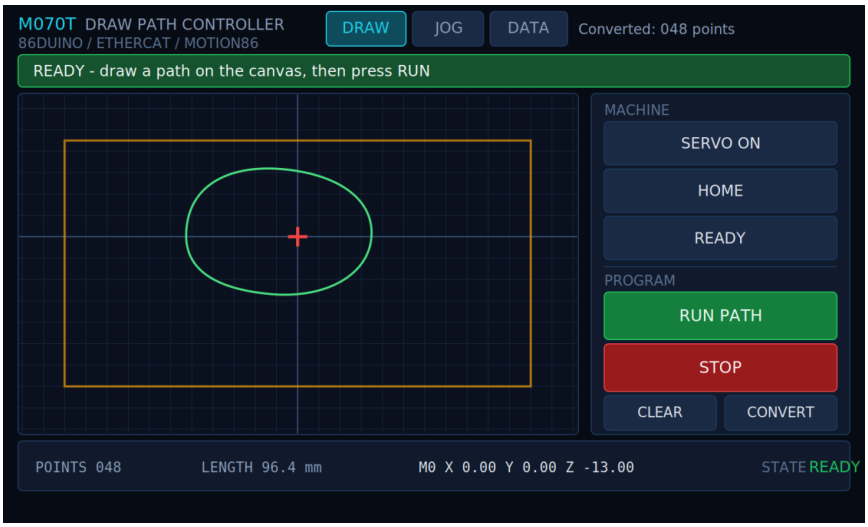
ICOP TECHNOLOGY · QEC 入門指南

[入門指南] QEC-M-070T × Motion86 – 在觸控螢幕上畫路徑,三軸即時執行



適用硬體	QEC-M-070T + QEC-R11MP3S
開發環境	86Duino IDE
對應範例	Motion86_DrawPath
文件語言	繁體中文

目錄

前言

這份入門指南示範如何用一台 **QEC-M-070T** 做出一個完整的**三軸路徑繪製控制器**：操作者用手指在 7 吋觸控螢幕上畫一條線，控制器把線條平滑化、轉成一連串座標點，再交給一台 **QEC-R11MP3S** 的三顆軸即時畫出來。

觸控畫面、EtherCAT 主站、路徑運算、位置紀錄全部跑在同一塊板子上，不需要外接 PC、PLC，也不需要另外插一張運動控制卡。

關於範例裡另外那兩組馬達

範例中還接了兩台 QEC-R11MP3S，讓它們同步複製第一台的動作。**這兩台不是應用的一部分，而是同步控制的驗證用。**

1. 快速概覽

- **用手指畫，機器就照著走**—在畫布上畫一條線，手指放開後程式自動把它整理成一連串座標點（最多 600 點），不需要事先寫好圖形。
- **三顆軸連續動作**—X / Y / Z 依規劃好的路徑走，指令全部用毫米，不必自己換算成 pulse。
- **路徑不會走走停停**—程式會先把後面的點排進緩衝區，Motion86 把相鄰的線段接起來，整條路徑跑起來是一個連續動作。
- **碰到極限開關立刻停**—每個控制週期都檢查開關，一碰到就急停並鎖住，要等開關實際放開才能解除。
- **三個操作頁面**—Draw 畫圖、Jog 手動移動、Data 查看每一個點的執行狀況。

2. 所需設備

項目	說明
QEC-M-070T	EtherCAT 主站,內建 7 吋 800 × 480 觸控螢幕
QEC-R11MP3S × 1	**主機構:**三軸 CiA402 伺服模組,負責畫路徑
QEC-R11MP3S × 2	**選配:**同步驗證用,不影響 draw path 本身
24V 電源	Vs 供系統、Vp 供周邊
極限開關	接到主機構模組每一顆軸的數位輸入
開發環境	86Duino IDE

