Latchline Controls

Legacy program in. Verified documentation out.

Functional Design Specification & Migration Review

PROJECT	PowerFlex 525 Device Control
CONTROLLER / PROGRAM	DEVPAC
DATE	02 September 2026

SAMPLE DELIVERABLE

This sample was generated from a publicly available Studio 5000 library program (a PowerFlex 525 device object). It contains no client data.

hello@latchlinecontrols.com

About this document

What this is

A complete engineering documentation package produced from a single export of a running legacy program (Studio 5000 .L5X, or RSLogix 500 .SLC for SLC-500 and MicroLogix). Deterministic analysis recovers the structure — every tag, every reference, every call, every piece of dead or disabled logic — and an AI documentation pass, verified by a controls engineer, recovers the intent. The result is what a migration, audit, or FAT actually needs: a functional specification, a tag dictionary, a full cross-reference, an engineering review findings register, and an executable test plan.

How to read it

Sections 1-3 describe what the program does and where every signal comes from. Section 4 lists every item that needs a human ruling before migration — disabled logic, uncalled routines, suspect tags. Section 5 is a factory acceptance test plan derived directly from the logic: every boolean output the program writes, with the conditions that drive it and drafted test steps. Statements the analysis could not prove from the export alone are marked VERIFY and collected for review — nothing is silently guessed.

The offer

Fixed price, fixed deadline, per program: legacy program in, verified documentation out. Conversion packages (PLC-5 / SLC-500 to ControlLogix with restructured code and FAT support) are quoted per system after a free structural analysis of your export.

Contents

Functional Design Specification

Scope: 1 program(s), 6 routine(s), 85 rung(s), 174 instruction(s), 16 tag(s).

Program: Dev_PF525

Program Dev_PF525 — Overview

What the program controls

Dev_PF525 is a single-device control program for one PowerFlex 525 variable-frequency drive on an Ethernet I/O connection (`Dev_PF525`, `Dev_PF525:I`, `Dev_PF525:O`). All drive interaction is carried by one `Dvc_PF525` add-on instruction acting on the device object `Dvc`. The ladder logic around it does three things only: conditions field feedback, evaluates interlocks and permissives, and raises alarms. The program writes no start, stop (the only stop command, `Dvc.PCmd_Stop`, is held permanently false), speed, or mode command to the drive. Operator commands must enter `Dvc` from outside this program's ladder logic (HMI or inside the AOI via `SysDevices`).

The program follows a standard device-class template: every routine has a NOP rung 0, and interlock/permissive blocks carry 32 input slots of which only a handful are used.

How the routines divide the work

Routine	Role	Produces	Consumed by
Main	Scheduler; runs the `Dvc_PF525` driver	`Dvc.Sts_*`, drive I/O, `ReadMsg`/`WriteMsg` messaging	Feedback, Interlocks, Alarms
Config	Structured Text, 0 rungs; content not analyzed in the digest	—	—
Feedback	Conditions three discrete inputs through `Dvc_DiscreteIn`	`Dvc_DI_Breaker.Sts`, `Dvc_DI_Disconnect.Sts` (`Dvc_DI_Feedback` is forced off)	Interlocks
Permissives	Builds and evaluates `Op_Permissive`	`Dvc.Inp_PermOK`, `Dvc.Inp_NBPermOK`	Main (driver)
Interlocks	Builds and evaluates `Op_Interlock`	`Dvc.Inp_IntlkOK`, `Dvc.Inp_NBIntlkOK`, `Interlocks.Sts_OK`	Main (driver), Permissives
Alarms	Two `Sys_Alarm` instances	`Alm_Fault`, `Alm_CommLoss` into `SysAlarms`	Plant alarm system (outside export)

Main operating sequence

Each scan, Main calls Config -> Feedback -> Permissives -> Interlocks -> Alarms, then forces `Dvc.PCmd_Stop` off, then executes the `Dvc_PF525` driver. The driver reads the four interlock/permissive OK bits written earlier in the same scan and updates the drive I/O image and `Dvc.Sts_*` status. There is no step sequence, no auto/manual logic, and no timing in the ladder; the drive runs whenever the driver (and whatever commands it) permits, and is blocked or stopped when interlocks or permissives are lost.

Two one-scan lags result from the call order: Permissives reads `Interlocks.Sts_OK` before Interlocks runs, and Alarms reads `Dvc.Sts_*` before the driver updates them. Preserve or consciously change this order in migration.

Interlock and alarm philosophy

Interlocks (Interlocks routine). Six live conditions, all "TRUE = OK": drive not faulted, drive connected, drive digital input 1 on, drive safety function not active, breaker feedback OK (`Local:3:I.Pt03.Data`), disconnect feedback OK (from `Dvc.Sts_DigIn1`). The safety-active interlock is a software mirror of drive STO status for blocking and annunciation only; it is not a safety-rated path. Inp.6 is forced TRUE; Inp.7–31 are forced FALSE from `AlwaysFalse`.

Permissives (Permissives routine). One live condition: `Interlocks.Sts_OK`. A lost interlock therefore also removes the start permissive. Inp.1–31 are spare (false).

Bypass. `Dvc.Sts_Bypass` from the drive object drives `Inp_Bypass` on both blocks. Which inputs are bypassable is configured inside the AOIs, not in the ladder. The non-bypassable results (`Sts_NBOK` -> `Dvc.Inp_NBIntlkOK` / `Inp_NBPermOK`) are passed separately so non-bypassable conditions remain enforced during bypass.

Alarms (Alarms routine). Two alarms: drive fault (suppressed while comm loss is active, so only one alarm is raised per event) and drive communication loss. Both are inhibited when `Dvc.Sts_Disabled` is on. Latching, delays, acknowledgement, and severity live inside `Sys_Alarm`.

Program-level review items

- VERIFY: Internals of the four AOIs (`Dvc_PF525`, `Dvc_DiscreteIn`, `Op_Interlock`, `Op_Permissive`, `Sys_Alarm`) are not in the export. In particular, confirm how unused inputs held false by `AlwaysFalse` (Interlocks Inp.7–31, Permissives Inp.1–31) are masked; if unmasked, the drive could never run. Also confirm which inputs are bypassable.
- VERIFY: `AlwaysFalse` is not written by any routine and has no description. Confirm it is a constant 0 and cannot be written externally. If set true it would assert every spare interlock/permissive slot and issue `Dvc.PCmd_Stop`.
- VERIFY: Whether the Config routine contains active statements or is empty; if empty, its JSR is dead.
- VERIFY: `Dvc_DI_Feedback` is permanently forced off and its status is not read anywhere. Confirm whether a real run-feedback signal was intended.
- VERIFY: `Dvc.Sts_DigIn1` is used both directly as interlock Inp.2 and as the source of the disconnect discrete input. Confirm the drive's digital input 1 wiring and that dual evaluation is intended.
- VERIFY: Where operator start/speed commands and bypass selection originate (HMI writes to `Dvc`, or AOI/`SysDevices`), and that bypass authority is appropriately restricted.
- VERIFY: `Dev_PF525` module configuration (RPI, IP, assembly layout), `ReadMsg`/`WriteMsg` setups, `Local:3` I/O wiring, and alarm parameter values are outside the export.
- All tags in the program lack descriptions; every role stated above is inferred from names and usage.

Routine: Alarms

Purpose

The Alarms routine derives two alarm request signals — drive fault and drive communication loss — from the PowerFlex 525 device status and passes them to the plant alarm class handler (`Sys_Alarm` add-on instruction) for management and reporting.

Functional description

The routine runs every scan from Main (rung 1). It produces no physical outputs and commands nothing. It only sets alarm inputs and executes two alarm-handler instances.

Source of status bits. All three status bits read here (`Dvc.Sts_Fault`, `Dvc.Sts_CommLoss`, `Dvc.Sts_Disabled`) are members of the `Dvc` device object, which is produced by the Main routine (see the Main routine, where `Dvc` is written from the drive I/O connection `Dev_PF525:I`). No tag description is available; the purposes below are inferred from the member names.

Drive fault alarm (`Alm_Fault`).

- `Alm_Fault.Inp` (alarm condition) is on when the drive reports a fault (`Dvc.Sts_Fault`) AND communication with the drive is healthy (`Dvc.Sts_CommLoss` off).
- The comm-loss qualifier suppresses the fault alarm while drive data is stale, so a communication loss does not also raise a spurious fault alarm. Only the comm-loss alarm is raised in that case.
- `Alm_Fault.Inp_Disable` follows `Dvc.Sts_Disabled`. Inferred: when the device is placed in a disabled state (out of service), the fault alarm is inhibited.
- `Sys_Alarm(Alm_Fault, SysAlarms)` is executed unconditionally on its own parallel branch every scan, so the alarm handler always processes the current input states.

Communication loss alarm (`Alm_CommLoss`).

- `Alm_CommLoss.Inp` follows `Dvc.Sts_CommLoss` directly.
- `Alm_CommLoss.Inp_Disable` follows `Dvc.Sts_Disabled`, same inferred meaning as above.
- `Sys_Alarm(Alm_CommLoss, SysAlarms)` executes unconditionally every scan.

Alarm handling. Both instances share the `SysAlarms` class structure, which is where system-wide alarm results are aggregated (also written by this routine per the digest). Alarm latching, acknowledgement, time delays, severity, and reset behavior all live inside the `Sys_Alarm` instruction and are not visible in this routine.

Relationship to other routines. `Dvc.Sts_Fault` is also consumed by the Interlocks routine as an interlock condition (see the Interlocks routine). This routine does not create that interlock; it only annunciates the fault.

Interlocks and permissives

None identified. This routine does not enable or block any output. Note that the drive fault it annunciates is separately used as an interlock condition in the Interlocks routine.

Alarms and faults

Alarm tag	Condition (Inp)	Inhibit (Inp_Disable)	Handler
`Alm_Fault`	`Dvc.Sts_Fault` on AND `Dvc.Sts_CommLoss` off	`Dvc.Sts_Disabled`	`Sys_Alarm(Alm_Fault, SysAlarms)`
`Alm_CommLoss`	`Dvc.Sts_CommLoss` on	`Dvc.Sts_Disabled`	`Sys_Alarm(Alm_CommLoss, SysAlarms)`

Behavior of note:

- A communication loss masks the fault alarm; only one alarm is expected during comm loss.
- A disabled device inhibits both alarms (subject to how `Sys_Alarm` treats `Inp_Disable`).

Review items

- VERIFY: `Sys_Alarm` AOI internals are not in the export. Confirm latching vs. auto-clear, acknowledge requirements, on/off delay timers, severity/priority, and how `Inp_Disable` is treated (suppress annunciation only, or also suppress logging/history).
- VERIFY: Configured parameter values for `Alm_Fault` and `Alm_CommLoss` (delay presets, severity, message text) are not in the export. Obtain from tag data or HMI alarm database before migration.
- VERIFY: Tag descriptions are absent for `Dvc.Sts_Fault`, `Dvc.Sts_CommLoss`, and `Dvc.Sts_Disabled`. The meanings used above (drive faulted, I/O connection lost, device taken out of service) are inferred from names. Confirm against the `Dvc_PF525` AOI definition in the Main routine.
- VERIFY: HMI presentation of `SysAlarms` (which displays, summary pages, horn) is outside the export.
- Rung 0 is a `NOP` placeholder. Harmless; likely a leftover from a template. Can be dropped in migration.
- Suppressing `Alm_Fault` during comm loss means a real drive fault that occurs at the same time as a comm loss is not separately annunciated until comms recover. This is intentional nuisance-alarm suppression but should be confirmed as acceptable by the plant engineer.

