(str(i)))

def change_save(self, val=0):
    self.message += '\npreset is: ' + str(val)
    if val in range(self.stor_size):
        self.set_storage[val] = [
            deepcopy(myfreq_copy),
            copy(config.ins),
            None,
            deepcopy(config.mm.mymidis),
            deepcopy(config.ins.x_step),
        ]

        # Add each string individually
        for string in config.ins.strings:
            self.set_storage[val].append(deepcopy(string))

        self.set_storage[val].append(z_positions)
        self.set_storage[val].append(tuner_positions)
```

```
            self.current_set = val
            items = gui.form.recall_preset.findItems(str(val), QtCore.Qt.MatchExactly)
            if len(items) == 0:
                gui.form.recall_preset.addItem(QtWidgets.QListWidgetItem(str(val)))
            if config.db.auto_write:
                self.dispatch('write')
        else:
            self.message += '\nout of range, must be ' + str(self.stor_size) + ' or less'

    def change_recall(self, val=0):
        """
        Safely recall a preset without overwriting the live `config.ins` object.
        This function now meticulously updates the attributes of the existing
        `config.ins` object from the loaded preset data, preserving the
        integrity of the live application state.
        """
        try:
            if val not in range(self.stor_size) or self.set_storage[val] is None:
                self.message += f'\npreset: {val} is empty'
                return

            stored_data = self.set_storage[val]

            # Safely update attributes of the live config.ins object
            loaded_ins = stored_data[1]
            for key, value in loaded_ins.__dict__.items():
                if hasattr(config.ins, key):
                    setattr(config.ins, key, value)

            # Safely update MIDI settings
            loaded_midis = stored_data[3]
            for i, midi_preset in enumerate(loaded_midis):
                if i < len(config.mm.mymidis):
                    config.mm.mymidis[i].program = midi_preset.program
                    config.mm.mymidis[i].trans = midi_preset.trans
                    config.mm.mymidis[i].gain = midi_preset.gain
                    config.mm.mymidis[i].show_notes = midi_preset.show_notes

            # Safely update X-Stepper
            loaded_x_step = stored_data[4]
            for key, value in loaded_x_step.__dict__.items():
                if hasattr(config.ins.x_step, key):
                    setattr(config.ins.x_step, key, value)

            # Safely update Strings
            string_num = config.ins.string_num
            for i in range(string_num):
                loaded_string = stored_data[5 + i]
                if i < len(config.ins.strings):
                    for key, value in loaded_string.__dict__.items():
                        if hasattr(config.ins.strings[i], key):
                            setattr(config.ins.strings[i], key, value)

            # --- CRITICAL FIX: Re-initialize steppers after loading preset ---
            # This ensures that all steppers are correctly assigned to their
            # Arduino boards and have the correct local indices, preventing
            # the state corruption that was disabling the Z-steppers.
            config.ins.steps_init()

            # Move steppers to their stored positions
            loaded_x_positions = stored_data[5]
            loaded_z_positions = stored_data[5 + string_num]
            loaded_tuner_positions = stored_data[6 + string_num]

            if config.ard.exist:
                # Move main board steppers
                z_steppers = [s for s in config.ins.all_steps if s and s.arduino == config.ard]
                for stepper in z_steppers:
                    if stepper.local_index < len(loaded_z_positions):
                        pos = loaded_z_positions[stepper.local_index]
                        self.change_abs_move(stepper, pos)

            if config.ard_T.exist:
                # Move tuner board steppers
                tuner_steps = [s for s in config.ins.all_steps if s.arduino == config.ard_T]
                max_tuner_local_index = max([s.local_index for s in tuner_steps if s.local_index is not None], default=-1)
                for stepper in tuner_steps:
                    if stepper.local_index < len(loaded_tuner_positions):
                        pos = loaded_tuner_positions[stepper.local_index]
                        self.change_abs_move(stepper, pos)

            # Update hardware settings
            if config.ard.exist or config.ard_T.exist:
                all_steppers_to_configure = config.ins.all_steps
                for stepper in all_steppers_to_configure:
                    if stepper and stepper.arduino:
                        msg = stepper.arduino.set_speed(stepper.local_index, stepper.speed)
                        if msg: self.message += msg
                        msg = stepper.arduino.set_accel(stepper.local_index, stepper.accel)
                        if msg: self.message += msg
            self.current_set = val
            update()

        except Exception:
            traceback.print_exc()
            self.message += '\nUnexpected error: ' + str(sys.exc_info())


    def get_storlist(self):
        self.storlist = []
        for i in range(self.stor_size):
            if self.set_storage[i]:
                self.storlist.append(i)

    def change_run(self,state):
        if state:
            # Sampler is initialized by config.py; plotting is disabled.
            if config.ard.exist:
                make_change.dispatch('initialize')
            self.message += '\nrun started'
            while gui.form.run.isChecked():
                try:
                    if self.exit:
                        self.exit = 0
                        return
                    # update() already calls bump_check; redundant here.
                    # make_change.change_bump_check()
                    update()
                except Exception:
                    traceback.print_exc()
                    self.message += '\nUnexpected error: '+str(sys.exc_info())
        else:
            self.message += '\nrun stopped'

    def change_sampler_init(self):
        try:
            if config.sampler.mysampler.is_alive():
                config.sampler.mysampler.stop()
            config.sampler.sd_init(show_plot=0)  # Called when sampler is initialized
            time.sleep(2)
            self.message += '\nsampler  initialized'
        except Exception:
            traceback.print_exc()
            print("Unexpected error:", sys.exc_info())

    def change_db_query(self,val):
        config.db.query = val

    def change_db_make_query(self,*args):
        if len(args) == 0:
            time_start = 1.637380e+09
            time_end = 1.637460e+09
            level = 5
        elif len(args) == 1:
            time_start = args[0]
            time_end = 2.000000e+09
            level = 5
        elif len(args) == 2:
            time_start = args[0]
            time_end = 2.000000e+09
            level = args[1]
        elif len(args) == 3:
            time_start = args[0]
            time_end = args[1]
            level = args[2]
        config.db.make_query(time_start,time_end,level)
        self.message += '\n'+str(config.db.query)

    def change_db_load(self,*args):
        if config.db.query is None:
            self.dispatch('db_make_query')
        self.message += '\nloading data...'
        try:
            config.db.get_data()
            self.message += '\ndata loaded, plotting...'
            config.db.plots()
        except Exception:
            traceback.print_exc()
            self.message += '\nUnexpected error: '+str(sys.exc_info())

    def change_db_play(self):
        if config.db.df is None:
            self.dispatch('db_load')
        global peaks
        config.ins.show_fft = 0
        while True:
            start = config.db.start_point
            end = config.db.steps + start
            change = False
            for ind in range(start,end):
                change_check()
                if self.exit:
```

```
                return
            if start != config.db.start_point or end != (config.db.steps + start):
                change = True
                break
            if config.db.play_enable:
                self.change_db_follow(ind) #parse the row
                change_check()
                peaks = config.db.get_peaks(ind)
                config.sampler.myfreq.peaks = peaks
                sam_play()
                if config.sampler.myfreq.show_plot:
                    config.sampler.spawn(config.sampler.myfreq.plot_sample)
                    config.sampler.myfreq.data = config.sampler.myfreq.p.get()
                else:
                    config.sampler.myfreq.plot_sample()
                gui.gui_sync()
                print_report()
                gui.lap_timer()
                if config.db.real_time_enable:
                    self.dispatch('z_adjuster_tuner')
            else:
                change_check()
                if self.exit:
                    return
                start = ind
        if not config.db.loop_enable and not change:
            config.db.play_enable = 0
            return

def change_db_follow(self,ind): #THIS needs work!
    db_row = config.db.df.loc[[ind]]
    gui.form.date_time.setPlainText(str(db_row['time']))
    # gui.form.date_time.setPlainText(str(datetime.fromtimestamp(db_row['time']).strftime('%Y-%m-%d %H:%M:%S')))
    gui.form.time_stamp.setPlainText(str(db_row['time']))
    if config.ard.exist:
        if config.ins.x_step.enable:
            self.change_abs_move(config.ins.x_step,int(db_row['X']))#*step_scale))
        if config.ins.z_steps[0].enable:
            self.change_abs_move(config.ins.z_steps[0],int(db_row['Z_In_0']))
        if config.ins.z_steps[1].enable:
            self.change_abs_move(config.ins.z_steps[1],int(db_row['Z_Out_0']))
        if config.ins.z_steps[2].enable:
            self.change_abs_move(config.ins.z_steps[2],int(db_row['Z_In_1']))
        if config.ins.z_steps[3].enable:
            self.change_abs_move(config.ins.z_steps[3],int(db_row['Z_Out_1']))
        # Handle tuner steps
        if config.ins.tuner_steps and len(config.ins.tuner_steps) > 0:
            if config.ins.tuner_steps[0].enable:
                self.dispatch('root', 0, float(db_row['root_0']))
        if config.ins.tuner_steps and len(config.ins.tuner_steps) > 1:
            if config.ins.tuner_steps[1].enable:
                self.dispatch('root', 1, float(db_row['root_1']))
    else:
        for widget in gui.app.allWidgets():
            #print('widget: ',widget.objectName())
            widget.blockSignals(True)
        gui.form.x_pos.setValue(int(db_row['X']))
        gui.form.x_position_0.setValue(int(db_row['X']))
        gui.form.x_position_1.setValue(int(db_row['X']))
        gui.z_rect_update(int(gui.form.x_pos.value()))

        gui.form.z_pos_in_0.setValue(int(db_row['Z_In_0']))
        gui.form.slider_z_pos_in_0.setValue(int(db_row['Z_In_0']))

        gui.form.z_pos_out_0.setValue(int(db_row['Z_Out_0']))
        gui.form.slider_z_pos_out_0.setValue(int(db_row['Z_Out_0']))

