Database Management System LINTER®

Version 5.9

Spdebug

Relational Expert Systems

Table of Contents

Introduction 3

Starting the Program 4

The User Interface 5

Menus and Menu Options 5
The Help Button 5
Moving Among Multiple Windows 5
Dialogue Windows 7
Message and Warning Windows 7

Viewing Object Code 8

Opening Stored Procedures 8
Opening Triggers 8
Viewing Source Code 8

Debugging 10

Choose the Debug Target 10
Creating a Debug Plan 10
Permanent Breakpoints 10
Adding Breakpoints 11
Deleting Breakpoints 12
Deactivating Breakpoints 13
Activating Breakpoints 13
Defining and Editing Breakpoint Properties 14
Execution 14
Execution on User Command 15
Execution on Event 15
Debugging Modes 16
Execution with Breakpoint Halts 16
Step-by-Step Execution 16
Execution to Temporary Breakpoint 16
Execute Until Return 16
Execution with Tracing 17
Viewing and Editing Debug Settings 18
Viewing Local Variables 18
Traced Variables and Expressions 18
Expression Evaluation 19
Viewing the Call Stack 20
Debug Log 20
Messages 21

Introduction

This document describes spdebug, Linter's interactive debugger of stored procedure and triggers. It is written for programmers who use triggers and stored procedures as a part of their application design.

Only the owner of the triggers and stored procedures can debug them. The following conditions must exist:

1) the Linter kernel must be running;
2) there must be one free channel for access to the DB;
3) **debugging targets**, **stored procedures** and **triggers** created with the debug option set must have been previously created.

Starting the Program

To launch the package, type spdebug on the command line.

As with any user, you must fill in Linter's standard login form, entering your name and password. If the server is on a remote system enter its name in the Server text box. The program will verify your access rights.

If the log-on is unsuccessful, you can cancel the log on and quit the debugger by pressing the ESC key.

After successful login, you will have the debugger's main menu on screen.

The User Interface

spdebug uses its own windowing system. A large main menu window provides the workspace within which each sub-window, resulting from selecting a menu or menu option or the output of a debugging process, is displayed.

The window displayed when spdebug is launched is the window from which all debugging functions start. Screen 1 shows the menu bar of that window.

Screen 1 – Main Debug Menu Bar

Menus and Menu Options

The ENTER key drops down a list of options from a menu selected on the menu bar. It also launches a menu option that has been selected.

The arrow keys move from menu to menu on the menu bar. They also move through the drop-down menu options. When a menu or menu option is selected, it is highlighted.

The ESC key closes a drop down menu. If no drop down menu or other window is open ESC closes the main menu and exits spdebug. You may want to use the hot-key combinations for menu options instead of having to select them from menus. Such combinations are listed to the right of the option they launch.

Menu options are grayed out when the context of the requested option makes execution impossible or inappropriate, e.g., when the active window does not contain the procedure code needed by the option.

