ABAP Debug Script — A Practitioner's Reference

Preface

While the standard debugger helps you trace one execution path at a time, Debug Script lets you **programmatically monitor, control, and repeat** debugging logic.
This makes it a silent companion for diagnosing hidden issues, investigating performance bottlenecks, or tracking variable changes deep inside complex frameworks.

This document is written in *human terms* but with SME-level precision — for those who truly want to understand what happens **inside** the ABAP runtime.

---

1. Understanding ABAP Debug Script

1.1 What It Really Is

A Debug Script is essentially a *mini ABAP program* that the debugger executes while you are debugging another ABAP program.

It extends the **New ABAP Debugger Framework** by adding scripting capabilities, meaning:

You can write ABAP logic that runs *within* the debugger, automating observations and actions during debugging.

You'll find it under:
Script Tab → Script Editor / Script Wizard / Script Monitor

---

2. Why Debug Script Exists

2.1 The Typical Developer Pain

- You want to know where a variable changes value, but you can't watch every assignment.
- You want to trace when a flag or structure field is updated, but it happens deep inside nested methods.
- You want to debug a background job or RFC call — where `/h` won't help.
- You want to collect runtime behavior metrics without stopping the process.

2.2 Debug Script Solves This

- It automates repetitive debugging steps.
- It can read runtime values and react automatically.
- It works even for background jobs and external sessions.
- It can capture patterns, log information, and trigger breakpoints under specific conditions.

In short: **Instead of you debugging the code, Debug Script debugs for you.**

---

3. Key Components

Component	Description
Script Editor	Where you write your ABAP script logic.
Script Wizard	Helps you generate skeletons and templates.
Script Monitor	Displays output, logs, and runtime info.
Execution Control	Start, stop, or step through script execution.

---

4. Anatomy of a Debug Script

Every debug script is an ABAP class implementing the interface `IF_TPDA_SCRIPT_TOOL`.

Core Methods

Method	Description
`SCRIPT_START`	Triggered when script execution begins.
`SCRIPT_STEP`	Called during each debugger step (main logic here).
`SCRIPT_STOP`	Triggered when debugging session ends.

Sample Structure

```
CLASS zcl_my_debug_script DEFINITION.
  PUBLIC SECTION.
    INTERFACES: if_tpda_script_tool.
ENDCLASS.

CLASS zcl_my_debug_script IMPLEMENTATION.

  METHOD if_tpda_script_tool~script_start.
    WRITE: / 'Debug Script Started'.
  ENDMETHOD.

  METHOD if_tpda_script_tool~script_step.
    DATA lv_var TYPE i.
    TRY.
        lv_var = tpda_script_data_descr->get_variable_value( 'MY_VAR' ).
        IF lv_var > 100.
          tpda_script_breakpoint->break( ).
        ENDIF.
```

```
      CATCH cx_tpda_script_error INTO DATA(lx_err).
        WRITE: / lx_err->get_text( ).
    ENDTRY.
  ENDMETHOD.

  METHOD if_tpda_script_tool~script_stop.
    WRITE: / 'Debug Script Completed'.
  ENDMETHOD.

ENDCLASS.
```

---

5. Practical Scenarios

Scenario 1: Detect Variable Changes

Goal: Identify which part of the program changes a global variable like `GV_STATUS`.

```
METHOD if_tpda_script_tool~script_step.
  DATA(lv_curr) = tpda_script_data_descr->get_variable_value( 'GV_STATUS'
).
  IF lv_curr <> gv_prev.
    WRITE: / 'GV_STATUS changed to:', lv_curr.
    gv_prev = lv_curr.
  ENDIF.
ENDMETHOD.
```

Use Case:
Finding where `GV_STATUS` flips during a workflow or pricing routine.

