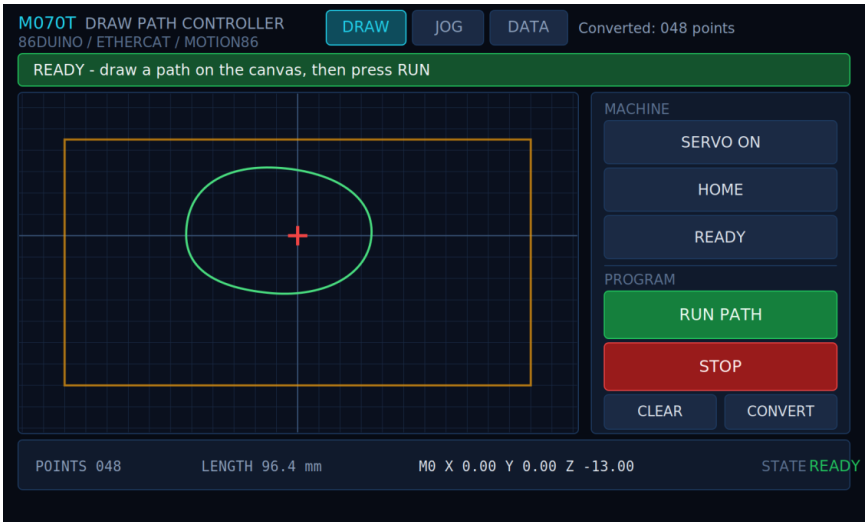
ICOP TECHNOLOGY · QEC START GUIDE

[Start Guide] QEC-M-070T × Motion86 — Draw a Path on the Touch Screen, Three Axes Run It



Hardware	QEC-M-070T + QEC-R11MP3S
Environment	86Duino IDE
Example sketch	Motion86_DrawPath
Language	English

Contents

※ If the page numbers are blank, press Ctrl+A then F9 in Word to update the fields.

Introduction

This start guide shows how to build a complete **three-axis path-drawing controller** on one **QEC-M-070T**. The operator draws a line on the 7" touch screen with a finger; the controller smooths the stroke, turns it into a list of waypoints, and hands it to the three axes of one **QEC-R11MP3S** to draw in real time.

The touch screen, the EtherCAT main device, the path planning and the position logging all run on the same board. No external PC, no PLC, no separate motion control card.

About the two extra motor groups in this example

Two more QEC-R11MP3S modules are wired in, mirroring the first one's motion. **They are not part of the application — they are there to verify synchronised control.**

1. Quick Overview

- **Freehand Path Generation —** Draw a path directly on the canvas. When the drawing is completed, the program automatically converts the trajectory into a waypoint list of up to 600 points, without requiring any predefined shape.
- **Continuous Three-Axis Motion —** The X, Y, and Z axes follow the generated trajectory continuously. All motion commands are defined in millimetres, eliminating the need for manual pulse conversion.
- **Smooth Path Execution — U**pcoming waypoints are queued in a motion buffer, while Motion86 blends adjacent path segments to maintain continuous movement throughout the trajectory.
- **Real-Time Limit Switch Protection —** Limit switches are monitored during every control cycle. Once triggered, the system immediately performs an emergency stop and remains latched until the activated switch is released.
- **Three operating pages** — **Draw** for path creation, **Jog** for manual axis control, and **Data** for monitoring waypoint and motion status.

2. What You Need

Item	Notes
QEC-M-070T	EtherCAT main device with a built-in 7" 800 × 480 touch panel
QEC-R11MP3S × 1	**Primary mechanism**: three-axis CiA402 servo module, draws the path
QEC-R11MP3S × 2	**Optional**: for the synchronisation check; does not affect the draw path itself
24 V supply	Vs for the system, Vp for the peripherals
Limit switches	Wired to a digital input on each axis of the primary module
Development environment	86Duino IDE

The program calls the three machine groups M0, M1 and M2.

M0 is the primary mechanism and the subject of this guide: it has the home switches, real travel limits, and it is the one that draws your path.