        gui.form.z_pos_in_1.setValue(int(db_row['Z_In_1']))
        gui.form.slider_z_pos_in_1.setValue(int(db_row['Z_In_1']))

        gui.form.z_pos_out_1.setValue(int(db_row['Z_Out_1']))
        gui.form.slider_z_pos_out_1.setValue(int(db_row['Z_Out_1']))

        gui.form.root_0.setValue(float(db_row['root_0']))
        gui.form.root_1.setValue(float(db_row['root_1']))

        for widget in gui.app.allWidgets():
            widget.blockSignals(False)

def change_db_plots(self,*args):
    if len(args) == 0:
        time_start=1643815000
        level=0
        depth=4
    else:
        time_start = args[0]
        level = args[1]
        depth = args[2]
    config.db.make_query(time_start=time_start,level=level)
    print(config.db.query)
    config.db.get_data()
    config.db.plots(depth=depth)
    def get_speeds(self):
        if config.ard.exist:
            all_speeds = [0,0]*len(config.ins.all_steps)
            for i,stepper in enumerate(config.ins.all_steps):
                all_speeds[i] = [stepper.accel,stepper.speed]
            return all_speeds
        else:
            self.message += '\nno Arduino'

    def z_range_check(self,pos):
        if not config.ins.z_steps[0].min_pos <= pos <= config.ins.z_steps[0].max_pos:
            self.message += '\nout of range'
            self.change_z_calibrate()
            return False
        else:
            return True

    def change_ard_init(self):
        if config.ard.exist:
            config.ard.arduino_init()
        if config.ard_T.exist:
            config.ard_T.arduino_init()

        # After Arduinos are confirmed, re-run instrument setup
        # to correctly assign boards and enable steppers.
        config.ins.steps_init()

    def change_check_memory(self):
        if config.ard.exist:
            self.message += config.ard.check_memory()



def linear_sweep(start, end, steps):
    """
    Generates a linear sweep of values from start to end with the given number of steps.

    Parameters:
    start (float or int): Starting value of the sweep.
    end (float or int): Ending value of the sweep.
    steps (int): Number of values to generate between start and end.

    Returns:
    np.ndarray: Array of linearly spaced values.
    """
    return np.linspace(start, end, steps)

def tolist(mess):
    l = list()
    for i in mess.split(','):
        try:
            val = int(i)
        except ValueError:
            try:
                val = float(i)
            except ValueError:
                val = i
        l.append(val)
    return l

def change_check():
    gui.app.processEvents()
#    if config.myjoy.joystick is not None:
#        config.myjoy.butcheck()
#    inval = input_box1.update()
#    if inval != '':
#        make_change.box_dispatch(inval.split(','))

def waiter(wait: float = 0):
    """Pause for <wait> seconds while keeping the GUI responsive.

    Uses a nested QEventLoop driven by a single-shot QTimer, so no busy CPU
    loops and no starvation of the main event loop.
    """

    if wait <= 0:
        return

    # Flush any pending events first (old behaviour of change_check()).
    change_check()

    loop = QtCore.QEventLoop()
    QtCore.QTimer.singleShot(int(wait * 1000), loop.quit)
    loop.exec_()
    # One more pass to catch anything that arrived during the rest.
    change_check()

def sam_play():
    if peaks is not None:
        # if config.sampler.s.getIsStarted():
        #     config.sampler.all_off()
```

```
            # for i in config.sampler.mysines:
            #     i.samplayer(peaks)
        for i in config.mm.mymidis:
            i.midplayer(peaks=peaks,time=config.ins.lap_rest)
    waiter(config.ins.lap_rest)

def tight_list(mylist):
    if mylist is None:
        return ''
    else:
        # Format as "freq, amp" using 2-dp frequency and integer amplitude
        return ' || '.join([f'{mylist[n][0]:.2f}, {int(mylist[n][1])}' for n in range(len(mylist))])

def print_positions():
    print('X Position: ', x_position)
    print('Z Positions: ', z_positions)
    if tuner_positions:
        print('Tuner Positions: ', tuner_positions)
    for i,j in np.ndenumerate(peaks):
        print('String_',i,': ',tight_list(makeitemlist(j)))

def flatten(a):
    if not isinstance(a,(tuple,list)): return [a]
    if len(a)==0: return []
    return flatten(a[0])+flatten(a[1:])

# Database column definitions
tuner_keys = ('enable','index','turns','speed','accel')
x_keys = ('index','min_pos','max_pos','enable','speed','accel')
z_keys = ('down_step','up_step','index','enable','speed','accel')
freq_keys = ('duration','high_cut','low_cut','mpd','mph','voices',
                'prominence','threshold','z_max','show_fft',
                'show_pitches','show_plot','filter_gain','crosstalk_scale','noise_gate_threshold')
ins_keys = ('z_variance_threshold','z_rest','x_rest','adjustment_level','level','retry_threshold','delta_threshold','bump_check_enable','bum
midi_keys = ('trans','program','show_notes','gain')
# sine_keys = ('trans','fade_in','fade_out','dur','gain')
string_keys = ('max_thresh','max_voice','min_thresh','min_voice') #,'root','fun','tolerance','follow','gauge','kind')
db_keys = ('predictvel','input_right_enable','table_enable')
# log_keys = ('input_select','carriage_coil_phase_0','carriage_coil_phase_1','fixed_coil_phase_0','fixed_coil_phase_1',
#              'piezo_left_level','piezo_right_level','carriage_coil_level','fixed_coil_level',
#              'input_left_le_level')

def makeheaders():
    if config.ard.exist:
        headers = ['time','command']
        headers.append('X')
        for i in range(config.ins.string_num):
            headers.append(['Z_In_' + str(i),'Z_Out_' + str(i)])
        for i in range(config.ins.string_num):
            headers.append(['Tuner_' + str(i)])
        for i in range(config.sampler.ichnls):
            for j in range(config.sampler.myfreq.voices):
                headers.append(['Freq_' + str(i) + '_' + str(j),'Amp_' + str(i) + '_' + str(j)])
        headers.append(namesfix(makekeylist(config.ins.x_step,x_keys),str(config.ins.x_step.index)))
        for i in range(len(config.ins.z_steps)):
            headers.append(namesfix(makekeylist(config.ins.z_steps[i],z_keys),str(config.ins.z_steps[i].index)))

        # Ensure tuner step header indices do not collide with Z/X indices.
        # If tuner indices are already beyond the max Z index, keep them.
        # Otherwise, offset tuner indices so all (accel_/speed_/etc.) names remain unique.
        try:
            max_z_index = max((s.index for s in config.ins.z_steps if s is not None), default=-1)
            tuner_indices = [s.index for s in config.ins.tuner_steps if s is not None]
            if tuner_indices:
                min_tuner_index = min(tuner_indices)
            else:
                min_tuner_index = 0
            needs_offset = any(idx <= max_z_index for idx in tuner_indices)
        except Exception:
            max_z_index = -1
            min_tuner_index = 0
            needs_offset = False

        for i in range(len(config.ins.tuner_steps)):
            step = config.ins.tuner_steps[i]
            # Skip placeholders when no real tuner hardware exists (index is None)
            if not step or step.index is None:
                continue

            if needs_offset:
                # Make a contiguous block after Z indices, preserving relative order
                suffix_idx = (step.index - min_tuner_index) + (max_z_index + 1)
                headers.append(namesfix(makekeylist(step, tuner_keys), str(suffix_idx)))
            else:
                headers.append(namesfix(makekeylist(step, tuner_keys), str(step.index)))
        headers.append(makekeylist(config.sampler.myfreq,freq_keys))
        for i in range(config.ins.string_num):
            headers.append(namesfix(makekeylist(config.ins.strings[i],string_keys),str(i)))
        headers.append(makekeylist(config.ins,ins_keys))
        headers.append(makekeylist(config.db,db_keys))
        # headers.append(makekeylist(config.ins.log,log_keys))
        headers = flatten(headers)
        return headers
    else:
        headers = ['time','command']
        for i in range(config.sampler.ichnls):
            for j in range(config.sampler.myfreq.voices):
                headers.append(['Freq_' + str(i) + '_' + str(j),'Amp_' + str(i) + '_' + str(j)])
        headers.append(makekeylist(config.sampler.myfreq,freq_keys))
        headers = flatten(headers)
        return headers

def safe_db_operation(func, *args, default_return=None):
    """Safely execute a database operation"""
    try:
        if not config.db.connected:
            return default_return
        return func(*args)
    except Exception as e:
        print(f"Database operation failed: {str(e)}")
        return default_return

def write_db(data_row):
    """Write a row to the current table, with auto-reconnection and schema-fixing."""
    if not config.db.save_db:
        return

    # Auto-reconnection logic
    try:
        config.db.db_connection.ping(reconnect=True, attempts=3, delay=1)
    except (sql.Error, AttributeError):
        print("INFO: Database connection lost. Attempting to re-initialize...")
        db_init()
        if not config.db.connected:
            print("ERROR: Reconnection failed. Cannot write to database.")
            return

