

An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP Investigative Tools for Production Problems

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 68)

In the first installment of this troubleshooting series,¹ we explored a variety of tools and techniques that can be used during development and testing to ensure that you are building high-quality ABAP applications before they ever reach your production systems. But in the real world, even with the most careful testing, sooner or later you will receive a complaint about bad performance, unexpected behavior, or even runtime errors in your production programs. Of course, production programs must be fixed as soon as possible, to avoid adversely affecting your critical business processes. Fortunately, SAP provides standard logging, analysis, trace, and debugging tools for identifying problems in your production programs.

Without expert knowledge of the appropriate use of these troubleshooting tools, however, bug hunting could cost you many extra hours of effort or even lost weekends. For example, while the ABAP Debugger is an invaluable tool in the right situation, when used in the wrong situation, it can turn an analysis that should take minutes into one that consumes days or even weeks — productive time you cannot afford to lose. This article focuses on when to use which tool for troubleshooting production problems, so you no longer need to resort to the ABAP Debugger every time. This knowledge is especially valuable for ABAP developers, but system administrators can also benefit from it, by learning how to determine which program is responsible for a terminated job, for example. I will also share some undocumented tricks to boost your efficiency and help you leverage the full power of the SAP-provided tools for troubleshooting your ABAP programs.

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*.

Scenario 1: Comparing Program Flow on Different Systems

Let's say your application runs on both a quality assurance system and a production system. While everything works fine on the quality assurance system, the application produces an error message on the production system, even though the programs on both systems are apparently identical.

Your challenge is to find the difference in the program flow between the two systems, which must be the cause of the error in the production system. Your first thought is to use the ABAP Debugger to check each statement that is executed, so you run the application in debug mode on both systems and step through the application in parallel. If you are unlucky, you can lose weeks running the debugger with a complex application before you find even the first difference.

Choosing the Right Tool: Analyzing ABAP Programs in Production Systems

When you are troubleshooting a production application, efficiency and speed are of primary importance. And using the right tool for the error situation is absolutely essential for troubleshooting efficiency. Imagine yourself in the scenario described in the sidebar above, where you need to find the difference in program flow between two systems.

As you will soon discover, the ABAP Debugger is not your only option, and in some cases, like the scenario described above, it is also not the best option. There is a better approach to solving the problem in this scenario. You could use ABAP Runtime Analysis (transaction *SE30*) as an ABAP trace tool, which will provide the complete program flow of the application (all call functions, call methods, message statements, etc.). Simply run the trace on both systems, download each trace to a file, and then use a standard file comparison tool (such as the Unix *diff* command) to compare the two traces. You will have a list of differences within minutes, instead of potentially days or even weeks with the debugger. (I discuss this trace functionality in more detail later in the article.)

This scenario is only one example of the debugger not always being the best tool for starting to troubleshoot an application in production. To give you an

idea of the possibilities, **Figure 1** introduces some of the most important tools and the appropriate error situations for using them.

In some situations, one tool is not sufficient, and you actually need to use several to solve the issue. But how do you select the optimal combination in a real-world troubleshooting session?² To answer this question, this article outlines the following scenarios to illustrate typical situations that I have encountered and successfully resolved without using the ABAP Debugger³:

- Using the system log and ABAP Dump Analysis to determine why a background job terminated.
- Using ABAP Runtime Analysis as an ABAP trace tool to identify where in the source code a specific message is launched. Since past *SAP Professional Journal* articles on performance tuning have covered the SQL Trace tool in detail,⁴ here I will focus on the lesser-known trace functionality of ABAP Runtime Analysis.

² This article focuses on how to approach the troubleshooting process and how the various tools apply. For a detailed description of how to use these tools, refer to the online help for each tool.

³ For new and better ways of using the ABAP Debugger, watch for an upcoming *SAP Professional Journal* article.

⁴ "Optimize Database Access and Increase System Performance Through More Efficient ABAP Programming" (May/June 2002) and "Performance Problems in ABAP Programs: How to Find Them" (May/June 2003).

Figure 1 *Tools for Troubleshooting ABAP Programs in Production*

Problem Type	Appropriate Tool
Starting point for analyzing a crashed program or background job	System Log (SM21)
Runtime error	ABAP Dump Analysis (ST22)
Unexpected program behavior (such as the display of an unexpected message) that requires program flow analysis	ABAP Runtime Analysis (SE30)
Finding a specific statement, function, method, etc.	ABAP Runtime Analysis (SE30)
Performance problem	ABAP Runtime Analysis (SE30) or SQL Trace (ST05)
Problem with batch input or screen processing (including F1 and F4 help)	Screen Trace (ST20)*
Detailed error analysis at the source code level to determine the value of a specific variable or other object	ABAP Debugger
* The Screen Trace tool is available starting with SAP Basis Release 4.6B. This topic is large enough to warrant its own article and is thus beyond the scope of this article series. Refer to SAP Note 365940 for more information.	

Post-Mortem Analysis of a Terminated ABAP Program

Let's begin our troubleshooting journey with the scenario described in the sidebar below, in which

we face the challenge of analyzing the cause of a terminated background job.

Where do we begin our analysis of this terminated job? We could run the ABAP Debugger, but when we

Scenario 2: Investigating a Terminated Background Job

Suppose a batch job that you run every day terminated unexpectedly yesterday. As an experienced system administrator or ABAP developer, you first check the job log by pressing the Job Log button in transaction SM37 (Job Overview), which displays the following information:

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 ZTSTECHE02_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

From the job log, you learn that the ABAP program ZTSTECHE02_SUBMIT_JOB was started at 15:00:33, and that sometime afterward Abend message 208(00) was launched. The last three columns in the job log identify the message: the message class (00), the message number (208), and the message type (A for Abend). You can view these messages in transaction SE91 (Message Maintenance).

While it's clear that this Abend message canceled the job, the message text is simply "Error," which will not help you find the cause of the error.

run the program directly in dialog mode, either via transaction *SA38* (ABAP: Execute Program) or transaction *SE38* (ABAP Editor), we quickly find that the error does not occur. The program only crashes when it runs in the background, so the debugger is not going to help in this situation.

In addition, using the debugger is only a useful approach if you already have a reasonably good understanding of the problem. You at least need to know which program the root error occurs in before the debugger can help you analyze the problem further. In this scenario, we only know that an error occurred.

First, we need a tool that gives us a general understanding of the error situation. We need to determine the source of this error message, and in particular, where the root error occurred. For more detailed analysis, we can then switch to a specialized tool like the ABAP trace or the debugger to investigate the problem at the source code level.

So, we begin our analysis with the system log (transaction *SM21*), which will provide a rough chronological overview of the error situation.

Using the System Log to Understand the Error

The system log is a specialized file that is located on the application server of an SAP R/3 system. Every generated error or warning on a running system is written automatically to this file, including termination or error messages from an ABAP program, runtime errors, or even system crash errors at

the SAP Web Application Server (SAP Web AS) kernel or operating system level. For each entry, the system log shows the timestamp, the work process number, the work process type, and user/transaction information. Refer to the online documentation for additional information about the log structure and customization options.

You can use system log files to better understand errors that happen in a given time frame. To access the system log, enter transaction code *SM21* or choose *Tools → Administration → Monitor → System Log* from the SAP Easy Access Menu.

Let's take a look at how we can use the system log to gain a better understanding of our example terminated background job.

Using the System Log to Investigate the Example Terminated Background Job

Remember from the sidebar on the previous page that we first checked the job log via transaction *SM37* (Job Overview) on the server where the job was canceled to determine the job start and end times. We learned that the Abend message that canceled the job happened at 15:00:45 (see **Figure 2**).

Next, we display the system log on this server, limiting the output to the job runtime, as shown in **Figure 3** (the relevant system log entries are marked). Analyzing the system log reveals the entry labeled ❶, which is the same event recorded in the job log, occurring at the same time. From this entry, we learn that Abend message *00 208* happened in batch work process 10. But it is much more interesting to see

Figure 2 The Job Log for the Terminated Background 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 ZTSTECH002_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

Tips for Using the System Log

When using the system log, remember the following essentials and possible pitfalls:

- ✓ The system log is a local file on each application server. Make sure that you analyze the system log on the server where the error occurred, or you will not find the relevant error log entries. Especially with background jobs, Remote Function Calls (RFCs), or asynchronous update problems, it is not always obvious on which server the error occurred. To ensure that you get information about the error, you can use either the central system log or remote system logs. Both options are available from the system log by following menu path *System Log* → *Choose* and then choosing either *Central system log* or *All remote system logs*.
- ✓ The system log provides a chronological list of errors and warnings for each application server. You can readily link trace entries to the error you want to analyze, because these trace entries show either all timestamps within a given time frame or all entries for your user account. By analyzing the trace entries, you can easily find which error occurred first, and then separate it from follow-up errors that are irrelevant (such as subsequent errors that occur during error handling).
- ✓ To avoid getting lost in a potentially long list of system log entries, you can limit the amount of information displayed when analyzing the log. For example, you can select only the log entries within the time frame in which the error occurred. Keep in mind, however, that it may not be wise to limit the log entries to those associated with your user account, because an RFC could be running under another user account (such as the default RFC user), for example.
- ✓ The system log file is circular. When the maximum file size* is reached, the system starts writing from the beginning of the file again, overwriting the first section. Therefore, take care to analyze the system log as soon as possible after an error occurs.