3. Feature Overview

3.1 Draw — drawing and live position

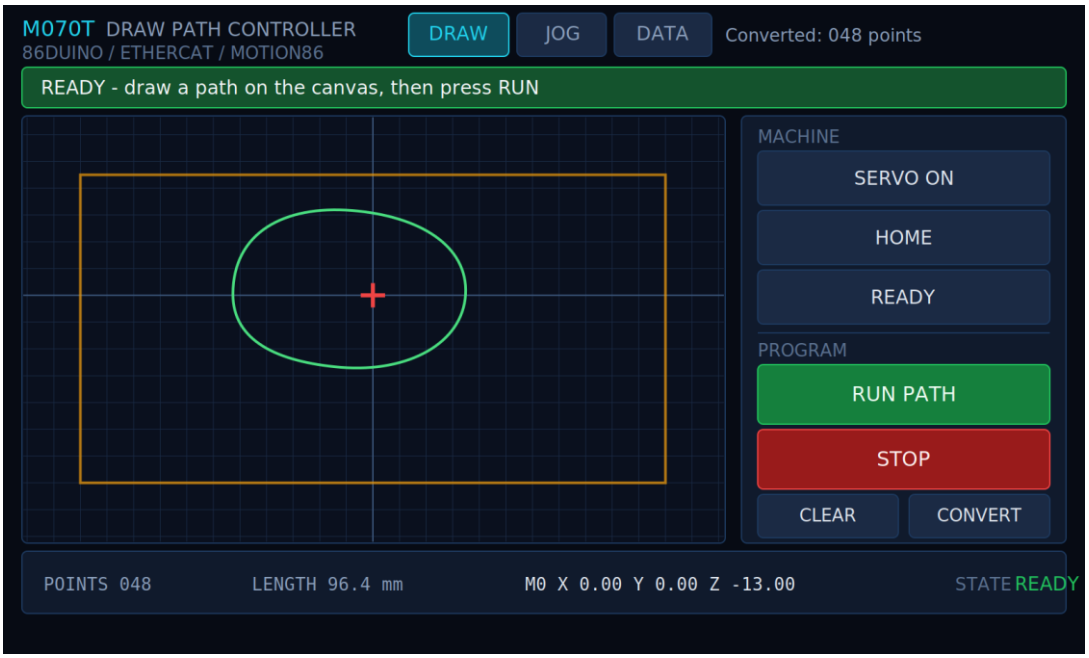


Figure 1 Draw page: grid canvas, amber safe-draw box, green converted path, red live position cursor, persistent control column on the right

Things you will see on this page:

- **Amber box** — the safe drawing area. Touches outside it are discarded before they are converted into waypoints, which keeps the operator from drawing a path that risks a crash. The box is the real travel inset by 25 mm (`DRAW_XY_BUFFER_MM`).
- **Green line** — the converted path. It is calculated as soon as you lift your finger, with no extra button press needed (if you want to recalculate the path, there is a `CONVERT` button at the bottom right).
- **Red crosshair** — where the machine actually is. It appears once the motors are enabled, and lets the operator monitor the position live.
- **Status bar at the bottom** — point count, total path length, M0's live XYZ, and a state chip on the right (`OFF` / `ON` / `READY` / `RUN` / `JOG` / `LIMIT`).

3.2 The Draw page banner

The banner across the top of the screen tells the operator which button to press next.

Colour	Message	Meaning
Blue	`STEP 1 - press SERVO to enable the drives`	Motors not enabled yet
Blue	`STEP 2 - press HOME, or press READY to zero at this position`	No origin coordinate established yet
Green	`READY - draw a path on the canvas, then press RUN`	Ready to draw a path
Amber	`PATH RUNNING (M0 + M1 + M2) - press STOP to abort`	Running; press STOP to halt the drawing task
Red	`LIMIT LATCHED - release the switch, then press CLEAR`	A limit switch was hit; motion is latched off