    # Original logic continues here...
    data = str_to_nums(data_row)
    vals = "VALUES ({0})".format(', '.join(['%s'] * len(data)))
    query = (
        f"INSERT INTO {config.db.data_base}.{config.db.data_table} "
        f"({','.join(makeheaders())}) {vals}"
    )

    try:
        config.db.db_cursor.execute(query, data)
        config.db.db_connection.commit()
    except sql.Error as e:
        errno = getattr(e, 'errno', None)
        # 1146: table doesn't exist – create with proper headers then retry
        if errno == 1146:
            ensure_table(config.db.data_table, makeheaders())
            config.db.db_cursor.execute(query, data)
            conn = config.db.db_connection
            conn.commit()
        elif errno in (1136, 1054):  # 1136 = column count, 1054 = unknown column
            # ------------------------------------------------------------------
            # Provide richer diagnostics so the operator knows *why* it failed
            # without having to dive into MySQL logs.  DO NOT attempt any
            # automatic destructive action – just report and skip the row.
            # ------------------------------------------------------------------

            if errno == 1136:
                # Column-count mismatch: report expected vs actual counts
                expected_cols = len(makeheaders())
                actual_cols   = len(data)
                detail_msg = (
                    f"Column-count mismatch: expected {expected_cols} columns "
                    f"but got {actual_cols}."
                )
            elif errno == 1054:
                # Unknown column – expose the MySQL message for the missing name
                detail_msg = f"Unknown column reported by MySQL: {getattr(e, 'msg', str(e))}"
            else:
                detail_msg = ""  # Fallback (should not occur)

            print(
                f"WARN: Schema mismatch writing to {config.db.data_table} (errno {errno}). "
                f"{detail_msg} Row skipped. Align schema manually or enable a safe ALTER path."
            )
        else:
            # For other errors, mark the connection as dead so the next call will re-initialize.
            config.db.connected = False
            print(f"ERROR: A database write error occurred: {e}")
            # Do not re-raise, to allow the writer thread to continue.


def write_csv(data_row):
    global new_headers
    try:
        # Ensure data directory exists
        data_dir = os.path.join(CURR_DIR, 'data')
        os.makedirs(data_dir, exist_ok=True)

        with open(os.path.join(data_dir, config.db.data_file), 'a+') as myfile:
            wr = csv.writer(myfile,quoting=csv.QUOTE_ALL)
            if new_headers:
                new_headers = False
                if os.stat(os.path.join(data_dir, config.db.data_file)).st_size != 0:
                    with open(os.path.join(data_dir, config.db.data_file), newline='') as f:
                        reader = csv.reader(f)
                        row1 = next(reader)
                        if len(row1) != len(makeheaders()):
                            wr.writerow(makeheaders())
                else:
                    wr.writerow(makeheaders())
            wr.writerow(data_row)
    except Exception:
        traceback.print_exc()
        make_change.message += '\nUnexpected error: '+str(sys.exc_info())

def print_report():
    """Collect the current runtime state and persist one row.

    A fresh UTC timestamp is generated **for every call** so that each
    stored row differs at least in the `time` column, even when the
    Arduino is offline and therefore no new hardware report arrives.
    """

    global data_row
    global timestamp

    # Always use a new timestamp.  When the HardwareWorker subsequently
    # updates `timestamp` from a real Arduino report we will overwrite
    # this placeholder on the next cycle, so there is no risk of the
    # GUI showing stale times.
    timestamp = strftime("%Y-%m-%d %H:%M:%S", gmtime())
    # Add timestamp; include positions only if we are not in monitor-only mode.
    # The GUI layer already appends the canonical positions block once per
    # cycle, so avoid duplicating those lines here. Only stamp the timestamp.
    display_timestamp = strftime("%Y-%m-%d %H:%M:%S", localtime())
    make_change.message += f"\n{display_timestamp}"
    if config.db.print_report_enable:
        print('myfreq: ',tight_list(makeitemlist(config.sampler.myfreq,freq_keys)))
        for i,j in enumerate(config.mm.mymidis):
            print('mymidis_',i,': ',tight_list(makeitemlist(j,midi_keys)))
        # for i,j in enumerate(config.sampler.mysines):
        #     print('sampler.mysines_',i,': ',tight_list(makeitemlist(j,sine_keys)))
        print('program: ',tight_list(makeitemlist(config.ins,ins_keys)))
        if config.ard.exist:
            print('x_step: ',tight_list(makeitemlist(config.ins.x_step,x_keys)))
            for i,z_step in enumerate(config.ins.z_steps):
                print('z_step_',i,': ',tight_list(makeitemlist(z_step,z_keys)))
            for i,tuner_step in enumerate(config.ins.tuner_steps):
                print('tuner_step_',i,': ',tight_list(makeitemlist(tuner_step,tuner_keys)))
            for i,string in enumerate(config.ins.strings):
                print('string_',i,': ',tight_list(makeitemlist(string,string_keys)))
            print('X Position: ',x_position)
            print('Z Positions: ',z_positions)
            if tuner_positions:
                print('Tuner Positions: ',tuner_positions)
            for chan in range(peaks.shape[0]):
                print('String_',chan,': ',peaks[chan].tolist())
        make_change.get_storlist()
        print('presets: ',make_change.storlist,'\ncurrent_set: ',make_change.current_set)
    if config.db.save_csv or config.db.save_db:
        # Rely on positions already updated by the HardwareWorker.
        # Avoid calling config.ard.report() here because concurrent access
        # from multiple threads can corrupt the serial stream.
        if timestamp is None:
            timestamp = strftime("%Y-%m-%d %H:%M:%S", gmtime())

        data_row=[]
        if config.ard.exist:
            # Build deterministic block: X, Z_In_0, Z_Out_0, Z_In_1, Z_Out_1, …, Tuner_0, Tuner_1, …
            x_val = config.ins.x_step.position
            z_vals = []
            for string in config.ins.strings:
                zin = string.z_in.position
                zout = string.z_out.position
                z_vals.extend([zin, zout])

            tuner_vals = []
            for tuner_step in config.ins.tuner_steps:
                # Handle placeholder tuners (None or unassigned) on hosts without tuner hardware
                if tuner_step is None or getattr(tuner_step, 'position', None) is None:
                    tuner_vals.append(0)
                else:
                    tuner_vals.append(tuner_step.position)

            positions_block = [x_val] + z_vals + tuner_vals
            data_row = [timestamp,
                        command,
                        positions_block,
                        peaks.tolist(),
                        makevaluelist(config.ins.x_step,x_keys),
                        *[makevaluelist(step, z_keys) for step in config.ins.z_steps],
                        *[makevaluelist(step, tuner_keys) for step in config.ins.tuner_steps if step and step.index is not None],
                        makevaluelist(config.sampler.myfreq,freq_keys),
                        *[makevaluelist(string, string_keys) for string in config.ins.strings],
                        makevaluelist(config.ins,ins_keys),
                        makevaluelist(config.db,db_keys)]
            data_row = flatten(data_row)
        else:
            data_row = [timestamp,
                        command,
                        peaks.tolist(),
                        makevaluelist(config.sampler.myfreq,freq_keys)]
            data_row = flatten(data_row)
#           print('data_row: ',data_row)
    if config.db.save_csv or config.db.save_db:
        # Push to CSV writer queue only for CSV; log DB directly via Postgres
        if config.db.save_csv:
            try:
                event_q.put_nowait(data_row)
            except queue.Full:
                write_csv(data_row)  # synchronous fallback for CSV only

        if config.db.save_db:
            # Always log stepper states to PostgreSQL – no fallbacks
            pg_steppers.log_stepper_states(config.ins.all_steps, command=command if 'command' in globals() else None)

            # Show database status warning if unavailable
            ctx = pg_steppers.PgSteppersContext.get()
            status_msg = ctx.get_status_message()
            if status_msg:
                make_change.message += f'\n{status_msg}'

def peaks_update():
    # Get current data from shared memory
    data = config.sampler.myfreq.get_current_buffer()
    if data is not None:
        config.sampler.myfreq.update_peaks(data)
        amp_report()
        voice_count()  # Calculate voice count

        # Plotting disabled in Python; Rust handles visualization

        gui.gui_report()  # Update GUI data
        gui.gui_update()  # Refresh GUI display

def sum_amps(some_peaks):
    if isinstance(some_peaks, list):
        # Convert list of lists of tuples to numpy array
        peaks_array = np.array(some_peaks)
        # Sum the amplitudes (second value of each tuple)
        amp_sums = np.sum([amp for channel in peaks_array for _, amp in channel], axis=0)
    else:
        # Original numpy array handling
        amp_sums = np.sum(some_peaks[:, :, 1], axis=1)
    return amp_sums

def voice_count():
    global v_num
    v_num = [0] * peaks.shape[0]
    for i in range(peaks.shape[0]):
        v_num[i] = np.count_nonzero(peaks[i],axis=0)[1]
    make_change.message += f'\nvoice_count: {v_num}'


def amp_report():
    global current, delta, delta_p, peaks
    peaks = config.sampler.myfreq.peaks
    # Initialize current if it doesn't exist
    if 'current' not in globals():
        current = 0
    previous = current
    current = sum_amps(peaks)
    make_change.message += f'\namp_sums: {current.astype(int)}'
    delta = np.subtract(current, previous)
    make_change.message += f'\ndelta: {delta.astype(int)}'
    with np.errstate(divide='ignore', invalid='ignore'):
        delta_p = np.divide(current, previous, where=previous!=0)

def get_positions():
    """
    Build position arrays from Python-tracked Stepper.position fields.
    Legacy Arduino polling is retained below but commented out per new design.
    """
    global x_position, z_positions, tuner_positions, timestamp, report

    x_position = [config.ins.x_step.position]

    # Size the list to include **all** main-board Z-steppers (local index 0■N)
    max_local = max([s.local_index for s in config.ins.z_steps if s and s.local_index is not None], default=-1)
    z_positions = [0.0] * (max_local + 1)
    for z_step in config.ins.z_steps:
        if z_step and z_step.local_index is not None:
            z_positions[z_step.local_index] = z_step.position

    if config.ins.has_tuners:
        tuner_positions = [0] * config.ins.string_num
        for i in range(len(config.ins.tuner_steps)):
            tuner_positions[i] = config.ins.tuner_steps[i].position