Rung listing

```
[0] NOP();
[1] [XIO(Dvc.Sts_CommLoss) XIC(Dvc.Sts_Fault) OTE(Alm_Fault.Inp) ,XIC(Dvc.Sts_Disabled)
      OTE(Alm_Fault.Inp_Disable) ,Sys_Alarm(Alm_Fault,SysAlarms) ];
[2] [XIC(Dvc.Sts_CommLoss) OTE(Alm_CommLoss.Inp) ,XIC(Dvc.Sts_Disabled) OTE(Alm_CommLoss.Inp_Disable)
      ,Sys_Alarm(Alm_CommLoss,SysAlarms) ];
```

Routine: Config

No ladder rungs — ST routine, not analyzed.

Routine: Feedback

Purpose

The Feedback routine conditions three discrete input signals into the device's standard `Dvc_DiscreteIn` objects. It maps a physical breaker input, a drive-derived disconnect status, and a (currently disabled) feedback input so that downstream logic can consume their processed status bits.

Functional description

The routine processes three discrete inputs. Each is handled the same way: a source bit is copied into the object's `.Inp_Data` member, then the `Dvc_DiscreteIn` Add-On Instruction is executed against that object with `SysDevices` as the shared device-class reference.

- **Breaker input (`Dvc_DI_Breaker`).** The source is physical input `Local:3:I.Pt03.Data` (module in slot 3, point 03). Its state is written to `Dvc_DI_Breaker.Inp_Data`, then the `Dvc_DiscreteIn` instruction runs. The resulting `Dvc_DI_Breaker.Sts` is consumed in the Interlocks routine.
- **Disconnect input (`Dvc_DI_Disconnect`).** The source is `Dvc.Sts_DigIn1`, a status bit from the main drive device object (`Dvc`, type `Dvc_PF525`). This is not a hardwired field point — it is a status bit reported by the drive object, mapped here into a discrete-input object. Its state is written to `Dvc_DI_Disconnect.Inp_Data`, then the instruction runs. The resulting `Dvc_DI_Disconnect.Sts` is consumed in the Interlocks routine.
- **Feedback input (`Dvc_DI_Feedback`).** The source is `AlwaysFalse`. `Dvc_DI_Feedback.Inp_Data` is therefore forced off every scan. The `Dvc_DiscreteIn` instruction still runs, but on a constant-false input. See Review items.

Rung 0 is a NOP and has no effect.

The internal behavior of the `Dvc_DiscreteIn` AOI (debounce, inversion, fault handling, how `.Inp_Data` becomes `.Sts`) is not visible in this export.

Interlocks and permissives

None identified in this routine. The Feedback routine does not evaluate interlocks or permissives; it only conditions input signals. The processed status bits (`Dvc_DI_Breaker.Sts`, `Dvc_DI_Disconnect.Sts`) are used as interlock conditions in the Interlocks routine.

Alarms and faults

None identified in this routine. Any fault behavior associated with these discrete inputs is internal to the `Dvc_DiscreteIn` AOI and is not exposed by the rung logic here.

VERIFY: Whether the `Dvc_DiscreteIn` AOI generates fault or alarm status internally, and where that status is consumed. The AOI internals are not in this export.

Review items

- `Dvc_DI_Feedback` **is driven by** `AlwaysFalse`**.** This input object is permanently forced off. Its processed status `Dvc_DI_Feedback.Sts` does not appear to be read by any sibling routine in the digests. This is likely a placeholder or dead/disabled input. Confirm whether a real feedback signal should be wired here.
- VERIFY: The purpose and expected behavior of `Dvc_DI_Feedback`. If field feedback is required for this device, the current logic never asserts it.
- VERIFY: The source of the disconnect signal. `Dvc_DI_Disconnect.Inp_Data` is fed from `Dvc.Sts_DigIn1` (a drive object status bit), not a hardwired field contact. Confirm this is intended, i.e., that the disconnect state is reported by the drive over the network rather than wired to a physical point.
- VERIFY: I/O wiring for `Local:3:I.Pt03.Data`. The `Local:3:I` module is not defined in this export; confirm the physical point assignment and that point 03 is the breaker status contact.
- VERIFY: Tag descriptions. All tags in this routine have no description in the export; the roles above (breaker, disconnect, feedback) are inferred from tag names and usage.
- VERIFY: `Dvc_DiscreteIn` AOI internals, as noted under Alarms and faults.

Rung listing

```
[0] NOP();
[1] [XIC(Local:3:I.Pt03.Data) OTE(Dvc_DI_Breaker.Inp_Data) ,Dvc_DiscreteIn(Dvc_DI_Breaker,SysDevices) ];
[2] [XIC(Dvc.Sts_DigIn1) OTE(Dvc_DI_Disconnect.Inp_Data) ,Dvc_DiscreteIn(Dvc_DI_Disconnect,SysDevices) ];
[3] [XIC(AlwaysFalse) OTE(Dvc_DI_Feedback.Inp_Data) ,Dvc_DiscreteIn(Dvc_DI_Feedback,SysDevices) ];
```

Routine: Interlocks

Purpose

Collects the interlock conditions for the PowerFlex 525 drive (`Dvc`) into the `Interlocks` (Op_Interlock) block, evaluates them, and passes the resulting "interlocks OK" and "non-bypassable interlocks OK" results back into the drive object.

Functional description

Interlock input collection (Inp.0–Inp.5). The routine builds a 32-bit interlock input word `Interlocks.Inp`. Six bits carry live conditions. Every bit is written so that TRUE = "condition OK, do not block":

Bit	Source	OK when	Where the source is produced
Inp.0	`Dvc.Sts_Fault`	drive is **not** faulted	`Dvc` object, updated in the Main routine
Inp.1	`Dvc.Sts_Connected`	drive communication is connected	`Dvc` object, Main routine
Inp.2	`Dvc.Sts_DigIn1`	drive digital input 1 is ON	`Dvc` object, Main routine
Inp.3	`Dvc.Sts_SafetyActive`	drive safety function is **not** active	`Dvc` object, Main routine
Inp.4	`Dvc_DI_Breaker.Sts`	breaker discrete-input status is ON	`Dvc_DI_Breaker` object, Feedback routine
Inp.5	`Dvc_DI_Disconnect.Sts`	disconnect discrete-input status is ON	`Dvc_DI_Disconnect` object, Feedback routine

The meanings of `Sts_DigIn1`, `Sts_SafetyActive`, and the two discrete-input `.Sts` bits are inferred from tag names; none of these tags carries a description in the export.

Spare inputs (Inp.6–Inp.31). Inp.6 is written by an unconditional output and is therefore always TRUE. Inp.7 through Inp.31 are each driven from `AlwaysFalse` and are therefore always FALSE (assuming `AlwaysFalse` holds 0 — no routine in the program writes it). How the Op_Interlock block treats a permanently-FALSE input depends on the block's internal configuration; see Review items.

Bypass. `Interlocks.Inp_Bypass` follows `Dvc.Sts_Bypass` (produced by the `Dvc` object in Main). When the drive object reports bypass, the interlock block is told to bypass. Which inputs are bypassable and which are not is determined inside the Op_Interlock block, not in this routine.

Evaluation. The `Op_Interlock` add-on instruction is executed once per scan on the `Interlocks` instance. Its internals are not in the export. From its use here, it consumes `Inp` (32 bits) and `Inp_Bypass` and produces at least `Sts_OK` and `Sts_NBOK`.

Results to the drive object. `Dvc.Inp_IntlkOK` follows `Interlocks.Sts_OK`; `Dvc.Inp_NBIntlkOK` follows `Interlocks.Sts_NBOK`. These two bits are consumed by the `Dvc` drive object executed in the Main routine. `Interlocks.Sts_OK` is also read by the Permissives routine.

Rung 0 is a NOP placeholder with no effect.

Interlocks and permissives

This routine **is** the interlock builder for the drive. All of the following are interlocks; each must be TRUE (OK) for the drive to be allowed to run, subject to the bypass rules inside the Op_Interlock block:

1. **Drive not faulted** (`Dvc.Sts_Fault` = 0) -> Inp.0.
2. **Drive communication connected** (`Dvc.Sts_Connected` = 1) -> Inp.1.
3. **Drive digital input 1 ON** (`Dvc.Sts_DigIn1` = 1) -> Inp.2. Purpose of this hardwired drive input is inferred only; likely a field-wired OK/healthy signal.

4. **Drive safety function not active** (`Dvc.Sts_SafetyActive` = 0) -> Inp.3. **Safety-relevant.** This is a software mirror of the drive's safety status; it lets the control logic stop and block a start when the drive's safety function (e.g. safe torque off) has been asserted. It is not itself a safety-rated path — the safety function is carried by the drive hardware.
5. **Breaker status OK** (`Dvc_DI_Breaker.Sts` = 1) -> Inp.4. Field input conditioned in the Feedback routine.
6. **Disconnect status OK** (`Dvc_DI_Disconnect.Sts` = 1) -> Inp.5. Field input conditioned in the Feedback routine.

Bypass: `Dvc.Sts_Bypass` from the drive object enables interlock bypass in the Op_Interlock block. The non-bypassable result (`Sts_NBOK`) continues to be reported separately so that whatever the block defines as non-bypassable is still enforced through `Dvc.Inp_NBIntlkOK`.

Downstream: `Dvc.Inp_IntlkOK` and `Dvc.Inp_NBIntlkOK` gate the drive object in Main; `Interlocks.Sts_OK` also feeds the Permissives routine.

Alarms and faults

None identified in this routine. The Op_Interlock block may latch or annunciate first-out interlock trips internally; that behavior is inside the AOI and not visible in the export. Drive fault and communication-loss alarming is handled in the Alarms routine.

