An Integrated Approach to Troubleshooting Your ABAP Programs: Expert Tips for Making the Most of the ABAP Debugger

Boris Gebhardt

Boris Gebhardt joined SAP AG in 1998, where he currently works in the ABAP QM group. Boris is responsible for customer support and SAP internal consulting for the ABAP programming language and its surrounding tools. He is also involved in the development of ABAP tools, including the New ABAP Debugger, and was engaged in a development project for the public sector.

(complete bio appears on page 110)

The first installment of this troubleshooting series discussed a variety of tools and techniques you can use during application development and testing to ensure that you are building high-quality ABAP applications.[1] The follow-up to that article then provided you with recommended strategies for troubleshooting your applications that have reached production, and despite your best efforts are experiencing problems.[2] It examined a set of valuable tools, including the system log, ABAP Dump Analysis, and the ABAP trace features of ABAP Runtime Analysis. In combination, these tools can either solve the problem itself or at least deliver a very detailed picture of the situation.

As you may recall from the discussion in the second article, in many cases the ABAP Debugger is not the most appropriate tool to begin your troubleshooting session. But once you have a good understanding of the situation and find that all you need to solve the problem is the value of one specific variable, for example, the debugger is really the only tool that can deliver the solution. Or perhaps you need to search the source code of a complex application to find where an internal table is refreshed, which you have determined is the root cause of a problem. Here, too, only the debugger can provide the help you need.

This article extends the ABAP troubleshooting series to show you how to use the ABAP Debugger appropriately and effectively, so that when you must resort to using it, you will be able to do so with the

[1] See the article "An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP 'Check' Tools During Development and Testing" in the March/April 2004 issue of *SAP Professional Journal*.

[2] See the article "An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP Investigative Tools for Production Problems" in the May/June 2004 issue of *SAP Professional Journal.*

knowledge and confidence of an expert. Instead of focusing on straightforward features of the debugger, such as displaying variables or stepping through source code, the details of which are readily available in the online help, this article walks you through a series of real-world debugging scenarios and offers little-known tips that will save you time and effort.

Let's start by reviewing the different ways you can access the debugger.

Starting a Program or Transaction in Debug Mode

Sometimes you need to start an application in debug mode from the beginning. For example, suppose you want to start analyzing the program flow as soon as the application starts running or monitor changes in a particular variable throughout a program run. Imagine a troubleshooting scenario in which a field on the first screen of an application contains an incorrect value. In order to analyze and solve this problem, you need to determine where and how this field is populated with data. If you begin debugging (by entering */h* in the SAP R/3 command field) after the first application screen appears, it's already too late. The field has already been populated. Instead, you need to start the program or transaction in debug mode.

Starting reports or other executable programs in debug mode is straightforward. Run the ABAP Editor (transaction *SE38*) and start the debugger for the program by clicking on the *Debug* button in the ABAP Editor toolbar.

Starting a transaction in debug mode is a little less clear-cut. Our first troubleshooting scenario (summarized in the sidebar at the upper right) illustrates this process using transaction *SE09*, the Transport Organizer.

Identifying the Source Variable Before Starting the Transaction in Debug Mode

Figure 1 shows the initial screen of the Transport Organizer, started with transaction *SE09*. Notice that the system prepopulated the *User* field with my user name (*GEBHARDT)*. Our task is to determine where in transaction *SE09* this prepopulation occurs.

Scenario 1: Analyzing the Source of a Prepopulated Field

In our first scenario, we investigate how a particular field is populated — in this case, the User field in the Transport Organizer (see Figure 1 on the next page). The troubleshooting and debugging techniques we'll use for this scenario are useful whenever you need to investigate the reason behind an unexpected field value on a screen. For example, perhaps the auto-populated name for a particular user is incorrect and you need to determine its source.

Our first challenge is to identify the relevant variable and program so we can identify the appropriate point to start debugging the transaction. Then we can use the ABAP Debugger to determine where in the program the variable is changed.

What is our troubleshooting strategy? You might think an ABAP trace would be a good place to start. An ABAP trace would show the application flow logic, but it would not provide the contents of different variables. Because we want to observe when a specific variable is filled, only the ABAP Debugger can help us here. Before starting to debug, however, we must answer two questions:

- Which variable in which program stores the user name?
- When should we start debugging in order to watch this variable being filled?

To answer the first question, move the cursor to the *User* field and press the *F1 (Help)* key to display the Performance Assistant pop-up shown in **Figure 2**. The Performance Assistant provides online help for

Figure 1 *Initial Screen of the Transport Organizer*

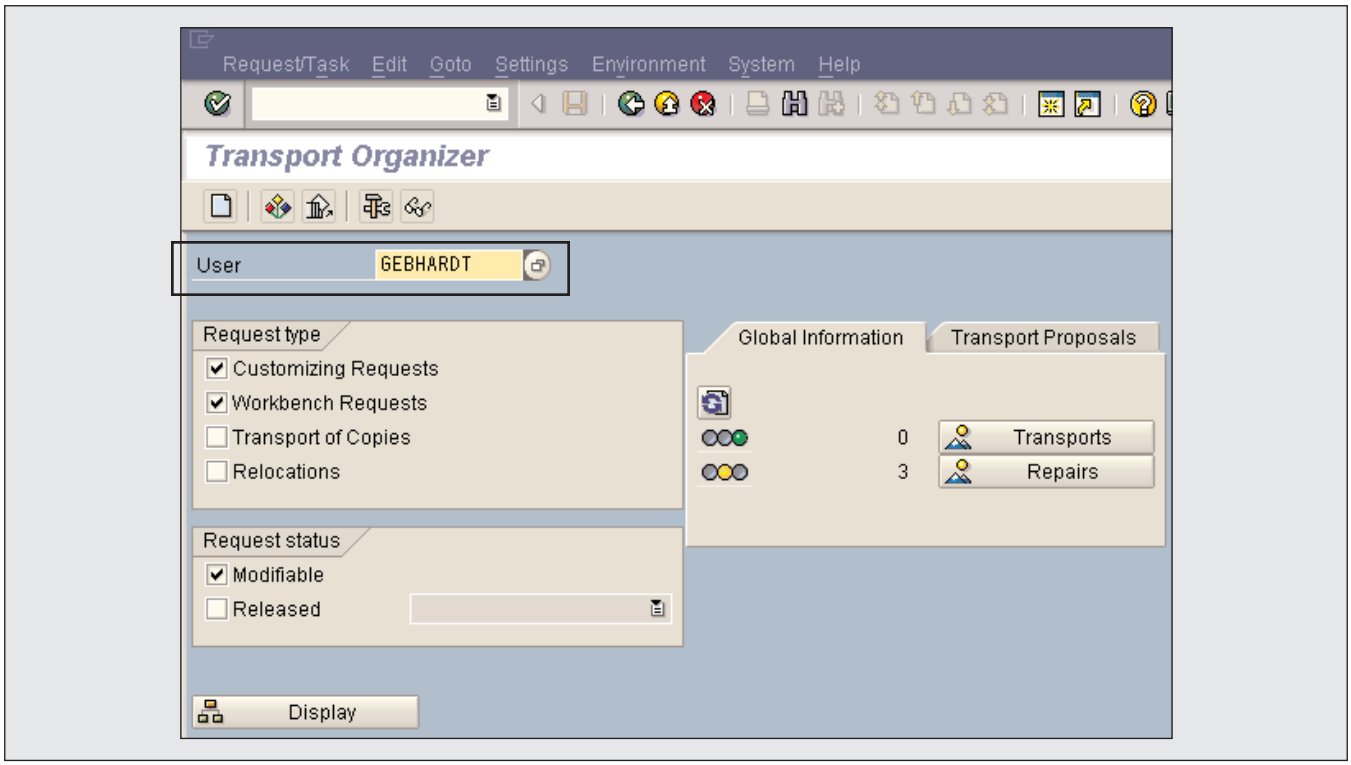


Figure 2 *The Performance Assistant*

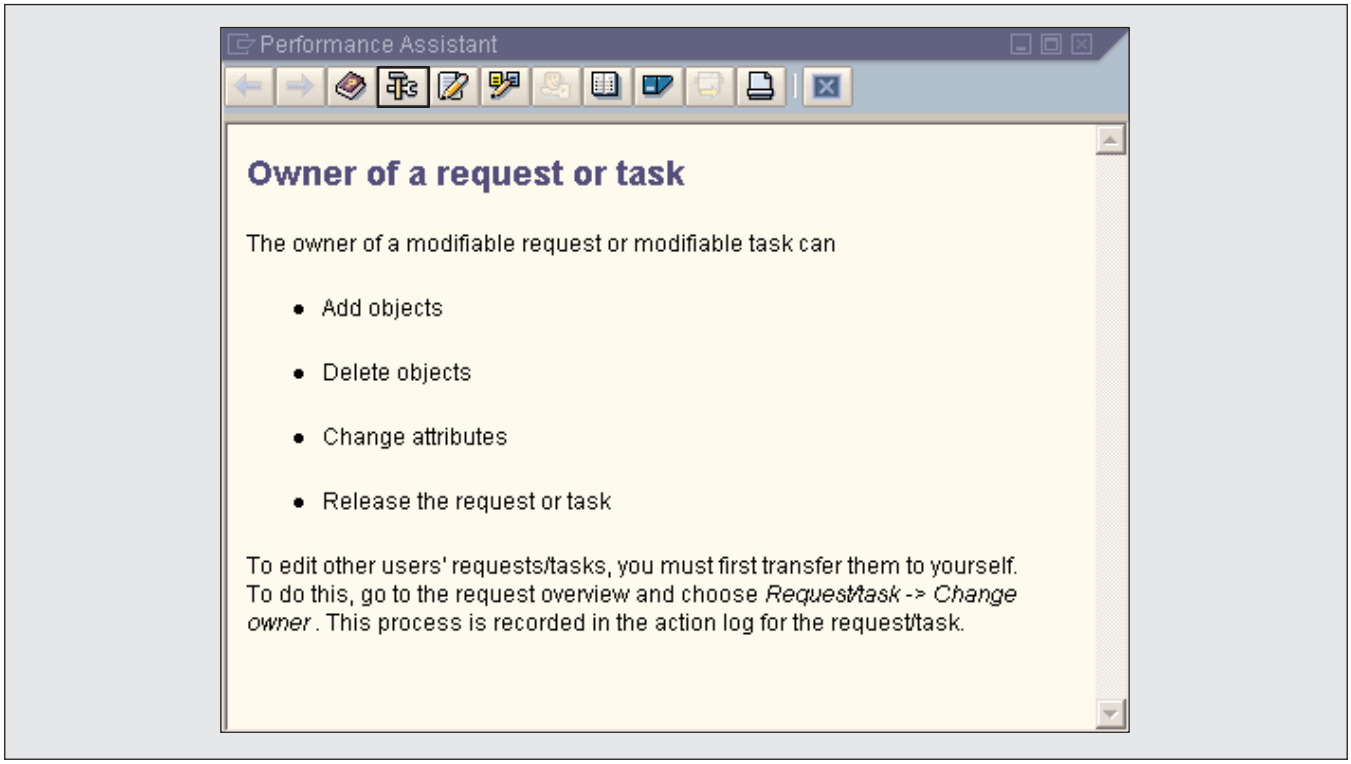
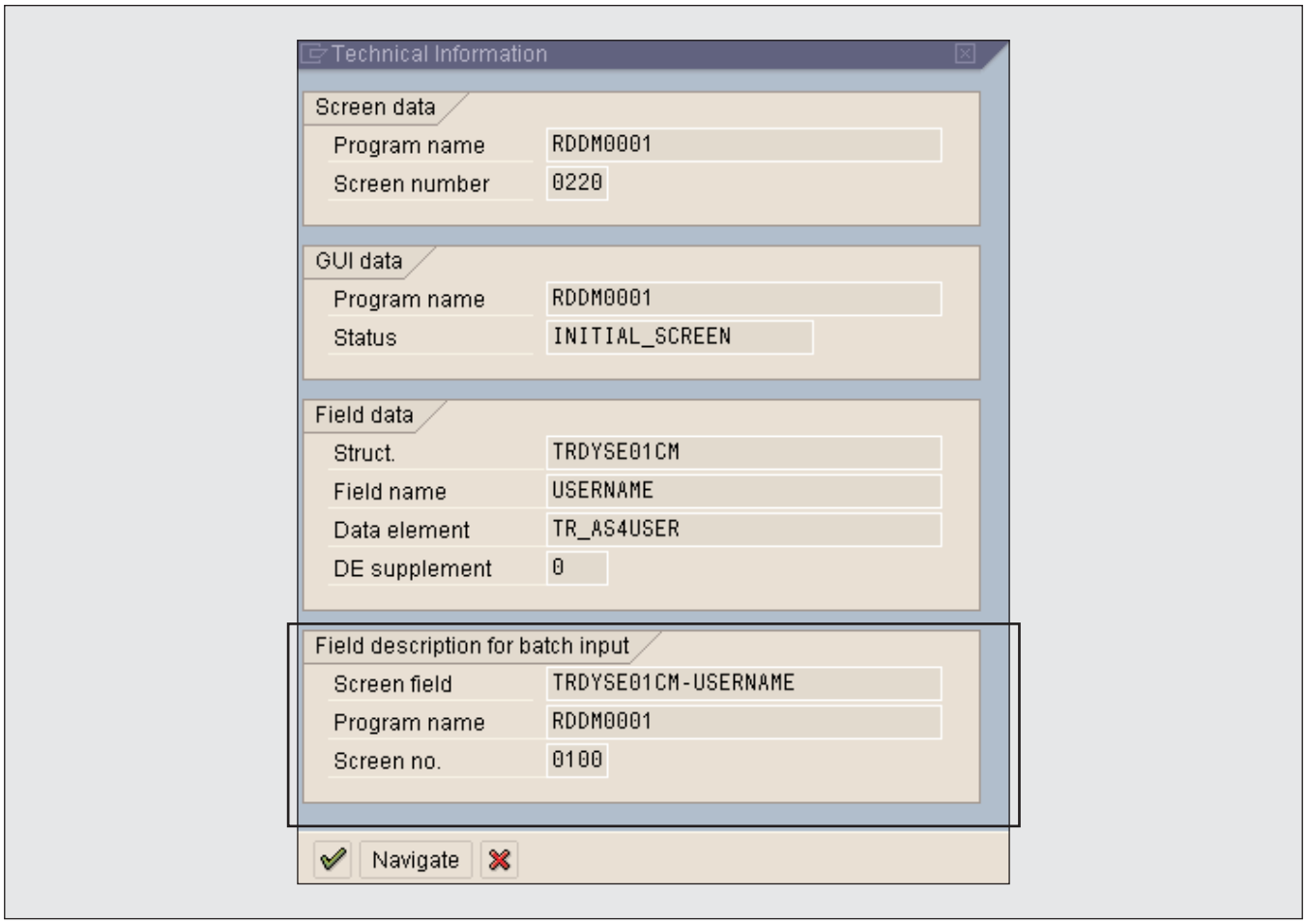


Figure 3 *Technical Information for a Field*



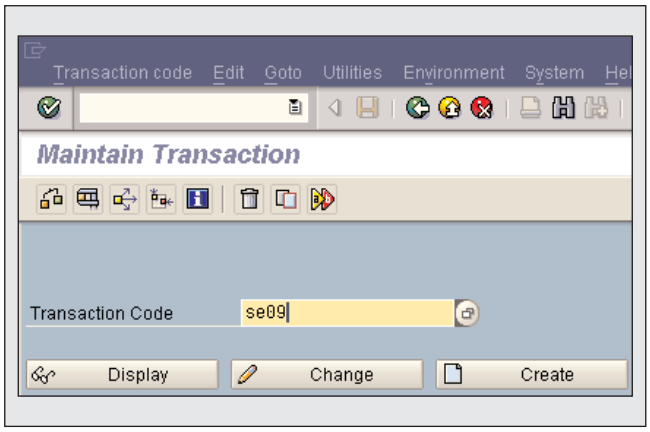
determining the purpose of the current field. To obtain technical details about the *User* field, click on the technical information button in the toolbar ( ). The Technical Information pop-up (see **Figure 3**) provides the answers we need in the *Field description for batch input* frame. The variable that stores my user name is *TRDYSE01CM-USERNAME* in the *RDDM0001* program. This frame also tells you the main screen on which the field is located — in this case, main screen number *0100*, but keep in mind that the field is actually on screen 0220 (see the *Screen data* frame in Figure 3), which is a subscreen included in main screen 0100.

Now answering the second question (when to start debugging) is easy. We need to start debugging before the *TRDYSE01CM-USERNAME* variable is filled. Starting the debugger after the first screen appears is clearly too late, because the field already displays the content *GEBHARDT*. Therefore, we need to start the transaction in debug mode from the beginning, which means we need to start it at the first ABAP statement of transaction *SE09*.