    # Timestamp remains a simple heartbeat of when we refreshed Python state
    timestamp = strftime("%Y-%m-%d %H:%M:%S", gmtime())

def update():
    # Moved to the top so it always runs even if the rest of the update
    # cycle is slow (e.g. during a blocking bump_check recovery).
    gui.lap_timer()
    change_check()
    # Always poll positions so tuner-only setups update correctly
    get_positions()
    # Run bump_check only when main Arduino exists and feature is enabled
    if config.ard.exist and config.ins.bump_check_enable:
        make_change.change_bump_check()
    peaks_update()
    print_report()
    sam_play()
    gui.gui_sync()
    # gui.lap_timer()  # MOVED to top of function
    gc.collect()

def _cleanup():
    """Gracefully shut down worker threads and external processes."""
    # Flush and stop DbWriter thread
    try:
        if '_db_writer_thread' in globals():
            event_q.put("QUIT", block=False)
            _db_writer_thread.join(timeout=2)
    except Exception:
        traceback.print_exc()

    # Attempt to terminate external helper processes gracefully
    for pattern in (
        "jackd",
        "qjackctl",
        "stream.sh",
        "gst-launch",
        "gst-launch-1.0",
        "rust_driver",
        "rust_gui",
        "xterm",
    ):
        try:
            subprocess.run(["pkill", "-f", pattern], check=False)
        except Exception:
            pass

class GuiInput():
    def __init__(self,ui_file='String_Driver_2.ui'):
        # Get existing QApplication instance or create new one
        self.app = QtWidgets.QApplication.instance() or QtWidgets.QApplication(sys.argv)

        Form,Window = uic.loadUiType(ui_file)
        self.window = Window()
        self.window.move(0,50)
        self.form = Form()
        self.form.setupUi(self.window)
        self.timer = time.time()
        self._update_string_tabs()
        self.gui_connect()

        # Ensure closing the main window triggers full application quit so
        # _cleanup() runs and external processes are terminated.
        def _on_close(event):
            try:
                make_change.change_quit(True)
            finally:
                event.accept()

        # Monkey-patch the closeEvent of the generated QMainWindow instance.
        self.window.closeEvent = _on_close  # type: ignore[assignment]

        # In case other hidden windows remain, force Qt to quit when all are
        # closed.
        self.app.setQuitOnLastWindowClosed(True)

    def _update_string_tabs(self):
        """Disables UI tabs for strings that are not in use."""
        try:
            num_strings = config.ins.string_num
            # Based on UI design, there are 6 tabs for individual strings (0-5).
            # We will only modify these tabs, leaving others like "All" untouched.
            MAX_STRING_TABS = 6
            for i in range(MAX_STRING_TABS):
                self.form.tabWidget.setTabEnabled(i, i < num_strings)

        except AttributeError:
            # This could happen if tabWidget doesn't exist in the UI file.
            print("WARN: Could not find 'tabWidget' to update string tabs.")
        except Exception as e:
            # Log other errors but don't crash the application.
            print(f"Could not update string tabs: {e}")


    def lap_timer(self):
        lap = float(time.time() - self.timer)
        lap_str = f'laptime: {round(lap, 2)}'
        try:
            # Works if laptime is a QPlainTextEdit
            self.form.laptime.setPlainText(lap_str)
        except AttributeError:
            # Fallback if it is e.g. a QLabel
            if hasattr(self.form.laptime, 'setText'):
                self.form.laptime.setText(lap_str)
            else:
                # Unexpected widget type – just print to console so we notice
                print('WARN: laptime widget has no setPlainText / setText')
        self.timer = time.time()

    def gui_update(self):
        # Check if exit was requested
        global exit_requested
        if exit_requested:
            return  # Shutdown already in progress, don't update GUI

        # Allow GUI updates if either main or tuner board exists
        if config.ard.exist or (hasattr(config, 'ard_T') and config.ard_T.exist):
            for widget in self.app.allWidgets():
                #print('widget: ',widget.objectName())
                widget.blockSignals(True)

            self.form.x_pos.setValue(config.ins.x_step.position)

            for i in range(config.ins.string_num):
                    getattr(self.form, f"x_position_{i}").setValue(config.ins.x_step.position)

            self.z_rect_update(int(self.form.x_pos.value()))

            # Update Z steppers (always on main board) – display float positions
            for i in range(config.ins.string_num):
                if i*2 < len(config.ins.z_steps) and config.ins.z_steps[i*2].local_index is not None:
                     z_step_in = config.ins.z_steps[i*2]
                     getattr(self.form, f"z_pos_in_{i}").setValue(z_step_in.position)
                     getattr(self.form, f"slider_z_pos_in_{i}").setValue(int(round(z_step_in.position)))

                if i*2+1 < len(config.ins.z_steps) and config.ins.z_steps[i*2+1].local_index is not None:
                     z_step_out = config.ins.z_steps[i*2+1]
                     getattr(self.form, f"z_pos_out_{i}").setValue(z_step_out.position)
                     getattr(self.form, f"slider_z_pos_out_{i}").setValue(int(round(z_step_out.position)))

            # Update tuner steppers (always on tuner board)
            if hasattr(config, 'ard_T') and config.ard_T.exist:
                for i in range(config.ins.string_num):
                    if i < len(config.ins.tuner_steps) and config.ins.tuner_steps[i].local_index is not None:
                        t_step = config.ins.tuner_steps[i]
                        getattr(self.form, f"tuner_pos_{i}").setValue(t_step.position)

            for widget in self.app.allWidgets():
                widget.blockSignals(False)

    def z_rect_update(self,pos):
        # Get X_MAX_POS from configuration to calculate proper scale for GUI display
        try:
            from instrument import host_config as _hc
            X_MAX_POS = _hc.get('X_MAX_POS', 2600)  # Default to stringdriver-2 range if not specified
        except Exception:
            X_MAX_POS = 2600  # Fallback default

        # Calculate scale based on the configured X_MAX_POS
        # The original scale of 2.5 was designed for 2600 range
        # So we scale proportionally: scale = 2.5 * (2600 / X_MAX_POS)
        scale = 2.5 * (2600 / X_MAX_POS)

        x = self.form.x_position_0.geometry().x()+10
        for i in range(config.ins.string_num):
            rect = getattr(self.form, f"rect_z_{i}").geometry()
            rect.moveTo(int(x+pos/scale),rect.y())
            getattr(self.form, f"rect_z_{i}").setGeometry(rect)

    def gui_report(self):
        pass

    def gui_message(self):
        if make_change.message:
```

```
        lines = []
        for line in make_change.message.split('\n'):
            stripped = line.strip()
            # Skip raw tuple-style debug lines (e.g. "('turns', 0, 10000)")
            if stripped.startswith('(') and stripped.endswith(')'):
                continue
            lines.append(line)

        if lines:
            msg = '\n'.join(lines)
            print(msg)
            self.form.message.appendPlainText(msg)

        # After printing the normal text, fall through to append positional
        # information so that *exactly* the same block (normal message plus
        # positions) is sent to both the terminal and the GUI list.

        # ------------------------------------------------------------------
        # Compose the authoritative status block (positions) and deliver it
        # once – to both terminal (print) and GUI – keeping the two outputs
        # identical.
        # ------------------------------------------------------------------

        # Format positions according to user's explicit requirements
        x_pos = []
        z_pos = []

        # Extract X position (should be single int)
        if config.ins.x_step.local_index is not None :
            x_pos = config.ins.x_step.position
        else:
            x_pos = [0]

        # Extract Z positions (should be floats)
        for i, z_step in enumerate(config.ins.z_steps):
            if z_step.local_index is not None:
                z_pos.append(z_step.position)
            else:
                z_pos.append(0.0)

        # Extract tuner positions (should be ints)
        tuner_pos = []
        for tuner in config.ins.tuner_steps:
            if tuner is not None and tuner.local_index is not None:
                tuner_pos.append(tuner.position)
            else:
                tuner_pos.append(0)

        status_lines = [
            f"x_position:\n {x_pos}",
            f"z_positions:\n {z_pos}",
            f"tuner_positions:\n {tuner_pos}"
        ]
        status_msg = '\n'.join(status_lines)
        print(status_msg)
        self.form.message.appendPlainText(status_msg)

        # House-keeping: clear queued message text and keep GUI scrolled.
        make_change.message = ''
        self.form.message.ensureCursorVisible()
        self.app.processEvents()

    def gui_sync(self):
        global GUI_UPDATING
        GUI_UPDATING = True
        for widget in self.app.allWidgets():
            #print('widget: ',widget.objectName())
            widget.blockSignals(True)
        #messages
        self.gui_message()
        # make_change.message is cleared inside gui_message after printing
        # make_change.message = ''
        #log
        # row = config.ins.log.input_select =='mix'
        # self.form.input_select.setCurrentRow(row)
        # self.form.carriage_coil_phase_0.setChecked(config.ins.log.carriage_coil_phase_0)
        # self.form.carriage_coil_phase_1.setChecked(config.ins.log.carriage_coil_phase_1)
        # self.form.fixed_coil_phase_0.setChecked(config.ins.log.fixed_coil_phase_0)
        # self.form.fixed_coil_phase_1.setChecked(config.ins.log.fixed_coil_phase_0)
        # self.form.piezo_left_level.setValue(config.ins.log.piezo_left_level)
        # self.form.piezo_right_level.setValue(config.ins.log.piezo_right_level)
        # self.form.carriage_coil_level.setValue(config.ins.log.carriage_coil_level)
        # self.form.fixed_coil_level.setValue(config.ins.log.fixed_coil_level)
        # self.form.input_left_level.setValue(config.ins.log.input_left_level)
        # self.form.input_right_level.setValue(config.ins.log.input_right_level)

        #ins
        self.form.lap_rest.setValue(config.ins.lap_rest)
        self.form.tune_rest.setValue(config.ins.tune_rest)
        self.form.x_rest.setValue(config.ins.x_rest)
        self.form.z_rest.setValue(config.ins.z_rest)
        self.form.adjustment_level.setValue(config.ins.adjustment_level)
        self.form.z_calibrate_enable.setChecked(config.ins.z_calibrate_enable)
        self.form.bump_check_enable.setChecked(config.ins.bump_check_enable)
        self.form.bump_check_repeat.setValue(config.ins.bump_check_repeat)
        self.form.retry_threshold.setValue(config.ins.retry_threshold)
        self.form.delta_threshold.setValue(config.ins.delta_threshold)
        self.form.z_variance_threshold.setValue(config.ins.z_variance_threshold)