Review items

- VERIFY: Op_Interlock AOI internals are not in the export. Confirm the exact evaluation: which of Inp.0–Inp.31 are bypassable versus non-bypassable, whether inputs are individually enabled/masked by configuration, whether trips latch and require reset, and whether the block generates any alarm or first-out status.
- VERIFY: Inp.6 is forced permanently TRUE while Inp.7–Inp.31 are forced permanently FALSE via `AlwaysFalse`. This is inconsistent for spare inputs. If the AOI treats every unmasked input as an active interlock, Inp.7–31 would hold the interlocks permanently not-OK; the drive evidently runs, so the AOI most likely masks unused inputs by configuration. Confirm the mask/enable configuration for bits 6–31 and confirm Inp.6 is a spare and not a removed condition.
- VERIFY: `AlwaysFalse` has no description and is not written by any routine. Confirm its initial/stored value is 0 and that it is not forced or written from an HMI or another program.
- VERIFY: Polarity and meaning of `Dvc.Sts_DigIn1`, `Dvc.Sts_SafetyActive`, `Dvc_DI_Breaker.Sts`, and `Dvc_DI_Disconnect.Sts` are inferred from tag names only. Confirm against the Dvc_PF525 and Dvc_DiscreteIn AOI definitions and the field wiring (normally-open vs normally-closed contacts for breaker and disconnect).
- VERIFY: `Dvc.Sts_DigIn1` is used directly as Inp.2 here and is also read by the Feedback routine as a source for one of the three discrete-input objects. Confirm whether the same physical signal is intentionally evaluated in two places, and whether the drive's digital input 1 is wired to a field device or is unused.
- VERIFY: `Dvc.Sts_SafetyActive` is used as a software interlock. Confirm that the drive's safety function (STO) is implemented and validated in hardware independently of this logic, and that this bit is for status/blocking only.
- VERIFY: Confirm `Dvc.Sts_Bypass` is the intended bypass source (e.g. a maintenance-bypass selection in the drive object or HMI) and that bypass authority is appropriately restricted.
- Rung 0 is a NOP with no function; harmless but note for migration.
- Rungs 8–32 (Inp.7–Inp.31 from `AlwaysFalse`) are functionally constant. They are spare placeholders, not dead logic, but carry no live conditions; the migration target can implement them as constants or omit them if the AOI masks unused inputs.

Rung listing

```
[0] NOP();
[1] XIO(Dvc.Sts_Fault)OTE(Interlocks.Inp.0);
[2] XIC(Dvc.Sts_Connected)OTE(Interlocks.Inp.1);
[3] XIC(Dvc.Sts_DigIn1)OTE(Interlocks.Inp.2);
[4] XIO(Dvc.Sts_SafetyActive)OTE(Interlocks.Inp.3);
[5] XIC(Dvc_DI_Breaker.Sts)OTE(Interlocks.Inp.4);
[6] XIC(Dvc_DI_Disconnect.Sts)OTE(Interlocks.Inp.5);
[7] OTE(Interlocks.Inp.6);
[8] XIC(AlwaysFalse)OTE(Interlocks.Inp.7);
[9] XIC(AlwaysFalse)OTE(Interlocks.Inp.8);
[10] XIC(AlwaysFalse)OTE(Interlocks.Inp.9);
[11] XIC(AlwaysFalse)OTE(Interlocks.Inp.10);
[12] XIC(AlwaysFalse)OTE(Interlocks.Inp.11);
[13] XIC(AlwaysFalse)OTE(Interlocks.Inp.12);
[14] XIC(AlwaysFalse)OTE(Interlocks.Inp.13);
[15] XIC(AlwaysFalse)OTE(Interlocks.Inp.14);
[16] XIC(AlwaysFalse)OTE(Interlocks.Inp.15);
[17] XIC(AlwaysFalse)OTE(Interlocks.Inp.16);
[18] XIC(AlwaysFalse)OTE(Interlocks.Inp.17);
[19] XIC(AlwaysFalse)OTE(Interlocks.Inp.18);
[20] XIC(AlwaysFalse)OTE(Interlocks.Inp.19);
[21] XIC(AlwaysFalse)OTE(Interlocks.Inp.20);
[22] XIC(AlwaysFalse)OTE(Interlocks.Inp.21);
[23] XIC(AlwaysFalse)OTE(Interlocks.Inp.22);
[24] XIC(AlwaysFalse)OTE(Interlocks.Inp.23);
[25] XIC(AlwaysFalse)OTE(Interlocks.Inp.24);
[26] XIC(AlwaysFalse)OTE(Interlocks.Inp.25);
[27] XIC(AlwaysFalse)OTE(Interlocks.Inp.26);
[28] XIC(AlwaysFalse)OTE(Interlocks.Inp.27);
[29] XIC(AlwaysFalse)OTE(Interlocks.Inp.28);
[30] XIC(AlwaysFalse)OTE(Interlocks.Inp.29);
[31] XIC(AlwaysFalse)OTE(Interlocks.Inp.30);
[32] XIC(AlwaysFalse)OTE(Interlocks.Inp.31);
[33] XIC(Dvc.Sts_Bypass)OTE(Interlocks.Inp_Bypass);
[34] Op_Interlock(Interlocks);
[35] XIC(Interlocks.Sts_OK)OTE(Dvc.Inp_IntlkOK);
[36] XIC(Interlocks.Sts_NBOK)OTE(Dvc.Inp_NBIntlkOK);
```

Routine: Main

Purpose

The Main routine is the scheduler for program Dev_PF525. It calls the five supporting routines in a fixed order, then executes the Dvc_PF525 device driver instruction that interfaces the controller to the PowerFlex 525 drive over its I/O connection and explicit messages.

Functional description

Routine sequencing (rung 1). Every scan, Main calls the sibling routines in this order: Config, Feedback, Permissives, Interlocks, Alarms. All calls are unconditional. The order matters:

- Feedback runs first so that `Dvc_DI_Breaker.Sts` and `Dvc_DI_Disconnect.Sts` are fresh before the Interlocks routine reads them (see the Feedback and Interlocks routines).
- Permissives runs before Interlocks, but the Permissives routine reads `Interlocks.Sts_OK`. That value is therefore from the previous scan (one-scan lag). See the Permissives routine.
- Alarms reads `Dvc.Sts_Fault`, `Dvc.Sts_CommLoss` and `Dvc.Sts_Disabled`, which are produced by the Dvc_PF525 instruction on rung 3. Because Alarms is called before rung 3, alarm logic also sees status from the previous scan. See the Alarms routine.

Program stop command (rung 2). `Dvc.PCmd_Stop` is driven directly from `AlwaysFalse`. No routine in Dev_PF525 writes `AlwaysFalse`, so as long as it is held false the program stop command to the drive is permanently off. This rung is effectively a placeholder; the program never issues a stop from logic. (Inferred: `PCmd_Stop` is the program-level stop command input to the device driver, based on its name and the naming pattern of the Dvc_PF525

type.)

Device driver (rung 3). The `Dvc_PF525` add-on instruction runs unconditionally with these arguments:

- `Dvc` — the device instance (status, commands, interlock/permissive inputs).
- `Dev_PF525`, `Dev_PF525:I`, `Dev_PF525:O` — the drive module reference and its cyclic input/output image (inferred from the `:I` / `:O` naming used for Ethernet I/O modules).
- `ReadMsg`, `WriteMsg`, `MsgData` — MESSAGE instructions and a data word for explicit (parameter) read/write to the drive (inferred from names and types).
- `SysDevices` — a system-wide device class structure shared with the Feedback routine.

The instruction consumes the interlock and permissive results written by the Interlocks and Permissives routines (`Dvc.Inp_IntlkOK`, `Dvc.Inp_NBIntlkOK`, `Dvc.Inp_PermOK`, `Dvc.Inp_NBPermOK`) and produces the `Dvc.Sts_*` bits used by Feedback, Interlocks, and Alarms. Its internal behavior is not visible in the export.

Rung 0 is a NOP with no effect.

Operator commands. Main contains no logic that writes start, speed reference, or mode commands to `Dvc`. The only command bit written in this program is `Dvc.PCmd_Stop` (forced off). Any run or speed commands must originate outside this program's ladder logic (for example, an HMI writing directly to `Dvc`, or inside the AOI).

Interlocks and permissives

Main does not evaluate interlock or permissive conditions itself. It determines when they are evaluated and hands the results to the drive driver:

- **Interlocks** — `Dvc.Inp_IntlkOK` and `Dvc.Inp_NBIntlkOK` are written by the Interlocks routine (called on rung 1) from drive connection, fault, safety-active status and the breaker/disconnect feedback. The Dvc_PF525 instruction on rung 3 consumes them in the same scan. See the Interlocks routine for the conditions.
- **Permissives** — `Dvc.Inp_PermOK` and `Dvc.Inp_NBPermOK` are written by the Permissives routine and consumed on rung 3. See the Permissives routine.
- **Program stop** — `Dvc.PCmd_Stop` is permanently de-energized by rung 2, so program logic never stops the drive. Stopping relies on interlocks, permissives, operator action, or the drive itself.

How the AOI acts on a false interlock or permissive (immediate stop, block start only, non-bypassable vs bypassable behavior) is inside the AOI and not shown.

Alarms and faults

None raised directly in Main. Drive fault and communication loss alarms are handled in the Alarms routine from `Dvc.Sts_Fault` and `Dvc.Sts_CommLoss`, which the Dvc_PF525 instruction on rung 3 produces. Because the Alarms routine runs before rung 3, alarm inputs lag device status by one scan.

Review items

- VERIFY: Internal behavior of the `Dvc_PF525` add-on instruction (command handling, response to `Inp_IntlkOK` / `Inp_NBIntlkOK` / `Inp_PermOK` / `Inp_NBPermOK`, bypass behavior, fault and comm-loss detection, messaging sequence) is not in the export. Obtain the AOI definition before migration.
- VERIFY: `AlwaysFalse` is not written by any routine in Dev_PF525. Confirm it is a controller-scoped constant that is never set true by another program. If it can become true, rung 2 stops the drive.
- Rung 2 is functionally dead logic: `Dvc.PCmd_Stop` is always off. Confirm with the plant that no program-initiated stop was intended (e.g., a sequence or E-stop bit that was never wired in).
- Rung 0 is a NOP placeholder. Harmless; likely a leftover.
- The Config routine is Structured Text with 0 rungs and its content is not analyzed in the structural digest. See the Config routine section. VERIFY: whether Config contains any active statements or is empty; if empty, the JSR is dead.
- Scan-order lag: Permissives reads `Interlocks.Sts_OK` before Interlocks runs; Alarms and Feedback read `Dvc.Sts_*` before the AOI updates them. Each is a one-scan delay. Acceptable for most drive applications, but

preserve or consciously change this order in the migrated platform.

- VERIFY: Configuration of the `Dev_PF525` module (connection type, RPI, IP address, I/O assembly layout) and the `ReadMsg` / `WriteMsg` message setups (target parameters, paths) are not in the export.
- No start, speed, or mode commands are written to `Dvc` anywhere in this program's ladder logic. VERIFY: Where operator commands enter the device object (HMI direct writes to `Dvc`, or inside the AOI via `SysDevices`).
- Tag purposes for `Dev_PF525`, `Dev_PF525:I`, `Dev_PF525:O`, `ReadMsg`, `WriteMsg`, `MsgData`, and `Dvc.PCmd_Stop` are inferred from names and usage; none carry descriptions in the export.

Rung listing

```
[0] NOP();
[1] JSR(Config,0)JSR(Feedback,0)JSR(Permissives,0)JSR(Interlocks,0)JSR(Alarms,0);
[2] XIC(AlwaysFalse)OTE(Dvc.PCmd_Stop);
[3] Dvc_PF525(Dvc,Dev_PF525,Dev_PF525:I,Dev_PF525:O,ReadMsg,WriteMsg,MsgData,SysDevices);
```

Routine: Permissives

Purpose

Collects the start permissive conditions for the PowerFlex 525 drive (`Dvc`), evaluates them through the `Op_Permissive` instruction, and passes the resulting permissive-OK status bits to the drive control object. Only one of the 32 permissive inputs is in use; the remaining 31 are held false as spares.