The Help Button

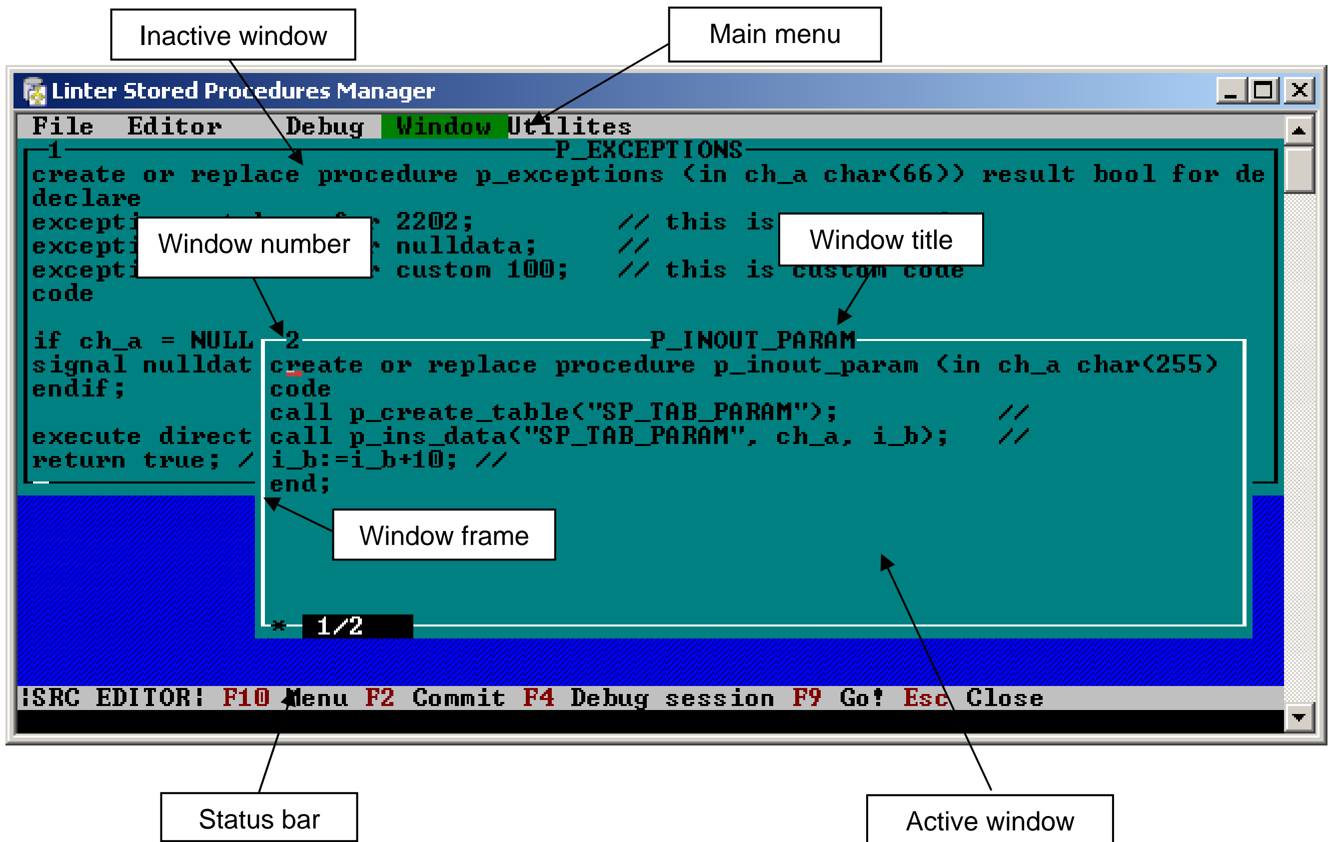
The lower portion of the screen displays the help prompt that shows the active command and lists the most common commands that are available in the present window's context.

Moving Among Multiple Windows

All information generated by selecting menu options is displayed in windows. These windows exist in the workspace between the menu bar and the help button. You can work with more than one window, e.g., one window displaying object source code, another showing debugging information.

Every window has a title, a number, and a frame. Only one window can be active at a time. When a window is active, its frame is white. When a window is maximized it overlays all other opened windows, hiding them. To change the active window, you have the following options:

Next window	F6 or Window > Next menu option.
Prior window	ALT+F6 or Window > Previous menu option.
Specific window	ALT+<window number>or select needed window from the drop-down list of open windows in the Window menu.



Screen 2 – Main Menu with Two Open Windows

To access the main menu from the active window use the F10 key. To return to the active window from the main menu use the ESC key.

The active window may be moved, resized and maximized.

Move the window:

1. Press Ctrl+F5 or choose the **Window** > **Move** menu option. The window frame will become red to indicate that it may be moved.
2. Use the arrow keys to position the window at the desired location.
3. Press the ENTER key to confirm the new window position. The window frame's initial color will be restored.

Resize the window:

1. Press CTL+F6 or choose the **Window** > **Resize** menu option. The window frame will become red to indicate that it may be resized;
2. Use the arrow keys to resize the window;
3. Press the ENTER key to confirm the new window size. The window frame's initial color will be restored.

Maximize the window by pressing F5 or choosing the **Window** > **Maximize** menu option. The maximized window will take up all of the workspace, hiding all other open windows. To restore the initial window size, repeat the previous commands which are toggles.

If there are so many opened windows that their overlapping makes them difficult to locate, the **Window** > **Cascade** menu option is a convenient way to organize them for availability. Each successive window will be positioned a little lower and to the right of the prior one so that the window titles maybe seen.

The active window may be closed by pressing ESC or choosing the **Window** > **Close** menu option.

Dialogue Windows

Dialogue windows are presented when, during the debugging process, new information needs to be entered or a choice needs to be made. Dialogue windows are modal i.e., when a dialogue window is open other windows may not be activated. Dialogue windows cannot be moved or resized. As a rule, they contain one or more input elements such as input fields, buttons, selection toggles, and lists. Screen 3 and Screen 4 show dialogue windows.

To move between input elements in the dialogue window, use TAB to move forward and SHI FT+TAB to move back. Arrow keys may be used as well if they are not being used within a specific input element:

In an input field, arrow keys are used to move the cursor.

In lists and columns of selection toggles, the arrow keys are used to select items.

ALT+<letter>, where <letter> is the highlighted button label letter, can be used to activate a button.

ENTER is used to confirm strings in input fields, choose an element in a list, change a selection toggle status, and to activate a button.

To close a dialogue window without saving the information entered press ESC. To save the entered information and close the dialogue window, press ENTER if the dialogue window has only one input element or select the relevant button with TAB and then hit ENTER.