        #data_base
        self.form.save_csv.setChecked(config.db.save_csv)
        # Do not overwrite while the user is typing.
        self.form.save_db.setChecked(config.db.save_db)
        self.form.play_enable.setChecked(config.db.play_enable)
        self.form.loop_enable.setChecked(config.db.loop_enable)
        self.form.real_time_enable.setChecked(config.db.real_time_enable)
        self.form.start_point.setValue(config.db.start_point)
        self.form.steps.setValue(config.db.steps)
        self.form.predict_enable.setChecked(config.db.predict_enable)
        self.form.table_enable.setChecked(config.db.table_enable)
        #self.form.auto_write.setChecked(config.db.auto_write)
        if not self.form.preset_file.hasFocus():
            self.form.preset_file.setText(config.db.preset_file)

        #wiz
        self.form.model.setText(config.wiz.tag)

        #mymidis
        for i in range(len(config.mm.mymidis)):
            getattr(self.form, f"midi_show_notes_{i}").setChecked(config.mm.mymidis[i].show_notes)
            getattr(self.form, f"midi_program_{i}").setValue(config.mm.mymidis[i].program)
            getattr(self.form, f"midi_trans_{i}").setValue(config.mm.mymidis[i].trans)

        #gain_sliders
        for i in range(len(config.mm.mymidis)):
            getattr(self.form, f"midi_gain_{i}").setValue(int(config.mm.mymidis[i].gain*127))

        #tuner_steps

        for i in range(config.ins.string_num):
            getattr(self.form, f"fun_{i}").setChecked(config.ins.strings[i].fun)
            getattr(self.form, f"root_{i}").setValue(config.ins.strings[i].root)
            getattr(self.form, f"tolerance_{i}").setValue(config.ins.strings[i].tolerance)
            getattr(self.form, f"follow_{i}").setValue(config.ins.strings[i].follow)
            # Always show tuner configuration in GUI – tuners may reside on the
            # main Arduino (stringdriver-1) or on a separate tuner board. The
            # presence of a dedicated ard_T board must NOT gate these updates.

            tuner = config.ins.tuner_steps[i]
            if tuner is not None:
                getattr(self.form, f"turns_{i}").setValue(tuner.turns)
                getattr(self.form, f"tuner_accel_{i}").setValue(tuner.accel)
                getattr(self.form, f"tuner_speed_{i}").setValue(tuner.speed)
                getattr(self.form, f"tuner_enable_{i}").setChecked(tuner.enable)
                getattr(self.form, f"tuner_pos_{i}").setValue(tuner.position)
                getattr(self.form, f"tuner_min_pos_{i}").setValue(tuner.min_pos)
                getattr(self.form, f"tuner_max_pos_{i}").setValue(tuner.max_pos)

        # Keep GUI increment as defined in UI; values reflect true float positions via updates above

        #x_step
        self.form.left_limit.setValue(left_limit)
        self.form.right_limit.setValue(right_limit)
        self.form.x_enable.setChecked(config.ins.x_step.enable)
        self.form.x_accel.setValue(config.ins.x_step.accel)
        self.form.x_speed.setValue(config.ins.x_step.speed)

        #z_steps
        for i in range(config.ins.string_num):
            getattr(self.form, f"z_in_enable_{i}").setChecked(config.ins.z_steps[i*2].enable)
            getattr(self.form, f"z_out_enable_{i}").setChecked(config.ins.z_steps[i*2+1].enable)
            getattr(self.form, f"min_voice_{i}").setValue(config.ins.strings[i].min_voice)
            getattr(self.form, f"min_thresh_{i}").setValue(config.ins.strings[i].min_thresh)
            getattr(self.form, f"max_voice_{i}").setValue(config.ins.strings[i].max_voice)
            getattr(self.form, f"max_thresh_{i}").setValue(config.ins.strings[i].max_thresh)
            getattr(self.form, f"down_step_{i}").setValue(config.ins.z_steps[i*2].down_step)
            getattr(self.form, f"down_step_{i}").setValue(config.ins.z_steps[i*2+1].down_step)
            getattr(self.form, f"up_step_{i}").setValue(config.ins.z_steps[i*2].up_step)
            getattr(self.form, f"up_step_{i}").setValue(config.ins.z_steps[i*2+1].up_step)
            getattr(self.form, f"z_min_pos_{i}").setValue(config.ins.z_steps[i].min_pos)
            getattr(self.form, f"z_max_pos_{i}").setValue(config.ins.z_steps[i].max_pos)
            getattr(self.form, f"z_accel_in_{i}").setValue(config.ins.z_steps[i].accel)
            getattr(self.form, f"z_accel_out_{i}").setValue(config.ins.z_steps[i*2+1].accel)
            getattr(self.form, f"z_speed_in_{i}").setValue(config.ins.z_steps[i*2].speed)
            getattr(self.form, f"z_speed_out_{i}").setValue(config.ins.z_steps[i*2+1].speed)

        for widget in self.app.allWidgets():
            widget.blockSignals(False)
        GUI_UPDATING = False

    def on_run_clicked(self,state):
        make_change.dispatch('run',int(state))

    def on_midi_program_valueChanged(self,index):
        if index == 6:
            make_change.dispatch('midi_program',int(getattr(self.form, f"midi_program_{index}").value()))
        else:
            make_change.dispatch('midi_program',index,int(getattr(self.form, f"midi_program_{index}").value()))

    def on_midi_show_notes_clicked(self,index,state):
        if index == 6:
            make_change.dispatch('midi_show_notes',int(state))
        else:
            make_change.dispatch('midi_show_notes',index,int(state))

    def on_midi_trans_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('midi_trans',val)
        else:
            make_change.dispatch('midi_trans',index,val)

    def on_midi_gain_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('midi_gain',round(val/127,3))
        else:
            make_change.dispatch('midi_gain',index,round(val/127,3))
    #globs
    def on_save_preset_valueChanged(self):
        make_change.dispatch('save',int(self.form.save_preset.value()))

    def on_recall_preset_itemClicked(self,item):
        make_change.dispatch('recall',int(item.text()))

    def on_preset_file_editingFinished(self):
        make_change.dispatch('preset_file',self.form.preset_file.text())

    def on_load_presets_pressed(self):
        make_change.dispatch('load')

    def on_auto_write_clicked(self,state):
        make_change.dispatch('auto_write',int(state))

    def on_print_report_clicked(self,state):
        make_change.dispatch('print_report_enable',int(state))

    def on_save_csv_clicked(self,state):
        make_change.dispatch('save_csv',int(state))

    def on_z_calibrate_enable_clicked(self,state):
        make_change.dispatch('z_calibrate_enable',int(state))

    def on_bump_check_enable_clicked(self,state):
        make_change.dispatch('bump_check_enable',int(state))

    def on_bump_check_repeat_valueChanged(self):
        make_change.dispatch('bump_check_repeat',int(self.form.bump_check_repeat.value()))

    def on_lap_rest_valueChanged(self):
        config.ins.lap_rest = self.form.lap_rest.value()  # No change needed here

    def on_tune_rest_valueChanged(self):
        make_change.dispatch('tune_rest',round(self.form.tune_rest.value(),2))

    def on_x_rest_valueChanged(self):
        make_change.dispatch('x_rest',round(self.form.x_rest.value(),2))

    def on_z_rest_valueChanged(self):
        make_change.dispatch('z_rest',round(self.form.z_rest.value(),2))

    def on_adjustment_level_valueChanged(self):
        make_change.dispatch('adjustment_level',int(self.form.adjustment_level.value()))

    def on_retry_threshold_valueChanged(self):
        make_change.dispatch('retry_threshold',int(self.form.retry_threshold.value()))

    def on_delta_threshold_valueChanged(self):
        make_change.dispatch('delta_threshold',int(self.form.delta_threshold.value()))

    def on_z_variance_threshold_valueChanged(self):
        make_change.dispatch('z_variance_threshold',int(self.form.z_variance_threshold.value()))

    def on_model_editingFinished(self):
        make_change.dispatch('predictors',self.form.model.text())

    def on_save_db_clicked(self,state):
        make_change.dispatch('save_db',int(state))

    def on_play_enable_Clicked(self,state):
        make_change.dispatch('play_enable',int(state))

    def on_loop_enable_Clicked(self,state):
        make_change.dispatch('loop_enable',int(state))

    def on_real_time_enable_Clicked(self,state):
        make_change.dispatch('real_time_enable',int(state))
    def on_start_point_valueChanged(self):
        make_change.dispatch('start_point',int(self.form.start_point.value()))

    def on_steps_valueChanged(self):
        make_change.dispatch('steps',int(self.form.steps.value()))

    def on_inputs_valueChanged(self):
        make_change.dispatch('inputs',int(self.form.inputs.value()))

    def on_outputs_valueChanged(self):
        make_change.dispatch('outputs',int(self.form.outputs.value()))

    def on_tuner_pos_valueChanged(self,index,val):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                make_change.dispatch('abs_move', tuner, val)
        else:
            tuner = config.ins.tuner_steps[index]
            make_change.dispatch('abs_move', tuner, val)

    def on_tuner_min_pos_valueChanged(self,index,val):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                tuner.min_pos = val
        else:
            config.ins.tuner_steps[index].min_pos = val

    def on_tuner_max_pos_valueChanged(self,index,val):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                tuner.max_pos = val
        else:
            config.ins.tuner_steps[index].max_pos = val

    def on_tuner_enable_clicked(self,index,state):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                tuner.enable = int(state)
        else:
            config.ins.tuner_steps[index].enable = int(state)

    def on_root_valueChanged(self,index,val):
        if index == 6:
            for i in range(config.inst.string_num):
                make_change.dispatch('root',i,round(val,2))
        else:
            make_change.dispatch('root',index,round(val,2))