程式裡把三組機台分別叫做 M0、M1、M2。

M0 是主機構,也是這份指南的主角:它裝了原點開關、設定了真實的行程極限、負責把你畫的路徑走出來。

3. 功能總覽

3.1 Draw — 畫圖與即時位置

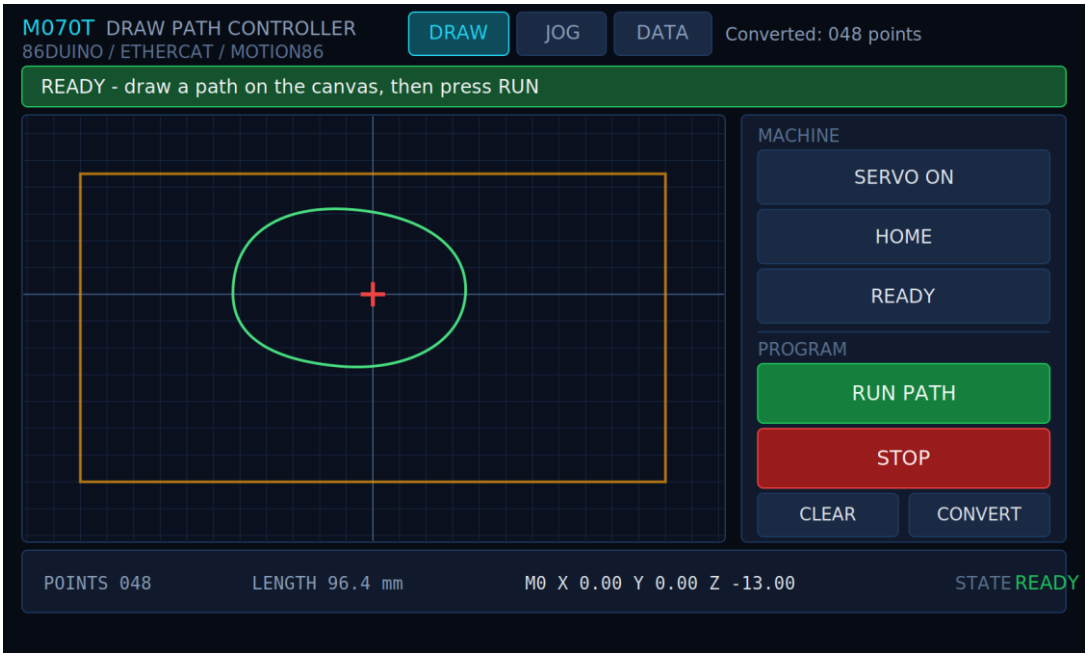


圖 1 Draw 畫圖頁:格線畫布、橘色安全框、綠色轉換後路徑、紅色即時位置游標,右側是常駐控制欄

畫面上會看到這幾樣東西:

- **橘色方框**—安全繪圖範圍。框外的觸控在還沒轉換成座標點之前就會被排除,避免使用者畫出造成撞機風險的路徑。這個範圍是從實際行程往內縮 25 mm(`DRAW_XY_BUFFER_MM`)得到的。
- **綠色線條**—轉換完成的路徑。手指一放開就會自動計算,不用另外按按鈕(若要重新計算路徑右下角有 `CONVERT` 按鈕可以重新計算)。
- **紅色十字**—機台現在的實際位置。馬達啟動就會出現。使用者可以根據此十字的位置,進行即時監控。
- **底部狀態列**—點數、路徑總長度、`M0` 的即時 XYZ,右邊還有一個狀態燈號(`OFF` / `ON` / `READY` / `RUN` / `JOG` / `LIMIT`)。

3.2 Draw 頁面

畫面上方的橫幅會直接告訴使用者下一步該按哪一顆按鈕。

顏色	訊息	意思
藍	`STEP 1 - press SERVO to enable the drives`	還沒啟動馬達
藍	`STEP 2 - press HOME, or press READY to zero at this position`	還沒建立原點座標
綠	`READY - draw a path on the canvas, then press RUN`	可以進行路徑繪製
橘	`PATH RUNNING (M0 + M1 + M2) - press STOP to abort`	執行中,按下 STOP 暫停繪製任務
紅	`LIMIT LATCHED - release the switch, then press CLEAR`	撞到極限開關,已鎖定