Message and Warning Windows

Warning and message windows usually contain just text and one, or several if the user needs to make a choice, buttons. To close these windows, select the appropriate button with TAB and press ENTER. If the window contains only one button just press ENTER.

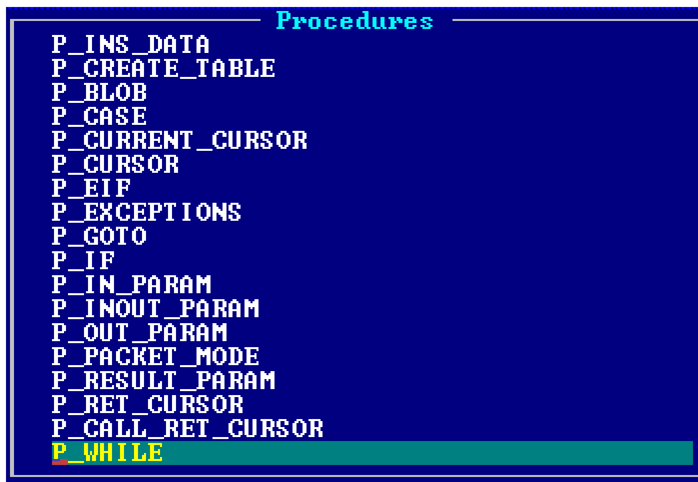
Viewing Object Code

Opening Stored Procedures

Choose the **File** > **Open procedure** menu option to open the Select Procedure Dialogue. Using the arrow keys, select a procedure and press ENTER to confirm your selection. A window containing the procedure code will be opened.

Screen 2 shows the debugger with two open debug targets: the PARAMS procedure and UPDEPERSON trigger.

To cancel the dialogue window and the selection process press ESC.



Screen 3 – Select Procedure Dialogue

Opening Triggers

Choose **File** > **Open trigger** to open the trigger selection window that is similar to the Select Procedure Dialogue window, Screen 3. Using the arrow keys, select a trigger and press ENTER to confirm your selection. A window containing the trigger code will be opened

To cancel the dialogue window and the selection process press ESC.

Viewing Source Code

After the object to be debugged has been loaded into a window, use the arrow keys to move the cursor through the source code. If the window does not contain the complete code text, you may scroll the contents of the window. Place the cursor near the edge of the window you wish to see beyond. Press the arrow key that points in the direction of the edge. You may also move through the text with the following keys:

PageUp	Move up one window height page
PageDn	Move down
CTL+ PageUp	Jump to first line of text
CTL+ PageDn	Jump to last line of text
CTL+→	Move one word to right
CTL+←	Move one word to left

Home	Move to start of line
End	Move to end of line

You may search for any word or string in the text using the search function. Press ALT+S or choose the **Edit > Find** menu option to call up the search dialogue pop-up.

Screen 4 – Search Dialogue

Enter the search string into the text field. Select the Ok button and hit ENTER. If the search is successful, the cursor will be positioned at the beginning of the text found. If unsuccessful, you will have a message that the text does not contain the string.

To find the next instance of the string press ALT+N or choose the **Edit** > **Find** next menu option.

To terminate the search, press ESC or select the Cancel button and hit ENTER.

Debugging

Choose the Debug Target

To choose the stored procedure or trigger to debug, windows need to be opened with all the necessary procedure(s) and/or trigger(s). See section “Opening Stored Procedures” and “Opening Triggers”.

Creating a Debug Plan

The debugging plan is defined by the complexity of the object and by your objectives. When creating the plan, the following considerations are significant:

1. Do you need permanent breakpoints, which are defined prior to the execution? Possibly, using temporary breakpoints, which are created on the fly during execution and depend on the output produced, would be a more usable solution.
2. If permanent breakpoints are to be used, what is the type of breakpoint to be set that will halt execution?

Regular:	Halt every time
Conditional:	Halt only break if condition is met
Var change:	Halt when a variable value changes

3. What initial state is to be assigned to breakpoints? Breakpoints may be temporarily active or inactive. Regardless of state, breakpoints are always present in the source code. However, inactive breakpoints do not influence execution of the debugging process.
4. Is source code execution or a user command to be used to launch the object generating a break? A trigger may only be executed when generated by an event in the DB for which the trigger was created. Initialization of such an event must be planned for via something like a plug in launch or execution of an SQL query. Execution of a stored procedure may be initialized by the user or in the same manner as described for a trigger.
5. The need to trace objects.
6. The need to view the call stack. Call stack information is useful when working with objects containing multiple instances of procedure calls. The call stack shows the order in which the procedures are being executed.
7. Which variables are to be displayed for analysis of object execution results? A complete list of variables or a list of selected variables may be shown. If the complete listing is too large, then, for every event a separate list of variables and/or expressions that need to be traced and displayed in the information window may be created.