    def on_turns_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('turns',int(val))
        else:
            make_change.dispatch('turns',index,int(val))

    def on_tolerance_valueChanged(self,index,val):
        if index == 6:
            for i in range(config.ins.string_num):
                make_change.dispatch('tolerance',i,round(val,2))
        else:
            make_change.dispatch('tolerance',index,round(val,2))

    def on_follow_valueChanged(self,index,val):
        # Prevent spurious follow commands when the value hasn't really changed.
        def _current_follow(idx: int) -> int:
            try:
                return config.ins.strings[idx].follow
            except (AttributeError, IndexError):
                return -1

        if index == 6:
            for i in range(config.ins.string_num):
                if int(val) != _current_follow(i):
                    make_change.dispatch('follow', i, int(val))
        else:
            if int(val) != _current_follow(index):
                make_change.dispatch('follow', index, int(val))

    def on_fun_clicked(self,index,state):
        if index == 6:
            for i in range(config.ins.string_num):
                make_change.dispatch('fun',i,int(state))
        else:
            make_change.dispatch('fun',index,int(state))

    def on_tuner_accel_valueChanged(self,index,val):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                make_change.dispatch('set_accel', tuner, int(val))
        else:
            if index < len(config.ins.tuner_steps):
                make_change.dispatch('set_accel', config.ins.tuner_steps[index], int(val))
    def on_tuner_speed_valueChanged(self,index,val):
        if index == 6:
            for tuner in config.ins.tuner_steps:
                make_change.dispatch('set_speed', tuner, int(val))
        else:
            if index < len(config.ins.tuner_steps):
                make_change.dispatch('set_speed', config.ins.tuner_steps[index], int(val))

    def on_x_pos_valueChanged(self):
        make_change.dispatch('abs_move',config.ins.x_step,int(self.form.x_pos.value()))
        self.z_rect_update(int(self.form.x_pos.value()))

    def on_left_limit_valueChanged(self):
        global left_limit
        left_limit = int(self.form.left_limit.value())

    def on_right_limit_valueChanged(self):
        global right_limit
        right_limit = int(self.form.right_limit.value())

    def on_x_enable_clicked(self,state):
        make_change.dispatch('enable',config.ins.x_step,int(state))

    def on_table_enable_clicked(self,state):
        make_change.dispatch('table_enable',int(state))

    def on_predict_enable_clicked(self,state):
        make_change.dispatch('predict_enable',int(state))

    def on_x_accel_valueChanged(self):
        make_change.dispatch('set_accel',config.ins.x_step.index,int(self.form.x_accel.value()))

    def on_x_speed_valueChanged(self):
        make_change.dispatch('set_speed',config.ins.x_step.index,int(self.form.x_speed.value()))

    def on_z_pos_in_valueChanged(self,objectName):
        # print("objectName: ",objectName)
        index = int(str(objectName[-1]))*2
        # print("index: ", index)
        if index == 12:
            for z_step in config.ins.z_steps[::2]:
                val = float(getattr(self.form, f"{objectName}").value())
                make_change.dispatch('abs_move', z_step, val)
        else:
            z_step = config.ins.z_steps[index]
            val = float(getattr(self.form, f"{objectName}").value())
            # If user nudged by exactly ±1, treat as a relative move of ±1 raw step
            delta_ui = val - z_step.position
            if delta_ui == 1:
                make_change.dispatch('rel_move', z_step, z_step.up_step)
            elif delta_ui == -1:
                make_change.dispatch('rel_move', z_step, z_step.down_step)
            else:
                # Otherwise perform absolute move using counts derived from physical value
                make_change.dispatch('abs_move', z_step, val)

    def on_z_pos_out_valueChanged(self,objectName):
        # print("objectName: ",objectName)
        index = int(str(objectName[-1]))*2+1
        # print("index: ", index)
        if index == 13:
            for z_step in config.ins.z_steps[1::2]:
                val = float(getattr(self.form, f"{objectName}").value())
                make_change.dispatch('abs_move', z_step,val)
        else:
            z_step = config.ins.z_steps[index]
            val = float(getattr(self.form, f"{objectName}").value())
            delta_ui = val - z_step.position
            if delta_ui == 1:
                make_change.dispatch('rel_move', z_step, z_step.up_step)
            elif delta_ui == -1:
                make_change.dispatch('rel_move', z_step, z_step.down_step)
            else:
                make_change.dispatch('abs_move', z_step, val)

    def on_z_min_pos_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps:
                z_step.min_pos = val
        else:
            make_change.dispatch('min_pos',config.ins.z_steps[index*2],val)
            make_change.dispatch('min_pos',config.ins.z_steps[index*2+1],val)

    def on_z_max_pos_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps:
                z_step.max_pos = val
        else:
            make_change.dispatch('max_pos',config.ins.z_steps[index*2],val)
            make_change.dispatch('max_pos',config.ins.z_steps[index*2+1],val)

    def on_z_in_enable_clicked(self,index,state):
        if index == 6:
            for z_step in config.ins.z_steps[::2]:
                z_step.enable = int(state)
        else:
            make_change.dispatch('enable',config.ins.z_steps[index*2].index,int(state))

    def on_z_out_enable_clicked(self,index,state):
        if index == 6:
            for z_step in config.ins.z_steps[1::2]:
                z_step.enable = int(state)
        else:
            make_change.dispatch('enable',config.ins.z_steps[index*2+1].index,int(state))

    def on_min_thresh_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('min_thresh',val)
        else:
            make_change.dispatch('min_thresh',index,val)

    def on_min_voice_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('min_voice',val)
        else:
            make_change.dispatch('min_voice',index,val)

    def on_max_thresh_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('max_thresh',val)
        else:
            make_change.dispatch('max_thresh',index,val)

    def on_max_voice_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('max_voice',val)
        else:
            make_change.dispatch('max_voice',index,val)

    def on_down_step_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('down_step',val)
        else:
            make_change.dispatch('down_step',index,val)

    def on_up_step_valueChanged(self,index,val):
        if index == 6:
            make_change.dispatch('up_step',val)
        else:
            make_change.dispatch('up_step',index,val)

    def on_z_accel_in_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps[::2]:
                make_change.dispatch('set_accel',z_step.index,val)
        else:
            make_change.dispatch('set_accel',config.ins.z_steps[index*2].index,val)

    def on_z_accel_out_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps[1::2]:
                make_change.dispatch('set_accel',z_step.index,val)
        else:
            make_change.dispatch('set_accel',config.ins.z_steps[index*2+1].index,val)

    def on_z_speed_in_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps[::2]:
                make_change.dispatch('set_speed',z_step.index,val)
        else:
            make_change.dispatch('set_speed',config.ins.z_steps[index*2].index,val)

    def on_z_speed_out_valueChanged(self,index,val):
        if index == 6:
            for z_step in config.ins.z_steps[1::2]:
                make_change.dispatch('set_speed',z_step.index,val)
        else:
            make_change.dispatch('set_speed',config.ins.z_steps[index*2+1].index,val)

    def on_command_returnPressed(self):
        make_change.box_dispatch(list(self.form.command.text().split(',')))

    def on_command_list_itemClicked(self,item):
        make_change.box_dispatch(item.text().split(','))

    def on_z_step_size_itemClicked(self,item):
        ind = self.form.z_step_size.row(item)
        if ind == 0:
            sz = 0
            make_change.message += '\nz_step_size is 1\n'
        elif ind == 1:
            sz = 4
            make_change.message += '\nz_step_size is 1/2\n'
        elif ind == 2:
            sz = 2
            make_change.message += '\nz_step_size is 1/4\n'
        elif ind == 3:
            sz = 6
            make_change.message += '\nz_step_size is 1/8\n'
        elif ind == 4:
            sz = 7
            make_change.message += '\nz_step_size is 1/16\n'
        else:
            sz = 0
            make_change.message += '\nz_step_size is 1\n'
        make_change.dispatch('z_step_size',sz)

    def on_input_select_itemClicked(self,item):
        make_change.dispatch('input_select',item.text())

    def on_carriage_coil_phase_0_clicked(self,state):
        make_change.dispatch('carriage_coil_phase_0',int(state))

    def on_carriage_coil_phase_1_clicked(self,state):
        make_change.dispatch('carriage_coil_phase_1',int(state))

    def on_fixed_coil_phase_0_clicked(self,state):
        make_change.dispatch('fixed_coil_phase_0',int(state))

    def on_fixed_coil_phase_1_clicked(self,state):
        make_change.dispatch('fixed_coil_phase_1',int(state))

    def on_piezo_left_level_valueChanged(self,val):
        make_change.dispatch('piezo_left_level',val)

    def on_piezo_right_level_valueChanged(self,val):
        make_change.dispatch('piezo_right_level',val)

    def on_carriage_coil_level_valueChanged(self,val):
        make_change.dispatch('carriage_coil_level',val)

    def on_fixed_coil_level_valueChanged(self,val):
        make_change.dispatch('fixed_coil_level',val)

    def on_input_left_level_valueChanged(self,val):
        make_change.dispatch('input_left_level',val)

    def on_input_right_level_valueChanged(self,val):
        make_change.dispatch('input_right_level',val)


    # def on_sine_gain_1_slider_valueChanged(self,val):
    #     make_change.dispatch('sine_gain',1,round(val/10,2))

    def gui_connect(self):
        #myfreq
        self.form.run.clicked.connect(self.on_run_clicked)
        #mymidis
        for i in range(7): #config.ins.string_num + 1):
            getattr(self.form, f"midi_show_notes_{i}").clicked.connect(partial(self.on_midi_show_notes_clicked, i))
            getattr(self.form, f"midi_program_{i}").valueChanged.connect(partial(self.on_midi_program_valueChanged, i))
            getattr(self.form, f"midi_trans_{i}").valueChanged.connect(partial(self.on_midi_trans_valueChanged, i))
            getattr(self.form, f"midi_gain_{i}").valueChanged.connect(partial(self.on_midi_gain_valueChanged, i))