But how do we identify the first ABAP statement of the transaction, and how do we start debugging it from the beginning? We could use the ABAP Editor to analyze the structure of transaction *SE09*, or we might decide to use an ABAP trace, as described in the previous troubleshooting article,[3] to analyze the transaction's program flow. We could then use the structure or program flow details to identify the first

[3] "An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP Investigative Tools for Production Problems" in the May/June 2004 issue of *SAP Professional Journal*.

Figure 4 Initial Screen of Maintain Transaction



screen of transaction *SE09*, the first module of that screen, and so on. Eventually, we would find the first statement of the transaction and be able to set a breakpoint there.

While analyzing the structure or program flow will work, the following trick is much more effective. First, run transaction *SE93* (Maintain Transaction). On the initial screen, enter the code of the transaction you want to debug (in this case, *SE09*) in the *Transaction Code* field, as shown in **Figure 4**.

Select *Transaction code* → *Test* → *Debugging* from the Maintain Transaction menu. The transaction starts in the debugger (as shown in **Figure 5**), without the need for you to find the first statement of the transaction.

You may have noticed that the pointer (➡) to the line that is currently executing is missing in Figure 5. In this state of screen processing, you are actually in a system module of the first screen (screen 100 of program *RDDM0001*), which is not visible in the displayed screen flow. Press the *F5* (*Step Into*)

Figure 5 Transaction Started in Debug Mode

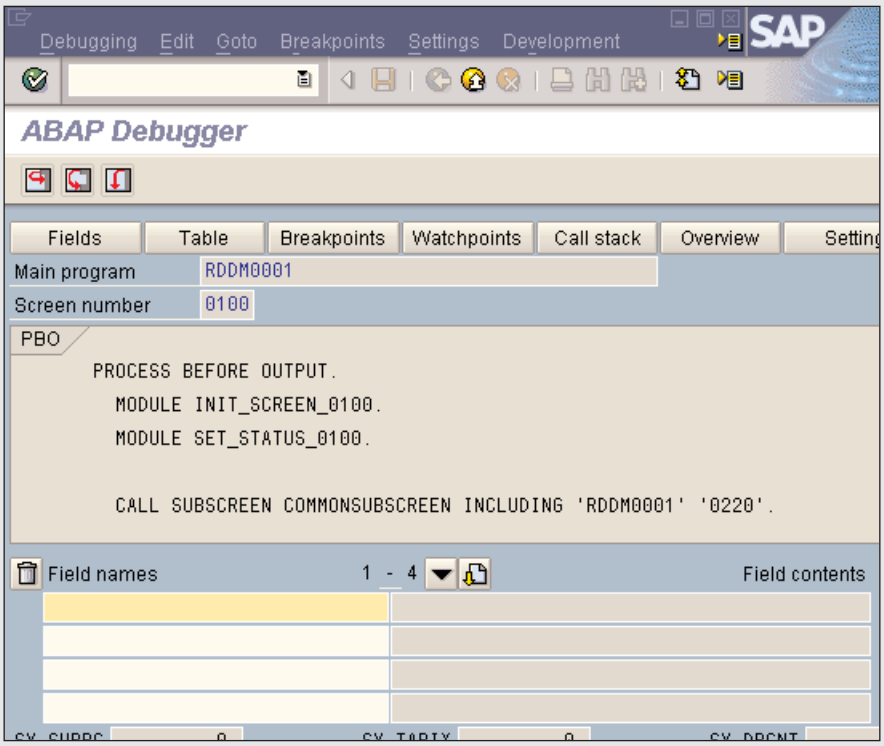
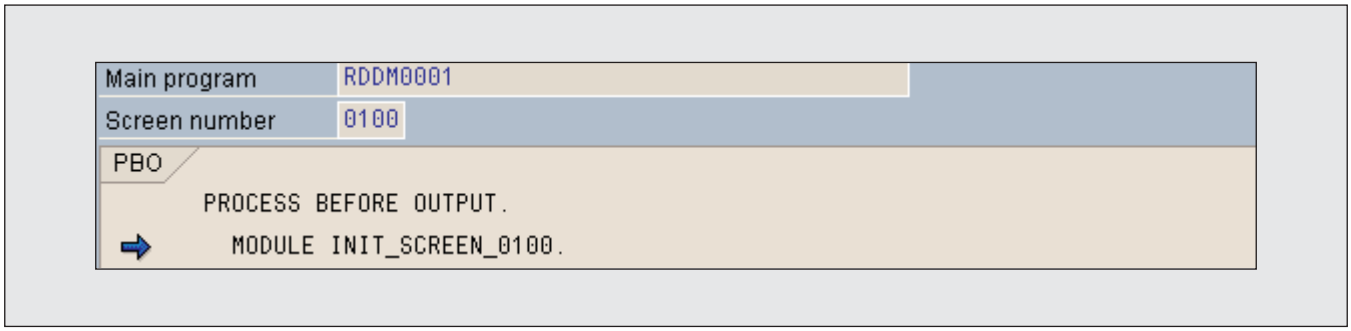


Figure 6 Advancing the Debugger to the First Module



key to step once in the debugger, and the line pointer will appear directly before the first module, *INIT_SCREEN_0100*, as shown in **Figure 6**.

We will continue this scenario later in order to find out when the *TRDYSE01CM-USERNAME* variable in program *RDDM0001* is actually filled. First, I'll show you how to start the debugger at a certain point in the transaction instead of from the very beginning, as well as how to start the debugger for background jobs. Finally, I'll show you how to use breakpoints and watchpoints, which is how we will determine when the example variable is filled.

Starting the Debugger in a Dialog Transaction

You now know how to start the debugger from scratch for a report (that is, an executable program) or a transaction. However, in most cases, you will probably want to start debugging at a specific point in a transaction rather than from the start. For example, suppose you want to find out why a particular message appears after screen 21. You could perform an ABAP trace. Or, you could start debugging directly after this screen by entering */h* in the command field on the main SAP R/3 screen. Upon the next user action (such as pressing the *Enter* key), you enter the debugger.

Sometimes you can't directly access the SAP R/3 command field, though, so you can't use it to turn on the debugger. For example, when you are on a pop-up window or message screen, you can't access the command field of the underlying main screen. Suppose you want to start the debugger directly after the Program Attributes dialog box shown in **Figure 7**, which shows the attributes of the *ZTSTECHED02_SE30_JOB* program in the ABAP Editor. You can't access the SAP R/3 command field, because the underlying main screen is inactive at that time and unavailable for user interaction, so you can't enter the */h* command to start the debugger.

In a situation like this, you can use a graphical user interface (GUI) shortcut to start the debugger. Using the SAPshortcut program,[4] you simply create a GUI shortcut on your desktop that stores the appropriate command, and then instead of entering the

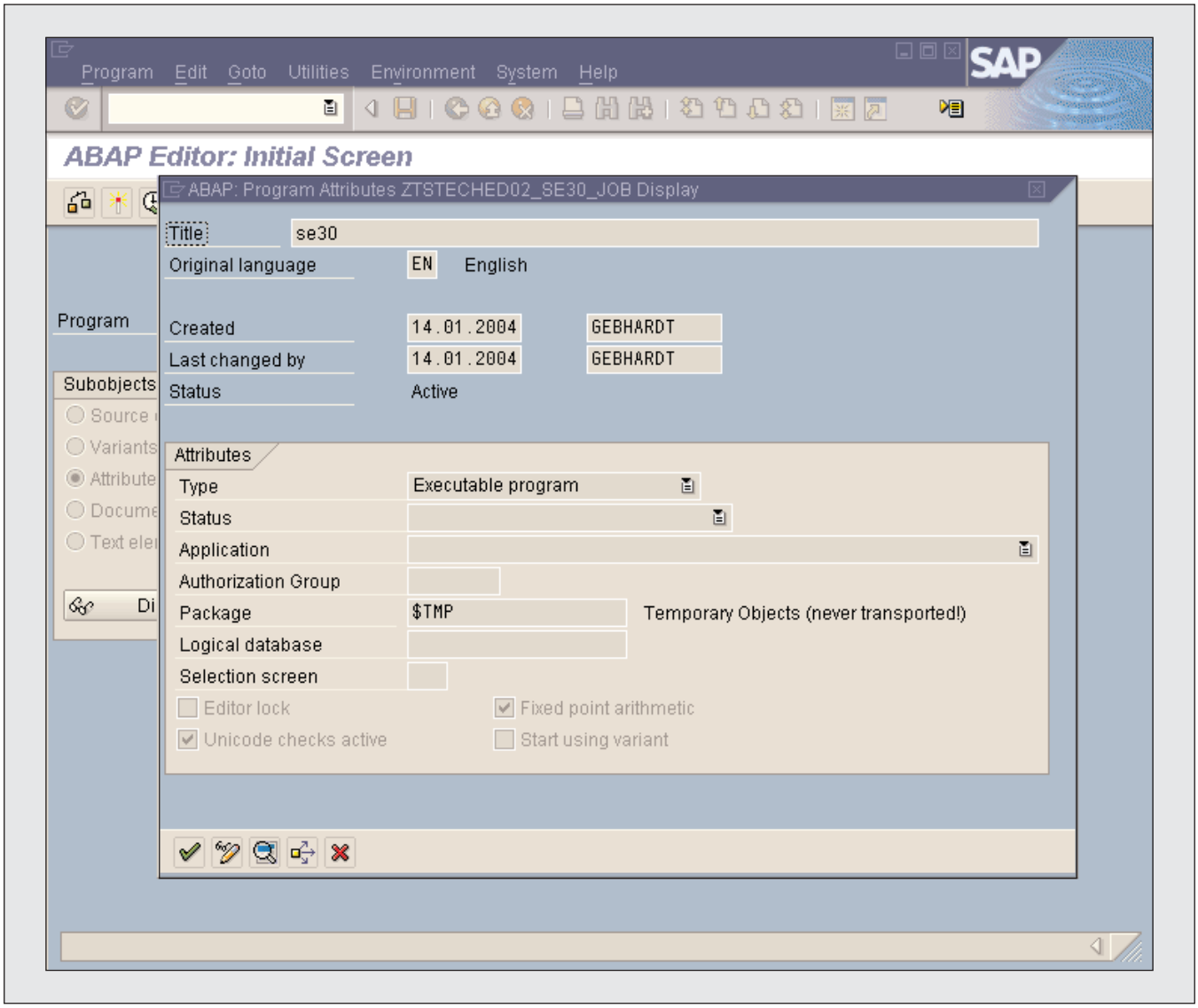
✓ *Alternate Ways to Start the Debugger*

In addition to using the /h command, you can also start the debugger by using the following commands:

- */hs — This command starts system debugging (for details on system debugging, see the download available at **www.SAPpro.com**)*
- */ha — This command skips screen processing and steps directly to the first ABAP statement*

[4] Refer to SAP Note 99054 for more information about the SAPshortcut program. It is available in SAP Basis 4.5A or higher, and you must be running SAPGUI 4.5A or higher.

Figure 7 ABAP Editor Program Attributes



✓ *Using GUI Shortcuts Across Your System Landscape*

GUI shortcuts can be used across the SAP R/3 systems in your system landscape, regardless of the particular SAP R/3 system you are logged on to, when you drag and drop them onto active SAP R/3 screens (only the shortcut command is executed). If you double-click on a shortcut, however, this will lead to a logon to the specified system, and afterward the shortcut command will be executed on this system.

command directly in the command field, you drag and drop the shortcut onto the active screen. The system executes the command as if you entered it directly in the command field.

Figure 8 *Creating a GUI Shortcut*

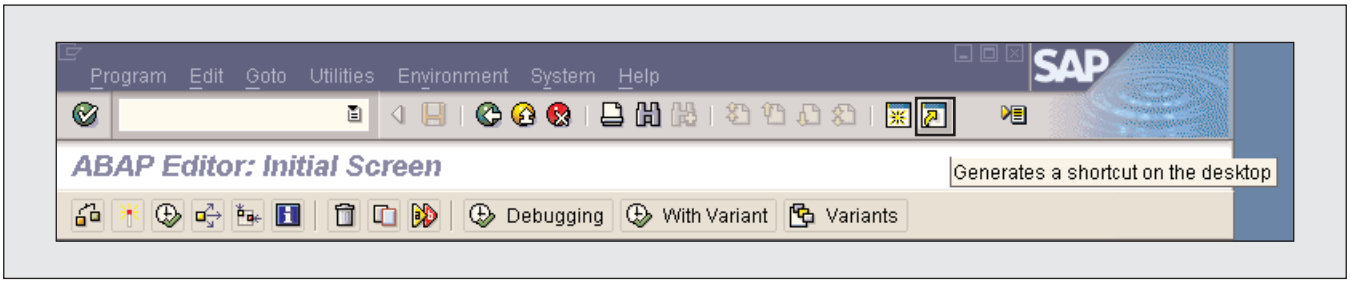
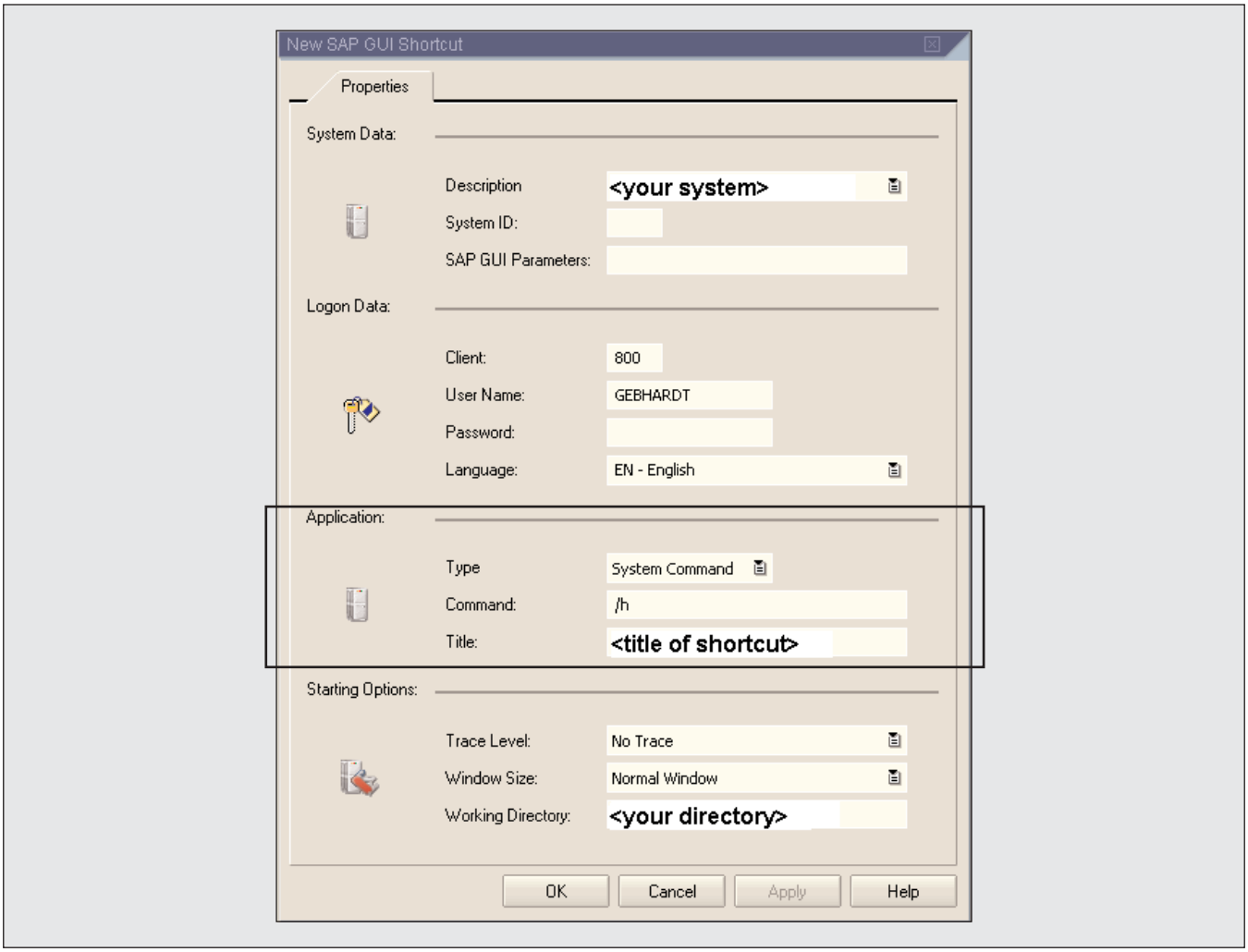