Considerations 1-3 and 5-7 can be handled prior to the test run and/or interactively, during the debugging process.

Permanent Breakpoints

A breakpoint is an operator in the source code of an object that causes the debugger to halt execution of the object. During the halt, you can view and analyze the debugging information and, if necessary, edit the properties of the test run. Execution of the object resumes on your command.

Permanent breakpoint are set during a debugging session when working with an object. To open a debugging session for an active window containing object source code use F4 or choose the **Debug** > **Debugging session** menu option.

The debugging session is launched automatically when executing or waiting for a procedure or trigger.

The word Debug appears in the top part of the window frame as notice that the debugging session is running.

Permanent breakpoints exist throughout the whole session. If the debugger is re-launched, breakpoint information from the prior launch is lost and must be set again.

The debugging session can be ended by using CTRL+F4 or choosing the **Debug** > **Close debugging session** menu option. In this case, the object source code remains open for viewing and all breakpoint information is saved. However, if the debugging session is resumed, all breakpoints will be set inactive and will need to be set active if they are to be used. See section "Adding Breakpoints". The same situation occurs if the window containing the object source code is closed and then opened again.

Adding Breakpoints

Breakpoints may only be set on executable operators. When attempting to set a breakpoint outside the operator constraints, the debugger alerts the user to the violation and the breakpoint is not set.

There are two ways to set a breakpoint: standard and custom.

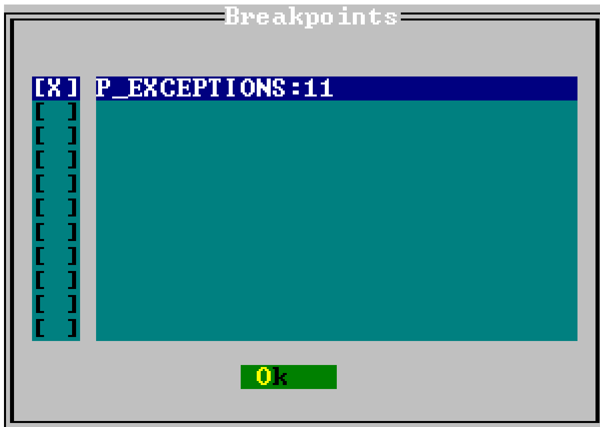
Standard

1. Activate the window containing the object's source code.
2. Move the cursor to the line in the code where you need the breakpoint.
3. Hit CTL+F8 or choose the **Debug** > **Switch breakpoint** menu option. If successful, the current line in the source code will be highlighted in red. If the current line does not contain an executable operator, the debugger will alert you to the violation. Setting the breakpoint will not produce any result if a debugging session has not been opened. See section "Permanent Breakpoints".
4. Repeat steps 1-3 for every breakpoint needed. Breakpoints set this way will be reguar breakpoints, i.e. the debugger will always halt execution when encountering them.

You must be in the currently active window to use this method of setting a breakpoint.

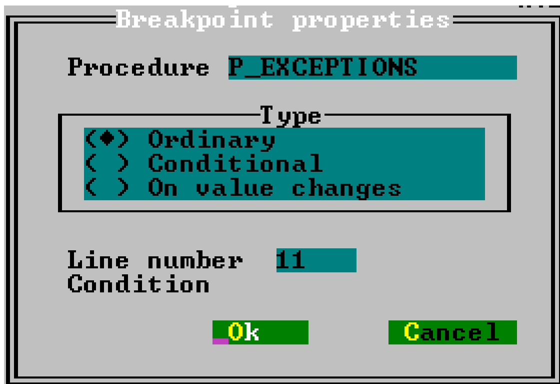
Custom

1. When in the active window containing the object source code, press ALT+F8 or choose the **Debug** > **Breakpoint** menu option to get the breakpoint dialogue window:



Screen 5 – Breakpoint Dialogue

2. Use the arrow keys to select a breakpoint, or the last empty line to add a custom breakpoint.
3. In the first column of the list, press ENTER to toggle the breakpoint status active or inactive. Alternatively, use the arrow keys to move to the right column of the list and press ENTER to open the breakpoint properties dialogue.



Screen 6 – Breakpoint Properties Dialogue

4. Select the desired object from the Procedure window box by moving to that box and pressing ENTER. Choose the object from a list similar to that shown on Screen 3. If there is no open object, the list will not be presented.
5. In the Line number field, type in the number of the line to which you wish to add a breakpoint. Line numbers are displayed in the lower left corner of the object window.
6. Select the Type, which specifies the first level of breakpoint properties, and press ENTER. The selection box to the left of the type name will be filled with a diamond. The default is Ordinary.
7. If necessary, set additional breakpoint properties. See section "Defining and Editing Breakpoint Properties".
8. Press the Ok button and a breakpoint will be set or a previously created breakpoint will be modified to correspond to the settings chosen. To cancel, use the Cancel button or hit ESC.
9. Repeat steps 2 - 8 for every breakpoint.
10. Press ESC to close the window without setting a breakpoint.