* The system administrator uses the profile parameter `rslg/max_diskspace/local` to set the maximum file size for the system log. See SAP Notes 548624 and 4063 for more information.

Figure 3 The System Log Output for the Specified Job Runtime

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		A00	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 whe
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

that this message is really only a follow-up error. Prior to this entry, the same user (*GEBHARDT*) received runtime error *TABLE_INVALID_INDEX* in dialog work process 0, as indicated by the entry labeled ❷. (See the note below for details on how the system log information is presented).

✓ *Interpreting the System Log*

- **Time:** Timestamp of the log entry.
- **Ty.:** Type of work process where the log entries were written: *BTC* (batch), *DIA* (dialog), etc.
- **Nr:** Number of the work process where the log entries were written. Use transaction *SM50* (Process Overview) to view the work process.
- **Cl.:** ID of the client in which the transaction was running.
- **Tcod:** The currently running transaction code.
- **MNo:** System log message number.
- **Text:** Long text of the system log message.

The times of system log entries ❶ and ❷ fall within the start and Abend times of our background job, which are 15:00:33 and 15:00:45 (refer back to Figure 2). We must conclude that our batch job started a dialog process (number 0) by calling an RFC module. This dialog process crashed with the runtime error *TABLE_INVALID_INDEX*. As a direct consequence, and at the same time, the Abend message was launched in the background process.

Our next step is to analyze the runtime error *TABLE_INVALID_INDEX* using ABAP Dump Analysis (transaction *ST22*), because we now know that it was the root cause of our example terminated background job.

Using ABAP Dump Analysis to Analyze Runtime Errors

Since we know from the system log that the root error of the terminated job was the runtime error *TABLE_INVALID_INDEX*, the next step in troubleshooting this problem is to determine why the runtime error happened. For this task, we turn to the ABAP Dump Analysis tool (transaction *ST22*).

One of the many advantages of ABAP is the detailed information that is written to the database after a runtime error occurs, known as the “ABAP dump.” You can display this information with ABAP Dump Analysis to review details about the error, the running transaction, and the error environment.

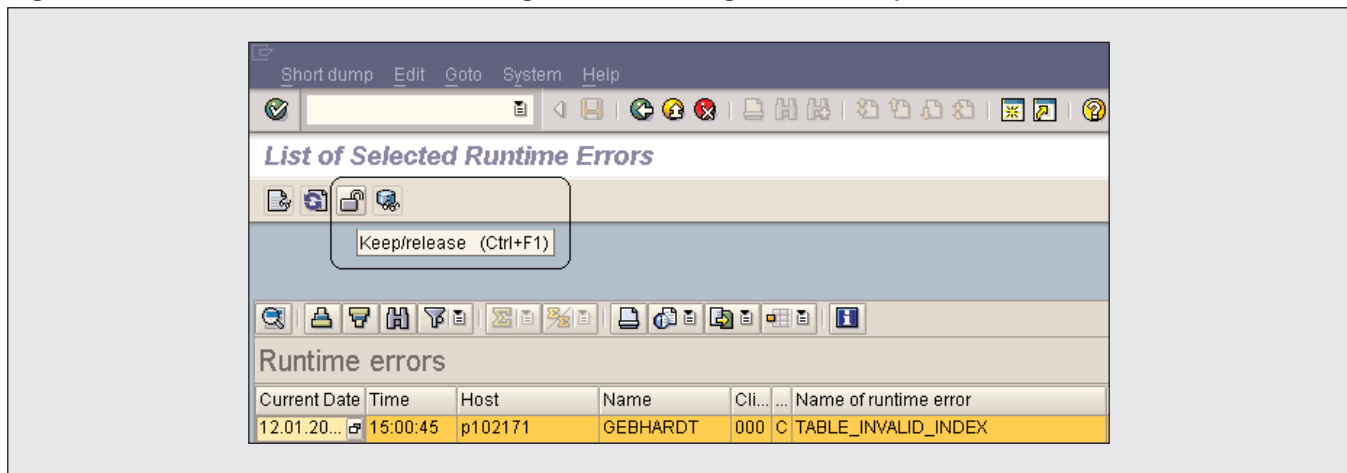
In my experience, simply reading the ABAP dump is often enough to find the bug in an ABAP program. In fact, sometimes the ABAP dump is all you have to investigate a problem — when the error is hard to reproduce or occurs only in a background job after several hours, for example. Therefore, you must know exactly what information you can extract from an ABAP dump and how to use these clues to solve the problem.

If a runtime error occurs in a dialog process, the system presents a screen that displays the ABAP dump as a list. Because this dump is also stored in the database, you can analyze it with ABAP Dump Analysis whenever you want. Keep in mind, however, that system administrators usually schedule a standard batch job that periodically runs the report *RSSNAPDL*. This batch job deletes all ABAP dumps that are older than one week.⁵ If you want to save a dump for later analysis, you must lock it in the ABAP Dump Analysis tool in order to protect it from automatic deletion. As shown in **Figure 4**, press the *Keep/release* button to lock the dump. The dump will then remain in the database until you unlock it using the same button.

⁵ Refer to SAP Note 11838 for more information about the deletion and reorganization of the ABAP dump database table.

Figure 4

Locking and Unlocking ABAP Dumps



✓ Tip

When you receive a live ABAP dump (that is, your running program terminates with an ABAP dump), you can easily switch from the dump list to the ABAP Debugger. Simply press the debugger button in the toolbar of the ABAP dump screen.⁶ After switching to the debugger, you can use its full power to check the content of internal tables or objects, or assess local variables in different stack frames. Keep in mind that this opportunity is gone once you leave the live dump — you cannot jump into the debugger from an ABAP dump in ABAP Dump Analysis. Only the ABAP dump stored in the database is available, not the ABAP runtime environment itself.

Now that you have a solid understanding of how the ABAP dump can help you determine why errors occur, our next step is to analyze the runtime error `TABLE_INVALID_INDEX`, which we identified as the root cause of our terminated batch job. But before we begin analyzing the details of the ABAP dump, in the next sections, I will provide you with guidelines for extracting as much information as possible from an ABAP dump and for navigating the dump itself.

⁶ You must have debug authorization to take advantage of this option. Refer to SAP Note 65968 for details.

Determining the Frequency and Location of a Runtime Error Using ABAP Dump Analysis

Taking a few minutes to consider some general characteristics about the error situation is especially helpful if the error occurs frequently and the root cause is unknown. You can use ABAP Dump Analysis for this purpose by entering transaction code `ST22` or choosing *Tools* → *Administration* → *Monitor* → *Dump Analysis* from the SAP Easy Access Menu. This tool can help you answer the following questions:

- **When did the dump first occur?** Perhaps you will find that a system error (such as `SYSTEM_CORE_DUMPED`) occurred immediately before the dump for your program. In this case, your dumps are probably only follow-up errors and not the root cause of the error.
- **Is only one user or program affected, or many?** If only one user is affected, perhaps individual user settings are affecting the program execution. For example, this user may have set parameter IDs that influence application behavior. You can view these parameter IDs in transaction `SU01` (User Maintenance). If you cannot reproduce the error with other user accounts, you may need to use this particular user account to reproduce and analyze the error. If only one program is affected, you may use the `RSDEPEND` report to find out which part of the program was changed recently (see the sidebar on the next page).

Need Even More Detail? How to Investigate Recent Program Changes

Sometimes you need to know which program parts, such as includes and data dictionary (DDIC) structures, were changed recently. This knowledge can help you identify inconsistencies between programs or between a program and a DDIC structure, or understand why a program that you did not change during the last year suddenly starts terminating with runtime errors.

The RSDEPEND report, which you start directly via the ABAP Editor, provides a list of all program components, including DDIC elements, includes, and screens. The report header shows the overall timestamp, which reflects the last change made to any element of the program. For each component, the report displays the date and time of the last change. The screenshot below shows an example of this report for the sample program SAPMSJOB.

