DETECTING MISSED PULSES IN MULTI-AXIS ABSOLUTE PSO MODE

1. Scope

This note covers how an application program can determine, after a multi-axis absolute PSO run, whether all expected trigger pulses were generated. It assumes the reader is already familiar with the multi-axis absolute PSO feature of the XPS-D controller and has read the relevant sections of the XPS-D Multi-Axis PSO User Guide. A companion Tech Note covers the selection of the capture radius value, which is the parameter most directly responsible for whether or not the pulse sequence completes successfully; this note assumes that radius selection has already been done according to that guidance, and covers how to verify the result at runtime.

This note does not cover identifying which pulses were missed during a run, or where in the trajectory the failure occurred. Detecting that a failure happened is a small, portable check; identifying what was missed and where is a larger problem requiring additional position logging and trajectory-correlated analysis. That broader topic is outside the scope of this note.

2. Why runtime detection matters

To support automation best practices, process-step confirmation, and downstream feedback, an application running multi-axis absolute PSO has a straightforward way to confirm that its trigger output completed as configured. After a run, a single register read indicates whether all expected pulses were generated. This check belongs in any application that participates in a process-control or quality-assurance chain.

A run that completed successfully and a run that truncated early due to a missed pulse look identical to the application from the outside. The trajectory executes to completion in both cases; the motion does not stop or fault; no error is returned by the execution call. The distinction lives only in the controller's pulse-generation state, and only for as long as the application chooses to look at it before the next operation overwrites or invalidates it.

Capturing the pulse count places this evidence on the application's side rather than leaving it in the controller's volatile state. When downstream investigation later asks whether triggers were generated for a given run, the application has logged evidence rather than a question that can only be re-investigated by re-running the part. The cost of capturing this evidence is one register read per run — a small fraction of trajectory execution time, and effectively zero compared to the cost of the investigation that has to start from scratch by re-running the part.

3. The register that holds pulse count

The XPS-D controller exposes a 32-bit register — register 706 in the controller's CIE register space — that holds the count of pulses remaining in the loaded multi-axis absolute PSO array. The register is read using the CIERegister32ValueGet API call. The generic form of the call is:

CIERegister32ValueGet(Group1.Positioner1, 706, int*)

The first argument identifies the positioner whose CIE board hosts the register. By controller convention, this is the first positioner in the PSO group. For a group named Group1 with positioners named Group1.Positioner1, Group1.Positioner2, and so on, the call uses Group1.Positioner1. The example script later in this note uses an XY group with positioners named XY.X and XY.Y, in which case the call resolves to:

CIERegister32ValueGet(XY.X, 706, int*)

The second argument is the register number; the third is the integer pointer into which the value is read.

The register's lifecycle is straightforward. When GroupMultiAxisPSOAbsolutePrepare completes, the register holds the total number of pulses in the loaded array — for the demonstration trajectory used in this note, 134,277. As the trajectory executes and pulses fire, the register decrements once per pulse. On a fully successful run, the register reads zero at the moment the trajectory completes. On a run that truncated due to a missed target, the register holds whatever count of pulses remained in the array at the moment the controller stopped advancing through it.

A nonzero remaining count at the end of a run is, therefore, definitive evidence that the run did not complete as configured. There is no other state in which the register reads nonzero at the end of a normal run; the only way to reach the end of trajectory execution with a nonzero remaining count is for the controller to have stopped firing pulses before all targets in the array were reached.

4. The detection pattern in a working script

The detection pattern is two register reads bracketing the trajectory execution, plus a subtraction. The first read captures the expected pulse count. The trajectory runs. The second read captures the remaining pulse count at completion. The triggered pulse count is the difference between the two. If the remaining count is nonzero, the run truncated.