Deleting Breakpoints

All breakpoints, active or inactive, can be deleted by either of two methods.

Debug > Switch breakpoint menu option

1. Activate the window containing the object source code.
2. Move the cursor to the line where a breakpoint is to be deleted.
3. Use CTL+F8 or choose the **Debug** > **Switch breakpoint** menu option. If successful, the lines red highlighting disappears.
4. Repeat steps 1 - 3 for each breakpoint to be deleted

Debug breakpoints menu option

1. Activate the window containing the object source code.
2. Use ALT+F8 or choose the **Debug** > **Breakpoint** menu option to produce the Breakpoint Dialogue window, Screen 5.
3. Use the arrow keys to choose a breakpoint and press Delete.
4. Select the Ok button or press the ESC key to close the dialogue.

Deactivating Breakpoints

A breakpoint may be temporarily excluded from the debugging process. Even when deactivated it retains all its information and properties. At any time, a deactivated breakpoint can be again set active:

1. Activate the window containing the object source code.
2. Use ALT+F8 or choose the **Debug** > **Breakpoint** menu option to produce the Breakpoint Dialogue window, Screen 5.
3. Use the arrow keys to choose an active breakpoint and move to the first column of the line on which the breakpoint is displayed. An active breakpoint is indicated by the presence of a cross in that column.
4. Press ENTER to deactivate the breakpoint.
5. Repeat steps 1 - 3 to deactivate all necessary breakpoints.

Select the Ok button or press the ESC key to close the dialogue. Inactive breakpoints are not highlighted in the source code and are invisible when viewing the object window.

Activating Breakpoints

An inactive breakpoint may be reset active:

1. Activate the window containing the object source code.
2. Use ALT+F8 or choose the **Debug** > **Breakpoint** menu option to produce the Breakpoint Dialogue window, Screen 5.
3. Use the arrow keys to choose an inactive breakpoint and move to the first column of the line on which the breakpoint is displayed. An inactive breakpoint is indicated by the absence of a cross in that column.
4. Press ENTER to activate the breakpoint.
5. Repeat steps 1 - 3 to activate all necessary breakpoints.
6. Select the Ok button or press the ESC key to close the dialogue.

Active breakpoints regain their visibility in the object source code.

Defining and Editing Breakpoint Properties

During breakpoint creation, you may set breakpoint properties and conditions. Properties include the three types of breakpoints: Ordinary, Conditional, and Variable.

Ordinary breakpoints

Ordinary breakpoints are conditionless. The debugger will always halt at these breakpoints. The default state of the breakpoint is conditionless.

To make any breakpoint ordinary, select the Ordinary type and press ENTER. See Screen 5, Breakpoint Properties Dialogue.

Conditional breakpoints

Conditional breakpoints halt a process only if the specified conditions have been met. A conditional expression consists of object variables and/or stored procedures.

To set or edit the conditional breakpoint properties:

1. Open the Breakpoint Properties Dialogue window as described in section “Adding Breakpoints” in the text above Screen 5.
2. Select the Conditional type and press ENTER.
3. In the Condition text box, enter a logical expression, using object variables.
4. Select the Ok button.
5. To cancel press the Cancel button.

Condition expression examples:

```
1 sum = null
2 result
3 sqlcode = 0 and index[i] > 100 and date( )<>'1998'
```

Variable Breakpoints

Variable breakpoints halt debugging execution only when the value assigned to the breakpoint variable has been modified. The default value of all non-initialized variables is NULL.

To set or edit the variable breakpoint properties:

1. Open the Breakpoint Properties Dialogue window as described in section “Adding Breakpoints” in the text above Screen 5.
2. Select the Variable type and press ENTER.
3. In the Variable text box, enter one or more variables and their values if non-NULL.
4. Select the Ok button and hit ENTER.
5. To cancel press the Cancel button or use ESC.

Temporary breakpoints

Temporary breakpoints, see section “Execution to Temporary Breakpoint”, have the same properties as ordinary breakpoints.

Execution

Initializing test execution of a stored procedure can be done directly, on user command or indirectly through a call to the procedure in question from another application.