3.3 The Jog page

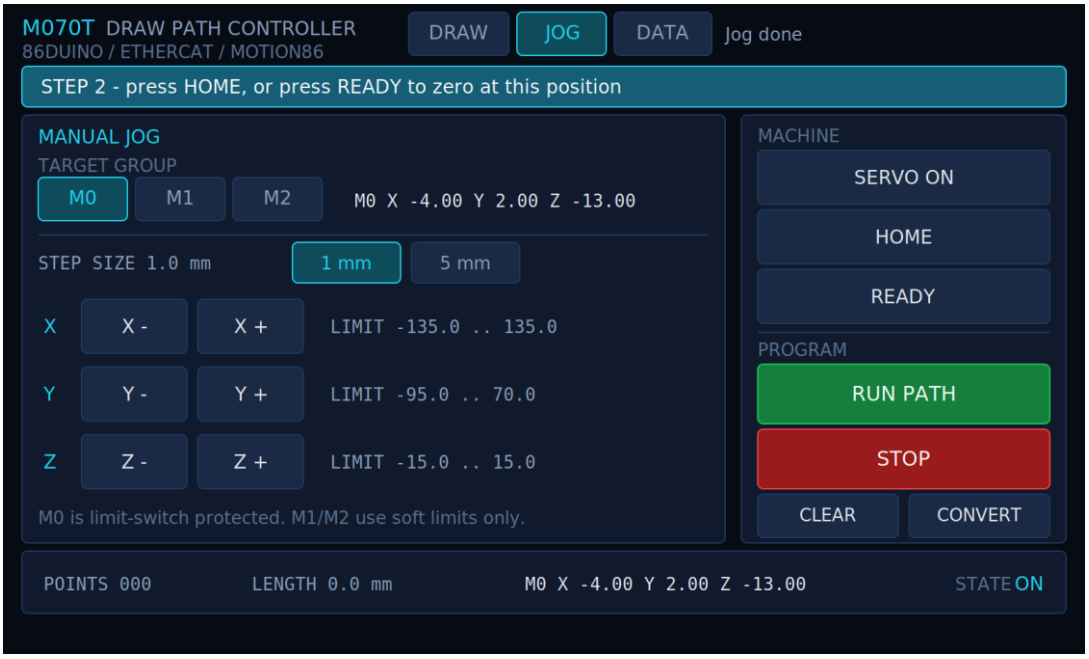


Figure 2 Jog page: machine group selector, step size, plus/minus buttons for X, Y and Z with their travel limits

Pick the group at the top (M0 / M1 / M2), pick the step size (1 mm or 5 mm), then press `X-` `X+` `Y-` `Y+` `Z-` `Z+`.

The right-hand side shows the live position of **the selected group** and each axis's travel range:

```
X    X -    X +     LIMIT  -135.0 .. 135.0
Y    Y -    Y +     LIMIT   -95.0 ..  70.0
Z    Z -    Z +     LIMIT   -15.0 ..  15.0
```

Select M1 or M2 and the range becomes `-1000 .. 1000` — with no home switches their absolute position is undefined, so all they get is a generous soft limit. The page says as much at the bottom: *M0 is limit-switch protected. M1/M2 use soft limits only.*

Worth noticing: **a jog is not sent to the drive as a raw command.** The program first constrains the target to the allowed range, then issues an absolute position command:

```
x = clampDouble(x + jogDeltaMm, xMin, xMax);
mach->line(x, y, z, JOG_FEEDRATE);
```