Functional description

Permissive input assembly (rungs 1–32). The routine maps 32 discrete conditions into the `Permissives.Inp.0` through `Permissives.Inp.31` bits of the `Op_Permissive` backing tag:

- `Inp.0` is driven by `Interlocks.Sts_OK`, the aggregate interlock-OK status produced by the `Op_Interlock` instruction in the Interlocks routine. This is the only active permissive. Its effect is that the drive's permissive status cannot be OK unless the interlock status is also OK; any interlock trip is also reflected as a lost permissive.
- `Inp.1` through `Inp.31` are each driven by `AlwaysFalse`. These are spare, unused permissive slots. `AlwaysFalse` has no description and is not written by any routine in this program (per the routine digests); it is inferred to be a constant-false BOOL used as a placeholder.

Bypass (rung 33). `Permissives.Inp_Bypass` follows `Dvc.Sts_Bypass`, a status output of the `Dvc_PF525` drive object executed in the Main routine. When the drive object reports bypass active, the permissive object is told to bypass its bypassable inputs. The Interlocks routine applies the same bypass bit to the interlock object.

Evaluation (rung 34). `Op_Permissive(Permissives)` executes the add-on instruction. Its internals are not in the export; it is expected to combine the `Inp.x` bits (honouring `Inp_Bypass` for inputs configured as bypassable) and produce `Sts_OK` and `Sts_NBOK`.

Outputs to the drive object (rungs 35–36).

- `Dvc.Inp_PermOK` follows `Permissives.Sts_OK` — the overall permissive-OK status, with bypass taken into account.
- `Dvc.Inp_NBPermOK` follows `Permissives.Sts_NBOK` — inferred from the name to be the "non-bypassable permissives OK" status, i.e. the state of only those inputs that cannot be bypassed.

Both bits are consumed by the `Dvc_PF525` instruction in the Main routine, which decides how a lost permissive affects the drive (typically blocking a start without stopping a running drive; see the Main routine).

Rung 0 is a NOP with no effect.

Interlocks and permissives

- **Permissive 0 — Interlocks OK (`Interlocks.Sts_OK`).** The drive permissive is satisfied only when the interlock object reports OK. Conditions feeding the interlock object (drive connected, no fault, safety not active, breaker and disconnect status, etc.) are documented in the Interlocks routine.

- **Permissives 1–31.** Unused; held false. Whether a false spare input blocks the permissive depends on how `Op_Permissive` treats unused inputs — see Review items.
- **Bypass.** `Dvc.Sts_Bypass` from the drive object bypasses bypassable permissive inputs. This is safety-relevant: with bypass active, `Dvc.Inp_PermOK` may be true while `Interlocks.Sts_OK` is false, if permissive input 0 is configured as bypassable inside `Op_Permissive`. `Dvc.Inp_NBPermOK` provides the non-bypassed view.

Alarms and faults

None identified. This routine generates no alarms; drive fault and communication-loss alarms are handled in the Alarms routine.

Review items

- VERIFY: `Op_Permissive` instruction internals are not in the export. Confirm (a) how inputs held false by `AlwaysFalse` (Inp.1–31) are treated — whether they must be masked/disabled by configuration in the `Permissives` tag or in the Config routine to avoid permanently blocking the permissive, (b) which inputs are configured as bypassable, and (c) the exact meaning of `Sts_OK` versus `Sts_NBOK`.
- VERIFY: `AlwaysFalse` has no description and is not written anywhere in the program. Confirm it is a constant-false tag (default value 0, not forced or written externally). If it were ever set true, spare permissives 1–31 would all assert simultaneously.
- VERIFY: Confirm the intended behaviour of `Dvc.Sts_Bypass` — where bypass is initiated (HMI, `Dvc_PF525` configuration) and whether bypassing the interlock-derived permissive is acceptable for this drive. Bypass allows the drive object to see permissives OK while interlocks are not OK, subject to the bypassable configuration above.
- VERIFY: Execution order in the Main routine. Permissives is called from Main rung 1 and reads `Interlocks.Sts_OK`; if the Interlocks routine is called after Permissives in the same scan, `Permissives.Inp.0` lags the true interlock state by one scan. See the Main routine for the call sequence.
- Note: Permissive input 0 duplicates the interlock status. This is a design choice (interlock trip also removes the start permissive), not an error, but the migration team should confirm it is intentional and not a placeholder awaiting real permissive conditions.
- Note: Rung 0 is a NOP and rungs 2–32 are effectively dead logic (spare slots). No functional impact if the spare inputs are correctly configured in the `Op_Permissive` instruction.

Rung listing

```
[0] NOP();
[1] XIC(Interlocks.Sts_OK)OTE(Permissives.Inp.0);
[2] XIC(AlwaysFalse)OTE(Permissives.Inp.1);
[3] XIC(AlwaysFalse)OTE(Permissives.Inp.2);
[4] XIC(AlwaysFalse)OTE(Permissives.Inp.3);
[5] XIC(AlwaysFalse)OTE(Permissives.Inp.4);
[6] XIC(AlwaysFalse)OTE(Permissives.Inp.5);
[7] XIC(AlwaysFalse)OTE(Permissives.Inp.6);
[8] XIC(AlwaysFalse)OTE(Permissives.Inp.7);
[9] XIC(AlwaysFalse)OTE(Permissives.Inp.8);
[10] XIC(AlwaysFalse)OTE(Permissives.Inp.9);
[11] XIC(AlwaysFalse)OTE(Permissives.Inp.10);
[12] XIC(AlwaysFalse)OTE(Permissives.Inp.11);
[13] XIC(AlwaysFalse)OTE(Permissives.Inp.12);
[14] XIC(AlwaysFalse)OTE(Permissives.Inp.13);
[15] XIC(AlwaysFalse)OTE(Permissives.Inp.14);
[16] XIC(AlwaysFalse)OTE(Permissives.Inp.15);
[17] XIC(AlwaysFalse)OTE(Permissives.Inp.16);
[18] XIC(AlwaysFalse)OTE(Permissives.Inp.17);
[19] XIC(AlwaysFalse)OTE(Permissives.Inp.18);
[20] XIC(AlwaysFalse)OTE(Permissives.Inp.19);
[21] XIC(AlwaysFalse)OTE(Permissives.Inp.20);
[22] XIC(AlwaysFalse)OTE(Permissives.Inp.21);
[23] XIC(AlwaysFalse)OTE(Permissives.Inp.22);
[24] XIC(AlwaysFalse)OTE(Permissives.Inp.23);
[25] XIC(AlwaysFalse)OTE(Permissives.Inp.24);
[26] XIC(AlwaysFalse)OTE(Permissives.Inp.25);
[27] XIC(AlwaysFalse)OTE(Permissives.Inp.26);
[28] XIC(AlwaysFalse)OTE(Permissives.Inp.27);
[29] XIC(AlwaysFalse)OTE(Permissives.Inp.28);
[30] XIC(AlwaysFalse)OTE(Permissives.Inp.29);
[31] XIC(AlwaysFalse)OTE(Permissives.Inp.30);
[32] XIC(AlwaysFalse)OTE(Permissives.Inp.31);
[33] XIC(Dvc.Sts_Bypass)OTE(Permissives.Inp_Bypass);
[34] Op_Permissive(Permissives);
[35] XIC(Permissives.Sts_OK)OTE(Dvc.Inp_PermOK);
[36] XIC(Permissives.Sts_NBOK)OTE(Dvc.Inp_NBPermOK);
```

Tag Dictionary

Tag	Scope	Type	Kind	Description	Reads	Writes
Alm_CommLoss	Dev_PF525	Sys_Alarm	Base		1	3
Alm_Fault	Dev_PF525	Sys_Alarm	Base		1	3
AlwaysFalse	Dev_PF525	BOOL	Base · constant		58	0
Dvc	Dev_PF525	Dvc_PF525	Base · Public parameter		13	6
Dvc_DI_Breaker	Dev_PF525	Dvc_DiscreteIn	Base · Public parameter		2	2
Dvc_DI_Disconnect	Dev_PF525	Dvc_DiscreteIn	Base · Public parameter		2	2
Dvc_DI_Feedback	Dev_PF525	Dvc_DiscreteIn	Base · Public parameter		1	2
Interlocks	Dev_PF525	Op_Interlock	Base		4	34
MsgData	Dev_PF525	DINT	Base		1	1
Permissives	Dev_PF525	Op_Permissive	Base		3	34
ReadMsg	Dev_PF525	MESSAGE	Base		1	1
WriteMsg	Dev_PF525	MESSAGE	Base		1	1
_StrCond	controller	STR_40	Base		0	0
_StrTemp	controller	STR_40	Base		0	0
SysAlarms	controller	ST_Sys_AlarmClass	Base		2	2
SysDevices	controller	ST_Sys_DeviceClass	Base		4	4

Cross-Reference

Routine call graph

- `Dev_PF525/Main` rung 1 -> `Config`
- `Dev_PF525/Main` rung 1 -> `Feedback`
- `Dev_PF525/Main` rung 1 -> `Permissives`
- `Dev_PF525/Main` rung 1 -> `Interlocks`
- `Dev_PF525/Main` rung 1 -> `Alarms`

Tag references

`Alm_CommLoss` (program `Dev_PF525`)

Access	Reference	Location	Instruction
write	`Alm_CommLoss.Inp`	Dev_PF525/Alarms rung 2	OTE
write	`Alm_CommLoss.Inp_Disable`	Dev_PF525/Alarms rung 2	OTE
read	`Alm_CommLoss`	Dev_PF525/Alarms rung 2	SYS_ALARM
write	`Alm_CommLoss`	Dev_PF525/Alarms rung 2	SYS_ALARM

`Alm_Fault` (program `Dev_PF525`)

Access	Reference	Location	Instruction
write	`Alm_Fault.Inp`	Dev_PF525/Alarms rung 1	OTE
write	`Alm_Fault.Inp_Disable`	Dev_PF525/Alarms rung 1	OTE
read	`Alm_Fault`	Dev_PF525/Alarms rung 1	SYS_ALARM
write	`Alm_Fault`	Dev_PF525/Alarms rung 1	SYS_ALARM

`AlwaysFalse` (program `Dev_PF525`)

Access	Reference	Location	Instruction
read	`AlwaysFalse`	Dev_PF525/Feedback rung 3	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 8	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 9	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 10	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 11	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 12	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 13	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 14	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 15	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 16	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 17	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 18	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 19	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 20	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 21	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 22	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 23	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 24	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 25	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 26	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 27	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 28	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 29	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 30	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 31	XIC
read	`AlwaysFalse`	Dev_PF525/Interlocks rung 32	XIC
read	`AlwaysFalse`	Dev_PF525/Main rung 2	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 2	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 3	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 4	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 5	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 6	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 7	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 8	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 9	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 10	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 11	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 12	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 13	XIC
read	`AlwaysFalse`	Dev_PF525/Permissives rung 14	XIC

Access	Reference	Location	Instruction
read	AlwaysFalse	Dev_PF525/Permissives rung 15	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 16	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 17	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 18	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 19	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 20	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 21	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 22	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 23	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 24	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 25	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 26	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 27	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 28	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 29	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 30	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 31	XIC
read	AlwaysFalse	Dev_PF525/Permissives rung 32	XIC