3.3 Jog 頁面

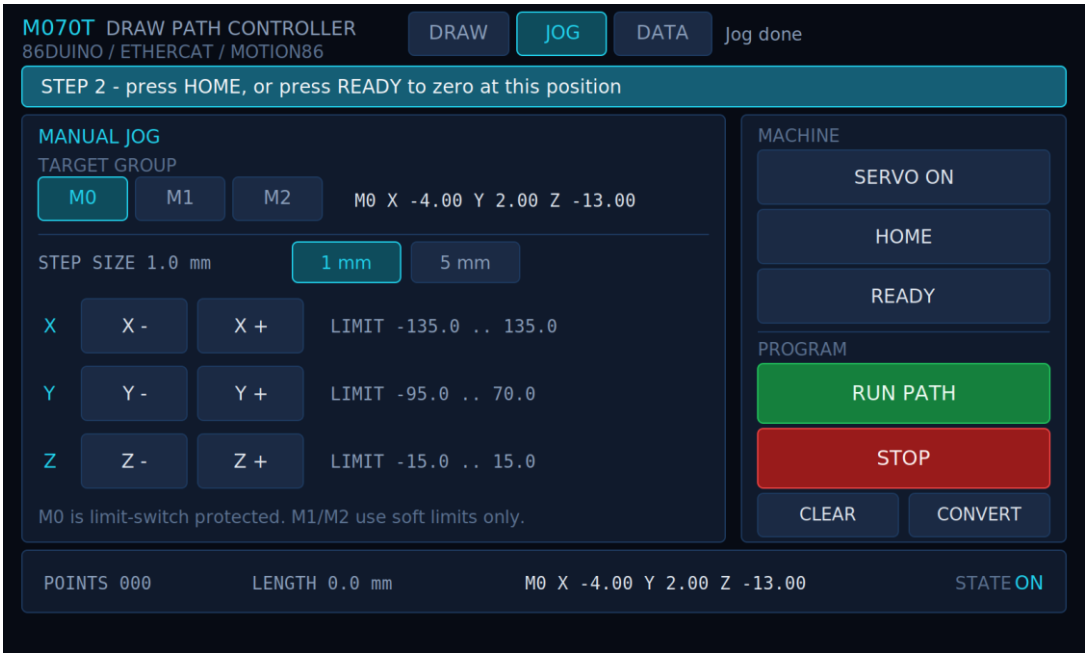


圖 2 Jog 手動頁:機台選擇、步進大小、XYZ 正負按鈕與各軸行程範圍

上面先選要動哪一組(M0 / M1 / M2),再選步進量(1 mm 或 5 mm),然後按 X- X+ Y- Y+ Z- Z+。

右邊會即時顯示**所選那一組**的位置,以及每顆軸的行程範圍:

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

選到 M1 或 M2 的時候,範圍會變成 -1000 .. 1000——因為它們沒有原點開關,絕對位置無從定義,只能給一個很寬鬆的軟體極限。頁面下方也直接寫了這句提醒:*M0 is limit-switch protected. M1/M2 use soft limits only.*

比較值得注意的是,**手動移動不是直接送指令出去**。程式先把目標值規範在一個範圍內,再下一個絕對位置指令:

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

3.4 Data 頁面

圖 3 Data 資料頁:座標點表格、進度條、目前執行中的那一列被標成橘色

這一頁把整條路徑攤開:步驟編號、目標 X / Y / Z、進給速度、狀態。

每一步會依序經過三種狀態:

狀態	顏色	意思
Wait	灰	等待中
Running	橘	執行中
Finish	綠	已完成

一頁顯示 7 列,`< PREV` / `NEXT >` 可以翻頁。`FOLLOW` 按鈕是自動跟隨:打開的時候,頁面會自己翻到正在執行的那一步;你只要手動翻過頁,它就會自動關掉。