3.4 The Data page

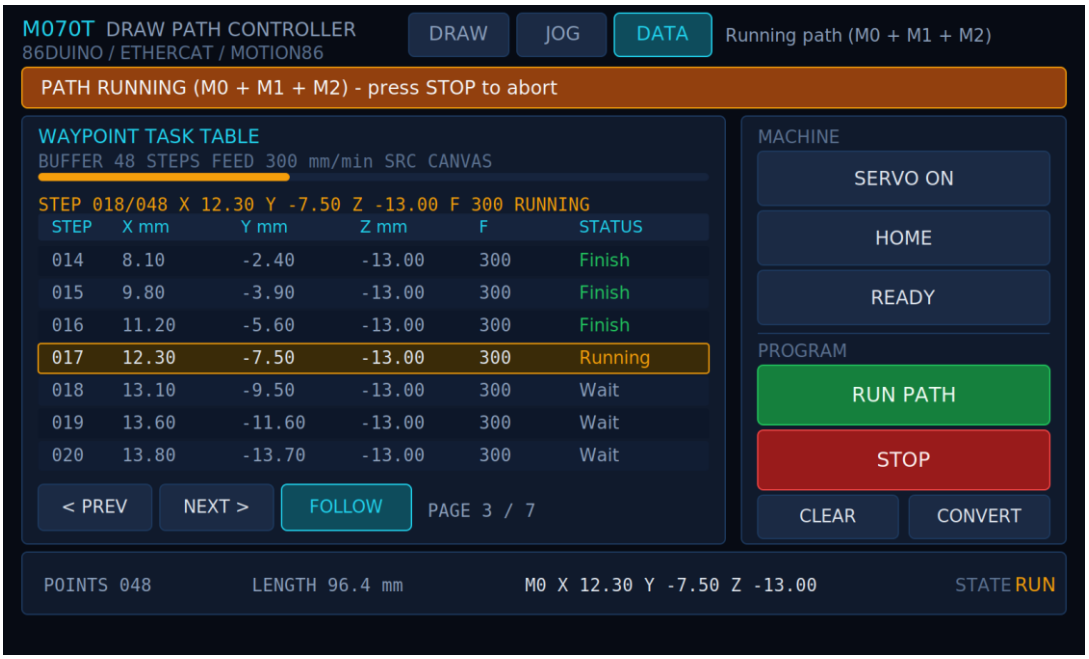


Figure 3 Data page: waypoint table, progress bar, the currently executing row highlighted in amber

This page lays the whole path out: step number, target X / Y / Z, feedrate, status.

Every step passes through three states:

Status	Colour	Meaning
Wait	Grey	Waiting
Running	Amber	Executing
Finish	Green	Done

Seven rows per page, with `< PREV` / `NEXT >` to page through. `FOLLOW` is auto-follow: while it is on, the page jumps to whichever step is executing; page manually and it switches itself off.

Above the table there is a progress bar and a one-line summary:

```
STEP 018/048   X   12.30  Y   -7.50  Z  -13.00  F 300   RUNNING
```

When the machine draws something other than what you expected, this page tells you whether the problem is in the coordinate conversion or in the mechanism and its parameters.

Every Z in the table is −13 mm, because Z does not take part in the drawing in this example (`PARK_Z_AT_HOME_LIMIT` is true); what gets replayed is the shape in the X / Y plane.

4. First End-to-End Run

Step 1: Press SERVO ON (twice)

First press — starts the EtherCAT main device, scans the three modules, and completes the configuration of the three `Machine` objects. The status line prints `ECAT: master.begin`, `ECAT: attach MP3S 0/1/2` and `ECAT: master.start` in turn, then shows `Press SERVO again to enable`.

Second press — switches each group's three axes to CSP mode and enables them, then calls `machineOn()`. The status line shows `M070T servo ready`, the state chip turns `ON`, and the red crosshair appears on the Draw page.

The two-press sequence is deliberate: the first press only initialises the network, so you can confirm the modules were all found before deciding to power the motors.

Step 2: Press HOME (optional)

M0 seeks its X / Y / Z home switches in turn, establishes the coordinate system with one line of `G92`, then backs off 3 mm to leave the switches. During the sequence the banner turns amber and reads `HOMING IN PROGRESS - do NOT touch any control`.

When it finishes, the status line shows `M0 home done. Press READY.`

This step can be skipped — if the machine is already sitting at a working origin, just press READY. How it works and what to watch out for is in 5.4.

Step 3: Press READY

M0 moves to the origin at (0, 0, −13), ready to start drawing.

M1 and M2, the two verification groups, define their current position as their starting point; every displacement from then on is counted from there.

The banner turns green and reads `READY - draw a path on the canvas, then press RUN`; the state chip turns `READY`.

Step 4: Draw on the canvas with a finger

Draw a shape inside the amber box. The moment you lift your finger the program does three things automatically: smooth the stroke → resample it at 2 mm spacing → build the waypoint list. The status line shows `Converted: 048 points`, the green path appears on the canvas, and the Data page table fills in at the same time.

Step 5: Press RUN PATH

M0 works through the waypoint list, with X / Y / Z moving continuously. The banner turns amber and the state chip turns `RUN`.

Switch to the **Data page** to watch it step through: the progress bar advances, the executing row is highlighted in amber, and the page follows along to whichever page is running.

At the same time, M1 and M2 reproduce each of M0's displacements. **This is the point to watch for synchronised control**: the M1 and M2 mechanisms should be moving in step.

When everything has run, all three return to the working origin — M0 to the origin, M1 and M2 to where `G92` zeroed them. The status line shows `Task done. M0/M1/M2 at zero.`

Step 6: STOP is available at any time

STOP halts all the motors immediately and clears every request that has not yet been executed.

5. What Motion86 Does

Everything above sits on surprisingly little code. This section is what the guide is really about.

5.1 Command in millimetres, no pulse arithmetic

Set "pulses per millimetre" once, and every command afterwards is in millimetres:

```
machine0.config_PPU(AXIS_X, 1600.0);   // 1 mm = 1600 pulses
machine0.config_PPU(AXIS_Y, 1600.0);
machine0.config_PPU(AXIS_Z, 1600.0);
machine0.config_ReverseDirection(AXIS_X);   // flip an axis that runs backwards
```

After that, `machine0.line(12.30, -7.50, -13.00, 300)` means "go to X=12.3, Y=−7.5, Z=−13 at 300 mm/min."