Dev_PF525

Access	Reference	Location	Instruction
read	Dev_PF525	Dev_PF525/Main rung 3	DVC_PF525
write	Dev_PF525	Dev_PF525/Main rung 3	DVC_PF525

Dev_PF525:I

Access	Reference	Location	Instruction
read	Dev_PF525:I	Dev_PF525/Main rung 3	DVC_PF525
write	Dev_PF525:I	Dev_PF525/Main rung 3	DVC_PF525

Dev_PF525:O

Access	Reference	Location	Instruction
read	Dev_PF525:O	Dev_PF525/Main rung 3	DVC_PF525
write	Dev_PF525:O	Dev_PF525/Main rung 3	DVC_PF525

Dvc (program Dev_PF525)

Access	Reference	Location	Instruction
read	Dvc.Sts_CommLoss	Dev_PF525/Alarms rung 1	XIO
read	Dvc.Sts_Fault	Dev_PF525/Alarms rung 1	XIC
read	Dvc.Sts_Disabled	Dev_PF525/Alarms rung 1	XIC
read	Dvc.Sts_CommLoss	Dev_PF525/Alarms rung 2	XIC
read	Dvc.Sts_Disabled	Dev_PF525/Alarms rung 2	XIC
read	Dvc.Sts_DigIn1	Dev_PF525/Feedback rung 2	XIC
read	Dvc.Sts_Fault	Dev_PF525/Interlocks rung 1	XIO
read	Dvc.Sts_Connected	Dev_PF525/Interlocks rung 2	XIC
read	Dvc.Sts_DigIn1	Dev_PF525/Interlocks rung 3	XIC
read	Dvc.Sts_SafetyActive	Dev_PF525/Interlocks rung 4	XIO
read	Dvc.Sts_Bypass	Dev_PF525/Interlocks rung 33	XIC
write	Dvc.Inp_IntlkOK	Dev_PF525/Interlocks rung 35	OTE
write	Dvc.Inp_NBIntlkOK	Dev_PF525/Interlocks rung 36	OTE
write	Dvc.PCmd_Stop	Dev_PF525/Main rung 2	OTE
read	Dvc	Dev_PF525/Main rung 3	DVC_PF525
write	Dvc	Dev_PF525/Main rung 3	DVC_PF525
read	Dvc.Sts_Bypass	Dev_PF525/Permissives rung 33	XIC
write	Dvc.Inp_PermOK	Dev_PF525/Permissives rung 35	OTE
write	Dvc.Inp_NBPermOK	Dev_PF525/Permissives rung 36	OTE

Dvc_DI_Breaker (program Dev_PF525)

Access	Reference	Location	Instruction
write	Dvc_DI_Breaker.Inp_Data	Dev_PF525/Feedback rung 1	OTE
read	Dvc_DI_Breaker	Dev_PF525/Feedback rung 1	DVC_DISCRETEIN
write	Dvc_DI_Breaker	Dev_PF525/Feedback rung 1	DVC_DISCRETEIN
read	Dvc_DI_Breaker.Sts	Dev_PF525/Interlocks rung 5	XIC

Dvc_DI_Disconnect (program Dev_PF525)

Access	Reference	Location	Instruction
write	Dvc_DI_Disconnect.Inp_Data	Dev_PF525/Feedback rung 2	OTE
read	Dvc_DI_Disconnect	Dev_PF525/Feedback rung 2	DVC_DISCRETEIN
write	Dvc_DI_Disconnect	Dev_PF525/Feedback rung 2	DVC_DISCRETEIN
read	Dvc_DI_Disconnect.Sts	Dev_PF525/Interlocks rung 6	XIC

Dvc_DI_Feedback (program Dev_PF525)

Access	Reference	Location	Instruction
write	Dvc_DI_Feedback.Inp_Data	Dev_PF525/Feedback rung 3	OTE
read	Dvc_DI_Feedback	Dev_PF525/Feedback rung 3	DVC_DISCRETEIN
write	Dvc_DI_Feedback	Dev_PF525/Feedback rung 3	DVC_DISCRETEIN

`Interlocks` (program `Dev_PF525`)

Access	Reference	Location	Instruction
write	`Interlocks.Inp.0`	Dev_PF525/Interlocks rung 1	OTE
write	`Interlocks.Inp.1`	Dev_PF525/Interlocks rung 2	OTE
write	`Interlocks.Inp.2`	Dev_PF525/Interlocks rung 3	OTE
write	`Interlocks.Inp.3`	Dev_PF525/Interlocks rung 4	OTE
write	`Interlocks.Inp.4`	Dev_PF525/Interlocks rung 5	OTE
write	`Interlocks.Inp.5`	Dev_PF525/Interlocks rung 6	OTE
write	`Interlocks.Inp.6`	Dev_PF525/Interlocks rung 7	OTE
write	`Interlocks.Inp.7`	Dev_PF525/Interlocks rung 8	OTE
write	`Interlocks.Inp.8`	Dev_PF525/Interlocks rung 9	OTE
write	`Interlocks.Inp.9`	Dev_PF525/Interlocks rung 10	OTE
write	`Interlocks.Inp.10`	Dev_PF525/Interlocks rung 11	OTE
write	`Interlocks.Inp.11`	Dev_PF525/Interlocks rung 12	OTE
write	`Interlocks.Inp.12`	Dev_PF525/Interlocks rung 13	OTE
write	`Interlocks.Inp.13`	Dev_PF525/Interlocks rung 14	OTE
write	`Interlocks.Inp.14`	Dev_PF525/Interlocks rung 15	OTE
write	`Interlocks.Inp.15`	Dev_PF525/Interlocks rung 16	OTE
write	`Interlocks.Inp.16`	Dev_PF525/Interlocks rung 17	OTE
write	`Interlocks.Inp.17`	Dev_PF525/Interlocks rung 18	OTE
write	`Interlocks.Inp.18`	Dev_PF525/Interlocks rung 19	OTE
write	`Interlocks.Inp.19`	Dev_PF525/Interlocks rung 20	OTE
write	`Interlocks.Inp.20`	Dev_PF525/Interlocks rung 21	OTE
write	`Interlocks.Inp.21`	Dev_PF525/Interlocks rung 22	OTE
write	`Interlocks.Inp.22`	Dev_PF525/Interlocks rung 23	OTE
write	`Interlocks.Inp.23`	Dev_PF525/Interlocks rung 24	OTE
write	`Interlocks.Inp.24`	Dev_PF525/Interlocks rung 25	OTE
write	`Interlocks.Inp.25`	Dev_PF525/Interlocks rung 26	OTE
write	`Interlocks.Inp.26`	Dev_PF525/Interlocks rung 27	OTE
write	`Interlocks.Inp.27`	Dev_PF525/Interlocks rung 28	OTE
write	`Interlocks.Inp.28`	Dev_PF525/Interlocks rung 29	OTE
write	`Interlocks.Inp.29`	Dev_PF525/Interlocks rung 30	OTE
write	`Interlocks.Inp.30`	Dev_PF525/Interlocks rung 31	OTE
write	`Interlocks.Inp.31`	Dev_PF525/Interlocks rung 32	OTE
write	`Interlocks.Inp_Bypass`	Dev_PF525/Interlocks rung 33	OTE
read	`Interlocks`	Dev_PF525/Interlocks rung 34	OP_INTERLOCK
write	`Interlocks`	Dev_PF525/Interlocks rung 34	OP_INTERLOCK
read	`Interlocks.Sts_OK`	Dev_PF525/Interlocks rung 35	XIC
read	`Interlocks.Sts_NBOK`	Dev_PF525/Interlocks rung 36	XIC
read	`Interlocks.Sts_OK`	Dev_PF525/Permissives rung 1	XIC

`Local:3:I`

Access	Reference	Location	Instruction
read	`Local:3:I.Pt03.Data`	Dev_PF525/Feedback rung 1	XIC

`MsgData` (program `Dev_PF525`)

Access	Reference	Location	Instruction
read	`MsgData`	Dev_PF525/Main rung 3	DVC_PF525
write	`MsgData`	Dev_PF525/Main rung 3	DVC_PF525

`Permissives` (program `Dev_PF525`)

Access	Reference	Location	Instruction
write	`Permissives.Inp.0`	Dev_PF525/Permissives rung 1	OTE
write	`Permissives.Inp.1`	Dev_PF525/Permissives rung 2	OTE
write	`Permissives.Inp.2`	Dev_PF525/Permissives rung 3	OTE
write	`Permissives.Inp.3`	Dev_PF525/Permissives rung 4	OTE
write	`Permissives.Inp.4`	Dev_PF525/Permissives rung 5	OTE
write	`Permissives.Inp.5`	Dev_PF525/Permissives rung 6	OTE
write	`Permissives.Inp.6`	Dev_PF525/Permissives rung 7	OTE
write	`Permissives.Inp.7`	Dev_PF525/Permissives rung 8	OTE
write	`Permissives.Inp.8`	Dev_PF525/Permissives rung 9	OTE
write	`Permissives.Inp.9`	Dev_PF525/Permissives rung 10	OTE
write	`Permissives.Inp.10`	Dev_PF525/Permissives rung 11	OTE
write	`Permissives.Inp.11`	Dev_PF525/Permissives rung 12	OTE
write	`Permissives.Inp.12`	Dev_PF525/Permissives rung 13	OTE
write	`Permissives.Inp.13`	Dev_PF525/Permissives rung 14	OTE
write	`Permissives.Inp.14`	Dev_PF525/Permissives rung 15	OTE
write	`Permissives.Inp.15`	Dev_PF525/Permissives rung 16	OTE
write	`Permissives.Inp.16`	Dev_PF525/Permissives rung 17	OTE
write	`Permissives.Inp.17`	Dev_PF525/Permissives rung 18	OTE
write	`Permissives.Inp.18`	Dev_PF525/Permissives rung 19	OTE
write	`Permissives.Inp.19`	Dev_PF525/Permissives rung 20	OTE
write	`Permissives.Inp.20`	Dev_PF525/Permissives rung 21	OTE
write	`Permissives.Inp.21`	Dev_PF525/Permissives rung 22	OTE
write	`Permissives.Inp.22`	Dev_PF525/Permissives rung 23	OTE
write	`Permissives.Inp.23`	Dev_PF525/Permissives rung 24	OTE
write	`Permissives.Inp.24`	Dev_PF525/Permissives rung 25	OTE
write	`Permissives.Inp.25`	Dev_PF525/Permissives rung 26	OTE
write	`Permissives.Inp.26`	Dev_PF525/Permissives rung 27	OTE
write	`Permissives.Inp.27`	Dev_PF525/Permissives rung 28	OTE
write	`Permissives.Inp.28`	Dev_PF525/Permissives rung 29	OTE
write	`Permissives.Inp.29`	Dev_PF525/Permissives rung 30	OTE
write	`Permissives.Inp.30`	Dev_PF525/Permissives rung 31	OTE
write	`Permissives.Inp.31`	Dev_PF525/Permissives rung 32	OTE
write	`Permissives.Inp_Bypass`	Dev_PF525/Permissives rung 33	OTE
read	`Permissives`	Dev_PF525/Permissives rung 34	OP_PERMISSIVE
write	`Permissives`	Dev_PF525/Permissives rung 34	OP_PERMISSIVE
read	`Permissives.Sts_OK`	Dev_PF525/Permissives rung 35	XIC
read	`Permissives.Sts_NBOK`	Dev_PF525/Permissives rung 36	XIC