表格上方還有一條進度條跟一行摘要:

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

當機器畫出來的東西跟你預期的不一樣時,可以透過此頁面查看是座標轉換的問題,還是是機構或參數設定的問題。

表格裡的 `Z` 全部是 `-13 mm`,因為這個範例的 `Z` 軸不參與繪圖(`PARK_Z_AT_HOME_LIMIT` 為 `true`),重播的是 `X / Y` 平面上的形狀。

4. 第一次端到端操作流程

Step 1:按 SERVO ON(要按兩次)

第一次按—啟動 EtherCAT 主站、掃描三台模組、完成三個 `Machine` 物件的設定。狀態列會依序印出 `ECAT: master.begin`、`ECAT: attach MP3S 0/1/2`、`ECAT: master.start`,結束後顯示 `Press SERVO again to enable`。

第二次按—依序把每一組的三顆軸切到 CSP 模式並啟動,然後 `machineOn()`。狀態列出現 `M070T servo ready`,底部狀態燈號變成 `ON`,Draw 頁上的紅色十字也會出現。

分兩次按是刻意的:第一次只做網路初始化,你可以先確認模組都掃到了,再決定要不要真的通電上馬達。

Step 2:按 HOME(可跳過)

M0 依序尋找 X / Y / Z 的原點開關,找到後用一行 `G92` 把座標系建立起來,再退開 3 mm 離開開關。過程中橫幅會變橘色並顯示 `HOMING IN PROGRESS - do NOT touch any control`。

完成後狀態列顯示 `M0 home done. Press READY.`

這一步可以跳過—如果機器本來就停在一個工作原點,直接按 READY 就好,原理與注意事項見 5.4。

Step 3:按 READY

M0 移動到原點 (0, 0, −13),準備開始畫圖。

同步驗證用的 M1、M2 則是定義當前位置為起點,之後的位移都從這裡算起。

橫幅轉綠,顯示 `READY - draw a path on the canvas, then press RUN`,狀態燈號變成 `READY`。

Step 4:用手指在畫布上畫圖

在橘色框內畫一個形狀。手指放開的瞬間程式就會自動做三件事:平滑化 → 依 2 mm 間距重新取樣 → 產生座標點清單。狀態列顯示 `Converted: 048 points`,綠色路徑同時出現在畫布上,Data 頁的表格也一起填好。

Step 5:按 RUN PATH

M0 依照座標點清單開始畫,X / Y / Z 三顆軸連續動作。橫幅變橘色,狀態燈號變成 `RUN`。

切到 **Data 頁**可以看到它一步一步跑:進度條前進、目前執行的那一列被標成橘色,頁面會自動翻到正在跑的那一頁。

同時,M1、M2 會複製 M0 的每一段位移。**這裡就是同步控制的觀察點**:M1 和 M2 機構應該同步執行動作。

全部跑完之後,三組同時回到工作原點——M0 回到原點,M1、M2 回到當初 `G92` 歸零的位置。狀態列顯示 `Task done. M0/M1/M2 at zero.`

Step 6:任何時候都可以按 STOP

STOP 會立刻馬達全部停下來,並且清掉所有還沒執行的請求。

5. Motion86 做了甚麼

上面看到的東西,底下其實只有很少的程式碼。這一節是這份指南真正想講的部分。

5.1 用毫米下指令,不用再自己算 pulse

設定一次「每毫米幾個 pulse」,之後所有指令都用毫米:

```
machine0.config_PPU(AXIS_X, 1600.0);   // 每 1 mm = 1600 個 pulse
machine0.config_PPU(AXIS_Y, 1600.0);
machine0.config_PPU(AXIS_Z, 1600.0);
machine0.config_ReverseDirection(AXIS_X);   // 方向反了就翻過來
```

之後 `machine0.line(12.30, -7.50, -13.00, 300)` 就是「走到 X=12.3、Y=−7.5、Z=−13 這個點,速度 300 mm/min」。