Scenario 2: Monitor Method Calls

Goal: Find out how many times a method executes unnecessarily.

```
METHOD if_tpda_script_tool~script_step.
  DATA lt_stack TYPE tpda_script_stackdescr_tab.
  lt_stack = tpda_script_stack_descr->get_callstack( ).

  LOOP AT lt_stack INTO DATA(ls_stack).
    IF ls_stack-progname CP '*ZCL_BILLING*' AND
       ls_stack-methodname = 'UPDATE_PRICES'.
      gv_count += 1.
      WRITE: / 'UPDATE_PRICES called:', gv_count.
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

Use Case:
Tracing duplicate method calls in SD pricing or invoice generation.

Scenario 3: Debug Background Jobs

Attach the debugger externally, run the script silently to monitor data changes, and log outputs when conditions occur.

Use Case:
Analyze RFC or batch job behavior where manual breakpoints cannot be used.

Scenario 4: Conditional Breakpoints

Trigger breakpoints only when a certain data pattern is found.

```
IF tpda_script_data_descr->get_variable_value( 'LV_AMOUNT' ) > 100000
   AND tpda_script_data_descr->get_variable_value( 'LV_CURRENCY' ) = 'INR'.
  tpda_script_breakpoint->break( ).
ENDIF.
```

Avoids stopping for every iteration.

Scenario 5: Performance Observation

Count SELECTs or LOOPs and time them using system variables to understand micro performance within a transaction.

---

6. Best Practices and Expert Tips

1. **Always use TRY...CATCH** – avoid crashing debugger.
2. **Do not modify data** – only observe, unless absolutely required.
3. **Use Script Wizard** – for template generation.
4. **Reuse and Share Scripts** – maintain a script library (XML files).
5. **Combine with External Debugging** – ideal for RFC, OData, and background.
6. **Use logs, not UI writes** – let the script monitor handle output.

---

7. Known Limitations

Limitation	Explanation
No DB operations	SELECT/INSERT not allowed.
No UI actions	`WRITE` works only in Script Monitor.
Scope bound	Ends when debugging session ends.
Non-transportable	Must be saved locally or as XML.

8. Advanced Debugging Patterns

These patterns elevate a good debugger into an SME-level investigator.

8.1 Dynamic Watchlist Pattern

Problem:
You don't know upfront which variable name holds a specific value (e.g., document number, status, or control flag). It might differ in each include or method.

Goal:
Automatically scan all local and global variables in every step, identify those containing a particular value (like a specific sales document), and log their path.

Pattern Approach:

1. Maintain a dynamic list of variable names.
2. Loop through them using `tpda_script_data_descr->get_variable_value()`.
3. Check if any variable holds your target value.
4. Log matches to the Script Monitor.

Sample Implementation:

```
DATA: lt_vars TYPE STANDARD TABLE OF string,
      lv_value TYPE string.

lt_vars = VALUE #( ( 'LV_VBELN' ) ( 'GV_VBELN' ) ( 'MS_HEADER-VBELN' ) (
'GS_DOC-VBELN' ) ).
lv_target = '0080001234'.

LOOP AT lt_vars INTO DATA(lv_varname).
  TRY.
      DATA(lv_val) = tpda_script_data_descr->get_variable_value( lv_varname
).
      IF lv_val = lv_target.
        WRITE: / 'Match found:', lv_varname, ' = ', lv_val.
      ENDIF.
    CATCH cx_tpda_script_error.
      CONTINUE.
  ENDTRY.
ENDLOOP.
```

Real-Time Scenario:
Used in SD order creation (VA01) when a specific VBELN appears wrongly duplicated after a BADI enhancement — the developer didn't know which structure was carrying the wrong data during runtime. The script auto-detected it in `MS_HEADER` of a customer include.

8.2 Conditional Stack Trace Pattern

Problem:
You need to know *which method or function* updates a field but only when a certain condition is true (e.g., when `STATUS = 'E'` and `AMOUNT > 0`).

Goal:
Instead of stopping every time the variable changes, collect a stack trace *only* when the condition is met.

Pattern Approach:

1. Evaluate the condition.
2. If condition true, fetch the current call stack using `get_callstack()`.
3. Log it to Script Monitor or an internal table.

Code Sample:

```
DATA: lt_stack TYPE tpda_script_stackdescr_tab.

DATA(lv_status) = tpda_script_data_descr->get_variable_value( 'GV_STATUS'
).
DATA(lv_amt)    = tpda_script_data_descr->get_variable_value( 'LV_AMOUNT'
).