`ReadMsg` (program `Dev_PF525`)

Access	Reference	Location	Instruction
read	`ReadMsg`	Dev_PF525/Main rung 3	DVC_PF525
write	`ReadMsg`	Dev_PF525/Main rung 3	DVC_PF525

`SysAlarms`

Access	Reference	Location	Instruction
read	`SysAlarms`	Dev_PF525/Alarms rung 1	SYS_ALARM
write	`SysAlarms`	Dev_PF525/Alarms rung 1	SYS_ALARM
read	`SysAlarms`	Dev_PF525/Alarms rung 2	SYS_ALARM
write	`SysAlarms`	Dev_PF525/Alarms rung 2	SYS_ALARM

`SysDevices`

Access	Reference	Location	Instruction
read	`SysDevices`	Dev_PF525/Feedback rung 1	DVC_DISCRETEIN
write	`SysDevices`	Dev_PF525/Feedback rung 1	DVC_DISCRETEIN
read	`SysDevices`	Dev_PF525/Feedback rung 2	DVC_DISCRETEIN
write	`SysDevices`	Dev_PF525/Feedback rung 2	DVC_DISCRETEIN
read	`SysDevices`	Dev_PF525/Feedback rung 3	DVC_DISCRETEIN
write	`SysDevices`	Dev_PF525/Feedback rung 3	DVC_DISCRETEIN
read	`SysDevices`	Dev_PF525/Main rung 3	DVC_PF525
write	`SysDevices`	Dev_PF525/Main rung 3	DVC_PF525

`WriteMsg` (program `Dev_PF525`)

Access	Reference	Location	Instruction
read	`WriteMsg`	Dev_PF525/Main rung 3	DVC_PF525
write	`WriteMsg`	Dev_PF525/Main rung 3	DVC_PF525

Engineering Review Findings

3 finding(s). Each item needs a human ruling before migration: keep, migrate as-is, or retire.

Disabled logic (AFI) (0)

None found.

Uncalled routines (0)

None found.

Empty routines (0)

None found.

Unscheduled programs (never execute) (0)

None found.

Calls into missing routines (0)

None found.

Pending online edits (0)

None found.

Source-protected routines (opaque) (0)

None found.

Unparsed routines (non-ladder) (1)

- `Dev_PF525/Config [ST]` — content not analyzed (ST is scanned for JSR calls only); tag usage inside it is invisible to the cross-reference

  Caveat: *some logic could not be analyzed (protected routines or unparseable rungs). Unused/read-only/write-only tag findings below may be false positives — the opaque logic may reference them.*

Unused tags (defined, never referenced) (2)

- `_StrCond`
- `_StrTemp`

Write-only tags (written, never read by logic) (0)

None found.

Read-only tags (read, never written by logic) (0)

None found.

Rungs that failed to parse (0)

None found.

FAT Test Plan

Output coverage matrix

Every boolean output written by logic, with the rung(s) that drive it. Each row needs at least one executed test.

Output	Description	Driven by	Tested	Result
Dev_PF525/Alm_CommLoss.Inp		Dev_PF525/Alarms rung 2 (OTE)	[ ]	
Dev_PF525/Alm_CommLoss.Inp_Disable		Dev_PF525/Alarms rung 2 (OTE)	[ ]	
Dev_PF525/Alm_Fault.Inp		Dev_PF525/Alarms rung 1 (OTE)	[ ]	
Dev_PF525/Alm_Fault.Inp_Disable		Dev_PF525/Alarms rung 1 (OTE)	[ ]	
Dev_PF525/Dvc.Inp_IntlkOK		Dev_PF525/Interlocks rung 35 (OTE)	[ ]	
Dev_PF525/Dvc.Inp_NBIntlkOK		Dev_PF525/Interlocks rung 36 (OTE)	[ ]	
Dev_PF525/Dvc.Inp_NBPermOK		Dev_PF525/Permissives rung 36 (OTE)	[ ]	
Dev_PF525/Dvc.Inp_PermOK		Dev_PF525/Permissives rung 35 (OTE)	[ ]	
Dev_PF525/Dvc.PCmd_Stop		Dev_PF525/Main rung 2 (OTE)	[ ]	
Dev_PF525/Dvc_DI_Breaker.Inp_Data		Dev_PF525/Feedback rung 1 (OTE)	[ ]	
Dev_PF525/Dvc_DI_Disconnect.Inp_Data		Dev_PF525/Feedback rung 2 (OTE)	[ ]	
Dev_PF525/Dvc_DI_Feedback.Inp_Data		Dev_PF525/Feedback rung 3 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.0		Dev_PF525/Interlocks rung 1 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.1		Dev_PF525/Interlocks rung 2 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.10		Dev_PF525/Interlocks rung 11 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.11		Dev_PF525/Interlocks rung 12 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.12		Dev_PF525/Interlocks rung 13 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.13		Dev_PF525/Interlocks rung 14 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.14		Dev_PF525/Interlocks rung 15 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.15		Dev_PF525/Interlocks rung 16 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.16		Dev_PF525/Interlocks rung 17 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.17		Dev_PF525/Interlocks rung 18 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.18		Dev_PF525/Interlocks rung 19 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.19		Dev_PF525/Interlocks rung 20 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.2		Dev_PF525/Interlocks rung 3 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.20		Dev_PF525/Interlocks rung 21 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.21		Dev_PF525/Interlocks rung 22 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.22		Dev_PF525/Interlocks rung 23 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.23		Dev_PF525/Interlocks rung 24 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.24		Dev_PF525/Interlocks rung 25 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.25		Dev_PF525/Interlocks rung 26 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.26		Dev_PF525/Interlocks rung 27 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.27		Dev_PF525/Interlocks rung 28 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.28		Dev_PF525/Interlocks rung 29 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.29		Dev_PF525/Interlocks rung 30 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.3		Dev_PF525/Interlocks rung 4 (OTE)	[ ]	

Output	Description	Driven by	Tested	Result
Dev_PF525/Interlocks.Inp.30		Dev_PF525/Interlocks rung 31 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.31		Dev_PF525/Interlocks rung 32 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.4		Dev_PF525/Interlocks rung 5 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.5		Dev_PF525/Interlocks rung 6 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.6		Dev_PF525/Interlocks rung 7 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.7		Dev_PF525/Interlocks rung 8 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.8		Dev_PF525/Interlocks rung 9 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp.9		Dev_PF525/Interlocks rung 10 (OTE)	[ ]	
Dev_PF525/Interlocks.Inp_Bypass		Dev_PF525/Interlocks rung 33 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.0		Dev_PF525/Permissives rung 1 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.1		Dev_PF525/Permissives rung 2 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.10		Dev_PF525/Permissives rung 11 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.11		Dev_PF525/Permissives rung 12 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.12		Dev_PF525/Permissives rung 13 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.13		Dev_PF525/Permissives rung 14 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.14		Dev_PF525/Permissives rung 15 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.15		Dev_PF525/Permissives rung 16 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.16		Dev_PF525/Permissives rung 17 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.17		Dev_PF525/Permissives rung 18 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.18		Dev_PF525/Permissives rung 19 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.19		Dev_PF525/Permissives rung 20 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.2		Dev_PF525/Permissives rung 3 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.20		Dev_PF525/Permissives rung 21 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.21		Dev_PF525/Permissives rung 22 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.22		Dev_PF525/Permissives rung 23 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.23		Dev_PF525/Permissives rung 24 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.24		Dev_PF525/Permissives rung 25 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.25		Dev_PF525/Permissives rung 26 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.26		Dev_PF525/Permissives rung 27 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.27		Dev_PF525/Permissives rung 28 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.28		Dev_PF525/Permissives rung 29 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.29		Dev_PF525/Permissives rung 30 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.3		Dev_PF525/Permissives rung 4 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.30		Dev_PF525/Permissives rung 31 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.31		Dev_PF525/Permissives rung 32 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.4		Dev_PF525/Permissives rung 5 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.5		Dev_PF525/Permissives rung 6 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.6		Dev_PF525/Permissives rung 7 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.7		Dev_PF525/Permissives rung 8 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.8		Dev_PF525/Permissives rung 9 (OTE)	[ ]	
Dev_PF525/Permissives.Inp.9		Dev_PF525/Permissives rung 10 (OTE)	[ ]	
Dev_PF525/Permissives.Inp_Bypass		Dev_PF525/Permissives rung 33 (OTE)	[ ]	

Drafted test steps

Dev_PF525/Alm_CommLoss.Inp

1. **Setup:** Force `Dvc.Sts_CommLoss` = 0. Run the Alarms routine (via Main scan).
   - **Action:** Set `Dvc.Sts_CommLoss` = 1.
   - **Expected result:** `Alm_CommLoss.Inp` energizes (1). Returning `Dvc.Sts_CommLoss` to 0 de-energizes it.

VERIFY: `Dvc.Sts_CommLoss` is a member of the `Dvc_PF525` device structure. Confirm how it is driven on the bench (device AOI internal / comms status), and how it can be simulated from the test panel.

Dev_PF525/Alm_CommLoss.Inp_Disable

1. **Setup:** Force `Dvc.Sts_Disabled` = 0.
   - **Action:** Set `Dvc.Sts_Disabled` = 1.
   - **Expected result:** `Alm_CommLoss.Inp_Disable` energizes (1). Returning `Dvc.Sts_Disabled` to 0 de-energizes it.

VERIFY: Confirm the source and bench-simulation method for `Dvc.Sts_Disabled`.

Dev_PF525/Alm_Fault.Inp

1. **Setup:** Force `Dvc.Sts_CommLoss` = 0 and `Dvc.Sts_Fault` = 0.
   - **Action:** Set `Dvc.Sts_Fault` = 1 (keep `Dvc.Sts_CommLoss` = 0).
   - **Expected result:** `Alm_Fault.Inp` energizes (1).
2. **Setup:** Force `Dvc.Sts_Fault` = 1 and `Dvc.Sts_CommLoss` = 0 (from test 1, `Alm_Fault.Inp` = 1).
   - **Action:** Set `Dvc.Sts_CommLoss` = 1.
   - **Expected result:** `Alm_Fault.Inp` de-energizes (0). Comm loss masks the fault alarm input.

VERIFY: Confirm source and bench-simulation method for `Dvc.Sts_Fault` and `Dvc.Sts_CommLoss`.

Dev_PF525/Alm_Fault.Inp_Disable

1. **Setup:** Force `Dvc.Sts_Disabled` = 0.
   - **Action:** Set `Dvc.Sts_Disabled` = 1.
   - **Expected result:** `Alm_Fault.Inp_Disable` energizes (1). Returning `Dvc.Sts_Disabled` to 0 de-energizes it.

VERIFY: Confirm source and bench-simulation method for `Dvc.Sts_Disabled`.