換減速比、換螺桿導程、換編碼器解析度,你只要改 `config_PPU()` 的數字。

5.2 兩個 callback,就接上任何驅動器

EtherCAT 函式庫給你的是**驅動器物件**,講的是編碼器 pulse:`setTargetPosition()`、`getPositionActualValue()`。

Motion86 給你的是 `Machine` **物件**,講的是毫米、進給速度、G-code。

中間就靠這兩個小函式接起來:

```
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 負責「給一個位置」跟「讀回一個位置」。

這件事的價值在哪? 以後你要更換其他馬達,甚至想先用純軟體的模擬軸把流程跑通,主要就是改寫這兩個函式。

5.3 超出行程的指令會被擋下來

```
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();
```

打開之後,只要目標超出範圍,Motion86 會在送出 pulse 之前先擋下這道指令,`line()` 回傳 false。

你不用在每一次下指令前手動寫 if 判斷,也不用擔心哪個路徑漏檢查。

5.4 原點復歸內建,但開關怎麼讀由你決定

歸 home 的動作 Motion86 幫你做,你只要告訴它「怎麼判斷開關到了沒」:

```
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();
```

考慮到在實務上每個應用場景的 IO 配線不盡相同,有人接驅動器的 DI、有人接主站的 IO、有人用 NPN 有人用 PNP。因此 Motion86 把「怎麼讀開關」這件事交給你決定,開關接法比較特別時,也還能沿用整套函式庫。

找到開關之後,**一行 G-code 就把座標系建立起來**:

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

`G92` 不會讓機器動,它只是告訴 Motion86:「你現在待的這個位置,就叫做這個座標。」

這也就是 Step 2 可以跳過的原因:

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

不過這裡有個取捨要留意:跳過 HOME 按鈕的時候,軟體極限是從「你當前的位置」開始計算。

5.5 將任務點走成一個連續動作

一條路徑可能有 600 個點,甚至更多。你**不是**送一個點、等它走完、再送下一個(那樣機器會不停抖動),而是持續把緩衝區填滿:

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

  if (anyEnabledCmdBufferFull()) {   // 緩衝滿了,同一個點等一下再試
    yield();
    sleep(1);
    continue;
  }

  machine0.line(p.x, p.y, p.z, p.feed);          // M0:絕對座標
  mirrorRelativeMove(p.x - prevX, p.y - prevY,   // M1、M2:相對位移
                     p.z - prevZ, p.feed);
  prevX = p.x; prevY = p.y; prevZ = p.z;
  i++;
}
```

`isCmdBufferFull()` 跟 `getCmdCount()` 讓你看得到佇列裡的狀況,所以你的程式可以一直往裡面塞、Motion86 一直往外拿。因為 Motion86 會把相鄰的線段自動銜接起來,48 個點的路徑會走成**一個連續動作**,而不是 48 次的走走停停。

`getCmdCount()` 還藏了一個很好用的小技巧。「還沒執行的指令數量」剛好就是「你送到哪裡」跟「軸實際走到哪裡」之間的差距:

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

Data 頁就是靠這一行判斷該把哪一列標成 Running,完全不用額外紀錄。

5.6 一台主站同時控制三組機構

到這裡為止講的都是 M0 這一組三軸。這一小節談的是範例裡另外兩組機台真正要證明的事。

多加一組機構,程式上就是多一個物件:

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

三個物件在同一個即時回呼裡依序更新,共用同一個 500 µs 的網路週期:

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

關鍵在於它們是三組各自獨立的座標系。 你可以讓它們跑三條不同的路徑,寫法跟這裡一樣—只是餵給 `line()` 的座標不同而已。

那為什麼範例讓它們做一樣的動作?**因為這樣比較看得出有沒有同步。** 三組跑不同路徑的時候,不容易用肉眼判斷它們是不是走在同一個節拍上;讓它們畫同一個形狀,落後或抖動就比較容易被看出來。