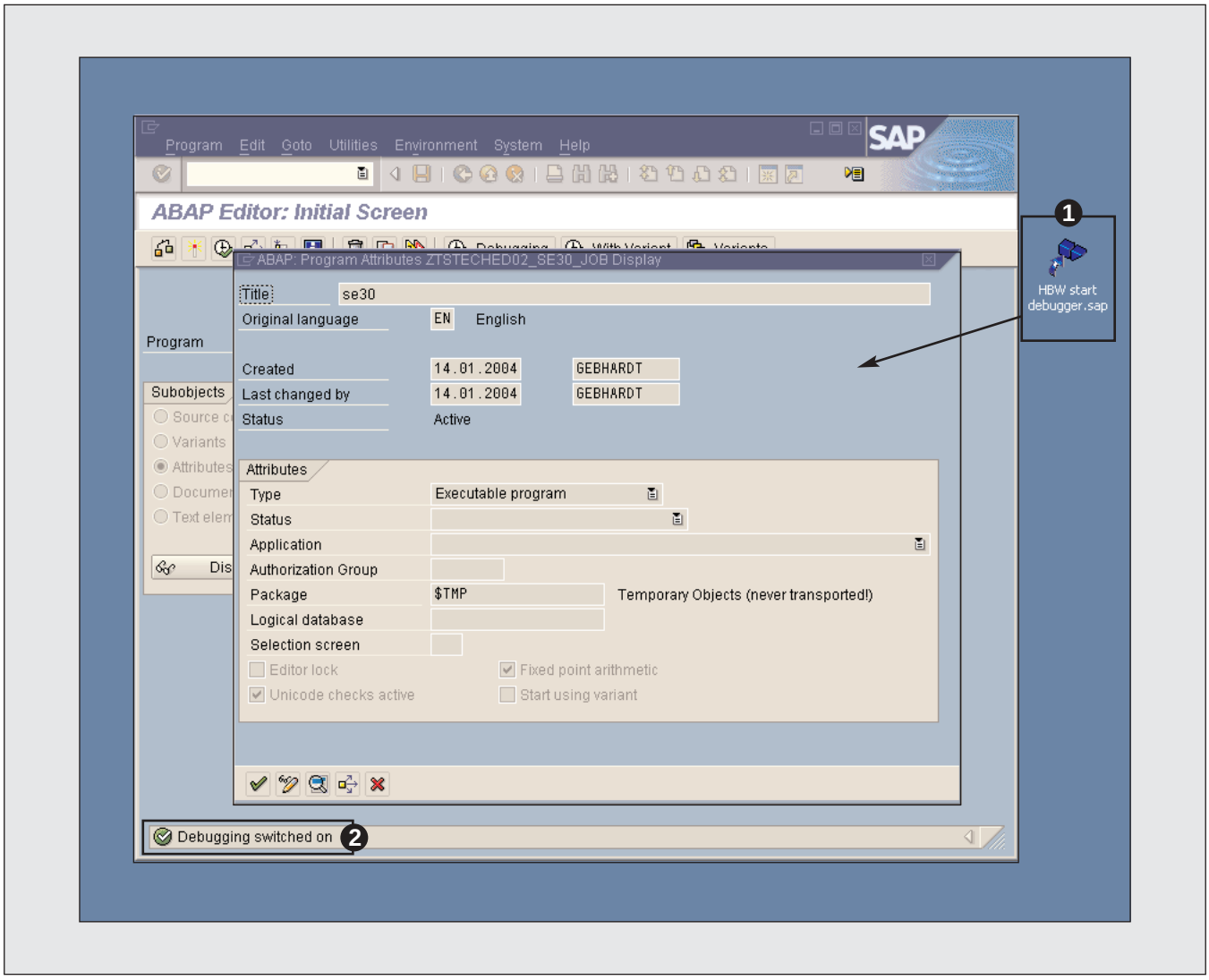
Figure 9 *Specifying the GUI Shortcut Details*



To create a shortcut, click on the shortcut button ( ) in the standard toolbar available on any SAP R/3 screen (see **Figure 8**).

Specify the command details in the New SAP GUI Shortcut dialog box shown in **Figure 9**. Only the *Application* frame is relevant for our debugger shortcut; the system data is only relevant if you want to execute the shortcut on a specified system by double-clicking on it, instead of dragging and

Figure 10 Turning on the Debugger via the GUI Shortcut



dropping it on an active SAP R/3 screen (see the note on page 83). Enter the type of command (system command, transaction, or report), the command itself, and a title for the shortcut. To create a shortcut to turn on the debugger, select *System command* as the application type and enter */h* (or */hs* to start system debugging) as the command.

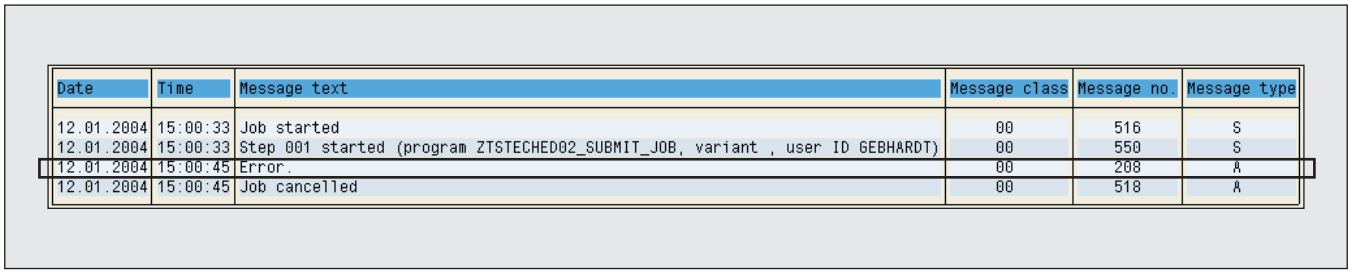
After clicking on the *OK* button, a shortcut with the specified name appears on your desktop (see the icon at the upper right of **Figure 10**).

When the Program Attributes dialog box appears, drag and drop the debugger shortcut onto the dialog window (see ❶ in Figure 10), which switches on debugging (see ❷ in Figure 10). The GUI shortcut command */h* is executed on the active SAP R/3 screen as if you entered it in the command field. The next user action (such as pressing *Enter* to close the dialog window) starts the debugger.

Starting the Debugger for a Background Job

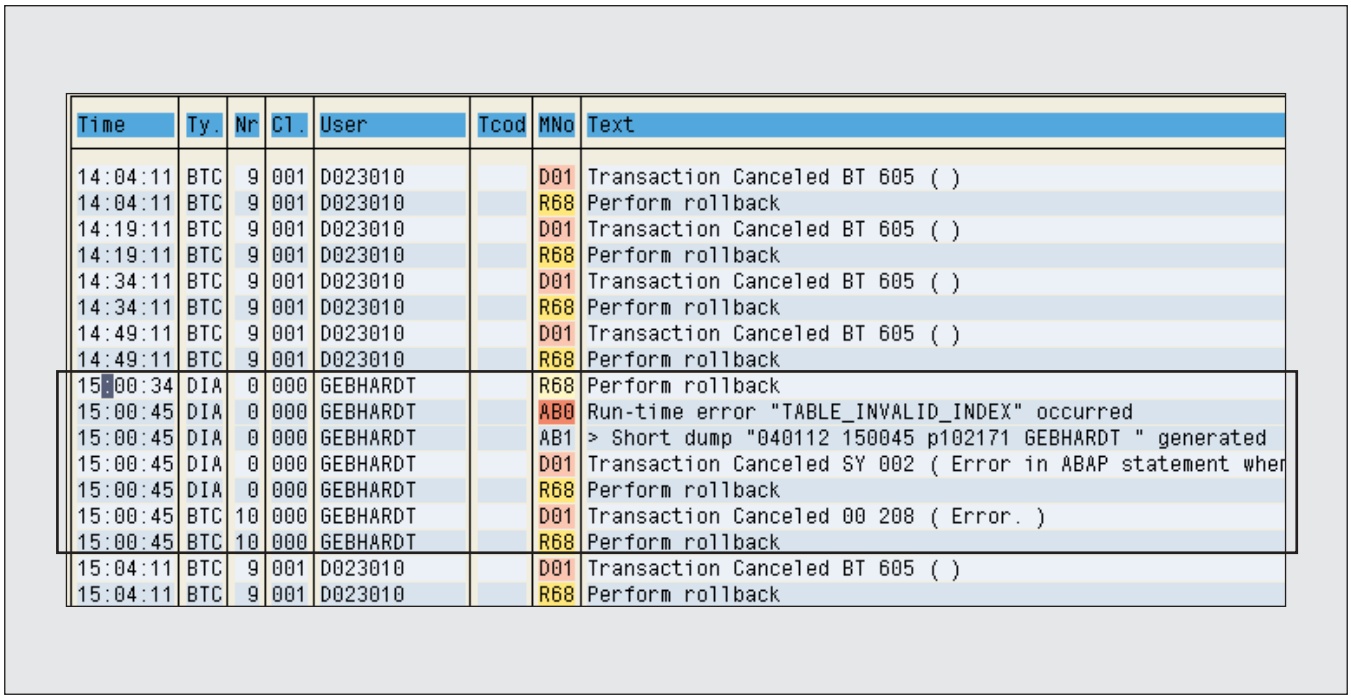
The troubleshooting article in the May/June 2004 issue examined a scenario in which a background job

Figure 11 Terminated Job in the Job Log



Date	Time	Message text	Message class	Message no.	Message type
12.01.2004	15:00:33	Job started	00	516	S
12.01.2004	15:00:33	Step 001 started (program ZTSTECHED02_SUBMIT_JOB, variant , user ID GEBHARDT)	00	550	S
12.01.2004	15:00:45	Error.	00	208	A
12.01.2004	15:00:45	Job cancelled	00	518	A

Figure 12 Root Error Details in the System Log



Time	Ty.	Nr	Cl.	User	Tcod	MNo	Text
14:04:11	BTC	9	001	D023010		D01	Transaction Canceled BT 605 ( )
14:04:11	BTC	9	001	D023010		R68	Perform rollback
14:19:11	BTC	9	001	D023010		D01	Transaction Canceled BT 605 ( )
14:19:11	BTC	9	001	D023010		R68	Perform rollback
14:34:11	BTC	9	001	D023010		D01	Transaction Canceled BT 605 ( )
14:34:11	BTC	9	001	D023010		R68	Perform rollback
14:49:11	BTC	9	001	D023010		D01	Transaction Canceled BT 605 ( )
14:49:11	BTC	9	001	D023010		R68	Perform rollback
15:00:34	DIA	0	000	GEBHARDT		R68	Perform rollback
15:00:45	DIA	0	000	GEBHARDT		ABO	Run-time error "TABLE_INVALID_INDEX" occurred
15:00:45	DIA	0	000	GEBHARDT		AB1	> Short dump "040112 150045 p102171 GEBHARDT " generated
15:00:45	DIA	0	000	GEBHARDT		D01	Transaction Canceled SY 002 (Error in ABAP statement wher
15:00:45	DIA	0	000	GEBHARDT		R68	Perform rollback
15:00:45	BTC	10	000	GEBHARDT		D01	Transaction Canceled 00 208 (Error.)
15:00:45	BTC	10	000	GEBHARDT		R68	Perform rollback
15:04:11	BTC	9	001	D023010		D01	Transaction Canceled BT 605 ( )
15:04:11	BTC	9	001	D023010		R68	Perform rollback

unexpectedly terminated. We discovered in the job log, shown in **Figure 11**, that the job was terminated because of an *ABORT* message.

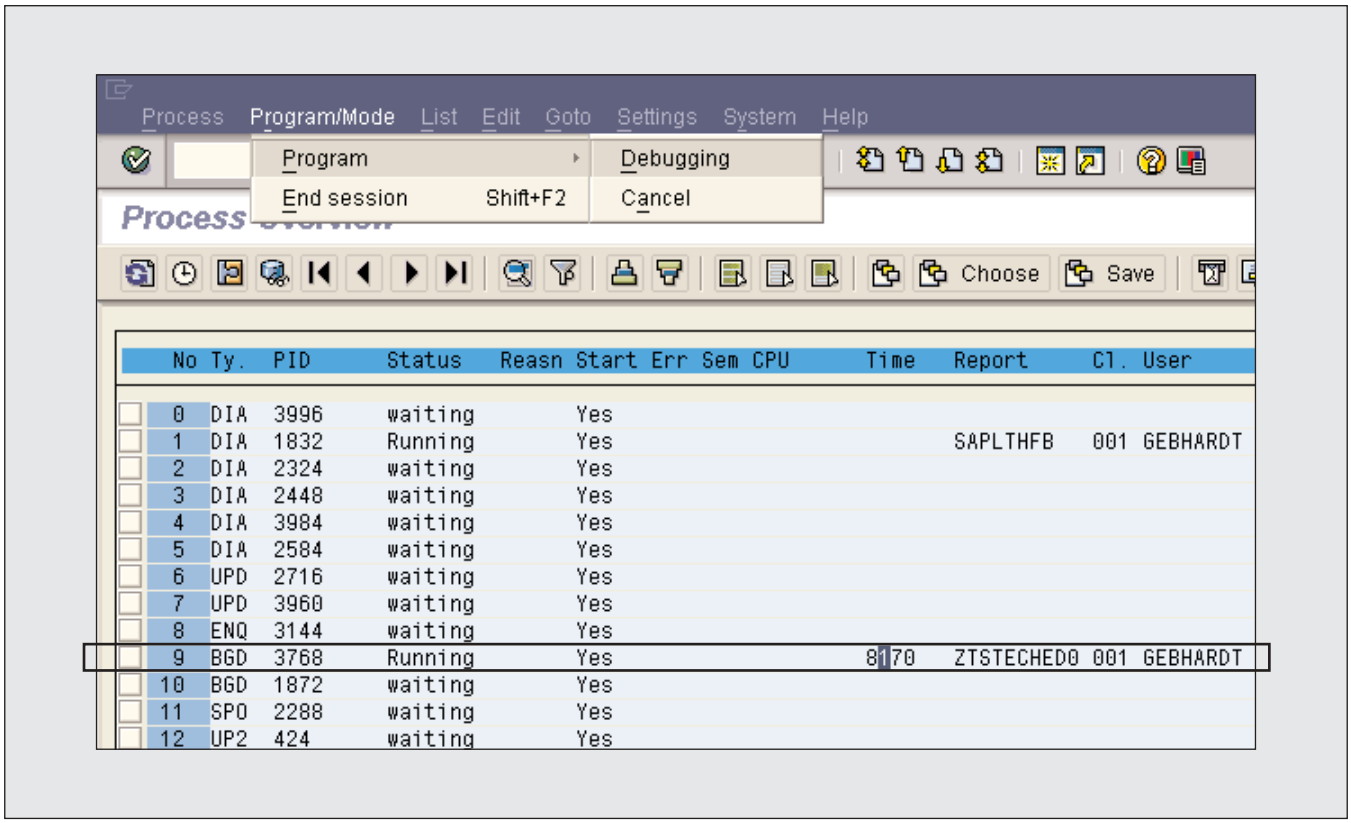
After analyzing the relevant part of the system log (see the highlighted area in **Figure 12**) and performing an ABAP Dump Analysis, we concluded that the root error of the terminated job was the runtime error *TABLE_INVALID_INDEX* in dialog work process 0.

Our theory was that a program running in a background process (batch work process 10) called a remote function call (RFC) module (running in dialog work process 0), which produced the ABAP dump *TABLE_INVALID_INDEX*. The program running in the background process then launched the *ABORT* message.

Unfortunately, the program didn't produce the error message when we ran it directly in the ABAP Editor (transaction *SE38*), so we need to debug the program while it is running in the background to validate the analysis.

Debugging a program running in the background is a bit complicated. If you set breakpoints in a program that is executed in background mode, program execution doesn't stop at the breakpoint. Because a

Figure 13 Starting the Debugger from the Process Overview for a Background Job



GUI session isn't attached to a background process, the debugger can't be displayed. Instead, the program ignores the breakpoints and continues executing. Even inserting a *BREAK-POINT* statement in the program doesn't start the debugger — it simply generates a *Breakpoint reached* entry in the system log.

It is possible to debug a background job, however, by using one of the following techniques:

- Jumping into the background job via Process Overview (transaction *SM50*)
- Restarting the background job in debug mode

Jumping into the Background Job via Process Overview (Transaction SM50)

First, run Process Overview (*SM50*) on the server where the background job is running. Then, identify the background process that is running the program (see process number 9 in **Figure 13**).[5]

Click on the checkbox next to the process and select *Program/Mode* → *Program* → *Debugging* from the Process Overview menu, as shown in Figure 13. In a few seconds, the debugger appears in a new window, and you can use the standard debug features, including breakpoints, watchpoints, and so on. Assuming that no breakpoints are reached, press the *F8* (*Continue*) key to close the debugger window and continue running the background job.

Unfortunately, this technique has a significant disadvantage. When you debug a running background job, you have no control over the point at which debugging begins. The debugger jumps into

[5] If you don't know on which server and in which process your background job is running, first run Job Overview (transaction *SM37*) and display the details for your job.

the application wherever it is at that time, which is imprecise at best. The situation is even worse if the batch job crashes very quickly, because there isn't enough time to jump into the debugger before it crashes.

A technique that offers you more control is to start debugging at a specific source line. To do this, insert an endless loop into your program directly before the relevant code:

```
DATA I TYPE I.
WHILE I <> 1.
ENDWHILE.

*** Start of the relevant code
```

The code causes the program to run forever in an endless loop, because the value of variable *I* is *0* and therefore always <> *1*. Start the background job, go get some coffee, and — after some time — the application will become trapped in the endless loop. Because you can be sure that the program is trapped in the loop, you can determine exactly which source line the batch job is running at the moment.