IF lv_status = 'E' AND lv_amt > 0.
  lt_stack = tpda_script_stack_descr->get_callstack( ).
  WRITE: / 'Triggered at:', sy-uzeit.
  LOOP AT lt_stack INTO DATA(ls_stack).
    WRITE: / ls_stack-progname, ls_stack-methodname, ls_stack-linenum.
  ENDLOOP.
ENDIF.
```

Real-Time Scenario:
While debugging a custom `BAPI_SALESORDER_CREATEFROMDAT2` wrapper, a pricing error flag was set deep inside a custom function module chain. This pattern revealed the exact enhancement spot where `GV_STATUS` flipped from blank to `E`.

8.3 Delta Pattern (Change Tracking)

Problem:
You have a structure (e.g., `GS_ORDER`) with dozens of fields, and you want to know which field changed between two execution steps.

Goal:
Automatically detect and display changed fields in structures during debugging.

Pattern Approach:

1. Take a snapshot of structure values at every step.

2. Compare with previous snapshot.
3. Log differences.

Code Sample:

```
DATA: ls_prev TYPE any,
      ls_curr TYPE any.

TRY.
    ls_curr = tpda_script_data_descr->get_variable_value( 'GS_ORDER' ).
    LOOP AT COMPONENTS OF ls_curr INTO DATA(lv_field).
      ASSIGN COMPONENT lv_field OF STRUCTURE ls_curr TO FIELD-
SYMBOL(<curr>).
      ASSIGN COMPONENT lv_field OF STRUCTURE ls_prev TO FIELD-
SYMBOL(<prev>).
      IF <curr> <> <prev>.
        WRITE: / 'Field changed:', lv_field, 'Old:', <prev>, 'New:',
<curr>.
      ENDIF.
    ENDLOOP.
    ls_prev = ls_curr.
  CATCH cx_tpda_script_error.
ENDTRY.
```

Real-Time Scenario:
In MM invoice posting (`MIRO`), `GS_HEADER` fields changed between BADI `INVOICE_UPDATE` and standard posting logic. This pattern caught the unexpected alteration of `BELNR` by a legacy Z-enhancement.

8.4 Event-Driven Script Activation

Problem:
You don't want your script to execute every single step — it slows debugging.
You only want it to "wake up" when a certain function or method is entered.

Goal:
Attach logic dynamically based on entry into a specific module or call stack.

Pattern Approach:

- Inspect the current top of the stack each step.
- Trigger the rest of your script logic only when it matches your target function.

Code Sample:

```
DATA(ls_top) = tpda_script_stack_descr->get_callstack( )[ 1 ].
IF ls_top-funcname = 'ZFM_ORDER_PRICE_CALC'.
  "Now execute watch logic
  DATA(lv_total) = tpda_script_data_descr->get_variable_value( 'LV_TOTAL'
).
  IF lv_total > 50000.
    WRITE: / 'High value detected:', lv_total.
```

```
  ENDIF.
ENDIF.
```

Real-Time Scenario:
In SD pricing flow, you only want to monitor variable `LV_TOTAL` inside the pricing function module, not in every routine. This pattern ensures zero noise outside the relevant logic.

8.5 Auto-Termination Pattern

Problem:
Your script keeps running after the target condition is already found.

Goal:
Stop script execution after specific conditions are fulfilled to save performance.

Pattern Approach:
Use a counter or condition flag and stop script programmatically.

Code Sample:

```
IF gv_found = abap_true.
  tpda_script_debugger_ctrl->stop_script( ).
  RETURN.
ENDIF.
```

Real-Time Scenario:
Used when debugging RFC-based ALE posting — once the target IDoc was found to be altered, script terminated automatically to prevent endless tracing in mass processing.

9. Real-World Use Cases by Expert ABAPers

Case 1 — Silent Mutation of Workflow Container Values (Workflow / SWF)

Context: Intermittent workflow step behavior — container element `APPROVAL_FLAG` flips to `'N'` during parallel events.
Why hard: Flag flips only in production load; local testing doesn't reproduce; many workflows/agent tasks touch same container.
Approach: Attach debug script to WFM runtime methods and use **Delta Pattern** on the container internal table. Log timestamp + stack when change occurs.
Pattern used: Delta Pattern + Conditional Stack Trace.
Snippet (conceptual):