範例用「相對模式」來做這件事:M0 走絕對座標,M1、M2 切成相對,所以每一道指令重複的是**位移量**而不是目的地—它們因此不需要自己的原點開關,也不需要跟 M0 對齊機械座標:

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

也因為這樣,M1、M2 的軟體極限只能設在 ±1000 mm,實質上沒有保護作用。這在本範例可行,是因為它們只複製 M0 的位移,而 M0 的行程本來就被限制住了。如果你要讓它們跑自己的路徑,建議先補上原點開關與正確的行程設定。

5.7 支援 G-code 指令

`gcode()` 隨時可以呼叫,跟 `line()` 混著用也沒問題。這個範例只用到 `G92`(定義目前位置的座標),但同一個入口也吃 `G0` / `G1` / `G2` / `G3`。

意思是:如果你的專案有一部分來自 CAM 輸出的 G-code,不用另外寫,丟給 `gcode()` 就好。

5.8 Motion86 到底幫你省了多少事

這件事	自己要處理的	用 Motion86
毫米 → pulse	每顆軸的換算、四捨五入誤差累積	`config_PPU()` 一次
三軸走一條直線	自己拆分向量、同步三軸	`line(x, y, z, f)`
加減速	自己寫 S 曲線或梯形	內建
行程保護	每次下指令前手動檢查	`enableSoftLimit()`
路徑連續	自己做前瞻與轉角速度	內建緩衝,`isCmdBufferFull()` 看狀態
原點復歸	自己寫尋找、減速、退開的狀態機	`home()` + 一個讀開關的 callback
換驅動器品牌	大部分要重寫	改兩個 callback

6. 程式怎麼組起來

6.1 五個檔案

```
Motion86_DrawPath/
├── Motion86_DrawPath.ino   主程式
├── config.h                所有可以調的參數
├── app_state.h             兩個模組共用的東西
├── ui.h                    UI 宣告
└── ui.cpp                  UI 畫面物件
```

`ui.cpp` 完全不下運動指令。按鈕不會直接去呼叫 `machine0.home()`,而是提出請求,由運動任務接手:

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

兩邊往來的所有函式都寫在 `app_state.h` 裡,一眼看得完:

```
// 在 .ino 實作,由 ui.cpp 呼叫
void requestMotion(MotionRequest req);
void requestJogMove(machineAxis_t axis, double deltaMm);
void requestStop();

// 在 ui.cpp 實作,由 .ino 呼叫
void uiInit();
void uiRefresh();
void setStatus(const char *text);
void setHint(const char *text, HintLevel level);
```

實際的好處是:**重畫螢幕、換配色、加第四個頁面,都不需要動到運動控制的程式。** 對一台已經在線上運轉的機器來說,能把改動限制在畫面這一側,風險會小很多。

6.2 設定一台機器

下方是三軸設定的程式:

```
machine0.config_PPU(AXIS_X, M0_PPU_X);          // 每毫米幾個 pulse
machine0.config_PPU(AXIS_Y, M0_PPU_Y);
machine0.config_PPU(AXIS_Z, M0_PPU_Z);
machine0.config_ReverseDirection(AXIS_X);       // 方向相反就翻過來
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);
```

所有大寫的數值都來自 `config.h`,如果今天要換一台機器來執行,主要就是改 `config.h` 這個檔案,運動控制的程式多半不用動。

6.3 讓路徑規劃的計算跑在即時週期裡

Motion86 的位置計算是放在 EtherCAT 主站的即時回呼裡執行的:

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

master.attachRTCyclicCallback(MyCyclicCallback, true);
master.start(ECAT_CYCLE_TIME, ECAT_SYNC);   // 500 µs,時鐘同步
```

每一次 `updateAxes()` 都會算出那台機器的下一個位置,再透過 callback 送出去。

6.4 四個任務怎麼分工

這支程式同時要做四件事,分成四個任務跑在同一顆 CPU 上:

任務	週期	負責什麼
`EtherCAT callback`	500 µs	三次 `updateAxes()`,順便取位置
`motionTask`	1 ms	執行 Servo / Home / Ready / Jog / Run 的請求
`hmiTask`	5 ms	跑 LVGL,每 100 ms 呼叫一次 `uiRefresh()`
`telemetryTask`	1 ms	把 250 Hz 的位置資料印到序列埠

其中 EtherCAT callback 的時間最不能被打斷—它必須在 500 µs 的網路週期內做完。所以它跟其他任務之間交換資料時,沒有用一般的「上鎖」做法(上鎖代表某一邊可能得停下來等),而是用一個計數器:

```
telemetrySeq++;                // 變成奇數 = 正在寫入
telemetrySampleX = x;  telemetrySampleY = y;  telemetrySampleZ = z;
telemetrySeq++;                // 變回偶數 = 寫完了
```

寫的那一邊(EtherCAT callback)只管往前加,不會停下來等。讀的那一邊(telemetryTask)則是這樣判斷:讀之前先看一次計數器,讀完再看一次—**兩次數字一樣,就表示這份資料在讀取過程中沒有被改掉**,可以放心用;不一樣就重讀一次。

按鈕的處理也是類似的想法:按下去不會直接呼叫運動函式,而是留下一個請求,由 `motionTask` 一次執行一件。所以復歸過程中不小心多按了幾下,也不會讓狀態錯亂—`canQueueMotionRequest()` 會把它擋掉,並在狀態列寫出被拒絕的原因。

7. 安全設計

這個範例把「防止撞機」拆成三層,分別擋掉不同階段的問題。

防護	擋掉什麼	寫在哪
畫面限制	不合理的輸入	`ui.cpp`
軟體極限	不合理的指令	`.ino`
硬體極限	真的撞上去了	`.ino`

第一層在使用者的手指上。 畫布上的橘色框比實際行程小一圈,框外的觸控在還沒變成座標點之前就被排除。問題在最前端就處理掉,後面的程式不會收到一個超出範圍的點。

第二層在指令送出之前。 `config_PosLimit()` 加上 `enableSoftLimit()` 之後,超出行程的目標值會被 Motion86 擋下來,不會有 pulse 送到驅動器。這一層擋的是程式端的錯誤—參數設錯、路徑算錯、或是有人改了程式忘記檢查。

第三層在機構上。 前兩層都是軟體判斷,前提是設定值跟實際機構相符。萬一不符(行程參數填錯、機構被移動過、跳過復歸而位置假設錯誤),就只剩實體開關。程式每個控制週期都讀一次開關,一旦觸發:

```
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()` 立刻停止動作,同時把 `limitSwitchLatched` 設起來鎖住狀態。**這個「鎖住」是刻意的:**開關被壓住的時候常常會有彈跳,如果只靠「開關放開就恢復」,機器可能在操作者還沒反應過來時又動起來。所以 `clearEMGStop()` 一定要等開關實際脫離、而且操作者主動按下 `CLEAR` 之後才會執行。

最後那行 `setHint()` 會立刻把橫幅塗紅,不必等下一次 100 ms 的畫面更新。UI 雖然住

在另一個檔案裡,運動控制這一側需要示警的時候仍然叫得動它。

8. 參數調校

所有速度都集中放在 `config.h`:

```
static const double PATH_FEEDRATE          = 300.0;  // mm/min,執行路徑
static const double READY_FEEDRATE         = 450.0;  // 回原點
static const double JOG_FEEDRATE           = 250.0;  // 手動微調
static const double HOME_BACKOFF_FEEDRATE  = 200.0;  // 復歸後退開
static const double HOME_SEEK_SPEED        = 600.0;  // 尋找開關
static const double PATH_SAMPLE_MM         = 2.0;    // 座標點的間距
```