        #globs
        self.form.save_preset.valueChanged.connect(self.on_save_preset_valueChanged)
        self.form.recall_preset.itemClicked.connect(self.on_recall_preset_itemClicked)
        self.form.preset_file.editingFinished.connect(self.on_preset_file_editingFinished)
        self.form.load_presets.pressed.connect(self.on_load_presets_pressed)
        #self.form.auto_write.clicked.connect(self.on_auto_write_clicked)
        self.form.save_csv.clicked.connect(self.on_save_csv_clicked)
        self.form.lap_rest.valueChanged.connect(self.on_lap_rest_valueChanged)
        self.form.tune_rest.valueChanged.connect(self.on_tune_rest_valueChanged)
        self.form.adjustment_level.valueChanged.connect(self.on_adjustment_level_valueChanged)
        self.form.x_rest.valueChanged.connect(self.on_x_rest_valueChanged)
        self.form.z_rest.valueChanged.connect(self.on_z_rest_valueChanged)
        self.form.z_calibrate_enable.clicked.connect(self.on_z_calibrate_enable_clicked)
        self.form.bump_check_enable.clicked.connect(self.on_bump_check_enable_clicked)
        self.form.bump_check_repeat.valueChanged.connect(self.on_bump_check_repeat_valueChanged)
        self.form.retry_threshold.valueChanged.connect(self.on_retry_threshold_valueChanged)
        self.form.delta_threshold.valueChanged.connect(self.on_delta_threshold_valueChanged)
        self.form.z_variance_threshold.valueChanged.connect(self.on_z_variance_threshold_valueChanged)

        #data_base
        self.form.save_db.clicked.connect(self.on_save_db_clicked)
        self.form.loop_enable.clicked.connect(self.on_loop_enable_Clicked)
        self.form.play_enable.clicked.connect(self.on_play_enable_Clicked)
        self.form.real_time_enable.clicked.connect(self.on_real_time_enable_Clicked)
        self.form.start_point.valueChanged.connect(self.on_start_point_valueChanged)
        self.form.steps.valueChanged.connect(self.on_steps_valueChanged)
        self.form.predict_enable.clicked.connect(self.on_predict_enable_clicked)
        self.form.table_enable.clicked.connect(self.on_table_enable_clicked)

        #wiz
        self.form.model.editingFinished.connect(self.on_model_editingFinished)
```

```
#tuner_steps
for i in range(config.ins.string_num):
    getattr(self.form, f"tuner_enable_{i}").clicked.connect(partial(self.on_tuner_enable_clicked, i))
    getattr(self.form, f"tuner_pos_{i}").valueChanged.connect(partial(self.on_tuner_pos_valueChanged, i))
    getattr(self.form, f"root_{i}").valueChanged.connect(partial(self.on_root_valueChanged, i))
    getattr(self.form, f"follow_{i}").valueChanged.connect(partial(self.on_follow_valueChanged, i))
    getattr(self.form, f"turns_{i}").valueChanged.connect(partial(self.on_turns_valueChanged, i))
    getattr(self.form, f"tolerance_{i}").valueChanged.connect(partial(self.on_tolerance_valueChanged, i))
    getattr(self.form, f"fun_{i}").clicked.connect(partial(self.on_fun_clicked, i))
    getattr(self.form, f"tuner_accel_{i}").valueChanged.connect(partial(self.on_tuner_accel_valueChanged, i))
    getattr(self.form, f"tuner_speed_{i}").valueChanged.connect(partial(self.on_tuner_speed_valueChanged, i))
    getattr(self.form, f"tuner_min_pos_{i}").valueChanged.connect(partial(self.on_tuner_min_pos_valueChanged, i))
    getattr(self.form, f"tuner_max_pos_{i}").valueChanged.connect(partial(self.on_tuner_max_pos_valueChanged, i))


#x_step
self.form.x_pos.valueChanged.connect(self.on_x_pos_valueChanged)
self.form.left_limit.valueChanged.connect(self.on_left_limit_valueChanged)
self.form.right_limit.valueChanged.connect(self.on_right_limit_valueChanged)
self.form.x_enable.clicked.connect(self.on_x_enable_clicked)
self.form.x_accel.valueChanged.connect(self.on_x_accel_valueChanged)
self.form.x_speed.valueChanged.connect(self.on_x_speed_valueChanged)

#z_steps
for i in range(config.ins.string_num):
    getattr(self.form, f"z_pos_in_{i}").valueChanged.connect(partial(self.on_z_pos_in_valueChanged, getattr(self.form, f"z_pos_in_{i
    getattr(self.form, f"z_pos_out_{i}").valueChanged.connect(partial(self.on_z_pos_out_valueChanged, getattr(self.form, f"z_pos_out
    getattr(self.form, f"z_in_enable_{i}").clicked.connect(partial(self.on_z_in_enable_clicked, i))
    getattr(self.form, f"z_out_enable_{i}").clicked.connect(partial(self.on_z_out_enable_clicked, i))
    getattr(self.form, f"z_min_pos_{i}").valueChanged.connect(partial(self.on_z_min_pos_valueChanged, i))
    getattr(self.form, f"z_max_pos_{i}").valueChanged.connect(partial(self.on_z_max_pos_valueChanged, i))
    getattr(self.form, f"min_thresh_{i}").valueChanged.connect(partial(self.on_min_thresh_valueChanged, i))
    getattr(self.form, f"min_voice_{i}").valueChanged.connect(partial(self.on_min_voice_valueChanged, i))
    getattr(self.form, f"max_thresh_{i}").valueChanged.connect(partial(self.on_max_thresh_valueChanged, i))
    getattr(self.form, f"max_voice_{i}").valueChanged.connect(partial(self.on_max_voice_valueChanged, i))
    getattr(self.form, f"down_step_{i}").valueChanged.connect(partial(self.on_down_step_valueChanged, i))
    getattr(self.form, f"up_step_{i}").valueChanged.connect(partial(self.on_up_step_valueChanged, i))
    getattr(self.form, f"z_accel_in_{i}").valueChanged.connect(partial(self.on_z_accel_in_valueChanged, i))
    getattr(self.form, f"z_accel_out_{i}").valueChanged.connect(partial(self.on_z_accel_out_valueChanged, i))
    getattr(self.form, f"z_speed_in_{i}").valueChanged.connect(partial(self.on_z_speed_in_valueChanged, i))
    getattr(self.form, f"z_speed_out_{i}").valueChanged.connect(partial(self.on_z_speed_out_valueChanged, i))

# All Strings tab controls (index 6) - always connect regardless of string count
if hasattr(self.form, "max_voice_6"):
    self.form.max_voice_6.valueChanged.connect(partial(self.on_max_voice_valueChanged, 6))
if hasattr(self.form, "min_voice_6"):
    self.form.min_voice_6.valueChanged.connect(partial(self.on_min_voice_valueChanged, 6))
if hasattr(self.form, "max_thresh_6"):
    self.form.max_thresh_6.valueChanged.connect(partial(self.on_max_thresh_valueChanged, 6))
if hasattr(self.form, "min_thresh_6"):
    self.form.min_thresh_6.valueChanged.connect(partial(self.on_min_thresh_valueChanged, 6))
if hasattr(self.form, "down_step_6"):
    self.form.down_step_6.valueChanged.connect(partial(self.on_down_step_valueChanged, 6))
if hasattr(self.form, "up_step_6"):
    self.form.up_step_6.valueChanged.connect(partial(self.on_up_step_valueChanged, 6))
if hasattr(self.form, "z_min_pos_6"):
    self.form.z_min_pos_6.valueChanged.connect(partial(self.on_z_min_pos_valueChanged, 6))
if hasattr(self.form, "z_max_pos_6"):
    self.form.z_max_pos_6.valueChanged.connect(partial(self.on_z_max_pos_valueChanged, 6))
if hasattr(self.form, "z_accel_in_6"):
    self.form.z_accel_in_6.valueChanged.connect(partial(self.on_z_accel_in_valueChanged, 6))
if hasattr(self.form, "z_accel_out_6"):
    self.form.z_accel_out_6.valueChanged.connect(partial(self.on_z_accel_out_valueChanged, 6))
if hasattr(self.form, "z_speed_in_6"):
    self.form.z_speed_in_6.valueChanged.connect(partial(self.on_z_speed_in_valueChanged, 6))
if hasattr(self.form, "z_speed_out_6"):
    self.form.z_speed_out_6.valueChanged.connect(partial(self.on_z_speed_out_valueChanged, 6))
if hasattr(self.form, "z_in_enable_6"):
    self.form.z_in_enable_6.clicked.connect(partial(self.on_z_in_enable_clicked, 6))
if hasattr(self.form, "z_out_enable_6"):
    self.form.z_out_enable_6.clicked.connect(partial(self.on_z_out_enable_clicked, 6))

#commands
self.form.command.returnPressed.connect(self.on_command_returnPressed)

#command_list
self.form.command_list.itemClicked.connect(self.on_command_list_itemClicked)

#z_step_size
self.form.z_step_size.itemClicked.connect(self.on_z_step_size_itemClicked)

#log
# self.form.input_select.itemClicked.connect(self.on_input_select_itemClicked)
# self.form.carriage_coil_phase_0.clicked.connect(self.on_carriage_coil_phase_0_clicked)
# self.form.carriage_coil_phase_1.clicked.connect(self.on_carriage_coil_phase_1_clicked)
# self.form.fixed_coil_phase_0.clicked.connect(self.on_fixed_coil_phase_0_clicked)
# self.form.fixed_coil_phase_1.clicked.connect(self.on_fixed_coil_phase_1_clicked)
# self.form.piezo_left_level.valueChanged.connect(self.on_piezo_left_level_valueChanged)
# self.form.piezo_right_level.valueChanged.connect(self.on_piezo_right_level_valueChanged)
# self.form.carriage_coil_level.valueChanged.connect(self.on_carriage_coil_level_valueChanged)
# self.form.fixed_coil_level.valueChanged.connect(self.on_fixed_coil_level_valueChanged)
# self.form.input_left_level.valueChanged.connect(self.on_input_left_level_valueChanged)
```