```
DATA(ls_curr) = tpda_script_data_descr->get_variable_value( 'LT_CONTAINER'
).
"Compare with ls_prev and log changed rows with stack
IF changed.
  lt_stack = tpda_script_stack_descr->get_callstack( ).
  WRITE: / sy-uzeit, 'Container changed by:', lt_stack[1]-progname,
lt_stack[1]-methodname.
ENDIF.
```

Outcome & Fix: Found a legacy background task `ZWF_AUTO_RESET` that executed on event callback and reset the flag — disabled the task and added enqueue protection.
Tips: Always log sy-uname and RFC caller info; use auto-terminate once flag flip captured.

Case 2 — Intermittent Pricing Condition Missing (SD / Pricing)

Context: Some orders fail pricing with "condition record not found" even though records exist. Happens sporadically per sales org.
Why hard: Pricing logic is dynamic with multiple enhancements and condition copies; intermittent makes ST05 noisy.
Approach: Use **Conditional Stack Trace** when `KOMV-KSCHL = 'ZDIS' AND KOMV-KBETR = 0` to capture where an unexpected override happens.
Pattern used: Conditional Stack Trace + Event Activation (only in pricing FM).
Snippet:

```
IF tpda_script_data_descr->get_variable_value( 'KOMV-KSCHL' ) = 'ZDIS'
   AND tpda_script_data_descr->get_variable_value( 'KOMV-KBETR' ) = 0.
  lt_stack = tpda_script_stack_descr->get_callstack( ).
  WRITE: / lt_stack[ 1 ]-progname, lt_stack[ 1 ]-methodname.
ENDIF.
```

Outcome & Fix: A dormant Z-formula (inactive in VK11) still executed due to bad copy logic in a Z routine — removed the copy and added explicit checks.
Tips: Reproduce for a single sales org and capture VO-chain (pricing procedure, access sequence).

Case 3 — Recursive BADI Leading to Infinite Loop (FI Interface)

Context: Financial interface enters recursion causing CPU spike and job hang.
Why hard: Recursive calls originate via COMMIT/ENQUEUE in enhancement and only show under certain data conditions.
Approach: Use **Event-Driven Activation** on the BADI method and **Auto-Termination** when recursive depth > N. Log entire callstack and caller.
Pattern used: Event Driven + Auto Termination.
Snippet:

```
lt_stack = tpda_script_stack_descr->get_callstack( ).
IF lines( lt_stack ) > 6.
  WRITE: / 'Recursion >6 - stack captured', lt_stack.
  tpda_script_debugger_ctrl->stop_script( ).
ENDIF.
```

Outcome & Fix: Found COMMIT WORK inside a BADI leading to re-entry; removed COMMIT and moved logic; put guards to avoid re-entry.
Tips: For CPU loops, also capture sy-uzeit and count loop hits per method to decide auto stop.

Case 4 — Background Job Writes Wrong Clearing Document (FI Background Job)

Context: Nightly clearing job occasionally posts wrong document. No breakpoint possible.
Why hard: Issue appears only under heavy parallel load and specific file inputs.
Approach: Use **External Debugging** + **Dynamic Watchlist** on `GV_DOC_NO`/`GV_CLEARED`; auto-terminate after N mismatches; write logs to shared table or directory for post analysis.
Pattern used: Dynamic Watchlist + Auto-Termination.
Snippet:

```
lv_doc = tpda_script_data_descr->get_variable_value( 'GV_DOC_NO' ).
IF lv_doc <> gv_prev AND lv_doc Is not initial.
  WRITE: / 'doc changed from', gv_prev, 'to', lv_doc, 'at', sy-uzeit.
  gv_prev = lv_doc.
ENDIF.
```

Outcome & Fix: Traced back to an ALV-reuse routine that reset fields for a test path; added boundary checks.
Tips: Persist debug outputs to a Z table with request id + timestamp for non-interactive review.

Case 5 — Performance Regression in PO Release (MM / ME29N)

Context: ME29N rose from 8s to 90s after enhancement transport.
Why hard: ST05 showed many selects but unclear origin; the enhancement was obscure.
Approach: Script counted SELECT calls inside release class `CL_ME_REL_STRATEGY` and logged caller stack each select.
Pattern used: Performance Observation (select counting) + Conditional Stack.
Snippet:

```
"Pseudo: increment counter when SELECT is executed (monitored by location
in code)
gv_select_hits += 1.
IF gv_select_hits MOD 10 = 0.
  WRITE: / 'SELECT hit count:', gv_select_hits.
```

```
ENDIF.
```

Outcome & Fix: Found `SELECT SINGLE` inside loop added by Z enhancement; refactored to single JOIN select; time back to ~8s.
Tips: Use short windows of tracing (auto-terminate) to avoid flood; correlate with DB statistics.

Case 6 — IDoc Processing Flag Reset (IDoc / EDI)

Context: Goods movement IDoc fails with "Document already posted" as posted flag resets to blank.
Why hard: IDoc processing is asynchronous; many includes access same flags.
Approach: Dynamic Watchlist scanning common flag names across includes (e.g., `LV_FLAG_POSTED`, `GS_HDR-PSTNG`). Log stack when found reset to space.
Pattern used: Dynamic Watchlist + Delta Pattern.
Snippet:

```
LOOP AT lt_vars INTO lv_var.
  TRY.
    lv_val = tpda_script_data_descr->get_variable_value( lv_var ).
    IF lv_val = space AND lv_prev[ lv_var ] = 'X'.
      WRITE: / 'Reset detected:', lv_var, 'by', tpda_script_stack_descr-
>get_callstack( )[1]-progname.
    ENDIF.
  ENDTRY.
ENDLOOP.
```

Outcome & Fix: Legacy Z enhancement for test IDocs reset the flag; commented it and added a check for production partners.
Tips: Always capture partner numbers and message type with the log.

Case 7 — Parallel RFCs Causing Duplicate Z-Table Entries (Concurrency)

Context: Same key inserted twice by parallel RFC calls — data corruption.
Why hard: Race condition only under concurrency.
Approach: Conditional Stack Trace when same `ZKEY` observed twice in short window; log caller node, RFC client, and enqueue usage.
Pattern used: Conditional Stack Trace + Dynamic Watchlist.
Snippet:

```
lv_zkey = tpda_script_data_descr->get_variable_value( 'ZKEY' ).
IF ztable_tracker[ lv_zkey ] = 'X'.
  WRITE: / 'Duplicate attempt for', lv_zkey, 'by', tpda_script_stack_descr-
>get_callstack( )[1]-progname.
ENDIF.
ztable_tracker[ lv_zkey ] = 'X'.
```

Outcome & Fix: Missing `ENQUEUE` call in wrapper; added `ENQUEUE_EZKEY` and retry logic.
Tips: For RFC scenarios, capture `sy-uname`, `sy-repid`, and `sy-sycno`.

Case 8 — Static Variable Retention Across Requests (Memory / Static)

Context: A static variable retains value between requests causing logic to behave wrong after long uptime.
Why hard: Static retention subtle; appears after specific sequence of calls.
Approach: Use script to watch static area variables and log creation stack of static assignment. Use **Delta Pattern** to detect first non-initial set.
Pattern used: Delta Pattern + Event Activation on relevant class loading method.
Snippet:

```
lv_static = tpda_script_data_descr->get_variable_value(
'zcl_cache=>gv_static' ).
IF lv_static <> gv_prev.
  WRITE: / 'Static set at', tpda_script_stack_descr->get_callstack( )[1]-
progname.
  gv_prev = lv_static.
ENDIF.
```

Outcome & Fix: Found lazy initialization in a utility class that executed once and then changed behavior; fixed with proper reset on RFC start.
Tips: Consider `SET/GET PARAMETER` difference and application server instance isolation.

Case 9 — OData Service Returning Wrong Payload (Gateway / Fiori)

Context: OData payload missing fields intermittently for certain users.
Why hard: Backend service uses user-specific authorization checks and a BADI that filters data. Issue only for some user groups.
Approach: Attach script in the gateway call sequence and monitor `et_entity` internal table before serialization; capture user and role info.
Pattern used: Event Activation + Dynamic Watchlist.
Snippet:

```
lt_entities = tpda_script_data_descr->get_variable_value( 'ET_ENTITY' ).
IF lines( lt_entities ) = 0.
  WRITE: / 'Empty payload for user', tpda_script_data_descr-
>get_variable_value( 'SY-UNAME' ), tpda_script_stack_descr->get_callstack(
)[1]-progname.
ENDIF.
```

Outcome & Fix: BADI applied additional filter for certain roles; adjusted logic and added role mapping.
Tips: Also check ICF and user mapping; capture HTTP headers if possible.

Case 10 — CDS View Unexpected Data (HANA / CDS)

Context: A CDS view exposed in Fiori returns fewer rows after a HANA rewrite. Standard trace shows nothing.
Why hard: View combines client-specific filters and a new AMDP logic changed null handling.
Approach: Use debug script to monitor the selection parameters passed to the view provider class and capture the generated SQL (if provider exists in ABAP).
Pattern used: Conditional Stack Trace + Data Snapshot.
Snippet:

```
lv_sel = tpda_script_data_descr->get_variable_value( 'LT_SELECTION' ).
IF lv_sel IS NOT INITIAL.
  WRITE: / 'Selection at', tpda_script_stack_descr->get_callstack( )[1]-
progname, lv_sel.
ENDIF.
```

Outcome & Fix: AMDP replaced implicit outer join with inner join under certain predicates — adjusted join and NULL handling.
Tips: For CDS, correlate with HANA plan when you suspect SQL rewrite.

Case 11 — Transport/Enhancement Sequence Causing Regression (DevOps)

Context: Post-transport regression where enhancement set in one system conflicts with another.
Why hard: Cross-system and sequence dependent — only appears after specific transport order sequence.
Approach: Use script to capture enhancement implementations in callstack and identify enhancement spot executing. Compare enhancement versions via code snippets captured in log.
Pattern used: Conditional Stack Trace + Dynamic Watchlist (enhancement name).
Snippet:
(When enhancement method found, log its INCLUDE/class name)

```
lt_stack = tpda_script_stack_descr->get_callstack( ).
IF lt_stack[1]-progname CP '*Z*ENH*'.
  WRITE: / 'Enhancement executed:', lt_stack[1]-progname.
ENDIF.
```

Outcome & Fix: Found conflicting enhancement reintroducing old logic; corrected transport order and added notes to transport description.
Tips: Maintain a transport dependency checklist for enhancement classes.

Case 12 — ALV Reuse Causing Wrong Layout/Data (UI / ALV)

Context: ALV shows wrong column values after switching users. Reuse container code shares static attributes.
Why hard: UI layer reused static objects across sessions causing data bleed.
Approach: Debug script monitors ALV memory objects (`GO_*` references) and logs when layout pointer changes unexpectedly.
Pattern used: Dynamic Watchlist + Delta Pattern.
Snippet:

```
lv_alv_ref = tpda_script_data_descr->get_variable_value( 'GO_GRID' ).
IF lv_alv_ref <> gv_prev.
  WRITE: / 'ALV ref changed at', sy-uzeit, 'by', tpda_script_stack_descr-
>get_callstack( )[1]-progname.
  gv_prev = lv_alv_ref.
ENDIF.
```

Outcome & Fix: Reused global variable for ALV grid declared in a local include; moved to method scope.
Tips: For UI issues always capture `sy-uname` and UI component ids.

Case 13 — Authorization Checks Passing Unexpectedly (Security)

Context: Users see data they should not; ABAP authorization check seems bypassed.
Why hard: Complex role inheritance; custom auth logic may short-circuit checks.
Approach: Conditional trace when `AUTHORITY-CHECK` returns success but user not in expected role — capture stack and variable values used in check.
Pattern used: Conditional Stack Trace + Dynamic Watchlist.
Snippet:

```
lv_auth_result = tpda_script_data_descr->get_variable_value(
'LV_AUTH_RESULT' ).
IF lv_auth_result = 'X' AND tpda_script_data_descr->get_variable_value(
'SY-UNAME' ) = 'userX'.
  lt_stack = tpda_script_stack_descr->get_callstack( ).
  WRITE: / 'Auth passed for', sy-uname, 'by', lt_stack.
ENDIF.
```

Outcome & Fix: Custom wrapper around `AUTHORITY-CHECK` returned true when role table empty (logic bug); fixed wrapper logic.
Tips: Also capture PFCG role assignments and su01 change timestamps.

Case 14 — Data Migration Script Producing Duplicate Keys (LSMW / Custom)

Context: Mass load created duplicates for a narrow set of master data.
Why hard: Bulk scripts behave differently under parallel batch processing.
Approach: Monitor the migration routine function when key generation happens; record inputs/outputs and calling host pid.
Pattern used: Dynamic Watchlist + Conditional Stack Trace.
Snippet:

```
lv_key = tpda_script_data_descr->get_variable_value( 'LV_NEWKEY' ).
IF zseen[ lv_key ] = 'X'.
  WRITE: / 'Duplicate generated for', lv_key, 'by',
tpda_script_stack_descr->get_callstack( )[1]-progname.
ENDIF.
zseen[ lv_key ] = 'X'.
```

Outcome & Fix: Collision due to key generator using SY-UZEIT only; added sequence + enqueue; reprocessed failed batch.
Tips: For migrations, include run id and batch number in debug logging.

Case 15 — Third-Party Interface Timeout Causing Partial Writes (Integration)

Context: External partner times out leading to half-applied changes in SAP.
Why hard: Partial transaction states depend on partner behavior and network; difficult to reproduce.
Approach: Watch transaction boundary FMs and detect `COMMIT` without subsequent confirmations; log callstack and payload.
Pattern used: Event Activation + Delta Pattern.
Snippet:

```
lv_commit_flag = tpda_script_data_descr->get_variable_value( 'GV_COMMITTED'
).
IF lv_commit_flag = 'X' AND tpda_script_data_descr->get_variable_value(
'GV_CONFIRMED' ) = space.
  WRITE: / 'Partial commit at', sy-uzeit, 'stack', tpda_script_stack_descr-
>get_callstack( )[1].
ENDIF.
```

Outcome & Fix: Partner retried on timeout causing duplicate partial posts; introduced idempotent keys and compensating logic.
Tips: For integrations, log correlation ids and network caller context.

---

10. SME-Level Observations

Insight	Meaning
Debug scripts are runtime inspectors.	They observe without disturbing live code.
Combined patterns yield precision.	e.g., Conditional Stack + Auto-Termination = Intelligent tracing.
Perfect for "grey zone" bugs.	Especially in asynchronous flows (RFC, IDoc, Workflow).
Save weeks of debugging effort.	One good script reveals root cause quickly.

---

11. Appendix — Important Interfaces and Classes

Object	Purpose
`IF_TPDA_SCRIPT_TOOL`	Interface every script must implement.
`CL_TPDA_SCRIPT_DATA_DESCR`	Access runtime variables.
`CL_TPDA_SCRIPT_STACK_DESCR`	Analyze call stacks.
`CL_TPDA_SCRIPT_BREAKPOINT`	Trigger conditional breakpoints.
`CX_TPDA_SCRIPT_ERROR`	Exception class for script handling.

---

12. Closing Thoughts

ABAP Debug Script transforms the debugger from a *reactive* tool into a *proactive* engineering platform.

For an ABAP SME, mastering this is like mastering the internal DNA of the ABAP runtime. You stop "following the code" — and start *orchestrating* it.

Once this tool becomes part of your arsenal, debugging large systems — whether ECC or S/4HANA — becomes intelligent, automated, and deeply insightful.

Final Notes (Practical checklist)

- **Always** wrap script logic in `TRY...CATCH` and handle `CX_TPDA_SCRIPT_ERROR`.

- **Limit scope**: event-drive or conditional triggers reduce noise and prevent performance drag.
- **Auto-terminate** scripts after capturing required evidence.
- **Persist logs** when debugging background jobs — Script Monitor output may be ephemeral. Use a Z table or file.
- **Capture context**: sy-uname, client, RFC caller, HTTP headers, run-id, transport number where relevant.
- **Sanitize** logs before sharing (PII, credentials).
- **Repository**: maintain a shared catalog of scripts (with description, version, expected side-effects, and safe usage instructions).