For tables (DDIC elements), you see both the last direct modification and the last ABAP update. If you add a column to a DDIC structure and activate the structure, the system checks for the ABAP programs that use it. These programs will be “touched,” which means their overall timestamp is updated as indicated by the timestamp in the ABAP update column. This “touch” (the updated overall timestamp) leads to an automatic regeneration of the program during its next use. This synchronization process ensures that the executable of this ABAP program is always up to date and includes any recent DDIC structure changes.

Display Dependent Objects			
Program SAPMSJOB			
Over all timestamps.....	03.09.2003	141837	
Main program changed.....	05.02.2001	12:49:06	
By.....	SAP		
Generated..... 12.09.2003 09:31:29 for hardware 560 (00000230)			
Include...	Last modification...		
<SYSINI>	04.02.2002 21:03:33		
MSJOB01	02.03.1998 12:48:13		
MSJOB101	03.09.2003 14:18:37		
MSJOB001	07.07.1998 16:35:46		
MSJOBTOP	07.07.1998 16:35:46		
Table.....	Last modification...	ABAP update.....	Screen update.....
ARC_PARAMS	01.02.2001 10:39:45	08.08.1997 14:45:20	02.02.2001 09:25:10
DYNPREAD	04.02.2002 20:32:44	04.02.2002 20:50:12	04.02.2002 20:50:12
PRI_PARAMS	07.11.2000 15:57:57	27.10.1998 15:01:35	07.11.2000 15:57:57
RALDB	07.11.2000 15:57:57	16.12.1996 19:30:46	07.11.2000 15:57:57
RSJOBINFO	01.02.2001 10:29:32	09.02.1995 13:25:17	02.02.2001 09:37:56
SCREEN	04.02.2002 20:18:04	04.02.2002 20:50:54	04.02.2002 20:50:54
SYST	04.02.2002 20:19:15	02.02.2001 09:44:49	09.10.2001 18:34:22
TBTCJOB	04.02.2002 20:19:22	02.02.2001 09:45:41	02.02.2001 09:45:41
TBTCO	01.02.2001 11:09:33	02.02.2001 10:05:50	02.02.2001 10:05:50
TRDIR	04.02.2002 20:30:19	05.10.1998 15:24:19	04.02.2002 20:54:09
Screen.....	Last load update...		
SAPMSJOB	0100 25.01.1999 16:56:03		
SAPMSJOB	0101 15.05.1996 13:41:15		
SAPMSJOB	0200 25.09.1997 21:47:14		

Some DDIC changes (such as a change to the table documentation) are not relevant to the dependent ABAP programs. In these cases, the Last modification timestamp is updated, but the update is not transferred to the dependent ABAP programs (in other words, the system does not “touch” the program), so the ABAP update timestamp remains unchanged.

- **Does the runtime error occur on only one server, even though the application runs on all servers?** This could indicate buffer problems on the server, which means you may need to refresh the buffer, by restarting the server, for example.

For an overview of your overall system situation and which ABAP dumps occurred in a specific time frame, follow the menu path *Goto* → *Overview* in ABAP Dump Analysis. **Figure 5** shows an example of the resulting display.

Figure 5 Overview of ABAP Runtime Errors

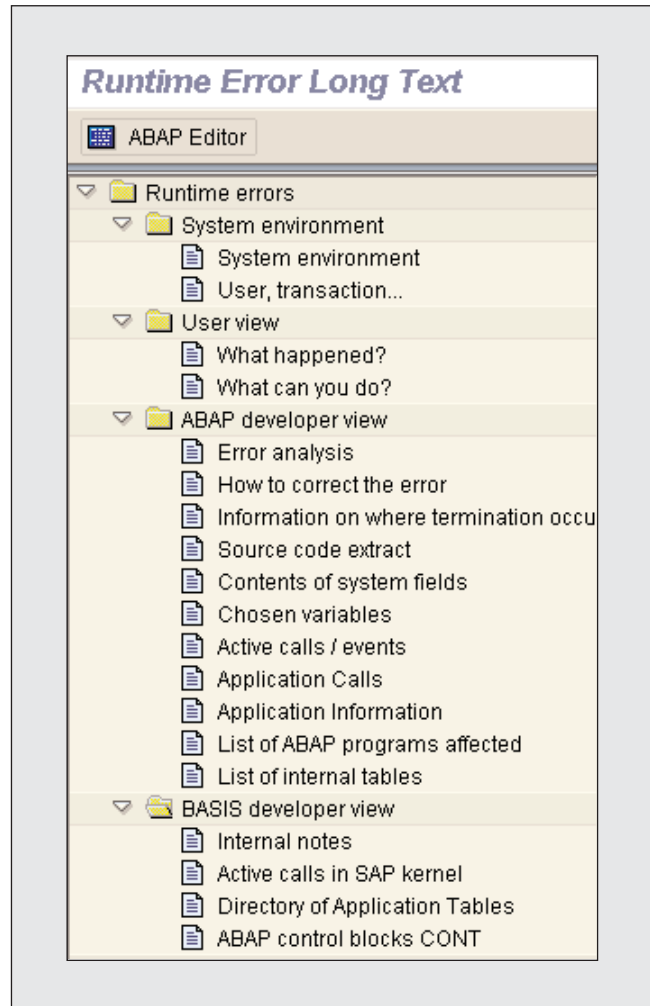
ABAP runtime errors		
Error ID	Number	Typ
LOAD_PROGRAM_LOST	102	S
SYSTEM_CORE_DUMPED	36	S
DDIC_TYPE_INCONSISTENCY	6	S
DDIC_TYPES_INCONSISTENT	2	S
ROLL_IN_ERROR	2	S
DDIC_TYPE_REF_ACCESS_ERROR	1	S
GEN_RSYN_ILLEGAL_KIND	1	S
TYPELOAD_LOST	1	S
SYSTEM_CANCELED	4	N
TSV_TNEW_PAGE_ALLOC_FAILED	15	C
RAISE_EXCEPTION	30	A
UC_OBJECTS_NOT_CONVERTIBLE	18	A
MESSAGE_TYPE_X	17	A
CONVT_NO_NUMBER	15	A

Note that the dumps are grouped by type to simplify analysis (see the note below for details on the type meanings). You can easily distinguish system errors (such as a signal from the operating system that leads to a *SYSTEM_CORE_DUMPED* error) from ABAP programming errors or other resource errors that may indicate a lack of memory.

✓ ABAP Dump Types

- **A:** ABAP programming error
- **C:** Capacity errors (lack of memory, due to incorrect system settings, for example)
- **N:** No error
- **S:** Internal error in ABAP or screen processing runtime
- **I:** Installation error
- **E:** External error
- **D:** Database error

Figure 6 Navigation Tree for an ABAP Dump



Navigating the Runtime Error ABAP Dump

The text of a runtime error ABAP dump, which is also referred to as the “long text,” consists of many pages. For ease of navigation, the dump is organized into folders such as *System environment*, *User view*, and so on, with nested sections. To navigate between sections, you can use either the navigation tree shown in **Figure 6** (which is available in SAP Basis⁷ Release 4.6D or later) or the *Overview* button in the application toolbar.

⁷ In Release 6.10, “SAP Basis” was renamed “SAP Web Application Server.”

Figure 7

ABAP Dump for TABLE_INVALID_INDEX

Runtime Errors	
TABLE_INVALID_INDEX	Occurred on 12.01.2004 at 15:00:45
Error in ABAP statement when processing an internal table.	
What happened?	
Error in ABAP application program.	
The current ABAP program "ZTSTECHE02_ST22 " had to be terminated because one of the statements could not be executed.	
This is probably due to an error in the ABAP program.	
What can you do?	
Print out the error message (using the "Print" function) and make a note of the actions and input that caused the error.	
To resolve the problem, contact your SAP system administrator. You can use transaction ST22 (ABAP Dump Analysis) to view and administer termination messages, especially those beyond their normal deletion date.	
is especially useful if you want to keep a particular message.	
Error analysis	
When changing or deleting one or more lines of the internal table "\PROGRAM=ZTSTECHE02_ST22\DATA=ITAB2[]" or when inserting in the table "\PROGRAM=ZTSTECHE02_ST22\DATA=ITAB2[]", 0 was used as the line index. An index less than or equal to zero is not allowed.	

The most relevant sections for troubleshooting ABAP are in the *ABAP developer view* folder. Here, you learn about the error environment from an ABAP perspective. The *BASIS developer view* folder contains SAP internal information about the kernel, the C stack, and so on. In most cases, this internal ABAP interpreter information is not directly relevant for an ABAP developer.