Change the gear ratio, the screw pitch or the encoder resolution and all you edit is the number in `config_PPU()`.

5.2 Two callbacks, and any drive is connected

The EtherCAT library gives you **drive objects** that speak in encoder pulses: `setTargetPosition()`, `getPositionActualValue()`.

Motion86 gives you a `Machine` **object** that speaks in millimetres, feedrates and G-code.

Two small functions join them:

```
void setRemoteTargetPos0(int axis, long pos) {
  if (servo[0][axis]) servo[0][axis]->setTargetPosition(pos);
}
long getRemoteTargetPos0(int axis) {
  if (servo[0][axis]) return servo[0][axis]->getPositionActualValue();
  return 0;
}

machine0.config_CustomActuators_CycleTime(ECAT_CYCLE_TIME);
machine0.config_CustomActuators_Callbacks(setRemoteTargetPos0, getRemoteTargetPos0);
```

Motion86's part is to "give a position" and "read one back".

Why this is worth having: moving to different motors later, or running the whole flow against simulated axes first, mainly means rewriting these two functions.

5.3 Out-of-travel commands get blocked

```
machine0.config_PosLimit(AXIS_X, -135.0, 135.0);
machine0.config_PosLimit(AXIS_Y,  -95.0,  70.0);
machine0.config_PosLimit(AXIS_Z,  -15.0,  15.0);
machine0.enableSoftLimit();
```

Once this is on, a target outside the range is blocked by Motion86 before any pulse goes out, and `line()` returns false.

You do not write an `if` before every move, and you do not have to worry about a rarely-taken path skipping the check.

5.4 Homing is built in — but you decide how the switch is read

Motion86 runs the homing sequence; you only tell it how to know whether a switch is made:

```
int myMachine0_HomeFeat(homeCommand_t feat) {
  if (feat == READ_HOME_X) return servo[0][0]->getDigitalInputs() & 0x01;
  if (feat == READ_HOME_Y) return servo[0][1]->getDigitalInputs() & 0x01;
  if (feat == READ_HOME_Z) return servo[0][2]->getDigitalInputs() & 0x01;
  return 0;
}

machine0.config_HomeController_Callback(myMachine0_HomeFeat);
machine0.setHomeSpeed(600.0);
machine0.home();
```

In practice the I/O wiring differs from application to application — some wire to the drive's DI, some to the main device's IO, some use NPN and some PNP. So Motion86 leaves "how to read the switch" up to you, and an unusual switch arrangement can still use the rest of the library.

Once the switch is found, **one line of G-code establishes the coordinate system**:

```
machine0.gcode("G92 X-135.000 Y-95.000 Z-15.000");
machine0.setAbsolute();
```

`G92` does not move anything. It simply tells Motion86: "the position you are standing in right now is called this."

That is also why Step 2 can be skipped:

```
if (!machineHomed && ALLOW_READY_WITHOUT_HOME) {
  machine0.gcode("G92 X0.000 Y0.000 Z-13.000");
  machine0.setAbsolute();
  machineHomed = true;
}
```

There is a trade-off to keep in mind: when the HOME button is skipped, the soft limits are calculated from your current position.

5.5 Turning a list of waypoints into one continuous move

A path can hold 600 points, or more. You do **not** send one, wait for it to finish and send the next (that would make the machine judder continuously). You keep the buffer topped up:

```
while (i < pathCount && !motionStopRequested) {
  MachinePoint p = clampToM0CommandBounds(pathPoints[i]);

  if (anyEnabledCmdBufferFull()) {   // buffer full: retry the same point shortly
    yield();
    sleep(1);
    continue;
  }

  machine0.line(p.x, p.y, p.z, p.feed);          // M0: absolute
  mirrorRelativeMove(p.x - prevX, p.y - prevY,   // M1, M2: relative
                     p.z - prevZ, p.feed);
  prevX = p.x; prevY = p.y; prevZ = p.z;
  i++;
}
```

`isCmdBufferFull()` and `getCmdCount()` make the state of the queue visible, so your code can keep pushing while Motion86 keeps pulling. Because Motion86 joins adjacent segments automatically, a 48-point path runs as **one continuous move** rather than 48 stop-starts.