The script below is an excerpt from a working test program that demonstrates the pattern in the context of a complete absolute-from-LineArc PSO run. Setup steps that are documented elsewhere in the User Guide are shown without rationale comments; the steps relevant to detection carry full rationale comments. The expected-count read is placed after GroupMultiAxisPSOAbsoluteEnable and before trajectory execution begins. This is where the register reliably reports the loaded array length.

```
// ********** Perform Setup **********

GroupMultiAxisPSORelativeDisable(XY,2,Immediate)

GroupMultiAxisPSOAbsoluteDisable(XY)

XYLineArcExecution(XY,Square120r10.
linearc,80,1000,1)

CIERegister32ValueGet(XY.X, 706, int*)

// If returned value is zero, run completed.

// If nonzero, run truncated.

GroupMultiAxisPSOAbsoluteDisable(XY)
```

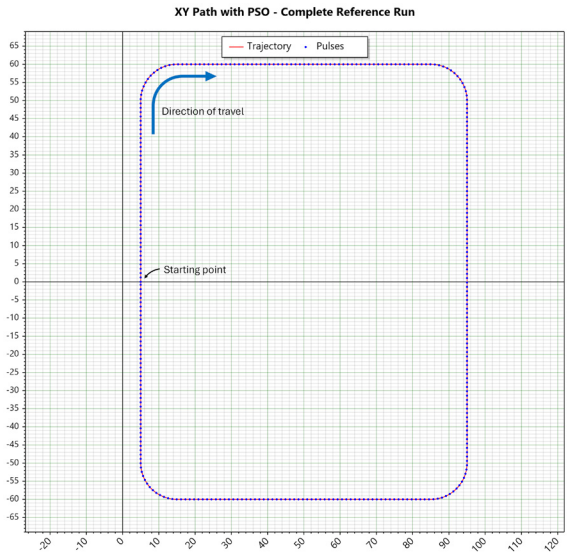
An application that wants more than a pass/fail bit can capture the expected pulse count as well, by adding a register read after GroupMultiAxisPSOAbsoluteEnable and before trajectory execution begins. This extra read returns the total number of pulses in the loaded array — the baseline against which the post-run remaining count can be compared. The triggered pulse count is the difference between the two. Section 4's working example shows this fuller form.

The extra read costs nothing measurable in time, and it produces information that some applications can use and others cannot. Cross-checking the controller's triggered count against an independent downstream count — a laser pulse counter, a vision system feature count, a metrology sample count — gives the application an independent confirmation that the process completed. Disagreements between counts are themselves useful: they point at where the count was lost between the controller and the downstream sensor. Applications that present an operator interface can use the same numbers to display run results in a more informative form than pass-or-fail. Whether either of these is worth the second read is a per-application decision; the minimum check works without it.

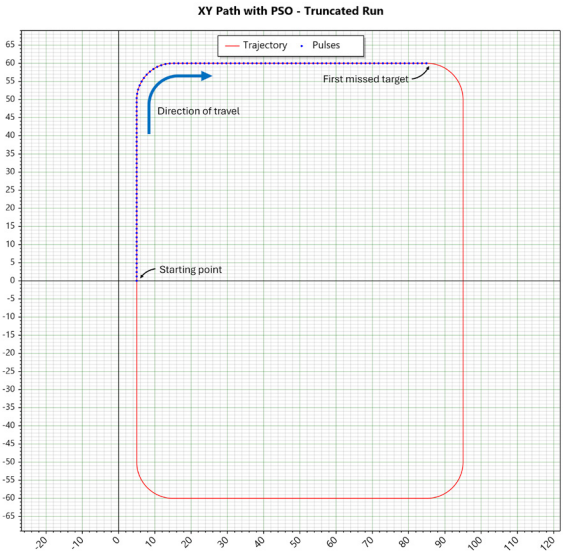
5. The minimum check, with an optional extension

The detection check in its minimum form is a single register read after trajectory execution, tested against zero. Any nonzero remaining count indicates that the run did not complete. This is sufficient for an application whose only response to a missed pulse is to alert or halt — proceed if zero, take corrective action if not. For likely causes when the run truncates and how to respond, see Section 6.

One sequencing constraint applies to both forms of the check: the post-execution register read must occur before GroupMultiAxisPSOAbsoluteDisable is called. Once Disable is issued, the register no longer reports a value that reflects the run's outcome. The constraint is on order: the read must come before Disable.



A complete reference run, with the full pulse set generated as expected.