Dev_PF525/Dvc.Inp_IntlkOK

1. **Setup:** Establish conditions so `Interlocks.Sts_OK` = 0 (at least one enabled interlock input not satisfied — see the Interlocks routine, which computes `Sts_OK` from `Interlocks.Inp.0…31`).
   - **Action:** Drive all enabled interlock inputs to their OK state so `Interlocks.Sts_OK` transitions to 1.
   - **Expected result:** `Dvc.Inp_IntlkOK` follows `Interlocks.Sts_OK`: 0 when `Sts_OK` = 0, 1 when `Sts_OK` = 1.

VERIFY: `Interlocks.Sts_OK` is an output of the `Op_Interlock` AOI. The internal logic that produces it (bypass handling, which Inp bits are enabled) is not visible in the rung export. Confirm the exact input combination required to force `Sts_OK` on the bench.

Dev_PF525/Dvc.Inp_NBIntlkOK

1. **Setup:** Establish conditions so `Interlocks.Sts_NBOK` = 0.
   - **Action:** Drive the non-bypassable interlock inputs to their OK state so `Interlocks.Sts_NBOK` transitions to 1.
   - **Expected result:** `Dvc.Inp_NBIntlkOK` follows `Interlocks.Sts_NBOK`: 0 when 0, 1 when 1.

VERIFY: `Interlocks.Sts_NBOK` ("non-bypassable interlocks OK" — inferred from name) is an `Op_Interlock` AOI output. Confirm which Inp bits are non-bypassable and the combination needed to force it on the bench.

Dev_PF525/Dvc.Inp_NBPermOK

1. **Setup:** Establish conditions so `Permissives.Sts_NBOK` = 0 (see the Permissives routine).
   - **Action:** Drive the non-bypassable permissive inputs to their OK state so `Permissives.Sts_NBOK` transitions to 1.
   - **Expected result:** `Dvc.Inp_NBPermOK` follows `Permissives.Sts_NBOK`: 0 when 0, 1 when 1.

VERIFY: `Permissives.Sts_NBOK` ("non-bypassable permissives OK" — inferred) is an `Op_Permissive` AOI output. Confirm which Inp bits drive it and the combination needed on the bench.

Dev_PF525/Dvc.Inp_PermOK

1. **Setup:** Establish conditions so `Permissives.Sts_OK` = 0.
   - **Action:** Drive all enabled permissive inputs to their OK state so `Permissives.Sts_OK` transitions to 1.
   - **Expected result:** `Dvc.Inp_PermOK` follows `Permissives.Sts_OK`: 0 when 0, 1 when 1.

VERIFY: `Permissives.Sts_OK` is an `Op_Permissive` AOI output. Confirm the input combination required to force it on the bench. Note the Permissives routine reads `Interlocks.Sts_OK`; confirm whether interlock state influences `Permissives.Sts_OK`.

Dev_PF525/Dvc.PCmd_Stop

1. **Setup:** Run the Main routine. Confirm `AlwaysFalse` = 0 (it is the hard-off/never-true bit inferred from its name; the Feedback and Interlocks routines also use it as a constant-false source).
   - **Action:** Attempt every normal operating sequence available on the test panel; do not (and you cannot) set `AlwaysFalse` = 1 through logic.
   - **Expected result:** `Dvc.PCmd_Stop` remains de-energized (0) at all times. This output is hard-wired off by design.

VERIFY: Confirm `AlwaysFalse` is intended to remain 0 in all operating states. If a stop command is expected to be functional after migration, this rung disables it — flag for engineering review.

Dev_PF525/Dvc_DI_Breaker.Inp_Data

1. **Setup:** Simulate the breaker input at `Local:3:I.Pt03.Data` = 0 (de-energize the corresponding channel on the test panel).
   - **Action:** Energize the input so `Local:3:I.Pt03.Data` = 1.
   - **Expected result:** `Dvc_DI_Breaker.Inp_Data` follows the input: 0 when the channel is off, 1 when on.

VERIFY: `Local:3:I.Pt03.Data` is a physical input point (slot 3, point 03). Confirm the field device / wiring assignment and the panel simulation method.

Dev_PF525/Dvc_DI_Disconnect.Inp_Data

1. **Setup:** Force `Dvc.Sts_DigIn1` = 0.
   - **Action:** Set `Dvc.Sts_DigIn1` = 1.
   - **Expected result:** `Dvc_DI_Disconnect.Inp_Data` follows `Dvc.Sts_DigIn1`: 0 when 0, 1 when 1.

VERIFY: The disconnect feedback is sourced from device status bit `Dvc.Sts_DigIn1` (a drive digital-input status, inferred from name), not from a discrete field point. Confirm this mapping and the bench-simulation method for `Dvc.Sts_DigIn1`.

Dev_PF525/Dvc_DI_Feedback.Inp_Data

1. **Setup:** Run the Feedback routine. Confirm `AlwaysFalse` = 0.
   - **Action:** Exercise all available inputs; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Dvc_DI_Feedback.Inp_Data` remains de-energized (0) at all times. This feedback input is hard-wired off by design.

VERIFY: Confirm whether `Dvc_DI_Feedback.Inp_Data` is intended to be permanently 0, or whether a real feedback source is to be wired in after migration. As written, no field signal drives this input.

Dev_PF525/Interlocks.Inp.0

1. **Setup:** Force `Dvc.Sts_Fault` = 1.
   - **Action:** Set `Dvc.Sts_Fault` = 0.
   - **Expected result:** `Interlocks.Inp.0` energizes (1) when `Dvc.Sts_Fault` = 0, and de-energizes (0) when `Dvc.Sts_Fault` = 1. This interlock input represents "no device fault" (inferred from the XIO of the fault status).

VERIFY: Confirm source and bench-simulation method for `Dvc.Sts_Fault`.

Dev_PF525/Interlocks.Inp.1

1. **Setup:** Force `Dvc.Sts_Connected` = 0.
   - **Action:** Set `Dvc.Sts_Connected` = 1.
   - **Expected result:** `Interlocks.Inp.1` follows `Dvc.Sts_Connected`: 0 when 0, 1 when 1. This interlock input requires device communication/connection.

VERIFY: Confirm source and bench-simulation method for `Dvc.Sts_Connected`.

Dev_PF525/Interlocks.Inp.10

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.
   - **Action:** None available; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Interlocks.Inp.10` remains de-energized (0) at all times. This interlock input is a spare/unused slot tied off false.

VERIFY: Confirm `Interlocks.Inp.10` is an intentionally unused interlock input (permanently 0). Note: an interlock input held at 0 may itself hold `Interlocks.Sts_OK` off depending on AOI enable configuration — confirm against the `Op_Interlock` setup.

Dev_PF525/Interlocks.Inp.11

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.
   - **Action:** None available; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Interlocks.Inp.11` remains de-energized (0) at all times. Spare/unused interlock input tied off false.

VERIFY: Confirm `Interlocks.Inp.11` is intentionally unused and how the `Op_Interlock` AOI treats an input held at 0.

Dev_PF525/Interlocks.Inp.12

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.
   - **Action:** None available; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Interlocks.Inp.12` remains de-energized (0) at all times. Spare/unused interlock input tied off false.

VERIFY: Confirm `Interlocks.Inp.12` is intentionally unused and how the `Op_Interlock` AOI treats an input held at 0.

Dev_PF525/Interlocks.Inp.13

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.
   - **Action:** None available; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Interlocks.Inp.13` remains de-energized (0) at all times. Spare/unused interlock input tied off false.

VERIFY: Confirm `Interlocks.Inp.13` is intentionally unused and how the `Op_Interlock` AOI treats an input held at 0.

Dev_PF525/Interlocks.Inp.14

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.

- **Action:** None available; `AlwaysFalse` cannot be set true through logic.
- **Expected result:** `Interlocks.Inp.14` remains de-energized (0) at all times. Spare/unused interlock input tied off false.

VERIFY: Confirm `Interlocks.Inp.14` is intentionally unused and how the `Op_Interlock` AOI treats an input held at 0.

Dev_PF525/Interlocks.Inp.15

1. **Setup:** Run the Interlocks routine. Confirm `AlwaysFalse` = 0.
   - **Action:** None available; `AlwaysFalse` cannot be set true through logic.
   - **Expected result:** `Interlocks.Inp.15` remains de-energized (0) at all times. Spare/unused interlock input tied off false.

VERIFY: Confirm `Interlocks.Inp.15` is intentionally unused and how the `Op_Interlock` AOI treats an input held at 0.

Interlocks.Inp.16

1. **Setup:** Establish normal test-panel conditions. Confirm the `AlwaysFalse` bit is held at 0 (it is an internal constant, not a field input).
   - **Action:** No field input drives this rung. Attempt no change; observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.16` remains 0 at all times. It is an unused interlock slot hard-driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` is confirmed to be initialized and held at 0. There is no field means to set this bit; it can only become 1 if `AlwaysFalse` is corrupted or written elsewhere.

Interlocks.Inp.17

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.17` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.18

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.18` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.19

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.19` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.2

1. **Setup:** Establish conditions so that `Dvc.Sts_DigIn1` is 0 (device digital-input status de-asserted). Confirm `Interlocks.Inp.2` reads 0.
   - **Action:** Drive `Dvc.Sts_DigIn1` to 1 by simulating the associated device digital input from the test panel.
   - **Expected result:** `Interlocks.Inp.2` follows to 1 while `Dvc.Sts_DigIn1` is 1, and returns to 0 when `Dvc.Sts_DigIn1` returns to 0.

VERIFY: Source of `Dvc.Sts_DigIn1`. It is a status member of the `Dvc` (Dvc_PF525) object and is also read by the Feedback routine; how it is populated (drive comms / device instruction internals) is not visible in this rung. Determine

the correct test-panel stimulus to set it.

Interlocks.Inp.20

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.20` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.21

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.21` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.22

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.22` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.23

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.23` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.24

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.24` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.25

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.25` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.26

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.26` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.27

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.27` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.28

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.28` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.29

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.29` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.3

1. **Setup:** Establish conditions so that `Dvc.Sts_SafetyActive` is 1 (device safety function active). Confirm `Interlocks.Inp.3` reads 0. (This rung is an XIO — inverted logic. Inp.3 represents the "safety not active" interlock condition.)
   - **Action:** Drive `Dvc.Sts_SafetyActive` to 0 by clearing the simulated safety-active condition from the test panel.
   - **Expected result:** `Interlocks.Inp.3` goes to 1 while `Dvc.Sts_SafetyActive` is 0, and returns to 0 when `Dvc.Sts_SafetyActive` is 1.

This is safety-relevant: `Interlocks.Inp.3` is de-asserted (0) whenever the device reports safety active. Confirm both transitions.

VERIFY: Source and meaning of `Dvc.Sts_SafetyActive`. It is a status member of the `Dvc` object; how it is populated (drive/safety instruction internals) is not visible in this rung. Determine the correct test-panel stimulus.

Interlocks.Inp.30

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.30` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.31

1. **Setup:** Confirm `AlwaysFalse` is held at 0.
   - **Action:** No field input drives this rung. Observe the output through one or more scans.
   - **Expected result:** `Interlocks.Inp.31` remains 0. Unused interlock slot driven by `AlwaysFalse`.

VERIFY: `AlwaysFalse` confirmed held at 0.