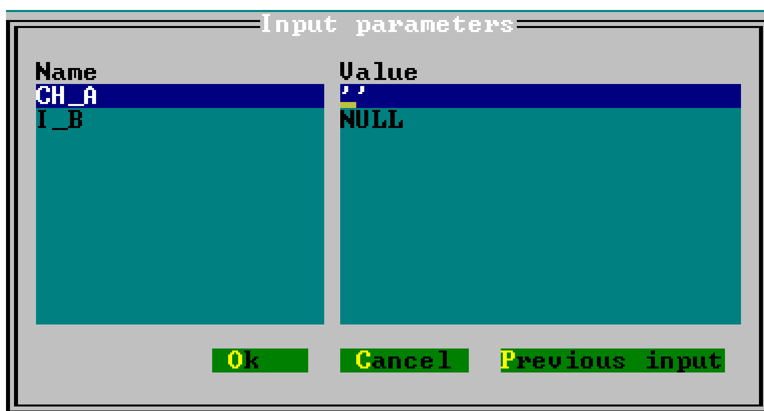
The test execution of a trigger can only be done indirectly, when an event occurs in the DB for which the trigger was created. Initialization of such an event must be done from another application.

Before test execution (for some commands, during object execution) you can specify the debugging mode (see section "Debugging Modes"). If the mode has not been specified, the debugger uses its default mode: regular execution with halts at all breakpoints that have been set.

Execution on User Command

This method of execution is available only for stored procedures. To execute via the user's command:

1. Activate the window containing the stored procedure source code.
2. Press F9 or choose the **Debug** > **Go** menu option. On this command, the procedure is loaded and variables, if any, are passed to it. If the procedure has incoming parameters, enter them in the Incoming Procedure Dialogue window. Procedure execution begins after the execution mode has been set. Breakpoints may be set before as well as after the Debug > Go command.



Screen 7 – Incoming Procedure Dialogue

Execution on Event

Execution-on-event can be applied to a single object or to a group o, not necessarily connected, objects.

For triggers, an event is a real event in the DB for which the trigger was created regardless of the event creation method.

For stored procedures, an event is a call to the procedure, including those from the debugger.

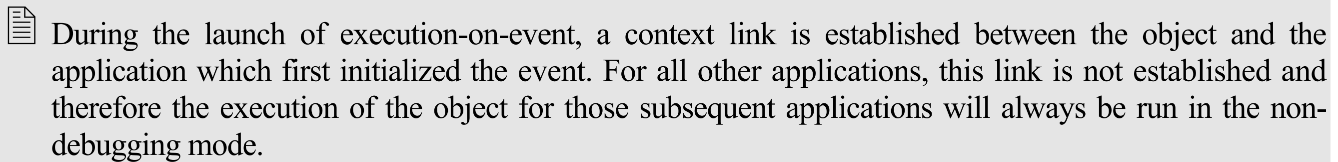
To execute a single object:

1. Activate the window containing the object source code.
2. Press CTL+F9 or chose the **Debug** > **Wait** for procedure/trigger menu option.

To execute a group of objects:

1. Open windows containing the source code of the objects to be executed.
2. Start a debugging sessions for each one by pressing F4 or choosing the **Debug** > **Debugging session** menu option.
3. Press CTL+F9 or chose the **Debug** > **Catch** any open menu option.

After execution launch, the objects enter an idle stage awaiting the events. During this idle period or during the pauses between execution cycles, the user may set or change the debugging modes.


During the launch of execution-on-event, a context link is established between the object and the application which first initialized the event. For all other applications, this link is not established and therefore the execution of the object for those subsequent applications will always be run in the non-debugging mode.

Debugging Modes

Execution with Breakpoint Halts

To execute an object with halts at the breakpoints:

1. Set the launch mode for the object. See section “Execution”.
2. If execution-on event-mode has been set, wait for the event or initialize the event from another application.
3. If execution-on-user-command mode has been set, launch the procedure as described in section “Execution on User Command”.
4. If all breakpoints have been deactivated or none have been set, execution of the object will not differ in any way from regular execution. Otherwise, on reaching a breakpoint and meeting any breakpoint conditions, execution will be halted.
5. To continue the test, press F9.

Step-by-Step Execution

In step-by-step execution mode, execution is halted after every evaluated operator in the object source code. It is the same as having conditionless breakpoints set on every executable operator.

To execute the step-by-step mode:

1. Set the launch mode for the object. See section “Execution”.
2. Press F8 or choose the **Debug** > **Step** menu option. During step-by-step execution, the current step is highlighted in blueю
3. Repeat step 2 to execute the next step.

Execution to Temporary Breakpoint

A temporary breakpoint is the line of the object source code containing the cursor. Since the cursor can only be at one position at any one time, there can be only one temporary breakpoint.

To launch temporary breakpoint debugging:

1. Place the cursor on the source code line where the breakpoint needs to be located.
2. Press CTL+F10 or choose the **Debug** > **Run** to cursor to cursor menu option.
3. Repeat steps 1 and 2 to execute the same or another object with a temporary breakpoint.

Execute Until Return

Execute until return is used with nested stored procedures. First, the breakpoints located in the nested procedure are evaluated. Then, the debugger halts execution inside the parent procedure after having encountering a return statement in the nested procedure.