`PATH_FEEDRATE` 建議一次調一點點,同時注意驅動器的追隨誤差。如果速度拉高之後曲線開始變得有稜有角,就把 `PATH_SAMPLE_MM` 調小,讓 Motion86 有更多點可以銜接—代價是同一個緩衝區能裝的路徑長度變短。

`config.h` 裡還放了行程極限、螢幕版面座標、配色。換一台機器來跑,通常從這一個檔案開始調就夠了。

9. 常見問題

只有一台 QEC-R11MP3S,可以跑 draw path 嗎?可以。先把 `config.h` 改一行:

```
static const uint8_t ENABLED_MACHINE_GROUPS = 1;   // 原本是 3
```

再把 `.ino` 裡 `initM070TMotion()` 中掛載第 2、3 台模組的那幾行註解掉(`slave[1].attach(...)` 與 `slave[2].attach(...)` 及其後的 `cia402GetServo()`)——這幾行沒有跟著 `ENABLED_MACHINE_GROUPS` 判斷,匯流排上找不到模組會卡在這裡。其餘程式不用動,餵點、極限保護、UI、Data 頁進度追蹤都照常運作。

按了 SERVO 沒反應,只顯示 `Press SERVO again to enable` 正常。第一次按只做 EtherCAT 初始化,要再按一次才會啟動驅動器。

橫幅是紅色的 `LIMIT LATCHED`**,按什麼都沒用**極限開關還被壓著。先手動把軸移開讓開關脫離,再按 `CLEAR`。程式刻意要求開關實際釋放後才解除鎖定,避免開關彈跳造成機器突然又動起來。

按 RUN 顯示 `Press READY before RUNdrawReady` 還是 `false`。Jog 過之後這個旗標會被清掉,所以手動微調完要再按一次 READY。

狀態列出現 `Point outside M0 limit` **或** `Motion command rejected` 軟體極限擋下來了。檢查 `config.h` 裡的 `M0_LIMIT_*` 是不是跟實際行程對得上,以及 `M0_PPU_*` 有沒有設對——PPU 設錯的話,毫米數字看起來正常,實際跑的距離卻會差好幾倍。

手指在畫布上畫不出線兩種可能:一是手指在橘色安全框外面,框外的觸控會直接被忽略;二是機器正在執行動作(`motionBusyForEditing()` 為 `true`),這時畫布會暫時停止接收輸入。

路徑跑起來曲線有稜角 `PATH_SAMPLE_MM` 太大或 `PATH_FEEDRATE` 太快。先把取樣間距調小到 `1.0 mm` 試試看。

同步驗證的 M1、M2 完全不動確認 `config.h` 裡的 `ENABLED_MACHINE_GROUPS` 是 3,並確認有按過 READY——M1、M2 是在 READY 時用 `G92` 歸零並切到相對模式的,沒歸零就沒有

起點。(M0 的 draw path 不受影響,少了這兩組一樣跑得完。)

底部狀態列的位置一直顯示 `----`伺服還沒啟動(`servoReady` 為 `false`)。先完成 Step 1。

序列埠看不到位置資料確認 `ENABLE_POSITION_TELEMETRY` 為 `true`,並把序列埠監控視窗的鮑率設成 `3000000`。

10. 資源

範例程式

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

五個檔案要放在同一個名為 `Motion86_DrawPath` 的資料夾裡。86Duino IDE 要求資料夾名稱必須跟 `.ino` 檔名一致。

11. 後續延伸

這個範例留了幾個容易延伸的方向:

- **把畫布換成檔案來源。** `pathPoints[]` 只是一個座標點陣列,改成從 SD 卡讀 CSV、或從網路收 G-code,後面的執行流程不用改。
- **讓 Z 軸真的參與。** 關掉 `PARK_Z_AT_HOME_LIMIT`,並在轉換時給每個點不同的 Z,就能做出提筆落筆的效果。
- **讓 M1、M2 跑自己的路徑。** 把它們從相對模式改回絕對模式、補上原點開關與行程極限,就從「同步驗證」變成三組真正各自作業的機構。