> ✓ ***Note!***
>
> *If other users also use the program, remember to restrict the endless loop to your user name. Otherwise, all users of the program will complain that it is hanging because they are trapped in the endless loop. Insert the following statement before the endless loop:*
>
> ```
> IF SY-UNAME = '<your user name>'.
> …endless loop…
> ENDIF.
> ```

Now you can control the point at which you start debugging when you jump into the debugger from the Process Overview (*SM50*). After the batch job starts executing the endless loop, change the variable *I* to *1* in the debugger, in order to leave the endless loop (*WHILE* condition *I <> 1*) and continue debugging.

Understandably, you may not want (or even be allowed) to change the program code to implement an endless loop. For example, you may not want to modify standard SAP code. Further, in a production environment, you may not be allowed to make program changes due to system settings that prevent changes to repository objects. Fortunately there is an alternative — you can restart the background job in debug mode.

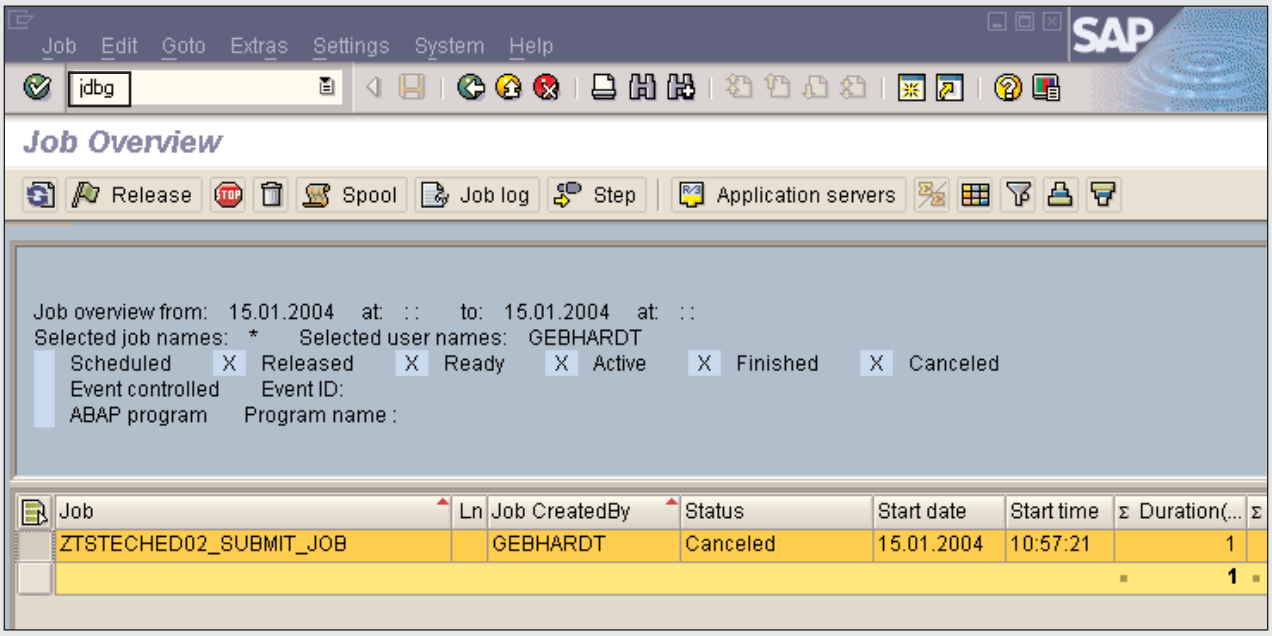
Restarting the Background Job in Debug Mode

Many developers are unaware of a valuable hidden feature that allows you to rerun a terminated or completed background job in debug mode. This technique allows you to debug a background job in a real background environment, including spool handling and execution of all job steps. This method may be your only option for investigating a background job problem, especially if the error doesn't occur when you run the program in dialog mode. The background job and its component steps rerun in a dialog process, which makes debugging possible, but it runs in an ABAP environment that is nearly identical to that of a background job.[6] You can observe the normal background spool handling and variant handling. In addition, *SY-BATCH* (the system variable that indicates if a program is running in the background) has a value of *X*, which, from an ABAP perspective, identifies the job as a background job.

To demonstrate how to rerun a background job in debug mode, let's revisit the example terminated background job. We can at last prove our analysis of the error (a runtime error occurred in a dialog process before the background job was terminated) and its root cause (an RFC module running in a dialog work

[6] Note that a few complex technical issues can't be simulated, such as memory allocation sequence or the fact that a batch process doesn't have an attached GUI.

Figure 14 Restarting a Background Job in Debug Mode in the Job Overview



> ***Caution!***
>
> *Rerunning a background job in debug mode is not merely a simulation, but a real job run. The system executes all updates, deletes, and other data modifications!*

process produced the ABAP dump *TABLE_INVALID_INDEX* after being called by a program running in a background process, which in turn launched the *ABORT* message).

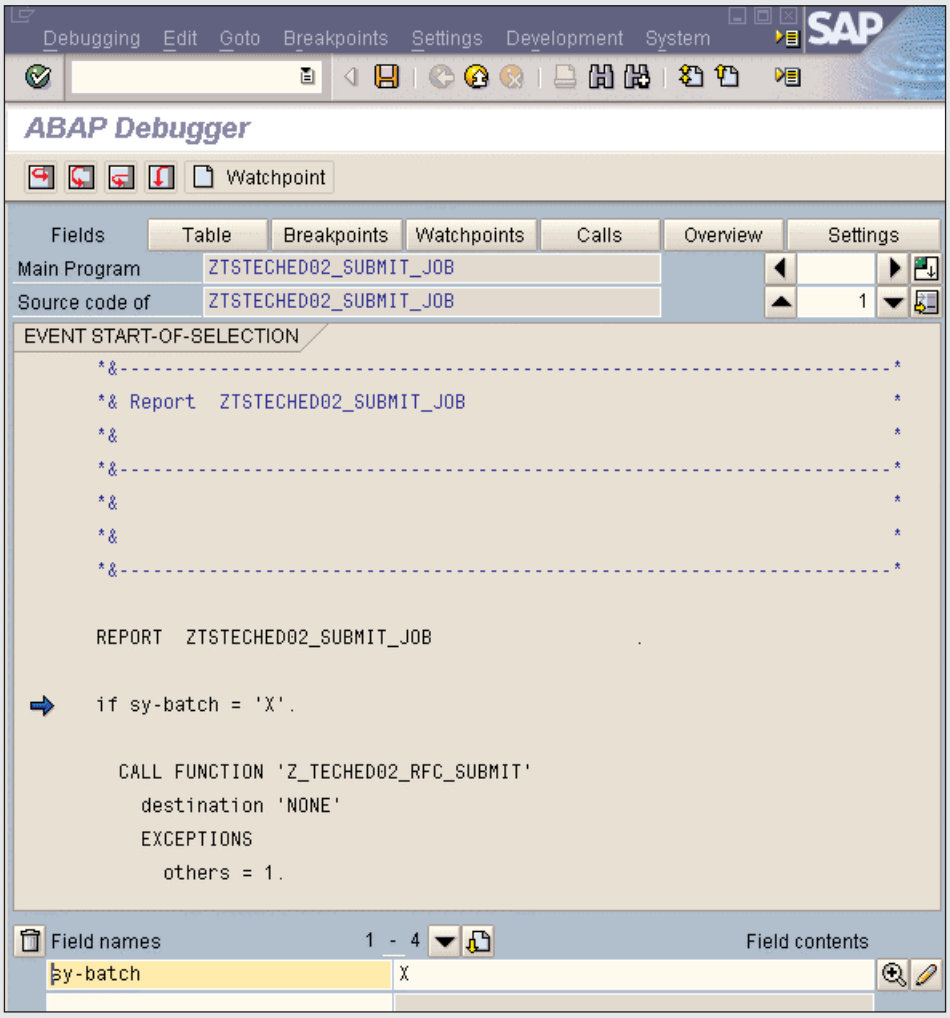
Debugging the Example Terminated Background Job

To restart a background job in debug mode, first run the Job Overview (transaction *SM37*). Highlight the terminated job and enter *jdbg* (the command for job debugging) in the command field, as shown in **Figure 14**. After executing the *jdbg* command, the system runs the highlighted job in debug mode.[7]

Spool handling (program *SAPLSPOO*) is automatically executed first for every batch job. Press the *F7* (*Step Return*) key several times to reach the first step in the batch job (that is, your ABAP program) and continue debugging. Alternatively, set a breakpoint in the ABAP Editor at the first line of the program, switch to the debugger, and press the *F8* (*Continue*)

[7] See SAP Note 573128 for details about the availability and limitations of this debugging technique.

Figure 15 Background Job Restarted in Debug Mode



key to reach the breakpoint. This technique takes you directly to the point of interest in the code, skipping spool handling.

The background job is now restarted in the debugger, as shown in **Figure 15**.

Note that the value of *SY-BATCH* (the system variable that indicates if the program runs in the background) equals *X*, just as it would for a "real" background job. Consequently, the test condition in the source code (*if sy-batch = 'X'*) is met, and the program reaches the branch where the RFC (*DESTINATION 'NONE'*) module *Z_TECHED02_RFC_SUBMIT* is called.

This behavior matches the theory we formulated from analyzing the system log — that a runtime error occurred in a dialog process before the background job terminated. We concluded that the batch job started a parallel dialog process by calling an RFC module. If we run this program in dialog mode, however, *SY-BATCH* is empty and the RFC function is not called, so we could not reproduce the error by running

Figure 16 Investigating What Happens Inside the RFC Module

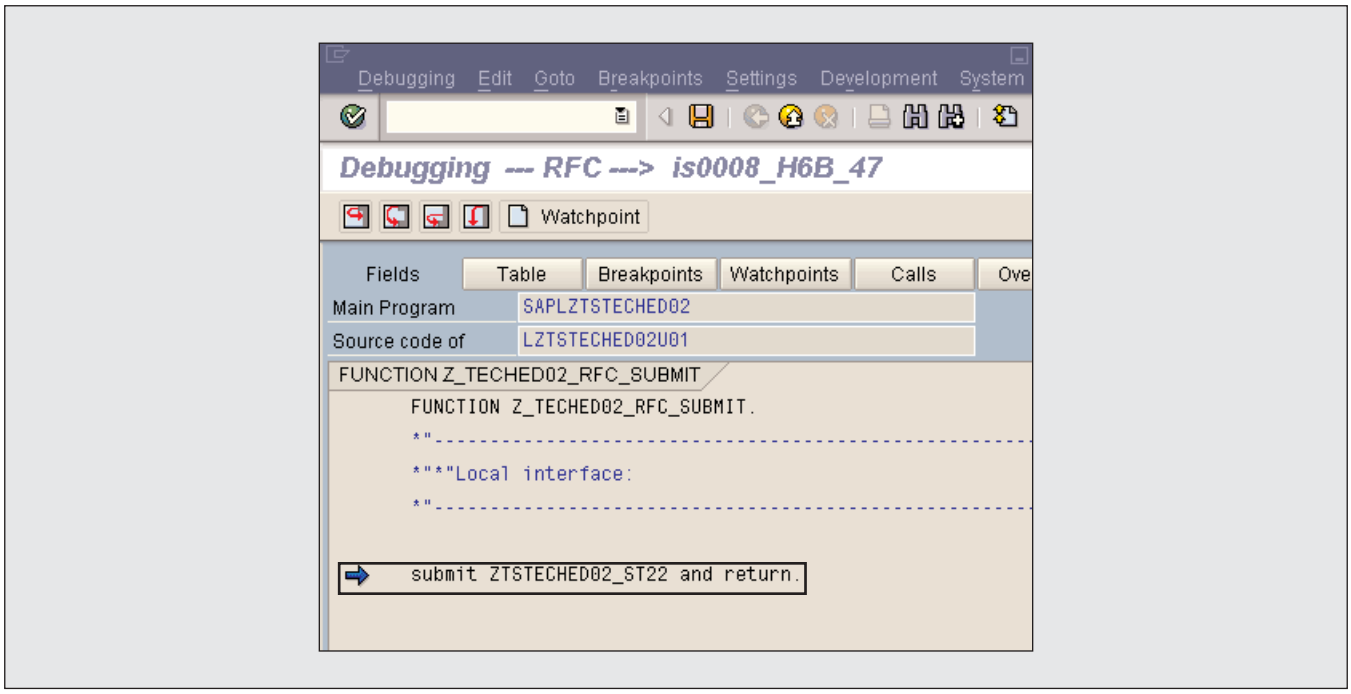
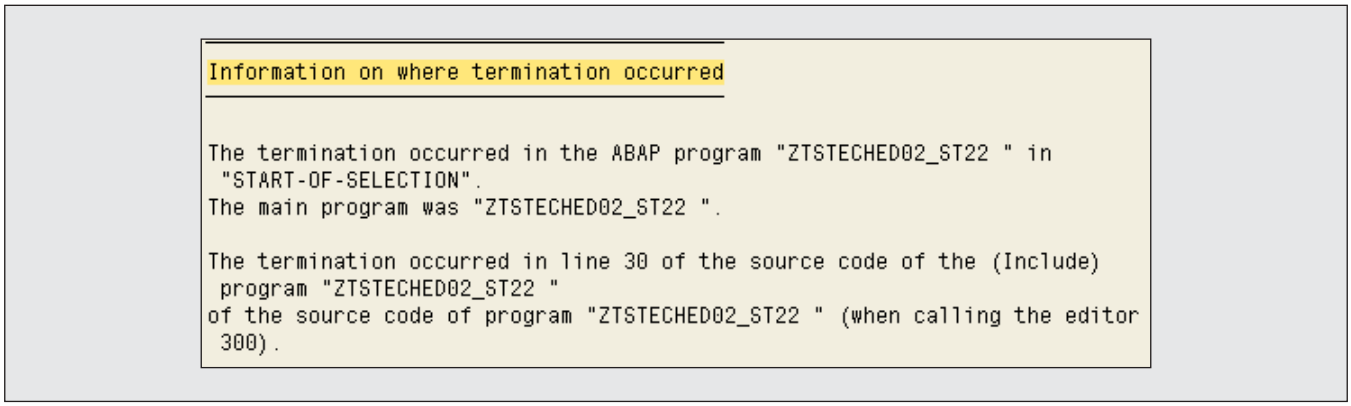


Figure 17 ABAP Dump Identifying the Terminated Program



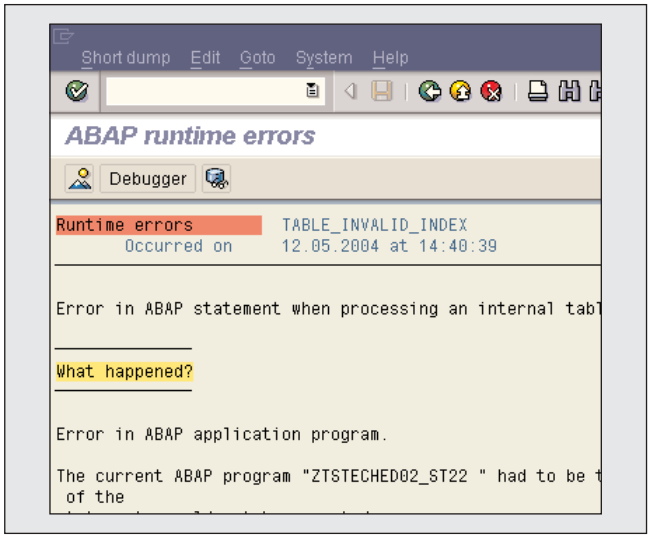
the program interactively. A common programming technique is to use *SY-BATCH* (or *SY-BINPT*, which indicates if a program is running in batch input mode) to separate the program flow for direct execution from the program flow for execution as a batch job. As a result, many programs, like the example program, behave differently in background mode than when executed directly.

As we continue debugging, we next discover that the report *ZTSTECHED02_ST22* is submitted inside this RFC module (see **Figure 16**).

We recognize this report from the ABAP Dump Analysis we performed in the previous article (see **Figure 17**), where we found that the ABAP dump *TABLE_INVALID_INDEX* occurred in this program.

Consequently, pressing the *F6* (*Step Over*) key in the debugger to execute the *SUBMIT* statement

Figure 18 Reproducing the ABAP Dump in the Debugger



produces the ABAP dump *TABLE_INVALID_INDEX*, as shown in **Figure 18**.