When you view an ABAP dump with ABAP Dump Analysis (transaction ST22), you see a display similar to the one shown in **Figure 7**. The ABAP dump list appears on the right, with sections separated by highlighted headers (such as *What happened?*). The navigation tree, which will lead you directly to the individual sections of the dump, appears on the left.

To demonstrate how to use the information in the *ABAP developer view* folder, we continue our troubleshooting scenario of investigating a terminated background job. We will analyze the main sections of the ABAP dump shown in Figure 7 in order to understand at the source code level why the error *TABLE_INVALID_INDEX* occurred.

Using ABAP Dump Analysis to Analyze the Example Terminated Background Job

To continue our investigation, we now closely examine each section of the dump to extract as much information as possible about the error.

Figure 8

The “Error analysis” Section

Error analysis

When changing or deleting one or more lines of the internal table
 "\PROGRAM=ZTSTECHE02_ST22\DATA=ITAB2[]" or when inserting in the table
 "\PROGRAM=ZTSTECHE02_ST22\DATA=ITAB2[]", 0 was used as
 the line index. An index less than or equal to zero is not
 allowed.

Figure 9

The “Information on where termination occurred” Section

Information on where termination occurred

The termination occurred in the ABAP program "ZTSTECHE02_ST22 " in
 "START-OF-SELECTION".
 The main program was "ZTSTECHE02_ST22 ".

The termination occurred in line 30 of the source code of the (Include)
 program "ZTSTECHE02_ST22 "
 of the source code of program "ZTSTECHE02_ST22 " (when calling the editor
 300).

Step 1: Identify the Source of the Error

We begin our analysis with the *Error analysis* section of the ABAP dump list (**Figure 8**), which provides the first clues to the source of the error. From the displayed text, in particular the extract `\PROGRAM=ZTSTECHE02_ST22\DATA=ITAB2[`, we learn that an internal table operation in the program `ZTSTECHE02_ST22` was executed on table `ITAB2` with an index of zero. This operation is not allowed, because the index must always be greater than or equal to one.

Step 2: Locate the Error

Next we move on to the *Information on where termination occurred* section (**Figure 9**). This section provides detailed information about the program and the specific source code line where the error occurred. Because of the many program and include names, this

section can be difficult to understand. Use the following guide to interpret its contents:

- The termination occurred in the ABAP program "`<program name>`" in "`<name of the event/function/form/method>`".
- The main program was "`<name of the program that is the header of the program group>`".⁸
- The termination occurred in line `<line in include/program>` of the source code of the (Include) program "`<include/program name>`" of the source code of program "`<program name>`" (when calling the editor 300).

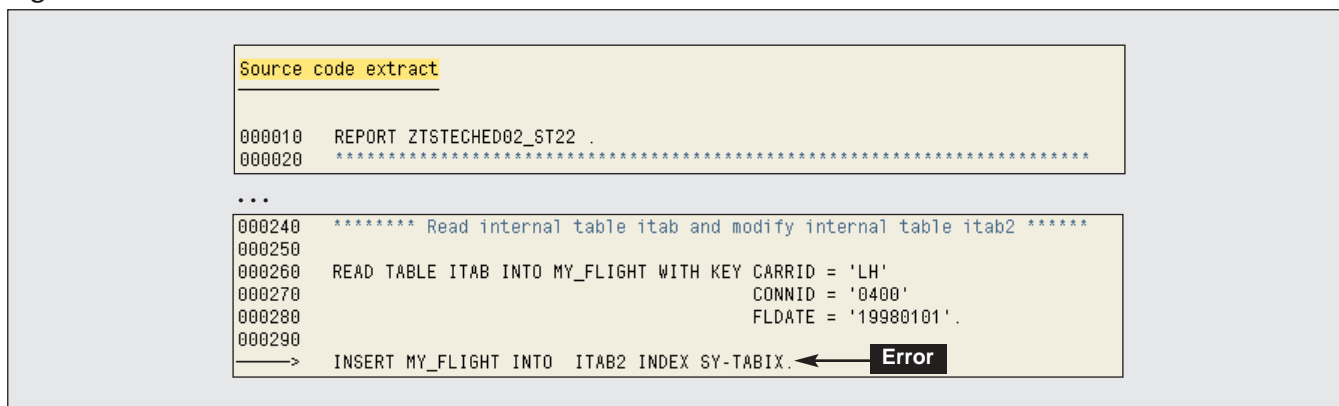
Step 3: Check the Relevant Source Code

Directly below this program/include information, you

⁸ For more details about program groups, search for “Modularization Units” in the ABAP keyword documentation.

Figure 10

The "Source code extract" Section



find the *Source code extract* section (Figure 10). It provides an extract of the relevant source code, indicating the specific line of code where the error occurred. Keep in mind that the affected source line may actually be located before the indicated line if the error happened after the program line pointer was incremented.

Here, we learn that the error occurred at line 30, and that the ABAP dump happened during the *INSERT* of a line into an internal table. In a previous statement (line 26), we also see that the program reads an entry from the internal table *ITAB*, and then tries to insert this line at the same position in the *ITAB2* table (line 30). The index used for the *INSERT* statement is the system variable *SY-TABIX*. (After a successful *READ* of an internal table, the system always inserts the index of the found line into *SY-TABIX*.)

Step 4: Examine the Values of Relevant System Fields

We next need to determine the value of *SY-TABIX*, which the program uses as an index for the *INSERT* statement that crashed (line 30 in Figure 10), so we turn to the *Content of system fields* section (Figure 11). This section contains the values of commonly used system (or *SY*-) fields, which are components of the system structure *SYST*.⁹ As you can see, the program executed the crashing *INSERT* statement with a value of 0 for *SY-TABIX*, which is why the error

Figure 11 The "Content of system fields" Section

SY field contents.....		SY field contents	
SY-SUBRC	4	SY-INDEX	0
SY-TABIX	0	SY-DBCNT	0
SY-FDPOS	0	SY-LSIND	0
SY-PAGNO	0	SY-LINNO	1
SY-COLNO	1	SY-PFKEY	
SY-UCOMM		SY-TITLE	..
SY-MSGTY		SY-MSGID	
SY-MSGNO	000	SY-MSGV1	
SY-MSGV2		SY-MSGV3	
SY-MSGV4			

occurred. (The index for an internal table *INSERT* must always be greater than or equal to zero.)

In addition, we find that the return code of ABAP statement *SY-SUBRC* is 4, which indicates that the *READ* statement (line 26 in Figure 10) before the *INSERT* statement (line 30 in Figure 10) failed (*SY-SUBRC* is set to 4 if a *READ* of an internal table is not successful). This explanation would fit with the initial value of *SY-TABIX*, because the system sets *SY-TABIX* to 0 if the *READ* fails. But we can't be sure — perhaps *SY-SUBRC* was set to 4 because of the error in the *INSERT* operation. So we need to determine whether the *READ* statement (line 26) before the crashing *INSERT* statement (line 30) failed. The *READ* statement will fail, and thus will not fill the result set *MY_FLIGHT*, if *ITAB* is either empty or contains no entries that match the specified key values, which are the following for our example:

⁹ For more information, search for "ABAP System Fields" in the ABAP keyword documentation.

Figure 12 Internal Table ITAB in the “Chosen variables” Section

Chosen variables	
1 EVENT	START-OF-SELECTION
ZTSTECHE02_ST22	
ZTSTECHE02_ST22	30

ITAB[]	Table IT_7[10x80]
...	42A50000000000000000000050000000FFFF0000B000
...	0460000020007000A0000000000000FFFF40000400

... + 40

Table administration header

Figure 13 Result Set Structure MY_FLIGHT in the “Chosen variables” Section

ITAB2[]	Table IT_8[0x80] 0000000000000000000050000000FFFF00000000 0000000050008000000000000000FFFF40000400
... + 40	1000C220 00001E00
MY_FLIGHT	000000000000##### 2222233333333333300000000222222222222 0000000000000000000000000000C0000000000000 ##### 20000000000000000000000000000000000000 00000000000000000000C00000000000000000
... + 40	

```
CARRID = 'LH'
CONNID = '0400'
FLDATE = '19980101'.
```

Therefore we must check if *ITAB* is empty and if *MY_FLIGHT* is initialized in order to prove that the *READ* failed.

Step 5: Verify the Content of Fields Near the Error

The next section of the ABAP dump, *Chosen variables*, displays the content of fields that were found in the source code near where the error occurred. Starting with SAP Web AS 6.10, the system displays variables not only at the top stack level, but for all ABAP stack levels.

We start with the top stack level, which is stack level 1:

1	EVENT	START-OF-SELECTION
---	-------	--------------------

This stack level contains the *READ* statement that we want to analyze:

```

READ TABLE ITAB INTO MY_FLIGHT
  WITH KEY CARRID = 'LH'
          CONNID = '0400'
          FLDATE = '19980101'.