If this mode is used when there are no embedded calls, execution continues uninterrupted until the entire procedure has been evaluated.

To execute an object until return:

1. Choose a testing plan such that the debugger will perform a halt inside a nested procedure. This can be done in either of two in ways:
   - create a breakpoint inside the procedure in question and use the breakpoint halt mode;
   - use the step-by-step mode with tracing.
2. When inside the procedure body, press SHIFT+f10, ALT+3, or, choose the **Debug** > **Run** until return menu option.

Example

Assume you have a procedure, mainproc( ), and two nested procedures, proc1 ( ) and proc2( ).

```
Procedure mainproc( ):
Operator        Operator
number
10              …
11              Call proc1( );
12              …
return;

Nested procedure proc1( ):
10              …
11              Call proc1( );
12              …
return;

Nested procedure proc2( ):
10              …
11              …
12              …
return;
```

If the execution is started on the 10th operator in mainproc, the procedure will be executed without halts until the end.

If the execution is started on the 10th operator in proc1, the halt will be performed on the 12th operator in mainproc.

If the execution is started on the 10th operator in proc2, then the halt will be performed on the 12th operator in proc1.

Execution with Tracing

During the debugging process, all nested procedures are regarded as a single executable operator, i.e., the debugger does not move through the procedure body.

If you need to enter the procedure body, there are two ways:

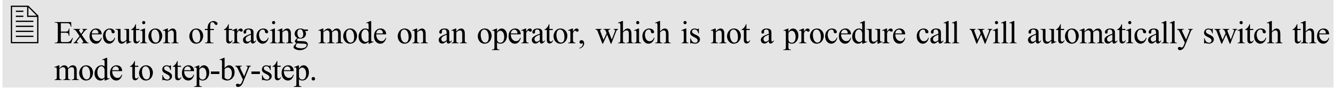
1) set a breakpoint inside the procedure body;
2) using execution with tracing.

If tracing mode is used, then, when a nested procedure call is encountered during execution the debugger moves into the nested procedure's window and performs a halt on the first operator being executed.

To execute with tracing:

1. Perform a halt on a nested procedure operator using any method - step-by-step, setting a breakpoint, etc.

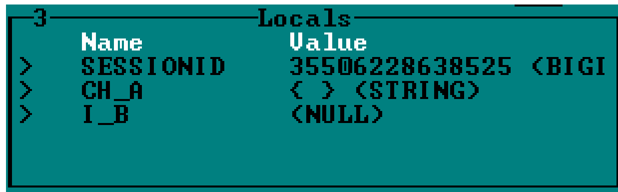
2. Press F7 or choose the **Debug** > **Trace** menu option. The window containing the procedure source code will be displayed and the cursor will be placed on the first executable operator.
3. Repeat steps 1 and 2 for all nested procedures.


Execution of tracing mode on an operator, which is not a procedure call will automatically switch the mode to step-by-step.

Viewing and Editing Debug Settings

Viewing Local Variables

During procedure debugging execution, the local variable window is automatically displayed



Screen 8 – Local Variables

If the local variable window is active, the user may scroll the window contents using the arrow keys.

Traced Variables and Expressions

Often, during the debugging process, the complete variable list is not needed. The variable list can become so large it slows execution.

If you only need a relatively few key variables and/or arithmetical expressions composed of local variables, you may create a custom list of variables and expressions. The values of these variables and expressions will then be traced at every breakpoint.

The Monitor window is used for working with such lists. To open it, choose the **Debug** > **Open monitor** menu option or press CTL+F7 particularly if the window containing the object source code is active.

Screen 9 – Monitor Window

Adding a variable or expression to be traced:

1. Activate the Monitor window.
2. Using the arrow keys, place the cursor on the last line.
3. Type an expression and move to another line or press ENTER. The second column will then display the expression value or information about an error.
4. Repeat steps 2 and 3 for any other variable or expression to be traced.

The variable/expression list is only kept for the duration of the current session. When re-launching the debugger, the list needs to be created again.

It is not necessary to add all of the variables/expressions at once. The list may be modified or edited at any time during the debuggin session.

Removing traced variables/expressions:

1. Activate the Monitor window.
2. Using the arrow keys, highlight a variable or expression.
3. Press ALT+D.
4. Repeat steps 2 and 3 for all variables and expressions to be deleted.

Expression Evaluation

A need for calculations often arises during the debugging process. For example, you may need to calculate values based on a comparison of debugging output values to values received during a test run.