Press the *F3* (*Back*) key to step back, leaving the terminated RFC module and the terminated logon created by the RFC call. Back in the debugger, we can now determine how the application handles the error that occurred during the RFC call. The dump in the RFC module leads to a system exception. The caller of the RFC module catches all exceptions[8]:

```
CALL FUNCTION…
EXCEPTIONS
  OTHERS = 1.
```

Consequently, all exceptions are caught within the RFC function, and the system sets the value of *SY-SUBRC* to 1. By checking the value of *SY-SUBRC* in the *Field names* section of the debugger (see **Figure 19**) after returning from the terminated RFC

[8] If you specify *OTHERS* = <*number*> (instead of a specific exception) in the *EXCEPTIONS* addition of the *CALL FUNCTION* statement, all exceptions are caught. If an exception is caught, the system sets *SY-SUBRC* to <*number*>.

Figure 19 Confirming the Value of SY-SUBRC

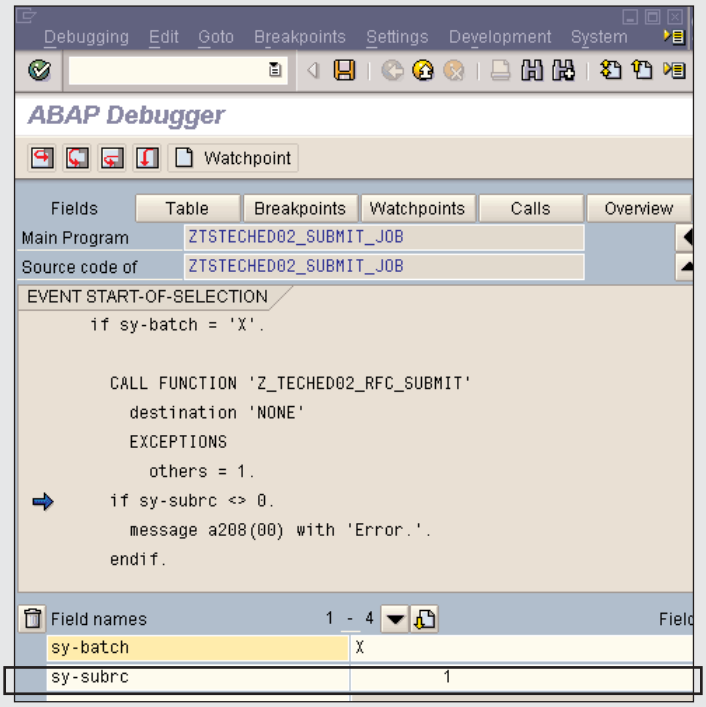
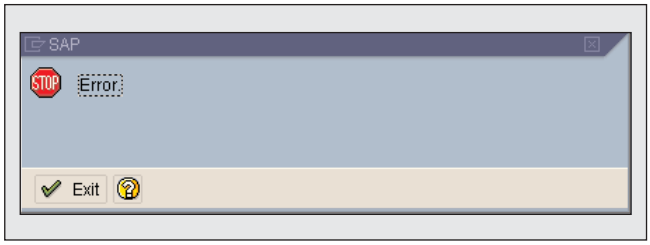


Figure 20 Reproducing the Error Message in the Debugger



call, we confirm that the value of *SY-SUBRC* does indeed equal 1.

Because *SY-SUBRC <> 0*, the debugger reaches the error-handling code of the application, which launches *ABORT* message 208(00) with the message text *Error* (see **Figure 20**). This error message matches the message we observed in the job log (see **Figure 21**).

By debugging the background job, we have verified that our analysis of the error situation was correct. The program running in the background calls an RFC module, which crashes with the ABAP dump *TABLE_INVALID_INDEX*. The system exception of the RFC module is caught, and the program launches an *ABORT* message. Case closed.

Now that you understand the various methods of starting the debugger, you're ready to learn how to use the debugger more effectively with breakpoints and watchpoints, which is how we'll determine when the example variable from earlier in the article (see Scenario 1 on page 78) is filled.

Using Breakpoints Effectively for Debugging

Starting the debugger is only the first step toward using it efficiently. Often you want to start debugging at a specific source line of a particular program.

Suppose your application has crashed with a runtime error, and you need to determine the values of some variables or the content of an internal table. You start in the ABAP dump by determining the precise source code position (program/include/line) where the error occurred. Next, you step to the ABAP Editor, display the source code, and set a breakpoint at the source line provided by the dump.[9] After restarting the application that crashed with the runtime error, you stop at the breakpoint in order to determine the values of the relevant variables.

In my experience, ABAP developers often complain about the debugger because they set breakpoints via the ABAP Editor or the debugger itself, yet the debugger doesn't stop. Most of these problems derive from not fully understanding the different breakpoint types. ABAP supports the following types of breakpoints,[10] which differ significantly in scope:

- **The ABAP statement BREAK-POINT:** This type of breakpoint is included in the program source code, and works for all sessions and all system users. To restrict this breakpoint to your

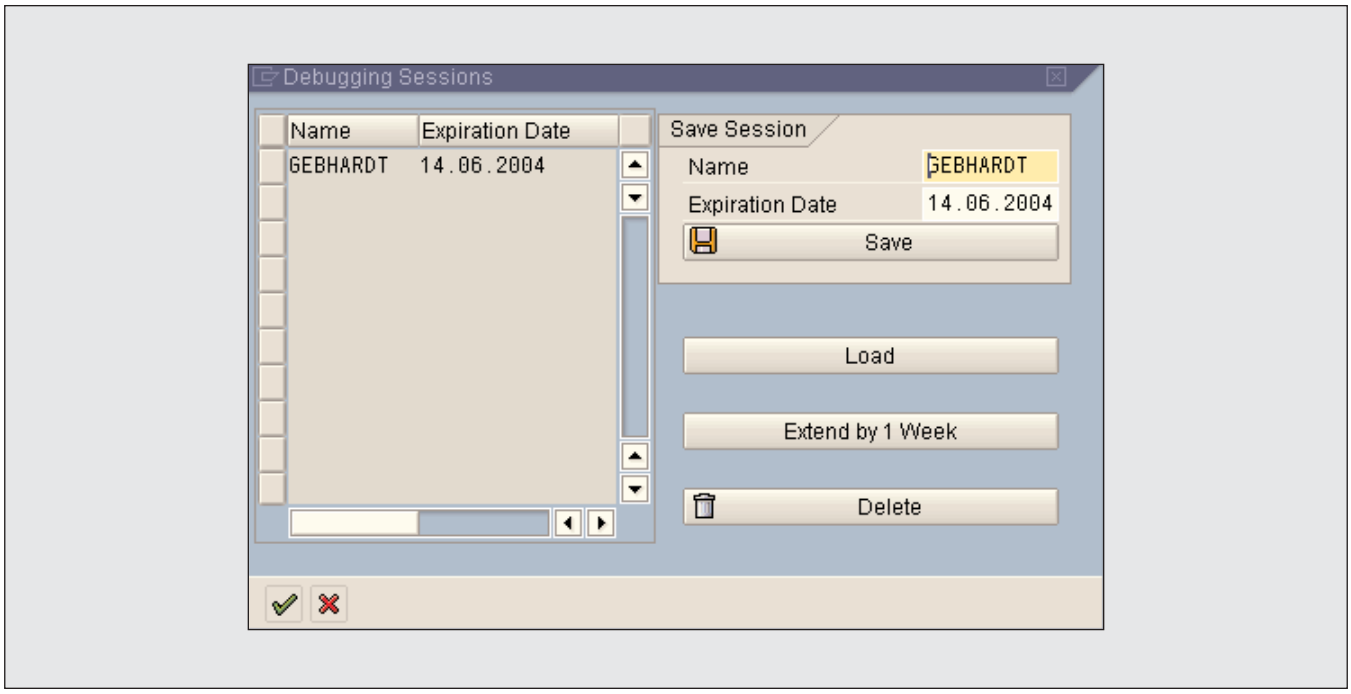
[9] For instructions on how to set a breakpoint in the ABAP Editor or the debugger, refer to the comprehensive online help for these tools.

[10] A fourth type of breakpoint, which is an external breakpoint used for HTTP and RFC debugging, is beyond the scope of this article.

Figure 21 Job Log Identifying the Error Message for the Terminated Job

Date	Time	Message text	Message class	Message no.	Message type
12.01.2004	15:00:33	Job started	00	516	S
12.01.2004	15:00:33	Step 001 started (program ZTSTECHED02_SUBMIT_JOB, variant , user ID GEBHARDT)	00	550	S
12.01.2004	15:00:45	Error.	00	208	A
12.01.2004	15:00:45	Job cancelled	00	518	A

Figure 22 Saving and Reusing Session Breakpoints in Debugging Sessions



user name, use the standard macro *BREAK <username>*.[11]

- **Session breakpoints:** Session breakpoints are set in the ABAP Editor, and are the same as saved debugger breakpoints (see the next bullet item). Session breakpoints are only available for your user name in the current logon session and, of course, all included external sessions (windows). If you log on again with your user account or another user account, the breakpoints are no longer available. Remember that changing the server on which you are running — via transaction *SM51* (SAP Servers), for example — implies a new logon, and thus your breakpoints are no longer available. However, as of SAP Web Application Server (SAP Web AS) 6.10, you can save session breakpoints in the database as debugging sessions. Select *Debugging* → *Sessions* in the debugger menu to display the Debugging Sessions dialog box (see **Figure 22**). To use these breakpoints again, load the appropriate debugging session using the same menu path.

- **Debugger breakpoints:** Breakpoints that are set in the debugger are only available in the current internal session. If your program contains a *SUBMIT* or *CALL TRANSACTION* that opens a new internal session, debugger breakpoints are not available in that new internal session. Only session breakpoints are available in all internal sessions. To ensure that breakpoints you set in the debugger are available for all internal and external sessions, save these debugger breakpoints by clicking on the save button (💾) in the debugger toolbar. These saved breakpoints are converted automatically to session breakpoints.

Understanding Source Line vs. Dynamic Breakpoints

Debugger and session breakpoints can be either source line or dynamic breakpoints.

[11] Standard macros are defined in the database table *TRMAC*. You can also define your own macros using the ABAP statement *DEFINE*.

To set a source line breakpoint in the debugger, simply double-click on a particular source code line and set the breakpoint.

Dynamic breakpoints are more flexible. To set dynamic breakpoints in the debugger, select *Breakpoints → Breakpoint at* and choose one of the following options:

- **Statements (stop at a specific statement):** For example, you might set a breakpoint at the *MESSAGE* statement to find out where a particular message is launched.[12] Or you might set a breakpoint at the *WRITE* statement to determine where the list output of an application starts. You can even set a breakpoint at *RFC* to stop execution if any RFC function is called.

- **Functions, Forms, Methods (stop at a specific module):** You can set a breakpoint in the Function Builder (transaction *SE37*) in order to stop execution at a specific function module. This breakpoint is available for all subsequently started debug sessions. However, suppose you are already in the debugger and decide that you want to stop at function *ABC*. If you don't want to restart your debug session, simply set a dynamic breakpoint at function *ABC* by selecting *Set breakpoint at → function* from the debugger menu.

- **Exceptions (stop when an ABAP exception *CX_xxx* is raised):** Modern ABAP programs typically use class-based exception handling, which is available as of SAP Web AS 6.10. You raise class-based exceptions with the ABAP statement *RAISE EXCEPTION TYPE class* to signal that an error occurred.[13] Suppose you find that an error occurs because exception *CX_ABC* was raised somewhere in the source code. Set a breakpoint at the exception *CX_ABC* so that the debugger will stop as soon as this exception is raised. Then analyze the source code in order to understand the error that caused the application to raise the exception.

- **System Exceptions (stop when a caught ABAP dump is launched):** This option is similar to the previous option, except that it applies to system exceptions.

While the first three options are fairly straightforward, the fourth requires a little more explanation. We'll look at system exception handling in more detail next.

Special Considerations for System Exception Breakpoints

Before the introduction of class-based exceptions, you could only catch some types of runtime errors with the statement *CATCH SYSTEM-EXCEPTIONS*. For example, a calculation that attempts to divide by zero (such as 1 / 0) would cause the runtime error *COMPUTE_INT_ZERODIVIDE*. The application would catch this runtime error with a code fragment such as:

```
data i type i.
CATCH SYSTEM-EXCEPTIONS
    COMPUTE_INT_ZERODIVIDE = 1.
  i = 1 / 0. " division by zero
…
endcatch.
if sy-subrc = 1.
  ** my error handling
endif.
```

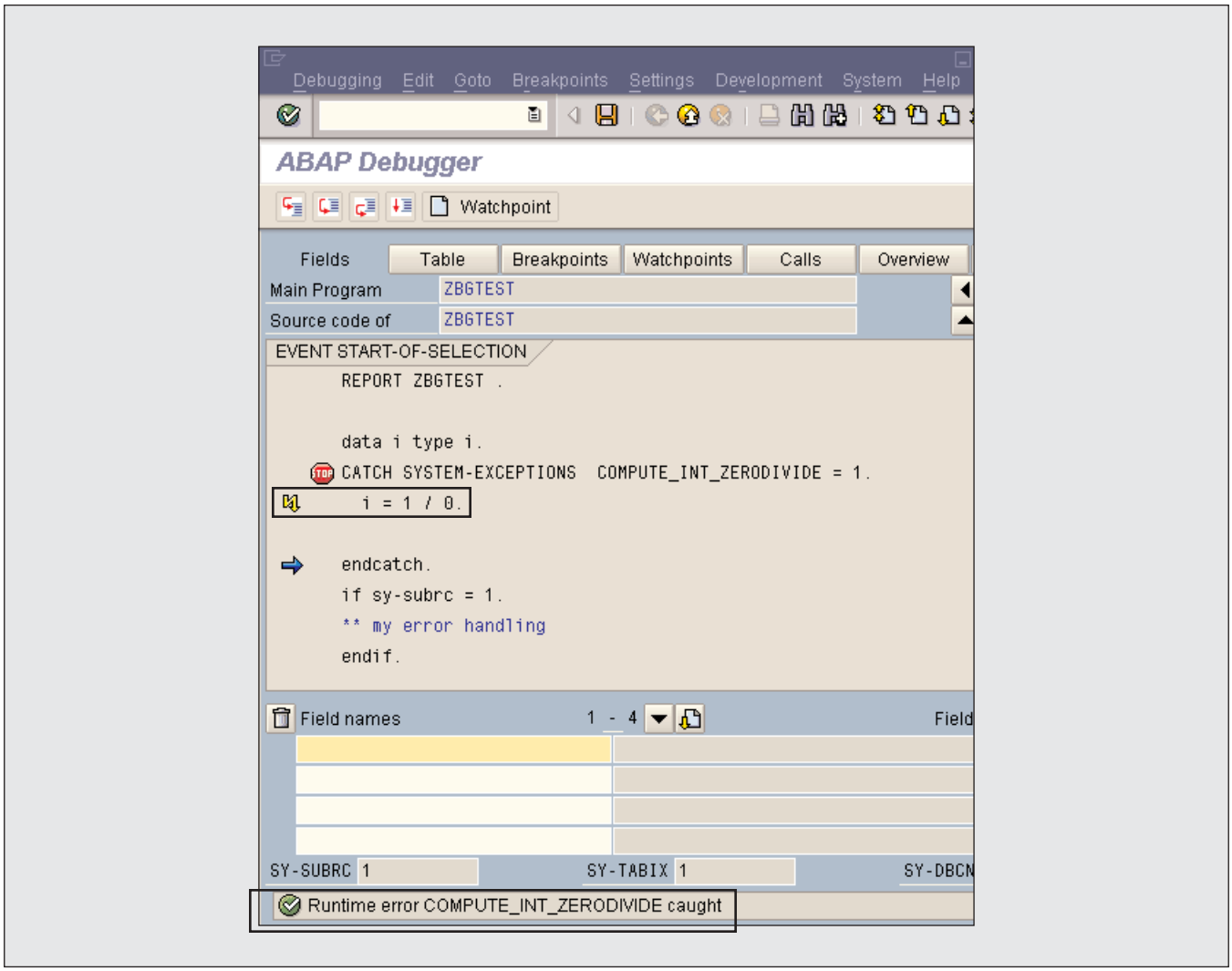
After the caught runtime error occurs, the system leaves the *CATCH/ENDCATCH* block and sets *SY-SUBRC* to the value specified in the *CATCH* statement (in this case, 1). The application then performs its own error handling after the *CATCH* block.

When the application uses this *CATCH* technique, the runtime error is typically not visible to the user. However, you may want to see these errors, because they could be the root cause of the error you are

[12] If you recall the scenario in the previous article in this series, where we were searching for a particular message, this technique offers another alternative.

[13] For more information, refer to the ABAP documentation for the *RAISE EXCEPTION TYPE class* statement and see the article "A Programmer's Guide to the New Exception-Handling Concept in ABAP" in the September/October 2002 issue of *SAP Professional Journal*.

Figure 23 Debugger Stopped at a Caught Runtime Error



analyzing. Setting a breakpoint on system exceptions causes the debugger to stop as soon as a runtime error is caught. As you can see in **Figure 23**, the debugger indicates the statement where the runtime error occurred (in this example, the calculation *i = 1 / 0*). The status bar across the bottom shows which runtime error was caught (in this case, *COMPUTE_INT_ZERODIVIDE)*.