```

On stack level 1, we first see the internal table *ITAB* (see **Figure 12**). We learn that *ITAB* (which has the internal name *IT_7*) has 10 lines and that the line width is 80 bytes. (Note that the hexadecimal code that appears below the text *TABLE IT_7[10x80]* represents the table administration header, not the content of the internal table *ITAB*). We can accurately conclude that internal table *ITAB* is not empty.

Next, we search for *MY_FLIGHT* to check whether the *READ* statement filled this structure. Further on in the same section, and again on stack level 1, we locate the result set structure *MY_FLIGHT* (see **Figure 13**).

The dump displays the content of the structure in 40-character segments. (The second segment starts at offset 40.) Each segment consists of the character representation of the displayed part of the structure. Vertically, you also see the hexadecimal representation, which is essential for displaying non-character fields. So, the first 40 characters of the structure *MY_FLIGHT* are:

```

000000000000#####
222222333333333300000000222222222222
000000000000000000000000C0000000000000
MANDT... PRICE

```

The structure *MY_FLIGHT* is declared in the program *ZTSTCHED02_ST22* as:

```
DATA MY_FLIGHT TYPE SFLIGHT.
```

You can check the Data Dictionary (transaction *SE11*) to determine the fields in this structure:

- The first field of the structure *MY_FLIGHT* is *MANDT* (*Char 3*). This field is empty and contains three spaces (*hex 20*).
- In the middle, we find a field with the value ##### and the hexadecimal value *hex 0C*. It is a non-character field (in this case, *PRICE* is actually a packed number field). The field is initialized (its value is *hex 0*, and *C* stands for the + sign), but the hexadecimal values cannot be displayed as characters, which is why you see the # symbols.

If we check the entire structure, we can easily see that it is initialized. Therefore, we can at last reach a conclusion. Because the value of *SY-SUBRC* is 4 (the return code of the *READ* statement), and the work area (or result set) *MY_FLIGHT* of the *READ* statement is empty, the *READ* statement in line 26 was not successful. Therefore, the value of *SY-TABIX* is 0, which produced a runtime error when used as an index for the subsequent *INSERT* statement (line 30).

Step 6: Devise a Solution

The source of this runtime error is the very common mistake of forgetting to check *SY-SUBRC* after the *READ* statement. A correction would be:

```

READ TABLE ITAB INTO MY_FLIGHT
  WITH KEY CARRID = 'LH'
          CONNID = '0400'
          FLDATE = '19980101'.
IF SY-SUBRC = 0.

  INSERT MY_FLIGHT INTO ITAB2
    INDEX SY-TABIX.
ELSE.
  ... "Error handling
ENDIF

```

Only by analyzing both the system log and the ABAP dump were we able to detect the error at the source code level and devise a solution. A future article on the ABAP Debugger will show you a trick for debugging this terminated background job to prove unequivocally that the analysis is correct. But in this article, we now turn to the world of ABAP traces in order to understand how easily you can use a trace to analyze the structure of an ABAP program.

Analyzing Program Structure with the ABAP Trace Functionality

ABAP Runtime Analysis (transaction *SE30*) was first designed for analyzing the performance of ABAP applications.¹⁰ Many developers aren't as familiar with its equally valuable trace features. For simplicity, we use the term "ABAP trace" when referring to the ABAP trace functionality of ABAP Runtime Analysis.

To illustrate ABAP trace capabilities, we'll use a new scenario (see the sidebar on the next page) in which we try to find the source code line in the ABAP Editor where a specific message is launched.

¹⁰ For more information, see the article "Performance Problems in ABAP Programs: How to Find Them" (*SAP Professional Journal*, May/June 2003).

Scenario 3: Finding a MESSAGE Statement

Suppose you want to analyze the program flow of an application or find a specific statement in a large application. As an example, we will analyze the ABAP Editor and try to find the line in the source code where a specific message is launched. We run the ABAP Editor by entering transaction code SE38 or following the menu path *Tools* → *ABAP Workbench* → *Development* → *ABAP Editor*, and then try to display the program ABC.

As you can see in the screenshot to the right, our attempt to display the program produces the message *Program ABC does not exist* in the status bar. Clicking on this status bar message takes you to the message long text, which includes identifier information we will need later in the analysis. The identifier for this message is DS017 (the message class is DS, and the message number is 017).

In order to understand why this message is appearing, we need to find out which source line is launching it and which checks are executed before that event. Therefore, we will trace this behavior with an ABAP trace.

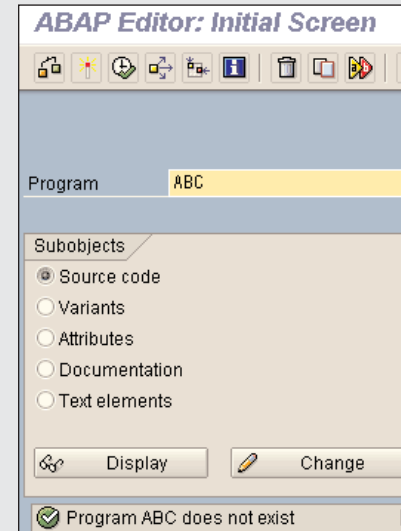


Figure 14 ABAP Statements That Are Eligible for Tracing

Purpose	Statements/Statement Groups
Database accesses	OPEN SQL, EXEC SQL
Modularization units	CALL FUNCTION, CALL METHOD, CALL SCREEN, SUBMIT, etc.
Internal table handling	APPEND, READ, INSERT, etc.
File handling	TRANSFER, OPEN DATASET, etc.
Other potentially time-consuming statements	EXPORT, IMPORT, MESSAGE, ASSIGN, COMMIT, ROLLBACK, etc.

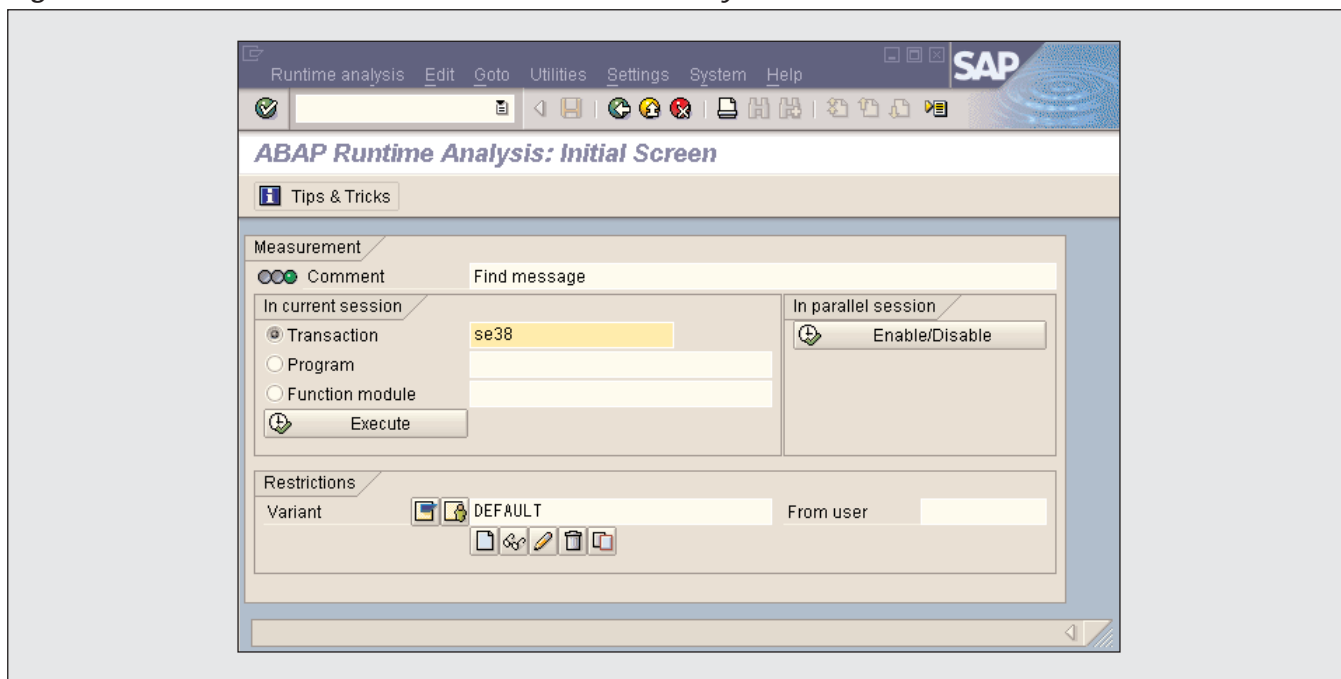
Using an ABAP Trace Effectively

The ABAP trace functionality of ABAP Runtime Analysis is your only option for tracing ABAP programs at the statement level. You typically want to perform an ABAP trace in order to understand the structure of a program, or to find out where in a program a specific function or statement is executed.