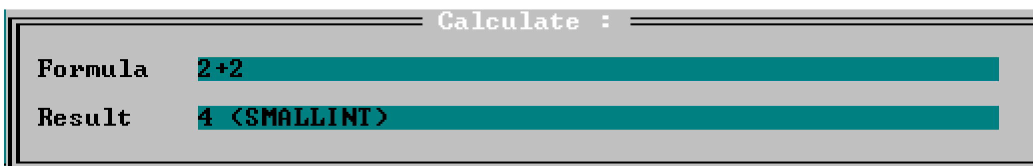
For these types of situations, the debugger provides an expression evaluation instrument. Any logically correct expression that adheres to the Linter procedural language standards maybe used as an argument. The expression may also contain the object's local variables however, in this case the expression can only be evaluated during execution of the object.

If the expression does not contain object variables, only constants, then the instrument may be considered a simplified calculator and may be used at any time, regardless of the presence of active debugging targets.

Constant expression evaluation

To evaluate a constant expression:

1. At any time, regardless whether debugging targets are present or not, choose the **Debug** > **Evaluate expression** menu option or press SHIFT+F9. The expression window will be displayed.
2. In the Formula text box, enter the desired expression and press ENTER.
3. The result will be displayed in the Result box.
4. Repeat steps 1 and 2 to evaluate other expressions.
5. When finished, press ESC to close the window.



Screen 10 – Expression Evaluation

Variable expression evaluation

To evaluate an expression containing variables:

1. Activate the object containing the variables to be used in the expression.
2. Execute the object and perform a halt in the source code area where an expression needs to be evaluated.
3. Choose the **Debug** > **Evaluate expression** menu option or press SHIFT+F9. The Expression Evaluation window will be displayed.
4. Enter the desired expression in the Formula text box and press ENTER.
5. The result will be displayed in the Result box.

6. Repeat step 4 to evaluate all needed expressions.
7. Press ESC to close the Expression Evaluation window.

Viewing the Call Stack

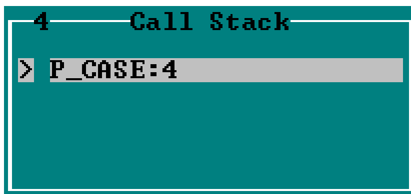
The call stack shows the current nesting level of the procedure being executed. It can be used effectively to trace a nested procedure that produces an exception during a testing run. If the stack is being monitored when an exception e.g., divide by zero occurs the debugger performs the following:

- Shows the name of the procedure that has produced the exception.
- Automatically opens a window containing the procedure source code with the cursor positioned on the line holding the operator that produced the exception.

If, during the debugging process, the trace command has been activated the call stack will also display information about the origin of the call.

To view the call stack:

1. Open the stack window by choosing the **Debug** > **Call stack** menu option or by pressing CTL+F3, particularly when the window containing object source code is active.
2. Move to the stack window.
3. Choose the desired line and press ENTER to view the results of the procedure.



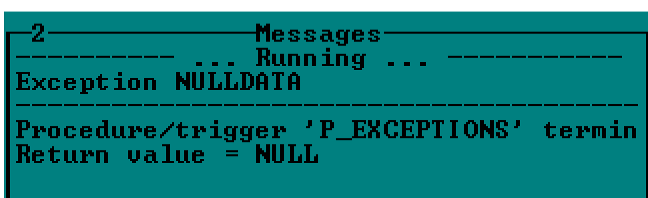
Screen 11 – Call Stack

Debug Log

During every test run, the debugger creates a log that contains the following information:

- The current state of the object: executing, idle, awaiting start, etc.
- All exceptions. The exception description includes information on the procedure, creating the exception, the line number, and the type of exception.
- All abnormal Linter termination codes.
- Output variable values, if there are any, or the trigger operation status, active or inactive, at termination.

The Message window automatically displays the debug log when debugging a stored procedure or trigger.



Screen 12 – Message

Messages

Message Text	Cause	Recommendations
Procedure is being debugged in a different session.	The currently selected object is in use by another user.	Coordinate the debugging sessions with the other users.
Debugging session not open.	An attempt to set a breakpoint has been made when a debugging session has not been opened.	Open the correct debugging target and/or open a debugging session.
Too many procedures.	The debugger can support a maximum of 273 objects simultaneously.	Do not open more than 273 objects at any one time.
Not enough memory for the operation.	Not enough free hard drive space. Internal debugger error.	Free more hard drive space. Contact tech-support.
Unknown error.	Internal debugger error.	Contact tech support.
Incorrect operator line number.	When setting a breakpoint, a line number has been specified which contains no operator or which already has a breakpoint.	Specify a correct line number.
Too many breakpoints.	Too many breakpoints have been set. The maximum number of regular breakpoints is 1000. The other types of breakpoints depend on the length of expressions or variable names.	Use the allowed number, or reduce the number, of breakpoints.
Procedure intended for debugging not launched.	During a debugging session, the object has been deleted by another user.	End the debugging session. If necessary, restore the object in the database and repeat debugging.