Trajectory plot showing a run with a deliberately-tight capture radius. The trajectory executes to completion, but pulse generation stops at the first missed target.

6. Likely causes and response

A truncated run means the stage moved past one of the targets in the pulse array without entering its capture radius. From that moment on, the controller waits for that target to be reached before advancing the array index, and no further pulses fire for the rest of the run. The companion Tech Note on capture-radius selection covers this mechanism in detail.

The likely cause depends on which form of absolute PSO is in use.

For absolute PSO from a LineArc trajectory, the pulse array is generated from the same trajectory the stage executes. The targets are guaranteed to lie on the commanded path. The cause of a missed target is therefore almost always that the stage's actual path deviated from the commanded path by more than the capture radius — that is, the worst-case following error along the trajectory exceeded the radius. The companion Tech Note covers radius sizing from following error.

For absolute PSO from an explicit point file, the pulse array is loaded from a separate file, independent of the trajectory the stage executes. Two failure modes are possible: following error large enough to miss a target, as above, or a mismatch between the point file and the trajectory such that the trajectory does not pass within the capture radius of every listed point. In practice, the second failure mode is the more common one, because point files and trajectories are often generated by different tools or different members of a team, and small geometric inconsistencies between them are easy to introduce.

Identifying which of these is the cause is a separate exercise from detecting that the run truncated. The register check tells the application that the run did not complete; investigation tells the engineer why.

The application's response to a detected truncation depends on what the application is for. The simplest response is to log the result and alert the operator or upstream system; this preserves the evidence that the run did not complete and stops further work that depends on the run having succeeded. An

application with operator interaction may present the run's expected, triggered, and remaining pulse counts directly, allowing the operator to decide whether to retry, abort, or escalate. An application that operates without supervision may attempt to re-execute the trajectory automatically — for example, after re-checking conditions or after applying a corrective adjustment to the capture radius or trajectory velocity — but automatic retry should be bounded; an underlying cause that produced a truncation once will often produce it again on the same configuration.

7. What the check cannot tell you

The register check tells the application that the run did not complete. It does not say which target was missed, where along the trajectory the miss occurred, or whether more than one miss is implicated in the truncation. The expected, remaining, and triggered counts say nothing about position — they describe how many pulses fired, not which.

The position of the last successful pulse can be recovered from gathered data: routing the PSO trigger output to a position-capture trigger input, and gathering both the captured positions and the commanded trajectory, produces a record of where each fired pulse landed. The position of the missed target — the one whose capture radius was not entered — is harder to reach. When the pulse array is loaded from an explicit point file, the file provides the targets directly, and the next-expected target after the last successful pulse can be read from the file. When the pulse array is generated from a LineArc trajectory, the array is internal to the controller and not externally visible, so the next-expected target cannot be read out at all. Inferring it from the trajectory file is also unreliable: gating in the LineArc trajectory file (sections defined with Line or Arc elements rather than LinePulse or ArcPulse) means the next-expected target may be a substantial distance away from the last successful pulse rather than at the next adjacent location along the path.

In practice, locating the missed target is investigation work for an engineer rather than runtime work for the application, and it requires tools the application does not have access to. The register check is sufficient for the application's job — knowing whether the run can be trusted as a successful production output. The investigative problem is meaningfully larger and is outside the scope of this note.

8. Summary

The detection pattern is two register reads bracketing trajectory execution. Read register 706 after GroupMultiAxisPSOAbsoluteEnable and before execution to capture the expected pulse count. Run the trajectory. Read register 706 after execution and before GroupMultiAxisPSOAbsoluteDisable to capture the remaining pulse count. The triggered pulse count is the difference. A nonzero remaining count means the run truncated and should not be treated as complete.

The minimum form of the check is a single register read after execution, tested against zero. Any application running multi-axis absolute PSO has reason to perform at least this much.

For radius-selection guidance, see the companion Tech Note on capture-radius selection. For the underlying API calls and their full specifications, consult the sections on the multi-axis absolute PSO API and on register access in the XPS-D Multi-Axis PSO User Guide.

DS-072602 Detecting Missed-Tech Note_07/26