You can trace all potentially time-consuming statements in your ABAP programs. However, you will not find every *MOVE* or *IF* statement in the resulting ABAP trace list. Otherwise, the ABAP trace file would become too large and the trace overhead during program execution would be too high. **Figure 14** lists the ABAP statements that are included in the resulting trace list.

Figure 15

The ABAP Runtime Analysis Initial Screen



✓ Managing ABAP Trace Files

An ABAP trace is stored in a file that is available in the folder specified in the profile parameter `abap/atrapath`. You display and maintain profile parameters via transaction `RZ11` (Maintain Profile Parameters). The system administrator can specify the maximum size of all ABAP trace files in this folder with the profile parameter `abap/atrasizequota`. The default size is 30,000 KB. If you receive warning messages that the file quota is exceeded, delete some old files in the `ATRA` directory of the current application server.

But why would you consider tracing the ABAP code that is executed while an application is running? Here are some possible reasons:

- To analyze the program flow of an application in order to understand its structure.
- To find the location of a specific statement

(this is our example scenario, as described in the sidebar on the previous page).

- To compare the application flow on different clients or systems in order to find the differences between them (as described on page 44 in Scenario 1). If you suspect that the odd behavior of an application may be caused by extensions in customer functions, field exits, or conversion exits, search for customer functions (`EXIT_`), field exit functions (`FIELD_EXIT_`), or conversion exit functions (`CONVERSION_EXIT_`).
- To trace the memory consumption of an application. An ABAP trace also provides the currently allocated memory for each trace entry. You can easily download this list from the trace to a spreadsheet and generate a chart to quickly identify the major memory consumers.

You turn on and analyze an ABAP trace from ABAP Runtime Analysis (see **Figure 15**). We now continue with the troubleshooting scenario described on the previous page to illustrate how to run an ABAP trace to find a specific `MESSAGE` statement.

Figure 16

Create a Trace Variant

Variant	ABAPTRACE	From user	GEBHARDT
Description	ABAP Trace		
<div> <div>Program (parts)</div> <div>Statements</div> <div>Durath/type</div> </div>			
Max. size of file	2.000	KB	
Maximum runtime	1.800	Seconds	
<div>Aggregation</div> <div> <input type="radio"/> Full <input type="radio"/> By call <input checked="" type="radio"/> None </div>			
<input type="checkbox"/> With memory use (if aggregation not used)			

Figure 17

The "Performance file" Frame

Performance file	
Application	SE38
Short description	Find message
Measurement date	14.01.2004 12:40:37
File size in KB	27
<div> <div>Analyze</div> <div>Other file...</div> <div>File info...</div> <div>Delete...</div> </div>	

Locating a Source Code Statement with an ABAP Trace

Here, we want to use the ABAP trace functionality of ABAP Runtime Analysis to find the target *MESSAGE* statement in the ABAP Editor source code. To start ABAP Runtime Analysis, enter transaction code *SE30* or follow the menu path *Tools* → *ABAP Workbench* → *Test* → *Runtime Analysis*. In the initial screen (Figure 15), specify the transaction we want to trace — in this example, *SE38* (the ABAP Editor).


In order to use ABAP Runtime Analysis for tracing ABAP programs, create a trace variant to run the trace with no aggregation (as shown in **Figure 16**). This setting prevents the runtime analysis from aggregating the ABAP trace records associated with the source code line. Otherwise, instead of


receiving the full ABAP trace, you would only see a list of the most time-consuming ABAP statements without direct links to the source code. Checking the *With memory use* checkbox also provides the current memory use for each ABAP trace line, which is useful for identifying the major memory consumers in your program.

Now we step back to the ABAP Runtime Analysis initial screen (Figure 15). Press the *Execute* button to execute transaction *SE38* with the new trace variant and reproduce the message *Program ABC does not exist* (see the sidebar on page 57). Then step back once again to the runtime analysis initial screen, where you should now see the *Performance file* frame at the bottom of the screen (see **Figure 17**). If you cannot find your trace file, press the *Other file* button to display all available trace files on this server.

Figure 18

ABAP Trace Results

Runtime Analysis Evaluation: Call Hierarchy			
			
Gross	Net	Lv	Call Hierarchy
485.984	72	0	Runtime analysis
485.912	54.324	1	Call transaction SE38
431.588	1.024	2	Program RSABAPPROGRAM
	218	3	Load report RSABAPPROGRAM
73.708	69.823	3	Screen entry
160	106	4	Load screen SAPMSSY0
	54	5	Load report SAPMSSY0
1.893	170	4	PBO screen SAPMSSY0
	36	5	Load report SAPMSSYD
33	17	5	PERFORM IS_EMERGENCY
	16	6	CALL C_GUI_SUPPORT


To start the analysis of the ABAP trace file, press the *Analyze* button. On the next screen, Runtime Analysis Evaluation: Overview, press the hierarchy button () or press *F8* to display the results of the ABAP trace. As you can see in **Figure 18**, the ABAP trace (also referred to as the “call hierarchy”) for our application contains a long list of trace entries.

The essential columns for finding and analyzing the *MESSAGE* statement of interest are call level (*Lv*), call hierarchy, and on the right (not shown), the program where the trace line is located. If you turned on memory tracing in your trace variant, you would see the allocated memory in bytes (*Mem.Reqmt*) on the right edge (also not shown).

In order to find the *MESSAGE* statement, we press *CTRL+F* to search the list for *MESSAGE* and find the entry *Message S017* in the call hierarchy (see **Figure 19**). Remember from the sidebar on page 57, we are looking for message number 017, class DS — the message we found here in the call hierarchy (S017) has the same message number (017) and is a success message (S). The program associated with this trace line is *SAPLWBABAP* (not shown in Figure 19). If we check this program, we find the following statement:

```
FUNCTION-POOL wbabap MESSAGE-ID ds.
```

The *MESSAGE-ID ds* statement tells us that the default message class of this function group is *DS*, which matches the message we are looking for (message number 017, message class *DS*, message type *S*). We will later prove definitively that this *MESSAGE* statement uses this default message class.

We could jump directly to the responsible source code line for this message trace line using the *Editor* button () in the application toolbar of the runtime analysis call hierarchy screen. But first let's try to analyze the trace. **Figure 20** shows the relevant excerpt from the trace results with the contents of the *Lv* column (which shows the call level) and the *Call Hierarchy* column (which shows the trace line).

Call level 0 is always *Runtime Analysis*, because it is the starting point of the trace. *Runtime Analysis* then calls transaction *SE38*, which is the transaction we are tracing. Inside transaction *SE38*, the call level is incremented as various functions and forms are called. The call level is decremented when a called function or form is exited.

The *MESSAGE* statement is on call level 25. Because the previous entry on call level 24 is *PERFORM CHECK_DIRECTORY*, we conclude that the message is in the *CHECK_DIRECTORY* form.