With class-based exception handling, explicitly catching runtime errors in this manner is unnecessary. Instead, you simply catch the exception that is linked to a runtime error. In our example, the exception *CX_SY_ZERODIVIDE* is linked to the runtime error *COMPUTE_INT_ZERODIVIDE*. This exception will be raised as soon as the associated runtime error occurs[14]:

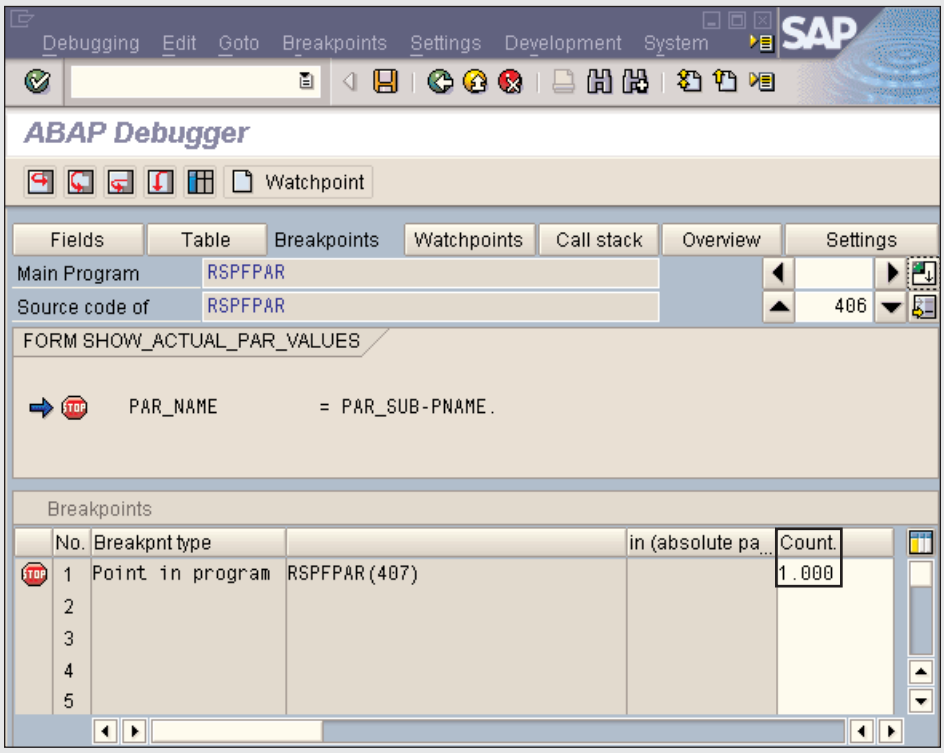
```
Runtime Errors          COMPUTE_INT_ZERODIVIDE
Exception               CX_SY_ZERODIVIDE
      Occurred on       17.06.2004 at 08:35:40
```

The following code fragment illustrates how to use class-based exceptions to catch the *COMPUTE_INT_ZERODIVIDE* runtime error:

[14] You can see the exception class for a runtime error in the header of a runtime error dump.

Skipping Breakpoints in Loops

Suppose you set a breakpoint in a loop that is executed 10,000 times. If you're only interested in the 1,000th execution, stopping execution after each loop and pressing the F8 (Continue) key would be painful. For source line breakpoints, you can specify a counter (such as 1,000, as you can see in the Count column in the screenshot below) that indicates how many times to skip the breakpoint until the program actually stops.

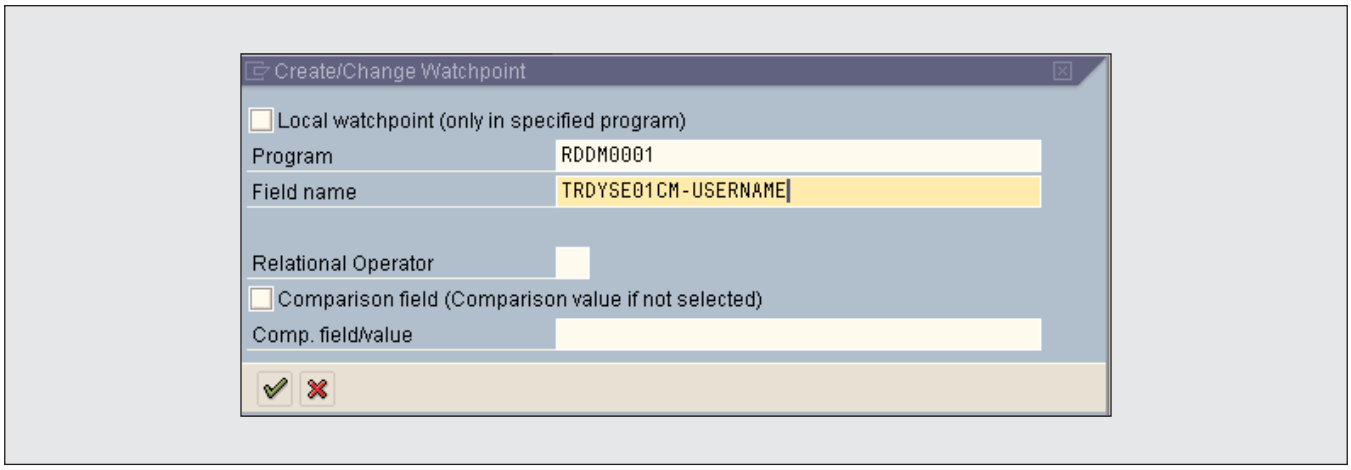


The real benefit of this feature is the ability to reach the point of interest quickly without losing time executing repetitive structures such as LOOPs, WHILE loops, or DO loops.

```
data i type i.
try.
  i = 1 / 0.
…
  catch CX_SY_ZERODIVIDE.
  ** my error handling
endtry.
```

Instead of setting a breakpoint on system exceptions, follow the menu path *Breakpoint at → Exception* in the debugger. Set a breakpoint for the *CX_SY_ZERODIVIDE* exception to stop as soon as the *COMPUTE_INT_ZERODIVIDE* runtime error occurs and the attached exception is caught. Use the *CX_SY_ARITHMETIC_ERROR* superclass to stop

Figure 24 Specifying the Watchpoint Details



execution for all caught arithmetic-based runtime errors, or the *CX_ROOT* superclass to stop execution for all exceptions.

In some troubleshooting situations, however, even dynamic breakpoints are not sufficiently dynamic. Suppose you want to determine when a specific field is populated or contains a particular value, as in our example earlier in the article. For this purpose, you need watchpoints.

Setting Watchpoints in the Debugger

Watchpoints enable you to watch the value of a variable during program execution and force the debugger to stop execution if a specified condition (such as a change of value) is fulfilled.

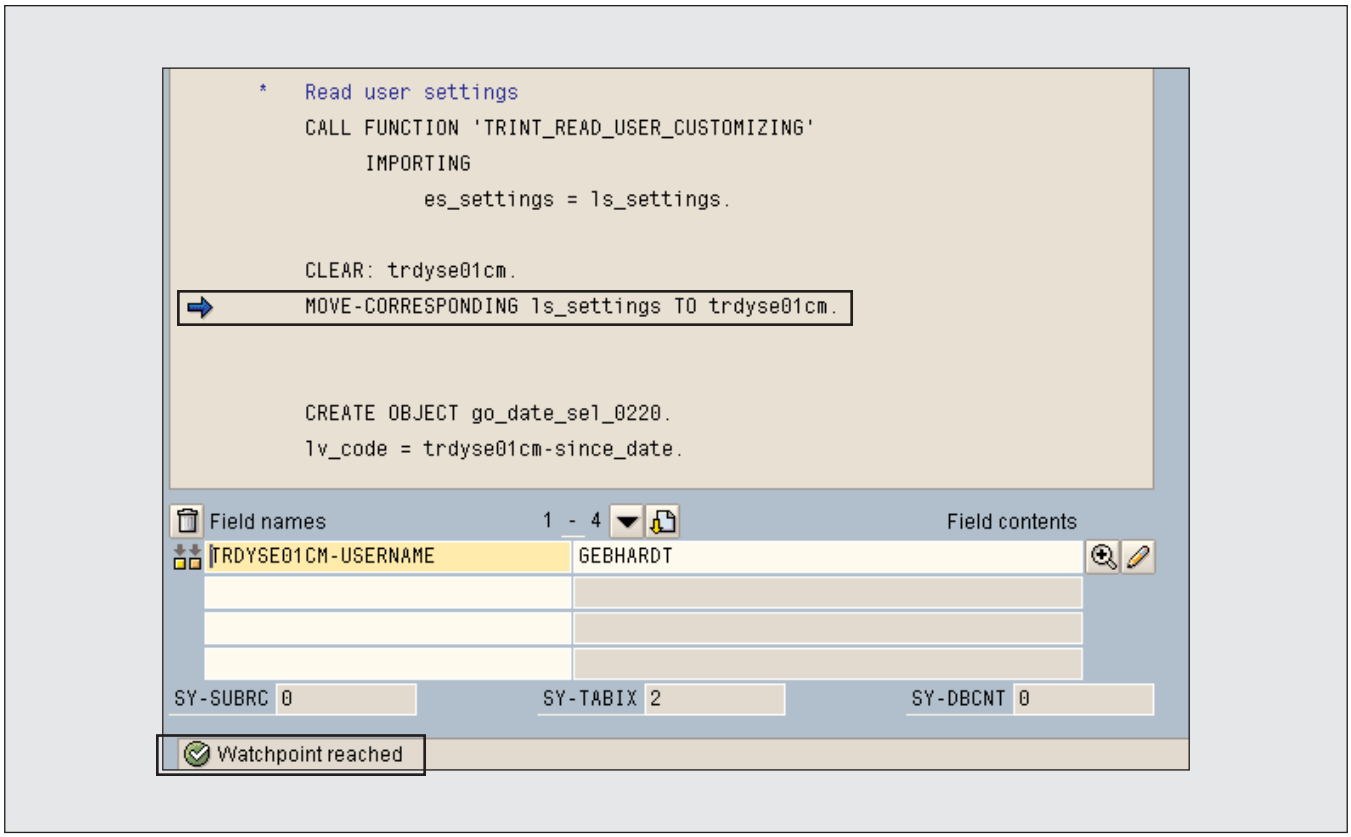
To illustrate the use of watchpoints, we continue with our earlier scenario of analyzing how a particular field is populated in the Transport Organizer (transaction *SE09*). Remember that from our initial analysis, we learned that the *User* field in the initial screen of the Transport Organizer is prepopulated with the variable *TRDYSE01CM-USERNAME* in the program *RDDM0001*. Additionally, we used Maintain Transaction (*SE93*) to start the Transport Organizer in debug mode.

We'll continue from that point, so we are now in the debugger before the first statement of transaction *SE09* is executed. We next want to set a watchpoint on the *TRDYSE01CM-USERNAME* variable, in order to find out when and how this variable is filled. Click on the *Create Watchpoint* button (Watchpoint) in the application toolbar of the debugger.

> ✓ ***The Create Watchpoint Button***
>
> *If you can't see the Create Watchpoint button, make sure that the debugger is running within the ABAP code and not in screen processing. If you are in screen processing, you see the screen flow logic (PBO/PAI, modules, etc.) in the debugger source display. The application toolbar only includes the Create Watchpoint button when the debugger is within ABAP code.*

In the Create/Change Watchpoint dialog box (shown in **Figure 24**), the *Program* field already contains the program you are debugging (*RDDM0001* in the example). In the *Field name* field, enter the variable you want to watch (in this case, *TRDYSE01CM-USERNAME*). Because we want to stop execution if there is *any* change to this variable, we don't use a relational operator (=, <, >, etc.) or a comparison value.

Figure 25 Debugger Stopped at the Specified Watchpoint



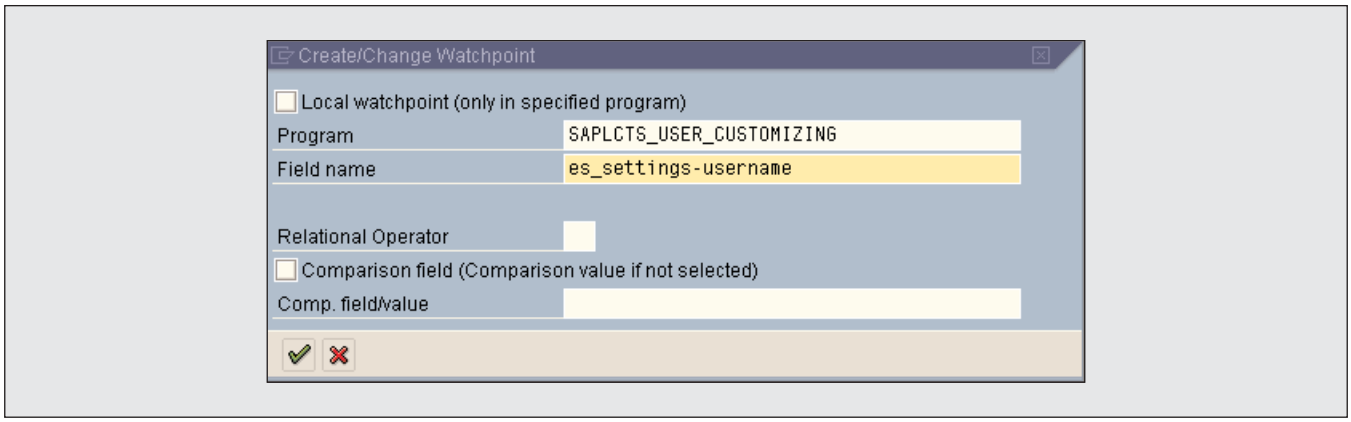
> ✓ ***Note!***
>
> *If you want to stop execution if the variable equals a specific user name, enter the = relational operator and enter the user name in the Comp. field/value field. Use the Comparison field checkbox to create watchpoint conditions such as "stop if SY-TABIX > SY-INDEX." These conditions compare the value of the watchpoint field with the value of another variable, instead of with a simple constant value.*

Click on the enter button (), which returns you to the debugger main screen. Press the *F8 (Continue*) key to continue. The debugger reaches our watchpoint (as indicated by the success message shown at the bottom of **Figure 25**) and stops at the line *MOVE-CORRESPONDING ls_settings TO trdyse01cm*.

The debugger stopped because the system detected a change in the *TRDYSE01CM-USERNAME* variable. Analyzing the information in the debugger provides further clues. After the *MOVE-CORRESPONDING ls_settings TO trdyse01cm* statement is executed, the *TRDYSE01CM-USERNAME* variable is filled with *GEBHARDT*. Remember that when we created the watchpoint for *TRDYSE01CM-USERNAME*, this variable was empty.

While this information is interesting, we still haven't solved the mystery of where the system obtains the user name that it passes to *TRDYSE01CM-USERNAME*. At the moment, we only know that the *MOVE-CORRESPONDING* statement (where we stopped because the watchpoint was reached) causes the *TRDYSE01CM-USERNAME* variable to be filled with the content of another variable, *LS_SETTINGS-USERNAME*. Therefore, our next step is to find out where *LS_SETTINGS-USERNAME* is filled. Directly

Figure 26 Setting a Watchpoint on the ES_SETTINGS Variable



before the *MOVE-CORRESPONDING ls_settings TO trdyse01cm* statement, we see that the variable *LS_SETTINGS* is filled as a result of the function *TRINT_READ_USER_CUSTOMIZING*:

```
CALL FUNCTION
      'TRINT_READ_USER_CUSTOMIZING'
  IMPORTING
    es_settings = ls_settings.
```

So, we set a breakpoint at this line, save (💾) the breakpoint, and then restart the application in the debugger by selecting *Debugging* → *Restart* in the debugger menu. When the debugger stops at the breakpoint, we press the *F5* (*Step Into*) key in order to step into the *TRINT_READ_USER_CUSTOMIZING* function. Now we need to set another watchpoint on the *ES_SETTINGS-USERNAME* result parameter that fills *LS_SETTINGS-USERNAME* (see **Figure 26**).

After creating the watchpoint and pressing the *F8 (Continue)* key, we stop again in the debugger (see **Figure 27**) — this time, directly after the line *es_settings-username = sy-uname*. This statement fills the *ES_SETTINGS-USERNAME* variable.