Interlocks.Inp.4

1. **Setup:** Establish conditions so that `Dvc_DI_Breaker.Sts` is 0. Confirm `Interlocks.Inp.4` reads 0.
   - **Action:** Drive `Dvc_DI_Breaker.Sts` to 1 by simulating the breaker discrete input from the test panel. (The Feedback routine processes the breaker input via `Dvc_DI_Breaker.Inp_Data`; the corresponding channel is `Local:3:I` — see the Feedback routine.)
   - **Expected result:** `Interlocks.Inp.4` follows to 1 while `Dvc_DI_Breaker.Sts` is 1, and returns to 0 when the breaker status returns to 0.

VERIFY: The physical input point and any on/off delay applied to `Dvc_DI_Breaker.Sts` inside the Dvc_DiscreteIn instruction. Instruction internals and I/O wiring are not visible here; confirm the correct test-panel channel and any filtering.

Interlocks.Inp.5

1. **Setup:** Establish conditions so that `Dvc_DI_Disconnect.Sts` is 0. Confirm `Interlocks.Inp.5` reads 0.
   - **Action:** Drive `Dvc_DI_Disconnect.Sts` to 1 by simulating the disconnect discrete input from the test panel. (Processed by the Feedback routine via `Dvc_DI_Disconnect.Inp_Data`.)
   - **Expected result:** `Interlocks.Inp.5` follows to 1 while `Dvc_DI_Disconnect.Sts` is 1, and returns to 0 when the disconnect status returns to 0.

VERIFY: The physical input point and any on/off delay applied to `Dvc_DI_Disconnect.Sts` inside the Dvc_DiscreteIn instruction. Instruction internals and I/O wiring are not visible here; confirm the correct test-panel channel and any filtering.

Dev_PF525/Interlocks.Inp.6

This output is driven by an unconditional OTE (rung 7 has no input conditions). It energizes on every scan while the Interlocks routine runs.

1. **Setup:** Download and run the program. Establish whatever conditions allow the Main routine to call the Interlocks routine each scan (normal run state).
2. **Action:** Observe `Interlocks.Inp.6` with no input manipulation.
3. **Expected result:** `Interlocks.Inp.6` is continuously energized (ON) whenever the Interlocks routine is scanned. There is no input condition that turns it off.

VERIFY: The purpose of this always-true interlock input bit is not documented. Confirm with the customer that bit 6 is intended to be permanently asserted.

Dev_PF525/Interlocks.Inp.7

Driven by `XIC(AlwaysFalse)`. `AlwaysFalse` has no description; it is inferred to be a tag held permanently at 0, used to hard-disable this rung.

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Interlocks routine with `AlwaysFalse` = 0.
3. **Expected result:** `Interlocks.Inp.7` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` is never written to 1 anywhere (it is written only in the Feedback routine reads per digest — confirm it is a constant). If it can be forced to 1, bit 7 would follow it.

Dev_PF525/Interlocks.Inp.8

Driven by `XIC(AlwaysFalse)`. See note on `AlwaysFalse` above (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Interlocks routine.
3. **Expected result:** `Interlocks.Inp.8` remains de-energized (OFF) at all times.

VERIFY: Same as Inp.7 — confirm `AlwaysFalse` cannot be set.

Dev_PF525/Interlocks.Inp.9

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Interlocks routine.
3. **Expected result:** `Interlocks.Inp.9` remains de-energized (OFF) at all times.

VERIFY: Same as Inp.7 — confirm `AlwaysFalse` cannot be set.

Dev_PF525/Interlocks.Inp_Bypass

Driven by `XIC(Dvc.Sts_Bypass)`. This bit mirrors the device bypass status into the interlock object. `Dvc.Sts_Bypass` is a status bit of the `Dvc` (Dvc_PF525) device object; it is not written by this routine.

1. **Setup:** Download and run. Establish `Dvc.Sts_Bypass` = 0 (device not in bypass).
2. **Action:** Force / command the device into bypass so that `Dvc.Sts_Bypass` transitions 0 -> 1, then back to 0.
3. **Expected result:** `Interlocks.Inp_Bypass` follows `Dvc.Sts_Bypass`: OFF when status = 0, ON when status = 1. No delay in logic.

VERIFY: `Dvc.Sts_Bypass` is set inside the `Dvc_PF525` device instruction/AOI. Confirm the test-panel method for asserting device bypass (HMI command, MSG, or direct force). AOI internals are not in this export.

Dev_PF525/Permissives.Inp.0

Driven by `XIC(Interlocks.Sts_OK)`. This carries the "interlocks OK" result into the permissive object as permissive input 0. `Interlocks.Sts_OK` is the status output of the `Interlocks` (Op_Interlock) object; it is evaluated in the Interlocks routine, not here.

1. **Setup:** Download and run. Drive the interlock inputs so that `Interlocks.Sts_OK` = 0 (interlocks not satisfied).
2. **Action:** Change interlock inputs so that `Interlocks.Sts_OK` transitions 0 -> 1, then back to 0.
3. **Expected result:** `Permissives.Inp.0` follows `Interlocks.Sts_OK`: OFF when interlocks are not OK, ON when interlocks are OK.

VERIFY: `Interlocks.Sts_OK` is produced by the Op_Interlock instruction (see the Interlocks routine). Confirm the exact input combination that drives `Sts_OK` true; the instruction internals are not in this export.

Dev_PF525/Permissives.Inp.1

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.1` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.10

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.10` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.11

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.11` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.12

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.12` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.13

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.13` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.14

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.14` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.15

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.15` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.16

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.16` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.17

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.17` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.18

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.18` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.19

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.

3. **Expected result:** `Permissives.Inp.19` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.2

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.2` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.20

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.20` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Dev_PF525/Permissives.Inp.21

Driven by `XIC(AlwaysFalse)` (inferred constant-false).

1. **Setup:** Download and run. Confirm `AlwaysFalse` = 0.
2. **Action:** Scan the Permissives routine.
3. **Expected result:** `Permissives.Inp.21` remains de-energized (OFF) at all times.

VERIFY: Confirm `AlwaysFalse` cannot be set to 1.

Permissives.Inp.3

Test PERM-03 — Spare permissive bit 3 held false Driving logic: Permissives rung 4 — `Permissives.Inp.3` follows `AlwaysFalse`. No test-panel signal, drive status, or HMI action feeds this rung. Purpose inferred from name and usage: unused permissive slot parked false (tag has no description).

1. Setup: Controller in Run, Main calling Permissives. No forces active. Confirm `AlwaysFalse` = 0 in the online monitor. Record `Permissives.Inp.3`.
2. Action: From the test panel, toggle every simulated field and drive condition available (Local:3:I.Pt03, drive connected/fault/safety/bypass). Do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.3` remains 0 throughout. It never energizes regardless of any input.

VERIFY: `AlwaysFalse` must hold value 0 by configuration. Confirm no logic or startup routine writes it. Bit is de-energized by design (spare slot).

Permissives.Inp.4

Test PERM-04 — Spare permissive bit 4 held false Driving logic: Permissives rung 5 — `Permissives.Inp.4` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.4`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.4` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.5

Test PERM-05 — Spare permissive bit 5 held false Driving logic: Permissives rung 6 — `Permissives.Inp.5` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.5`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.5` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.6

Test PERM-06 — Spare permissive bit 6 held false Driving logic: Permissives rung 7 — `Permissives.Inp.6` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.6`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.6` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.7

Test PERM-07 — Spare permissive bit 7 held false Driving logic: Permissives rung 8 — `Permissives.Inp.7` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.7`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.7` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.8

Test PERM-08 — Spare permissive bit 8 held false Driving logic: Permissives rung 9 — `Permissives.Inp.8` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.8`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.8` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.9

Test PERM-09 — Spare permissive bit 9 held false Driving logic: Permissives rung 10 — `Permissives.Inp.9` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.9`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.9` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.22

Test PERM-22 — Spare permissive bit 22 held false Driving logic: Permissives rung 23 — `Permissives.Inp.22` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.22`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.22` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.23

Test PERM-23 — Spare permissive bit 23 held false Driving logic: Permissives rung 24 — `Permissives.Inp.23` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.23`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.23` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.24

Test PERM-24 — Spare permissive bit 24 held false Driving logic: Permissives rung 25 — `Permissives.Inp.24` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.24`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.24` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.25

Test PERM-25 — Spare permissive bit 25 held false Driving logic: Permissives rung 26 — `Permissives.Inp.25` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.25`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.25` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.26

Test PERM-26 — Spare permissive bit 26 held false Driving logic: Permissives rung 27 — `Permissives.Inp.26` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.26`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.26` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.27

Test PERM-27 — Spare permissive bit 27 held false Driving logic: Permissives rung 28 — `Permissives.Inp.27` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.27`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.27` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.28

Test PERM-28 — Spare permissive bit 28 held false Driving logic: Permissives rung 29 — `Permissives.Inp.28` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.28`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.28` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.29

Test PERM-29 — Spare permissive bit 29 held false Driving logic: Permissives rung 30 — `Permissives.Inp.29` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.29`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.29` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.30

Test PERM-30 — Spare permissive bit 30 held false Driving logic: Permissives rung 31 — `Permissives.Inp.30` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.30`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.30` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp.31

Test PERM-31 — Spare permissive bit 31 held false Driving logic: Permissives rung 32 — `Permissives.Inp.31` follows `AlwaysFalse`.

1. Setup: Controller in Run, Permissives executing. No forces. Confirm `AlwaysFalse` = 0. Record `Permissives.Inp.31`.
2. Action: Exercise all available panel inputs; do not force `AlwaysFalse`.
3. Expected result: `Permissives.Inp.31` remains 0 throughout.

VERIFY: `AlwaysFalse` holds 0 by configuration. Spare slot de-energized by design.

Permissives.Inp_Bypass

Test PERM-BYP — Permissive bypass flag follows device bypass status Driving logic: Permissives rung 33 — `Permissives.Inp_Bypass` follows `Dvc.Sts_Bypass`. This bit signals that permissives are bypassed. Treat as safety-relevant: bypassing permissives defeats start protection. Confirm downstream use in the Op_Permissive instruction internals.

Part A — bypass inactive

1. Setup: Controller in Run, Permissives executing. Establish `Dvc.Sts_Bypass` = 0 (device not in bypass). Record `Permissives.Inp_Bypass`.
2. Action: Hold `Dvc.Sts_Bypass` = 0.
3. Expected result: `Permissives.Inp_Bypass` = 0.

Part B — bypass active

1. Setup: Controller in Run, Permissives executing. `Dvc.Sts_Bypass` = 0 initially, `Permissives.Inp_Bypass` = 0.
2. Action: Drive `Dvc.Sts_Bypass` = 1 (place the device in bypass by the intended mechanism).
3. Expected result: `Permissives.Inp_Bypass` energizes to 1 in the same/next scan. Returning `Dvc.Sts_Bypass` to 0 returns `Permissives.Inp_Bypass` to 0.

VERIFY: `Dvc.Sts_Bypass` is produced inside the `Dvc` (Dvc_PF525) device object; it is not written by any routine in this program per the digests. Confirm on the bench how bypass status is set (HMI command / AOI internal) so the witness can drive it. VERIFY: effect of `Inp_Bypass` inside the Op_Permissive instruction (whether it forces `Sts_OK`/`Sts_NBOK`) is internal to the instruction and not visible in this rung.