Figure 19

MESSAGE Search Result

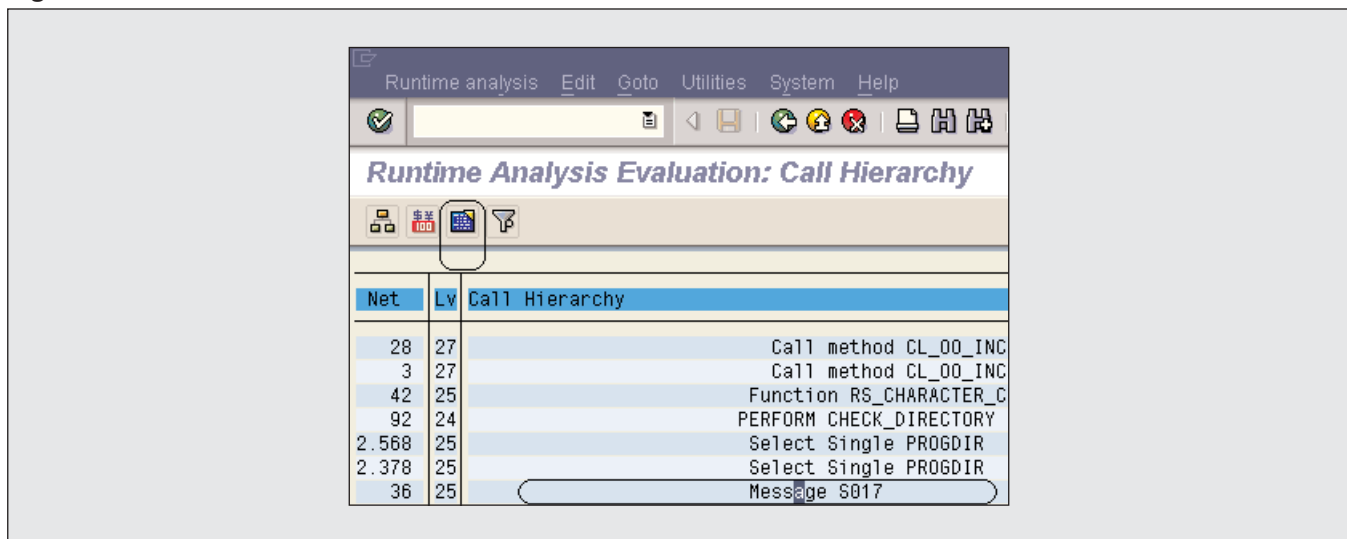


Figure 20

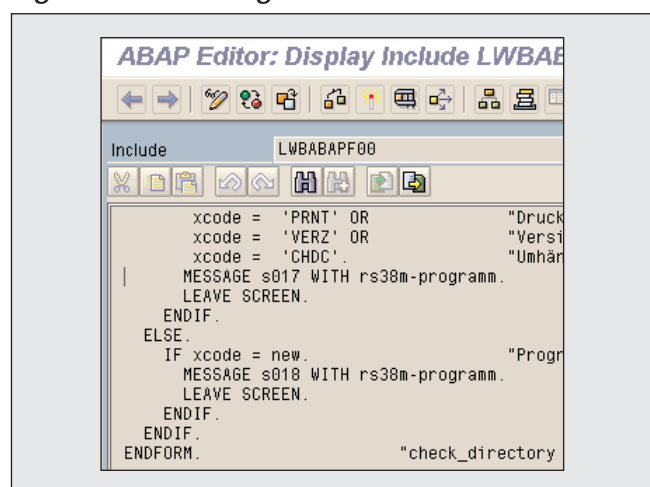
Analyzing the Example ABAP Trace

Call level	Trace line
0	Runtime Analysis
1	Call transaction se38
2	...
24	PERFORM CHECK_DIRECTORY
25	SELECT SINGLE PROGDIR
25	SELECT SINGLE PROGDIR
25	MESSAGE S017

We also see two *SELECT* statements on the database table *PROGDIR* on the same call level (25) as the message. The clues are all coming together. In the *CHECK_DIRECTORY* form, the transaction executes two selects on the *PROGDIR* table and then launches the message *DS017*.


If we use the *Editor* toolbar button (see Figure 19) to view the source code for the message statement, we now see **Figure 21**. Here we can confirm that the *MESSAGE* statement is in the *CHECK_DIRECTORY* form. We also see that no message class is specified, so the transaction does indeed use the default message class *DS* of the function group. Further analysis of the code would show that the program first tries to obtain

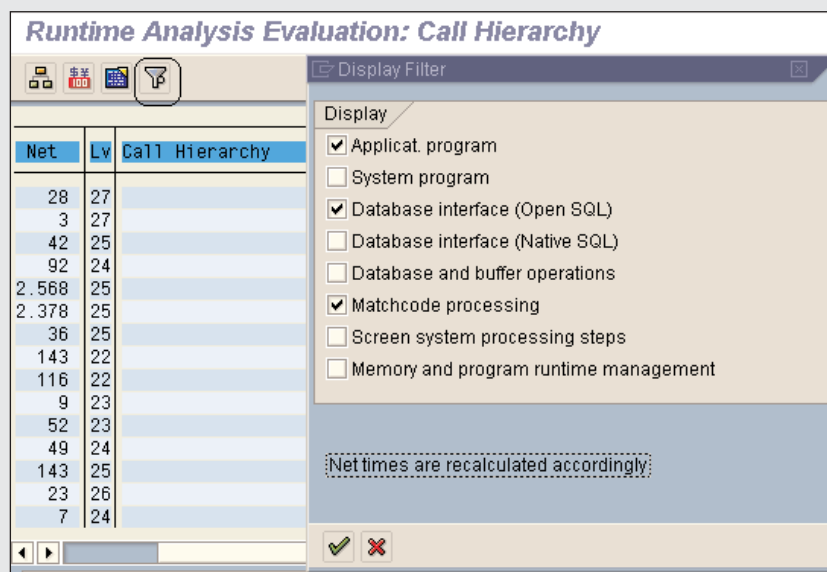
Figure 21 Message Statement Source Code



Tips for Using an ABAP Trace

Keep in mind the following possible pitfalls when using an ABAP trace:

- ✓ ABAP traces do not include ABAP code that is executed in RFCs or asynchronous updates. If you trace an application that calls RFC or update functions, this functionality will be missing from the ABAP trace.
- ✓ Traces are stored locally on the application server. Remember to analyze the ABAP trace on the same server where you performed the trace, or you will not find your ABAP trace file.
- ✓ If you cannot find particular statements in your ABAP trace, first check the trace variant maintenance screen to see whether you restricted your trace variant to specific statements or programs. If this is not the case, try turning on all Display Filter options (as shown in the screenshot below). Press the filter button () in the application toolbar of the runtime analysis call hierarchy screen to access the Display Filter pop-up. By default, some programs (such as system programs) are not displayed. For example, suppose you are looking for a message that is launched from a system program such as SAPMSSY0. If you do not check *System program* in the Display Filter pop-up, you will not find the target message in your ABAP trace.



the inactive version of the requested program from the *PROGDIR* database table. If that attempt fails, the program next tries to obtain the active version of the requested program. If this attempt also fails (in other words, the program does not exist), the transaction displays the message we were trying to find: *DS017*.

Using ABAP Runtime Analysis to Trace Programs Running in Parallel

Starting with SAP Basis Release 4.6B, you can also use an ABAP trace for applications running in a parallel session to the runtime analysis session. This feature is a valuable enhancement, because you can now

Figure 22 Starting an ABAP Trace for a Parallel Session

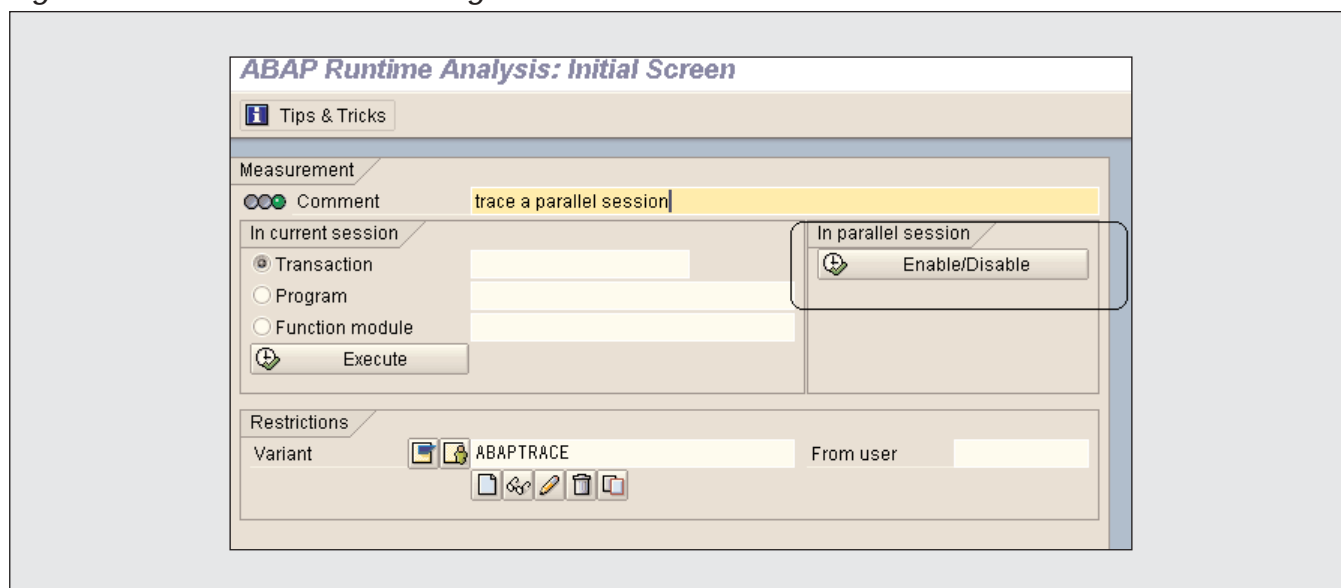
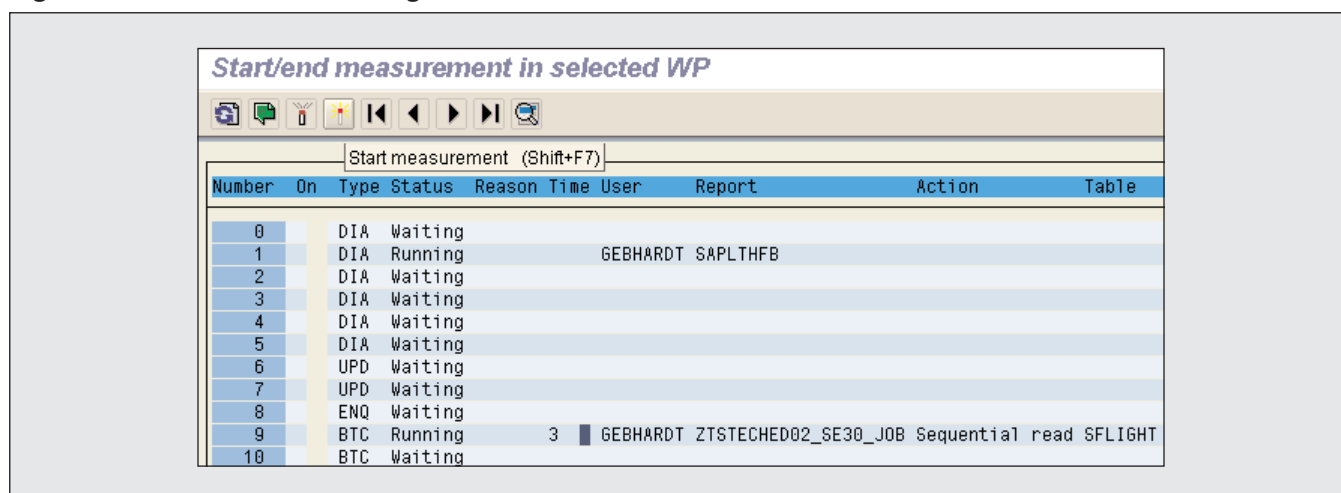