We have finally succeeded. First, the application looks in *CTSUSRCUST* (the standard Transport Organizer customizing table) to see if an alternative name is available for the current user[15]:

```
SELECT SINGLE * FROM CTSUSRCUST INTO
    ES_SETTINGS
  WHERE UNAME = SY-UNAME.
```

If this *SELECT* statement fails, *SY-SUBRC* equals 4 and *SY-DBCNT* (the system variable that provides the number of lines found after a *SELECT* statement) equals 0. As we can see in Figure 27, the bottom frame in the debugger shows exactly these values (*SY-SUBRC = 4* and *SY-DBCNT = 0*) for the system variables. Consequently, because there is no entry in *CTSUSRCUST* for the user name, the debugger reaches the *SY-SUBRC <> 0* branch of the *IF* statement that appears just after the *SELECT FROM CTSUSRCUST* statement. Finally, the *ES_SETTINGS-USERNAME* variable is filled with *SY-UNAME* in the following statement:

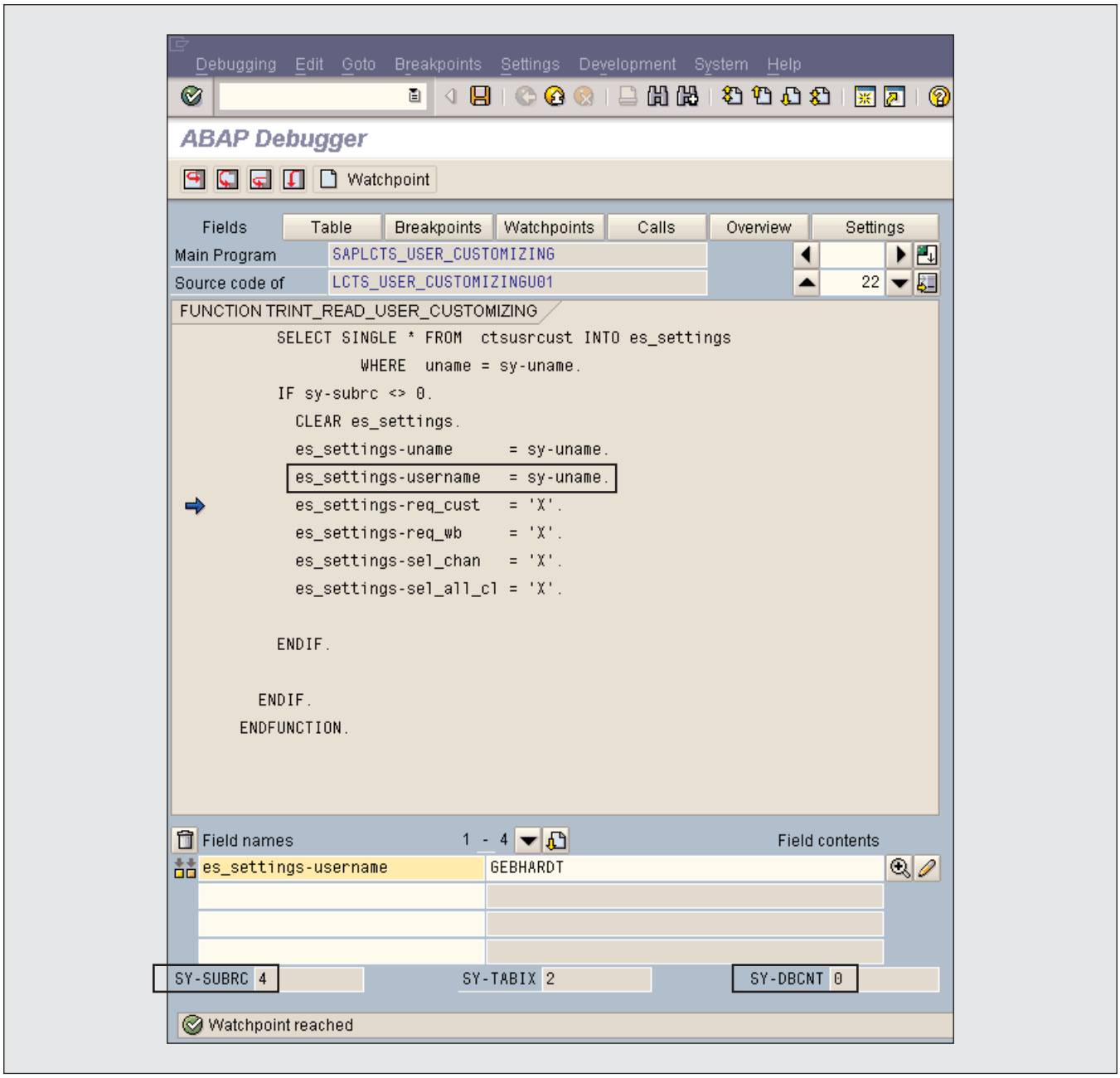
```
ES_SETTINGS-USERNAME = SY-UNAME
```

So, let's summarize our findings. If, as in our case, no entry can be found for the user in the *CTSUSRCUST* customizing table, the system uses the *SY-UNAME* variable as a default to fill the *ES_SETTINGS-USERNAME* variable. This variable in turn fills another variable, *LS_SETTINGS-USERNAME*. The *MOVE-CORRESPONDING* statement uses this variable to fill the variable that we are seeking (*TRDYSE01CM-USERNAME*):

```
MOVE-CORRESPONDING ls_settings TO
  trdyse01cm.
```

[15] The user name is always available in the system variable *SY-UNAME*.

Figure 27 Debugger Stopped at the Second Watchpoint

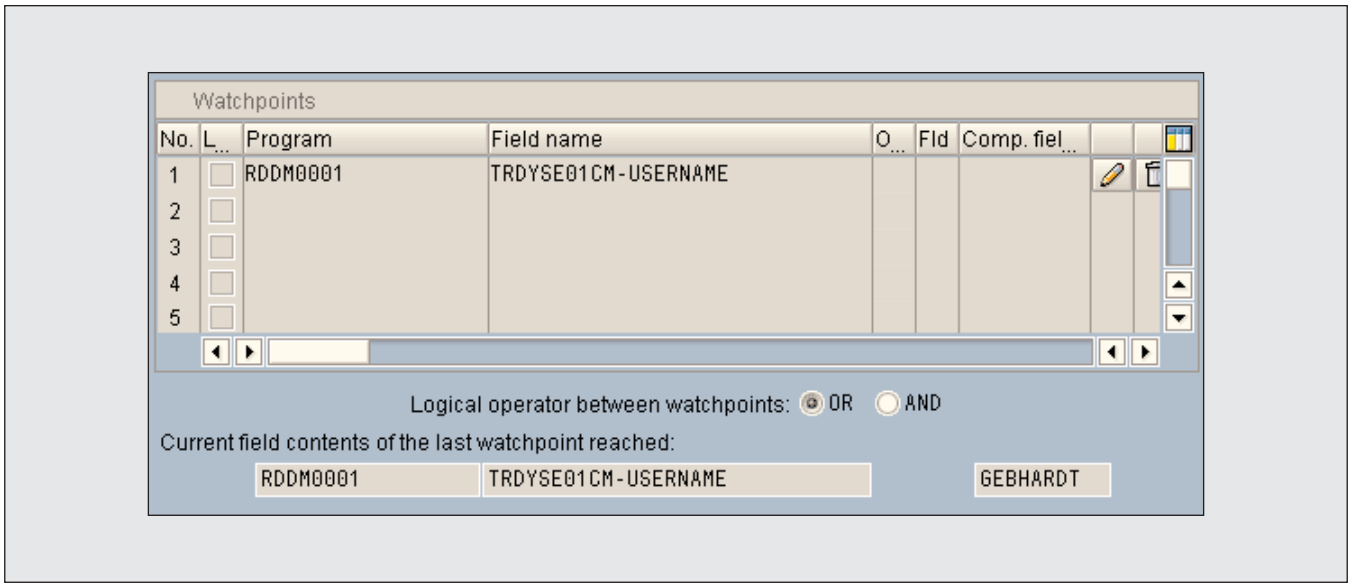


At last, this variable appears in the *User* field on the initial screen of the Transport Organizer (*SE09*).

Let's consider our troubleshooting scenario for a moment. Suppose some users report that the initial screen of the Transport Organizer displays an incorrect value in the *User* field for their user accounts. Using the demonstrated watchpoint technique, we might discover that an incorrect entry exists for this user in the *CTSUSRCUST* customizing table.

Within minutes, the watchpoint technique answered the question of how and where a specific

Figure 28 *The Watchpoint View in the Debugger*



variable is filled. You could waste hours trying to solve this problem by analyzing the code or single-stepping in the debugger.

If you like, you can view the last value of your watchpoint variable (in this case, *TRDYSE01CM-USERNAME*) in the Watchpoint view of the debugger (see **Figure 28**). You can also maintain (i.e., change or delete) watchpoints in this view. To go to the Watchpoint view, click on the *Watchpoints* button (Watchpoints) in the navigation bar of the debugger.

While watchpoints are very useful for monitoring the content of variables, it is important to keep the following restrictions in mind:

- ☑ Unlike breakpoints, watchpoints can't be saved. They disappear after you end the current internal session in the debugger. You need to re-create them in your new internal session. Because switching to a new internal session serves as an implicit end of the current debug session, any watchpoints are lost.

- ☑ If your application uses *TABLES* areas or *COMMON PARTS*[16] that are shared among several programs, consider restricting the watchpoint to your program in order to find out when it (rather than another program) changes the variable. Click on the *Local watchpoint* checkbox in the Create/Change Watchpoint dialog box.

- ☑ Due to technical restrictions, you can't set watchpoints directly on internal tables or object instances. The sidebar on the next page shows how you can monitor at least some internal table characteristics, such as the number of lines in an internal table. For object instances, you can set watchpoints on object attributes (for example, *ref->a*, where *ref* is an object reference), although not on the object itself.

The last bullet point is an unfortunate restriction. The desire to stop in the debugger due to a change (made by statements like *APPEND*, *INSERT*, *DELETE*, or *REFRESH*) in an internal table is a common situation. Fortunately, as I mentioned above, a workaround exists, as we learn from our next troubleshooting scenario, outlined in the sidebar on the next page.

[16] For more information, refer to the ABAP documentation on keywords.

Scenario 2: Investigating an Internal Table Refresh

Let's say that after hours of troubleshooting, you conclude that the root cause of all your problems is that an internal table ITAB is refreshed somewhere in the application code. Unfortunately, the application contains millions of lines of code. Finding the guilty code by reading the source code or single-stepping in the debugger would be a nightmare.

Clearly, the best strategy is to set a watchpoint on the internal table ITAB so that as soon as the number of lines in the table changes, the debugger stops execution, but you can't set a watchpoint on internal tables. Fortunately, however, there is a little-known feature that allows you to monitor at least some characteristics of an internal table — for example, you can monitor the line count or the loop counter (as incremented in a LOOP) of an internal table.

Each internal table has a table header that contains administrative details such as the table ID, the loop counter, and — very essential — the number of lines. The following table shows you how to display the table header of an internal table ITAB in the "Field name" frame of the debugger in different SAP R/3 kernel releases:

Kernel Release	Notation to Display the Table Header
Before 4.0B	itab-*sys*
4.0B	itab[]
4.5A, 4.5B	itab[]+0(128)
4.6A and higher	*itab[] (Note that the header is only populated when the table is not initial.)

(continued on next page)

Preview of the New ABAP Debugger

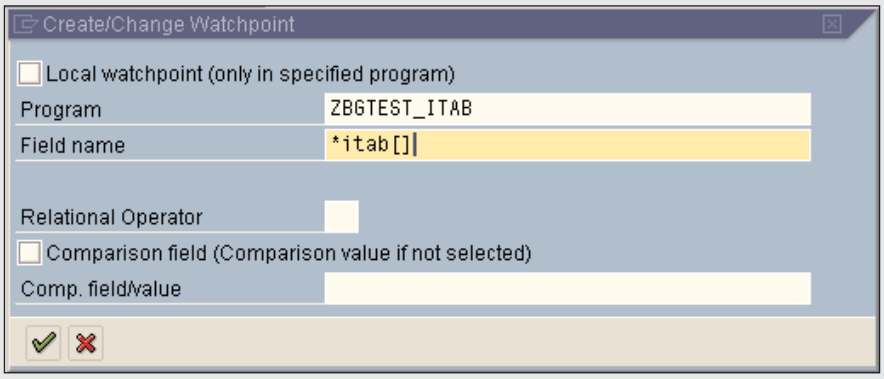
SAP NetWeaver '04 delivers the first version of the New ABAP Debugger.[17] The New ABAP Debugger is a broad enough topic to warrant its own article, but I wanted to provide you with a glimpse into the future of ABAP debugging, as it involves many helpful changes that will make your debugging life easier.

The main difference between the still-available Classic ABAP Debugger and the New ABAP Debugger is that the application and the debugger now run in separate external sessions (windows). You can view the running application in external session 1 (the "Debuggee"), and the debugger in a parallel external

[17] The New ABAP Debugger doesn't yet offer the full functionality of the Classic ABAP Debugger — for example, you can't set watchpoints or debug special sessions such as update, HTTP, or background jobs. For this reason, in NetWeaver '04, you can switch between versions while debugging.

(continued from previous page)

While you can't set a watchpoint on the internal table itself, you can set a watchpoint on its table header. The following screenshot shows the necessary settings:



As soon as the table header changes, because someone refreshed the table or appended a line, for example, the debugger stops.

Remember the following caveats when setting watchpoints on table headers:

☑ The table header is also changed when you execute a LOOP AT ITAB statement, because the loop counter that is stored in the table header is incremented.

☑ The table header is only available if the internal table was already populated. If you try to set a watchpoint on the table header of an internal table that has not yet been populated, the debugger displays an error message indicating that this variable does not exist and no watchpoint can be created.

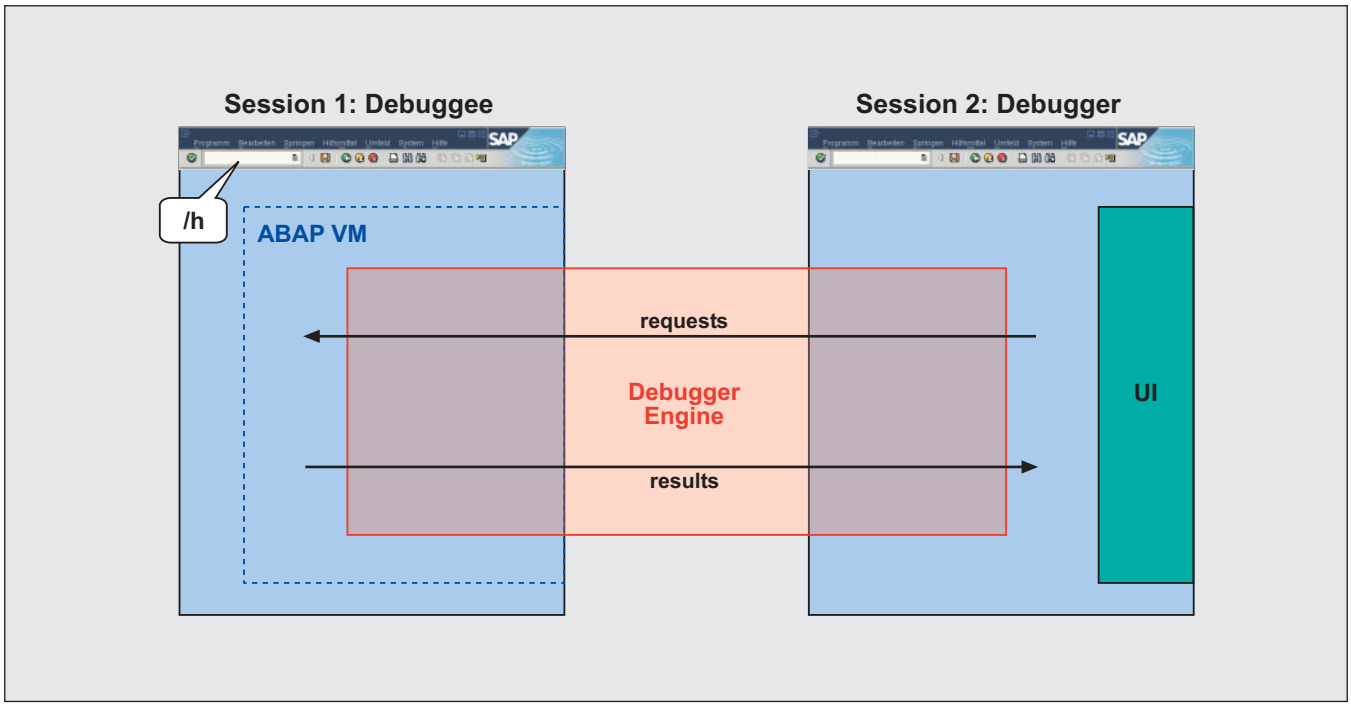
Using this technique, you can locate the REFRESH of an internal table in a complex application within seconds (or at least minutes). After finding the guilty REFRESH of the internal table, you can analyze the code to determine why the refresh occurs and how to correct the code.

session 2 (the "Debugger"), as you can see in **Figure 29**. The debugger engine transfers requests (such as a single step command) from the debugger to the debuggee, and then returns the results from the debuggee to the debugger.

The Classic ABAP Debugger runs in a single external session with the debugger screens inserted in the running application. Not surprisingly, this approach has some drawbacks:

- You can't debug ABAP that is called directly from the kernel (such as conversion exits or field exits).
- ABAP can't be used in the user interface of the Classic ABAP Debugger, because an ABAP

Figure 29 Architecture of the New ABAP Debugger



statement in the debugger could influence the application being debugged (for example, if the value of *SY-SUBRC* is changed).