`getCmdCount()` also hides a useful trick. The number of commands not yet executed is exactly the gap between "how far you have queued" and "where the axes actually are":

```
int pending   = machine0.getCmdCount();
int executing = runSentIndex - pending;
```

That single line is how the Data page knows which row to mark Running, with no extra bookkeeping.

5.6 One main device controlling three mechanisms

Everything up to here has been about M0 and its three axes. This section is about what the other two groups in the example are really there to demonstrate.

Adding a mechanism means adding an object:

```
Machine machine0(CUSTOM_MACHINE_0);
Machine machine1(CUSTOM_MACHINE_1);
Machine machine2(CUSTOM_MACHINE_2);
```

The three objects are updated in turn inside the same real-time callback, sharing one 500 µs network cycle:

```
void MyCyclicCallback() {
  machine0.updateAxes();
  machine1.updateAxes();
  machine2.updateAxes();
}
```

The key point is that these are three independent coordinate systems. You can give them three different paths, written the same way as here — only the coordinates fed to `line()` would differ.

So why does the example make them do the same thing? **Because that makes it easier to see whether they are in sync.** When three groups run different paths it is hard to judge by eye whether they are keeping the same beat; when they draw the same shape, lag or judder is easier to spot.

The example does this with relative mode: M0 runs absolute coordinates while M1 and M2 are switched to relative, so each command repeats a **displacement** rather than a destination. That is why they need neither their own home switches nor alignment with M0's machine coordinates:

```
machine0.setAbsolute();
machine1.setRelative();
machine2.setRelative();
```

It also means M1 and M2 can only be given soft limits of ±1000 mm, which offers no real protection. That is workable in this example because they only copy M0's displacements, and M0's travel is already bounded. If you want them to run their own paths, adding home switches and correct travel limits is the place to start.

5.7 G-code support

`gcode()` can be called at any time and mixed freely with `line()`. This example only uses `G92` (defining the coordinate of the current position), but the same entry point accepts `G0` / `G1` / `G2` / `G3`.

Which means: if part of your project comes out of CAM as G-code, there is nothing extra to write — hand it to `gcode()`.

5.8 How much Motion86 saves you

The job	Rolling your own	With Motion86
mm → pulse	Per-axis conversion, accumulating rounding error	`config_PPU()`, once
Three axes on one straight line	Split the vector yourself, keep the three axes in step	`line(x, y, z, f)`
Acceleration profiles	Write your own S-curve or trapezoid	Built in
Travel protection	Check by hand before every command	`enableSoftLimit()`
Continuous paths	Implement look-ahead and corner speeds yourself	Built-in buffer, `isCmdBufferFull()` to inspect
Homing	Write the seek / decelerate / back-off state machine	`home()` plus one switch-reading callback
Changing drive brand	Most of it gets rewritten	Rewrite two callbacks

6. How the Code Fits Together

6.1 Five files

```
Motion86_DrawPath/
├── Motion86_DrawPath.ino   main program
├── config.h                every tunable parameter
├── app_state.h             what the two modules share
├── ui.h                    UI declarations
└── ui.cpp                  UI screen objects
```

`ui.cpp` issues no motion commands at all. A button does not call `machine0.home()` directly; it raises a request, and the motion task takes it from there:

```
} else if (obj == homeBtn) {
  requestMotion(REQ_HOME);
} else if (obj == stopBtn) {
  requestStop();
} else if (obj == jogXMinusBtn) {
  requestJogMove(AXIS_X, -jogStepMm);
}
```

Everything that crosses between the two lives in `app_state.h`, short enough to read in one go:

```
// implemented in the .ino, called from ui.cpp
void requestMotion(MotionRequest req);
void requestJogMove(machineAxis_t axis, double deltaMm);
void requestStop();

// implemented in ui.cpp, called from the .ino
void uiInit();
void uiRefresh();
void setStatus(const char *text);
void setHint(const char *text, HintLevel level);
```

The practical payoff: **redrawing the screen, changing the colour scheme or adding a fourth page does not require touching the motion code.** On a machine already running in production, being able to confine a change to the UI side makes the risk much smaller.