```
        # self.form.input_right_level.valueChanged.connect(self.on_input_right_level_valueChanged)


    # ------------------------------------------------------------------
    # Internal helper to refresh all Z-position widgets (spin-boxes + sliders)
    # ------------------------------------------------------------------
    def _refresh_z_widgets(self):
        """Update every z_pos_* widget from the authoritative stepper.position."""

        for i in range(config.ins.string_num):
            # Z-in (even index)
            idx_in = i * 2
            if idx_in < len(config.ins.z_steps):
                z_step_in = config.ins.z_steps[idx_in]
                if z_step_in and z_step_in.local_index is not None:
                    getattr(self.form, f"z_pos_in_{i}").setValue(z_step_in.position)
                    getattr(self.form, f"slider_z_pos_in_{i}").setValue(int(round(z_step_in.position)))

            # Z-out (odd index)
            idx_out = i * 2 + 1
            if idx_out < len(config.ins.z_steps):
                z_step_out = config.ins.z_steps[idx_out]
                if z_step_out and z_step_out.local_index is not None:
                    getattr(self.form, f"z_pos_out_{i}").setValue(z_step_out.position)
                    getattr(self.form, f"slider_z_pos_out_{i}").setValue(int(round(z_step_out.position)))

def init():
    """One-time initialisation that must run *after* the Qt application object exists."""
    global make_change, _db_writer_thread, x_position, z_positions, tuner_positions, z_step_size, left_limit, right_limit

    # Ensure the instrument is fully initialised before sizing the position arrays
    config.ins.steps_init()

    #initialize z_step_size
    z_step_size = 1.0

    #initialize left_limit and right_limit
    left_limit = config.ins.x_step.min_pos + 100
    right_limit = config.ins.x_step.max_pos - 100

    # --- DEBUG: Prepare initial stepper enable states message ---
    enable_states_msg = "\n--- Initial Stepper Enable States ---"
    for stepper in config.ins.all_steps:
        if stepper:
            state = "ENABLED" if stepper.enable else "DISABLED"
            port = stepper.arduino.com_port if hasattr(stepper, 'arduino') and stepper.arduino else 'N/A'
            enable_states_msg += f"\nStepper {stepper.index} ({type(stepper).__name__} on {port}): {state}"
    enable_states_msg += "\n-----------------------------------"

    # --- FIX: Create two separate, correctly sized position arrays ---
    x_stepper = config.ins.x_step
    z_steppers = [s for s in config.ins.z_steps if s and s.arduino == config.ard]
    tuner_steps = [s for s in config.ins.all_steps if s and s.arduino == config.ard_T]

    # Size arrays by count, not by max local_index – avoids sparse arrays on -1
    z_positions = [0] * len(z_steppers)
    tuner_positions = [0] * len(tuner_steps)

    # Create the command dispatcher
    make_change = TextInputCommands()

    # --- DEBUG: Add the stored message now that make_change exists ---
    make_change.message += enable_states_msg
    make_change.message += f"\nInitialized z_positions with size: {len(z_positions)}"
    make_change.message += f"\nInitialized tuner_positions with size: {len(tuner_positions)}"

    # Ensure the Rust backend supervisor is running before the GUI needs audio data.
    ensure_persist_monitor()

    # --- Reset tuner positions on startup only if tuner board exists ---
    if config.ard_T.exist:
        make_change.message += "\n\n--- Resetting Tuner Positions ---"
        reset_msg = config.ard_T.reset_all()
        if reset_msg:
            make_change.message += reset_msg
        make_change.message += "\nTuner positions reset to zero"
        make_change.message += "\n-----------------------------------"

    # Start the database writer thread
    _db_writer_thread = DbWriter(event_q)
    _db_writer_thread.start()

    # Register signal handler for clean exit
    signal.signal(signal.SIGINT, signal_handler)

    # Ensure the database table exists
    ensure_table(config.db.data_table, makeheaders())

    print("System initialized successfully")
    # --- Diagnostic summary ---
    connected_steps = [s for s in config.ins.all_steps if s and s.arduino and getattr(s.arduino, "exist", False)]
    if connected_steps:
        print(f"Total steppers detected: {len(connected_steps)}")
        print(f"Z positions array size: {len(z_positions)}")
        if tuner_positions:
            print(f"Tuner positions array size: {len(tuner_positions)}")
        for stepper in connected_steps:
            idx = stepper.index if stepper.index is not None else "?"
            port = stepper.arduino.com_port if hasattr(stepper.arduino, "com_port") else "?"
            print(f"  Stepper {idx}: {stepper.__class__.__name__} on {port}")
    else:
        print("No stepper hardware detected on this host (monitor-only mode).")

    # Decide if this host runs in monitor-only mode (no strings, no motors)
    global monitor_only
    monitor_only = (config.ins.string_num == 0 and not (config.ard.exist or config.ard_T.exist))

# ----------------------------
# Stage-1: Background DbWriter setup
# ----------------------------

# Flush queued rows to storage
def _flush_rows(rows: List[list]):
    for row in rows:
        if config.db.save_csv:
            write_csv(row)
        if config.db.save_db:
            pg_steppers.log_stepper_states(config.ins.all_steps, command=command if 'command' in globals() else None)

# Start the background writer thread at import time (singleton)
if '_db_writer_thread' not in globals():
    _db_writer_thread = DbWriter(flush_callback=_flush_rows)
    _db_writer_thread.start()

# ----------------------------
# Stage-2: Hardware Worker setup
# ----------------------------
# NOTE: The worker thread requires a running Q(Core)Application; starting it
# before QApplication is constructed triggers warnings such as
# "QSocketNotifier: socket notifiers cannot be enabled".  We now create the
# worker *after* QApplication has been instantiated (see __main__ section
# below).
_hw_worker = None  # will be initialised lazily once QApplication exists
_hw_thread = None

# Flag to avoid piling up concurrent get_positions() requests
_getpos_inflight = False

# Track last GUI message to avoid spamming duplicates
# _last_gui_msg = ''  # deprecated – no longer used

# One-shot flags to avoid spamming the same status messages
_no_arduino_warned = False
_bump_disabled_warned = False

# Convenience wrappers that dispatch move commands to the worker thread
from PyQt5 import QtCore

def hw_amove(stepper_ref, target: int):
    """Absolute move using direct Arduino calls."""
    from instrument import Stepper as _Stepper

    if not isinstance(stepper_ref, _Stepper):
        raise RuntimeError("abs_move requires a Stepper object, not an int")
    stepper = stepper_ref
    if not stepper:
        return
    if stepper.arduino:
        target = max(stepper.min_pos, min(stepper.max_pos, target))

        # Use Arduino's amove command for absolute positioning
        msg = stepper.arduino.amove(stepper.local_index, target)
        if msg:
            make_change.message += msg
        # Update cached physical position
        stepper.position = target


def hw_rmove(stepper_ref, delta: int):
    """Relative move using direct Arduino calls."""
    from instrument import Stepper as _Stepper

    if isinstance(stepper_ref, _Stepper):
        stepper = stepper_ref
    else:
        stepper = _find_stepper(stepper_ref)

    if not stepper or not stepper.arduino:
        make_change.message += f"\nhw_rmove failed – bad stepper {stepper_ref}"
        return

    msg = stepper.arduino.rmove(stepper.local_index, delta)
    if msg:
        make_change.message += msg

    # Update Python position
    stepper.position = max(stepper.min_pos, min(stepper.max_pos, stepper.position + delta * z_step_size))

# ------------------------------------------------------------------
# Legacy helper used while migrating away from raw index look-ups.
# Returns the Stepper object whose .index matches *idx* or None.
# Callers MUST use stepper.arduino and stepper.index for routing.
# ------------------------------------------------------------------
def _find_stepper(idx: int):
    """Compatibility helper: map integer index -> Stepper object.

    WARNING: This helper exists ONLY to keep older text-command paths running
    while we finish migrating every call-site to pass a Stepper instance
    directly.  It must *never* be used to decide which Arduino to talk to –
    the returned Stepper already holds the correct ``.arduino`` reference and
    that (board, local_index) pair is the sole routing source of truth.

    Once grep of the codebase shows **zero** remaining usages of
    ``_find_stepper`` this function can be removed for good.
    """

    # Lazy import to avoid circular dependency during module load.
    import config as _cfg

    for s in getattr(_cfg, 'ins', None).all_steps if hasattr(_cfg, 'ins') else []:
        if s is not None and getattr(s, 'index', None) == idx:
            return s

    # No match – emit a gentle warning instead of crashing hard so that the
    # caller can handle ``None`` and report a user-friendly error.
    import warnings, traceback as _tb, os
    warnings.warn(f"_find_stepper: no stepper with index {idx}", RuntimeWarning)
    # Print traceback once for easier debugging (opt-in by env var to reduce spam)
    if os.getenv("DEBUG_FIND_STEPPER_TRACE") == "1":
        _tb.print_stack(limit=5)
    return None

if __name__ == '__main__':
    # Create the GUI application instance. This MUST be done before creating
    # any other Qt-dependent components (like the hardware worker).
    gui = GuiInput(ui_file='String_Driver_trim.ui')

    # Show the main window
    gui.window.show()

    # Now that the Qt event loop is running, we can safely initialize the rest
    # of the system.
    init()

    # Force the same quit path when the last window is closed (user clicks X)
    gui.app.lastWindowClosed.connect(lambda: make_change.change_quit(True))

    # Run the main event loop
    sys.exit(gui.app.exec_())
```