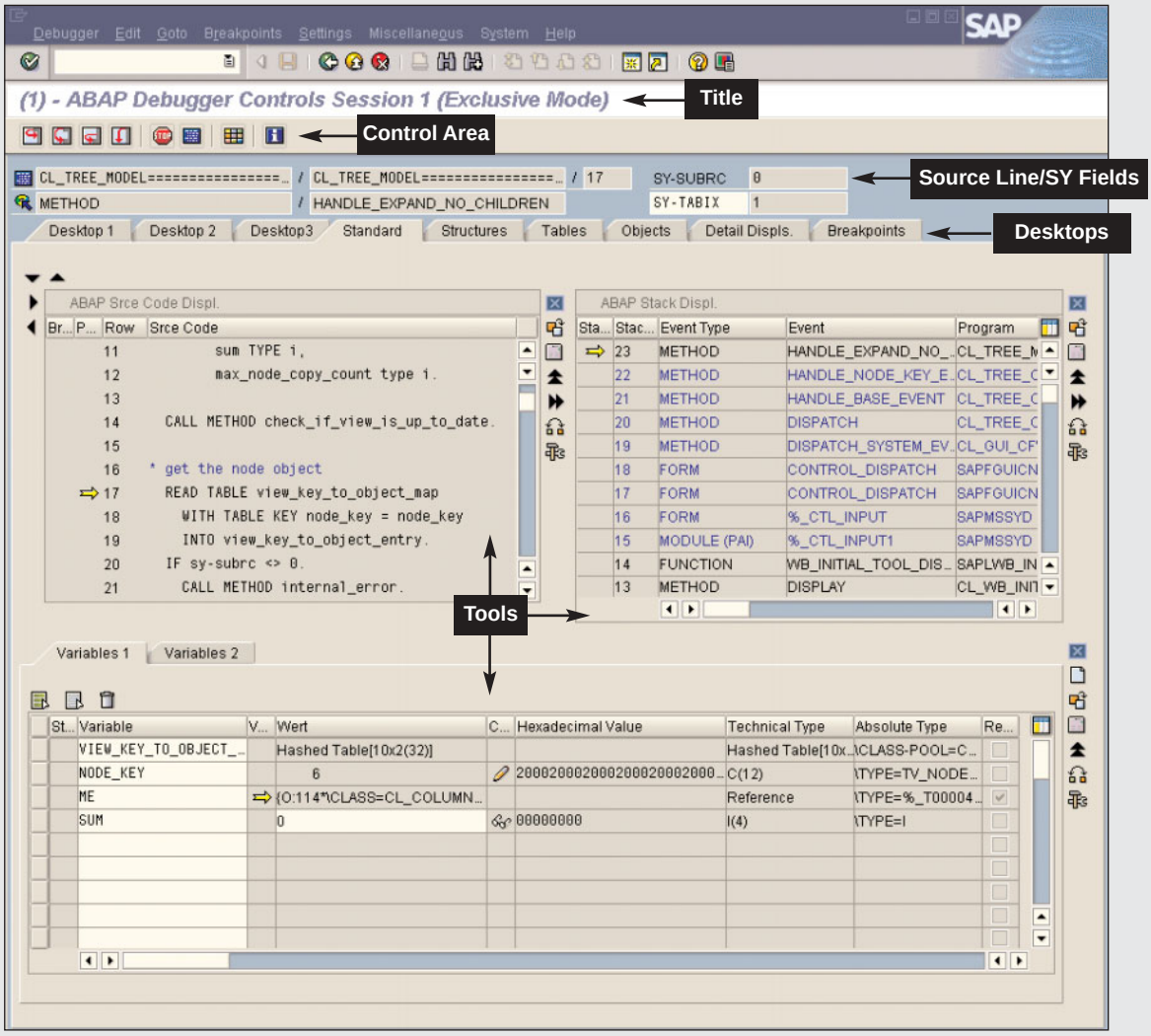
For this reason, the Classic ABAP Debugger is not really an ABAP application. Most of its functionality is implemented in the kernel. Consequently, enhancing the Classic ABAP Debugger or adding a state-of-the-art user interface with controls was nearly impossible, because these technologies are encapsulated in ABAP.

In contrast, the New ABAP Debugger (where the debugger and the application are separate) offers the following advantages:

- The New ABAP Debugger can leverage the full power of ABAP, because there is no danger of interference with the application. SAP is now able to offer a modern user interface for the debugger, which can be easily customized to your needs.
- Except for macros, you can debug all ABAP statements (including conversion exits and field exits).
- The New ABAP Debugger is connected to the external (debuggee) session, rather than to the internal session (as with the Classic ABAP Debugger). When the program is finished, the debugger is not closed. Instead, the inactive debugger session remains pending and is reactivated if you start debugging again. After reactivating the debugger session, your debugger user interface settings and breakpoints are still available. The New ABAP Debugger is closed automatically when you close the debuggee session.

  Alternatively, you can explicitly close or detach the debugger from the debuggee by entering */hx* in the command field of the active debuggee session. If the debugger is active, you can of course end debugging there as well.

Figure 30 The New ABAP Debugger User Interface



Let's take a brief look at the user interface for the New ABAP Debugger, as shown in **Figure 30**.

The screen layout is optimized for maximum flexibility, with many areas that you can configure to suit your needs. **Figure 31** describes the primary areas and features of the new screen layout.

As an example of how you might want to configure debugger desktops, you could easily display four internal tables for comparison (as shown in **Figure 32**).

The first three desktops are user-specific. After configuring and saving the layout for these desktops,

Figure 31 User Interface Areas in the New ABAP Debugger

User Interface Area	Description
Title (Process Info)	Identifies the debuggee session to which the debugger is connected.
Control Area	Contains the debugger control keys (F5, F6, etc.), which you'll recognize from the Classic ABAP Debugger.
Source Line/SY Fields	Displays program information and system fields (such as SY-TABIX).
Desktops	Enables you to run multiple desktops (up to nine) for a debugging session, displaying a different type of information in each window.
Tools	Enables you to populate each desktop individually with your choice of up to four tools, such as a stack display or an editor. For example, you might configure one desktop with four tools, and another desktop with one tool, displaying a single internal table in full-screen mode.

Figure 32 Comparing Four Internal Tables in the New ABAP Debugger

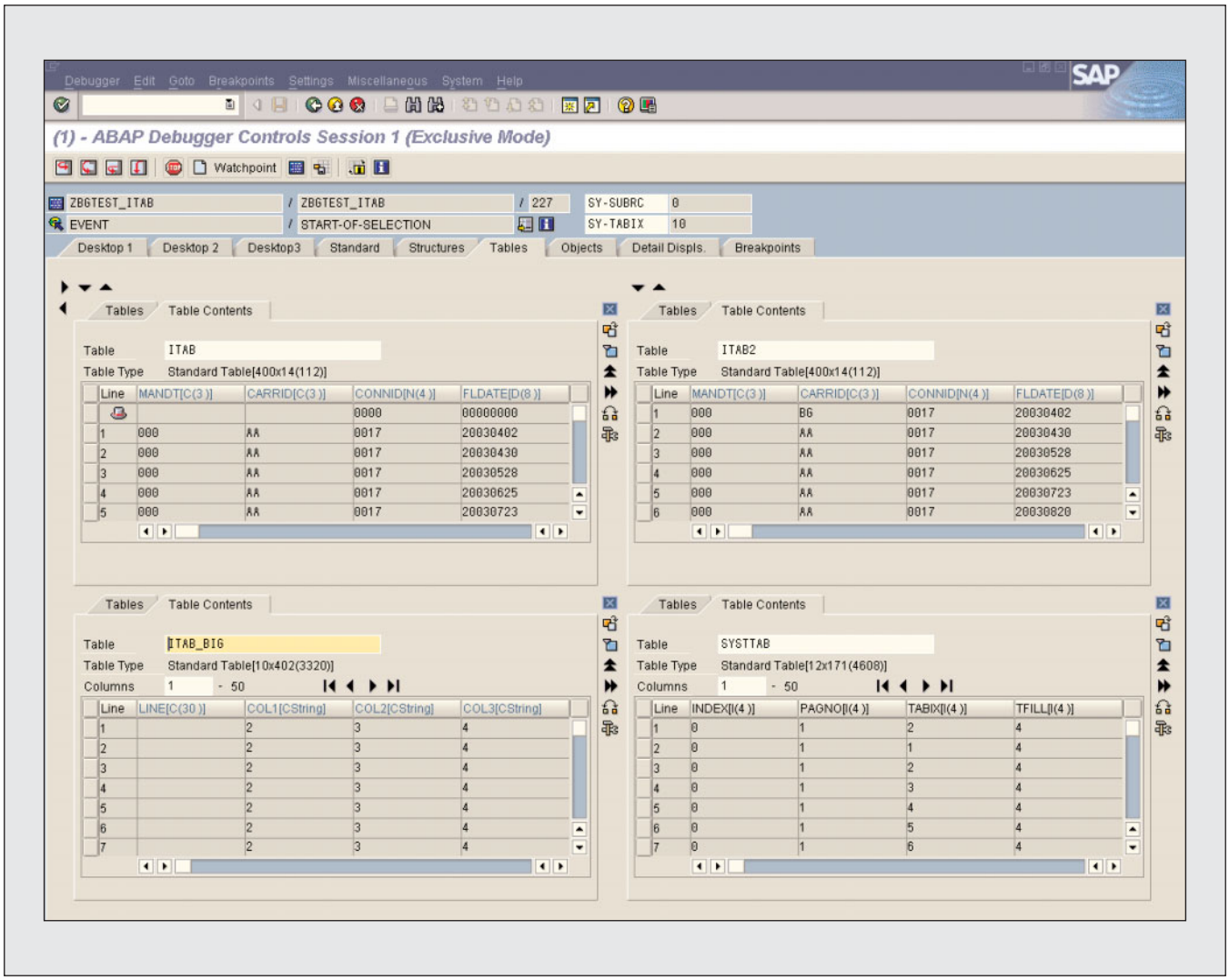


Figure 33 Tool Enhancements in the New ABAP Debugger

Tool	Enhancement
Source Display	Improved support of source lines longer than 70 characters.
Stack	Shows both the include and source line for each stack line, combined with the ability to navigate to the ABAP Editor.
Breakpoints	Supports integrated checkpoints.
Variables	For a variable, displays its value, hexadecimal value, technical type, and absolute type at a glance.
Structure View	Ability to change the content of a structure.
Internal Table View	Convenient navigation between internal table columns.
Object View	Displays the object inheritance relationship.
Simple Type View	Supports converting a hexadecimal value using a provided code page.

your layout settings are loaded automatically when you restart debugging. The other six desktops are preconfigured. Although you can change the layout for these desktops for the current session, you can't save your changes.

In addition to adding the ability to customize the debugger layout, all available tools were enhanced with a range of new features, which are described in **Figure 33**.

In the next release of NetWeaver, SAP plans to implement additional debugging tools (see **Figure 34**),[18] including:

- **Globals** for displaying all global variables of either the current program or all loaded programs
- **Locals** for displaying all local variables of, for example, a function module, and the parameters of the module
- **A new edit control** for displaying the source code

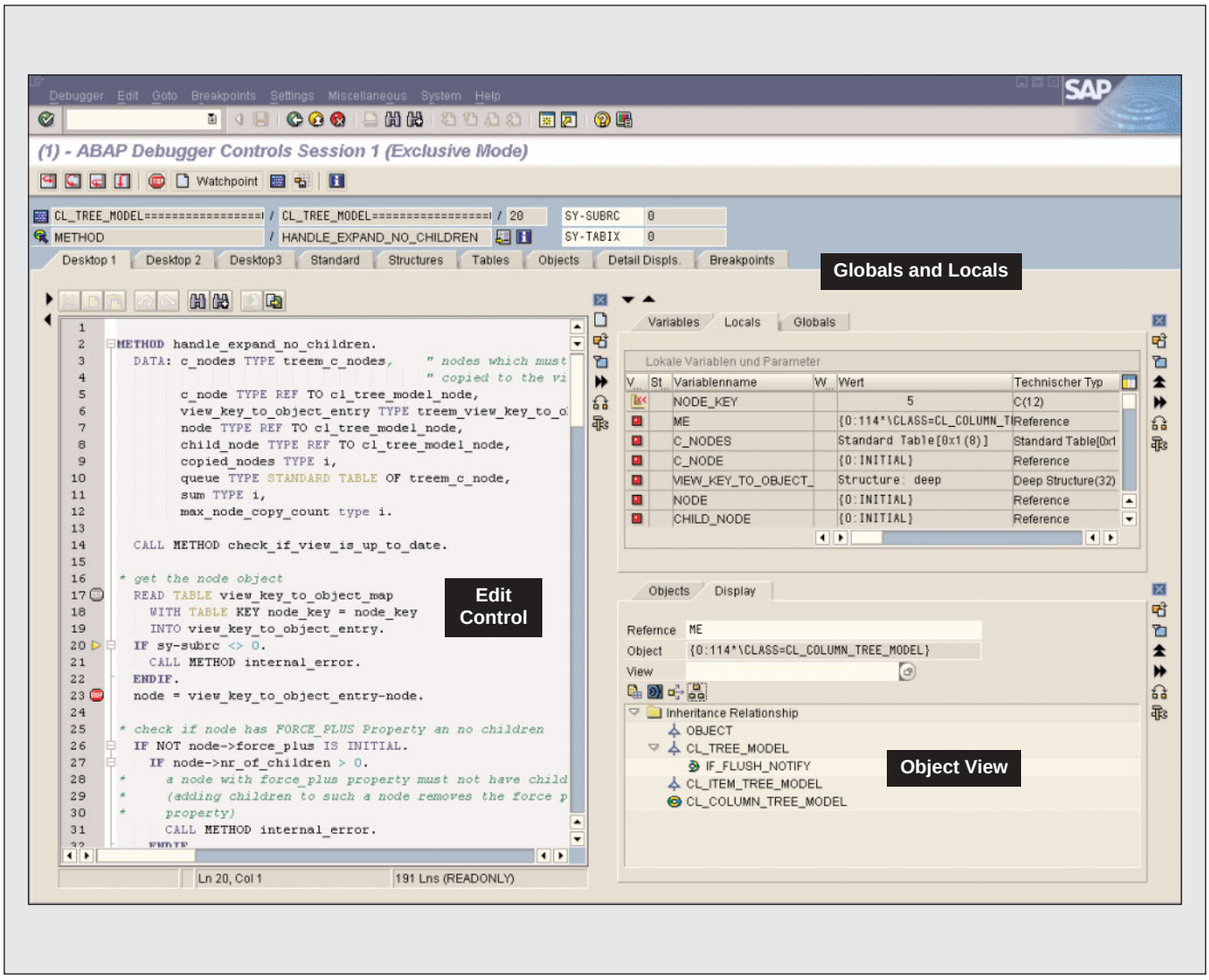
Conclusion

With this article on how to get the most from the ABAP Debugger, you now have a complete survival guide for troubleshooting your ABAP programs during development and testing, and in production. The scenarios and troubleshooting techniques offered in this article series originate from our daily experiences in ABAP development support at SAP. They are essentially our "best-of" collection for saving precious time during ABAP troubleshooting sessions.

You might recall the proven strategies for troubleshooting ABAP in a production environment that I provided in the previous article. I explained which tool is appropriate for which situation, and why the debugger shouldn't be your default tool of choice. But you will at some point encounter a complicated

[18] The Object view visible in Figure 34, which is already available in NetWeaver '04, displays all relevant information for a class or an object, such as the inheritance relationship, as shown in this example.

Figure 34 Upcoming Enhancements to the New ABAP Debugger



troubleshooting scenario where you have already used an ABAP trace or ABAP Dump Analysis. When only the debugger can help you obtain the last critical details, use it with the confidence and expert knowledge that will reduce your time and effort to a minimum:

- ☑ Start the debugger appropriately for the situation, including for transactions, pop-up windows, and batch jobs.
- ☑ Use the different breakpoint types effectively.
- ☑ Leverage watchpoints in order to find where a specific variable is changed or filled.
- ☑ Use little-known features such as debugging a batch job via the *jdbg* command or setting watchpoints on table headers in order to watch internal tables.

But there is more. It is beyond the scope (and space) of this article to provide detailed information about all the settings and system areas in the ABAP Debugger. However, it is important that you are aware of the opportunities, and even more important that you understand when you can benefit from these features. Visit the "Download Files" page at **www.SAPpro.com** for an introduction to these debugger settings and essential system areas that will help you on your way to becoming an ABAP Debugger expert yourself.

Boris Gebhardt studied physics at the University of Erlangen-Nürnberg, Germany. He joined SAP AG in 1998, where he currently works in the ABAP QM group. Boris is responsible for customer support and SAP internal consulting for the ABAP programming language and its surrounding tools. He is also involved in the development of ABAP tools, including the New ABAP Debugger, and was engaged in a development project for the public sector. Boris can be reached at boris.gebhardt@sap.com.