6.2 Configuring one machine

Here is the three-axis configuration:

```
machine0.config_PPU(AXIS_X, M0_PPU_X);         // pulses per millimetre
machine0.config_PPU(AXIS_Y, M0_PPU_Y);
machine0.config_PPU(AXIS_Z, M0_PPU_Z);
machine0.config_ReverseDirection(AXIS_X);      // flip a backwards axis
machine0.config_PosLimit(AXIS_X, M0_LIMIT_X_MIN, M0_LIMIT_X_MAX);
machine0.config_PosLimit(AXIS_Y, M0_LIMIT_Y_MIN, M0_LIMIT_Y_MAX);
machine0.config_PosLimit(AXIS_Z, M0_LIMIT_Z_MIN, M0_LIMIT_Z_MAX);
machine0.enableSoftLimit();
machine0.setDefaultFeedrate(PATH_FEEDRATE);
machine0.config_HomeController_Callback(myMachine0_HomeFeat);
```

Every uppercase value comes from `config.h`, so running this on a different machine mainly means editing `config.h`; the motion code usually does not have to be opened.

6.3 Running the path-planning maths inside the real-time cycle

Motion86 does its position calculations inside the EtherCAT main device's real-time callback:

```
void MyCyclicCallback() {
  machine0.updateAxes();
  machine1.updateAxes();
  machine2.updateAxes();
  emitPositionTelemetry250Hz();
}

master.attachRTCyclicCallback(MyCyclicCallback, true);
master.start(ECAT_CYCLE_TIME, ECAT_SYNC);   // 500 µs, clock-synchronised
```

Each `updateAxes()` computes that machine's next position and sends it out through the callback.

6.4 How the four tasks divide the work

This program has four things to do at once, split into four tasks running on the same CPU:

Task	Period	Responsibility
`EtherCAT callback`	500 µs	Three `updateAxes()` calls, plus a position sample
`motionTask`	1 ms	Executes Servo / Home / Ready / Jog / Run requests
`hmiTask`	5 ms	Runs LVGL, calls `uiRefresh()` every 100 ms
`telemetryTask`	1 ms	Prints the 250 Hz position data to serial

Of these, the EtherCAT callback is the one that must not be held up — it has to finish inside the 500 µs network cycle. So when it exchanges data with the other tasks it does not use an ordinary lock (a lock means one side may have to stop and wait); it uses a counter instead:

```
telemetrySeq++;              // now odd = write in progress
telemetrySampleX = x;  telemetrySampleY = y;  telemetrySampleZ = z;
telemetrySeq++;              // even again = writing finished
```

The writing side (the EtherCAT callback) only ever increments and never stops to wait. The reading side (`telemetryTask`) judges it like this: check the counter before reading and again afterwards — **if the two numbers match, the data was not changed while it was being read** and is safe to use; if they differ, it simply reads again.

Buttons work on the same idea: pressing one does not call a motion function directly, it leaves a request for `motionTask` to execute one at a time. A few stray taps during homing therefore cannot put the state out of order — `canQueueMotionRequest()` blocks them and writes the reason to the status line.

7. Safety Design

This example splits crash prevention into three layers, each catching a problem at a different stage.

Layer	What it catches	Where
Screen limits	Unreasonable input	`ui.cpp`
Soft limits	Unreasonable commands	`.ino`
Hardware limits	An actual collision	`.ino`

The first layer is at the operator's fingertip. The amber box on the canvas is smaller than the real travel, and touches outside it are discarded before they become waypoints. The problem is handled right at the front, so the rest of the program never receives an out-of-range point.

The second layer is before the command goes out. With `config_PosLimit()` and `enableSoftLimit()`, an out-of-travel target is blocked by Motion86 and no pulse reaches the drive. This layer catches errors on the program side — a wrong parameter, a miscalculated path, or a code change that forgot to check.

The third layer is on the mechanism. The first two layers are software judgements, and they assume the configured values match the real mechanism. When they do not — a mistyped travel parameter, a mechanism that has been moved, or a wrong position assumption after skipping homing — the physical switch is all that is left. The program reads the switches every control cycle, and once one triggers:

```
static void latchM0LimitStop(const char *source) {
  limitSwitchLatched = true;
  motionStopRequested = true;
  stopAllMachines();
  machine0.emgStop();
  setStatus(buf);
  setHint("LIMIT LATCHED - release switch, then press CLEAR", HINT_DANGER);
}
```

`emgStop()` halts motion immediately and sets `limitSwitchLatched` to lock the state. **The latch is deliberate**: a switch being held often bounces, and if recovery relied only on "the switch was released", the machine could start moving again before the operator has reacted. So `clearEMGStop()` runs only after the switch has physically cleared and the operator has pressed `CLEAR`.

That last `setHint()` turns the banner red immediately, without waiting for the next 100 ms screen refresh. The UI lives in another file, but the motion side can still call on it when it needs to

raise an alarm.

8. Tuning

Every speed lives in `config.h`:

```
static const double PATH_FEEDRATE          = 300.0;  // mm/min, path execution
static const double READY_FEEDRATE         = 450.0;  // returning to origin
static const double JOG_FEEDRATE           = 250.0;  // manual jog
static const double HOME_BACKOFF_FEEDRATE  = 200.0;  // backing off after homing
static const double HOME_SEEK_SPEED        = 600.0;  // seeking the switch
static const double PATH_SAMPLE_MM         = 2.0;    // spacing between waypoints
```

Raise `PATH_FEEDRATE` a little at a time and watch the drives' following error. If curves start to look faceted at higher speed, reduce `PATH_SAMPLE_MM` so Motion86 has more points to join — at the cost of how much path fits in the same buffer.

`config.h` also holds the travel limits, the screen layout coordinates and the colour scheme. When moving to a different machine, this one file is usually enough to start from.

9. Troubleshooting

Can I run the draw path with only one QEC-R11MP3S? Yes. First change one line in `config.h`:

```
static const uint8_t ENABLED_MACHINE_GROUPS = 1;   // was 3
```

Then comment out the lines in `initM070TMotion()` that attach the second and third modules (`slave[1].attach(...)`, `slave[2].attach(...)` and the `cia402GetServo()` calls that follow) — those lines are not guarded by `ENABLED_MACHINE_GROUPS`, and the program will stall there if the modules are not on the bus. Nothing else needs to change: waypoint feeding, limit protection, the UI and the Data page progress all keep working.

SERVO seems to do nothing and only shows `Press SERVO again to enable` Expected. The first press only initialises EtherCAT; press it again to enable the drives.

The banner is red with `LIMIT LATCHED` and nothing responds A limit switch is still made. Move the axis off the switch by hand, then press `CLEAR`. The program deliberately requires a physical release before clearing, so a bouncing switch cannot let the machine suddenly start moving again.

RUN reports `Press READY before RUN` `drawReady` is false. Jogging clears that flag, so press READY again after any manual move.

The status line shows `Point outside M0 limit` or `Motion command rejected` A soft limit blocked it. Check that `M0_LIMIT_*` in `config.h` matches the real travel, and that `M0_PPU_*` is correct — with the wrong PPU the millimetre figures look fine while the actual distance is off by a large factor.

Drawing on the canvas produces no line Two possibilities: your finger is outside the amber safe box, where touches are ignored; or the machine is executing a move (`motionBusyForEditing()` is true) and the canvas has temporarily stopped accepting input.

The executed path looks faceted `PATH_SAMPLE_MM` is too large or `PATH_FEEDRATE` is too high. Try reducing the sample spacing to 1.0 mm first.

The verification groups M1 and M2 do not move at all Check that `ENABLED_MACHINE_GROUPS` is 3 in `config.h`, and that READY has been pressed — M1 and M2 are zeroed with `G92` and switched to relative mode during READY, and without that they have no starting point. (M0's draw path is unaffected and completes without them.)

The position in the status bar stays at `----` The servos are not enabled yet (`servoReady` is false). Complete Step 1 first.

No position data on the serial port Check that `ENABLE_POSITION_TELEMETRY` is true and set the

serial monitor's baud rate to `3000000`.

10. Resources

Example program

```
Motion86_DrawPath/
├── Motion86_DrawPath.ino
├── config.h
├── app_state.h
├── ui.h
└── ui.cpp
```

All five files go in one folder named `Motion86_DrawPath`. The 86Duino IDE requires the folder name to match the `.ino` file name.

11. Where to Take It Next

The example leaves a few directions open:

- **Swap the canvas for a file source.** `pathPoints[]` is just an array of waypoints — read a CSV from an SD card or receive G-code over the network, and the execution flow after that does not change.
- **Bring Z into the drawing.** Turn off `PARK_Z_AT_HOME_LIMIT` and give each point its own Z during conversion to get pen-up / pen-down behaviour.
- **Let M1 and M2 run their own paths.** Switch them back from relative to absolute mode and add home switches and travel limits, and they go from being a synchronisation check to three mechanisms genuinely working on their own.