Figure 23 Turning on an ABAP Trace for a Parallel Work Process



turn on an ABAP trace for a running program, such as a background job that has already started. Previously, you could start a program or transaction with a trace turned on, but there was no way to turn on an ABAP trace for a program that was already running.

Suppose you find a job in transaction *SM50* (Process Overview) that has been running for days. Obviously you want to know what this job is doing. Is it still working or just hanging in an endless loop?

To find the answer, turn on an ABAP trace for this batch job and analyze the ABAP code that is currently executing in this background process. In the initial screen of ABAP Runtime Analysis, you normally fill in the *Transaction* or *Program* field in the *In current session* frame. Instead, press the *Enable/Disable* button in the *In parallel session* frame, as shown in **Figure 22**.

Next, you see the screen shown in **Figure 23**,

Tips for Using an ABAP Trace for a Parallel Session

Be aware of the following points when using an ABAP trace for a parallel session:

- ✓ If you are running the SAP Basis 4.6D kernel with patch level 1413 or higher (or the SAP Web AS 6.10 kernel with patch level 427 or higher), you can turn on an ABAP trace for processes that are either currently executing ABAP code or idle.* For example, you can turn on an ABAP trace for a background process before the job is actually started. This option is especially helpful for jobs that finish quickly, where it would be impossible to turn on an ABAP trace while the job is running. Simply turn on a trace for the first background process and schedule the job for this server. If no other job is launched in the meantime, the job will run in the background process for which you turned on the trace. (The system always uses the first available background work process to start a background job.)
- ✓ During one transaction, you may actually change work processes several times. When you run

* See SAP Note 586940, which describes this new feature for tracing idle processes.

Figure 24

Call Hierarchy for a Traced Background Process

Gross	Net	Lv	Call Hierarchy
	910	3	Fetch SFLIGHT
	0	3	Close Cursor SFLIGHT
928	1	1	PERFORM A
927	7	2	PERFORM B
	14	3	Open Cursor SFLIGHT
	906	3	Fetch SFLIGHT
	0	3	Close Cursor SFLIGHT
926	0	1	PERFORM A
926	7	2	PERFORM B
	14	3	Open Cursor SFLIGHT
	905	3	Fetch SFLIGHT
	0	3	Close Cursor SFLIGHT
917	0	1	PERFORM A
917	6	2	PERFORM B
	14	3	Open Cursor SFLIGHT
	897	3	Fetch SFLIGHT
	0	3	Close Cursor SFLIGHT
977	1	1	PERFORM A
976	6	2	PERFORM B
	14	3	Open Cursor SFLIGHT
	956	3	Fetch SFLIGHT
	0	3	Close Cursor SFLIGHT
988	1	1	PERFORM A
987	6	2	PERFORM B

a transaction (such as by entering /nse38 in the Command field), the dispatcher rolls your user context into an available dialog work process, which executes this task until the first screen comes up. The transaction is now waiting for your input. Because the work process would be idle in the meantime, the system rolls out your user context so that another user can use the work process. If you perform any action on the screen (such as pressing the Return key), the dispatcher will search for the next available work process and rolls in your user context again, this time into the new work process.

When tracing a parallel session, you turn on the ABAP trace for a specific work process number. In order to find which work process has the activated ABAP trace, press the Refresh button in the work process overview screen. If the user context was rolled out and rolled in again, the work process with the activated trace will have a work process number of -1 in the Number column of the process overview.

- ☒ The trace variant, which specifies which program parts to trace and whether to use aggregation, applies to ABAP traces for parallel sessions in the same way as for normal ABAP traces. Use your usual trace restriction variants with no aggregation to run an ABAP trace for a parallel session.

which shows an overview of current work processes. Here, you turn an ABAP trace on and off for a single work process. In our example, the job is running in work process 9. To turn the trace on, select the work process and press the *Start measurement* button (🔍) in the application toolbar. Don't trace for a long time; several minutes should be enough. Otherwise, the trace file will grow very quickly and could hit its maximum size before reaching the point of interest.

To turn off the trace, select the work process in the process overview and press the *End measurement* button (🛑) in the application toolbar. Then step back to the initial ABAP Runtime Analysis screen, where you can analyze the trace file and review the call hierarchy to determine which ABAP code executed in the meantime.

Let's examine the call hierarchy for the ABAP trace we just ran, as shown in **Figure 24**. Now we can determine why this background job has been running for days. This program appears to be in an endless loop, because the following fragment repeats continuously:

```
PERFORM A
PERFORM B
  OPEN CURSOR SFLIGHT
  FETCH SFLIGHT
  CLOSE CURSOR SFLIGHT
```

To further research the situation, we jump from the repeating *PERFORM A* trace line to the ABAP Editor, as shown in **Figure 25**. Here, we find that this

Figure 25 The Traced Source Code

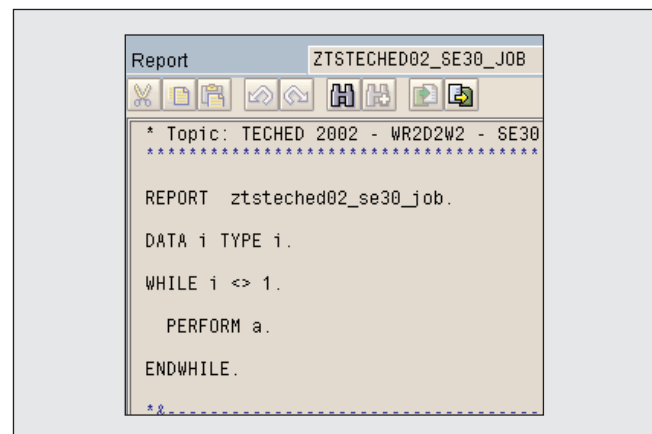


Figure 26 *The Particular Units Restriction for Trace Variants*

ABAP Runtime Analysis: Initial Screen

Variant: ABAPTRACE From user: GEBH

Description: ABAP Trace

Program (parts) Statements Duratn/type

☒ Within and below ☐ RFC, Update ☐ Particular units

Program (Parts)

P Program/gl. class/fct. group	Local class

program will actually *never* finish. The variable *i* that is used in the *WHILE* condition will never equal 1. This variable is initialized to 0, and further source code analysis would quickly reveal that it is never changed within the program.

We conclude from the ABAP trace that we can (and should!) safely terminate this batch job; otherwise, this loop will never end.

Tracing Partial Transactions

Sometimes you want to trace only a portion of a transaction, such as a subset of statements, rather than the entire transaction. For example, suppose you have a complicated transaction with many screens, and you are only interested in a message that appears after screen 21. It wouldn't make sense to trace all the ABAP code that is executed before this point of interest. Instead, you want to turn on the trace when execution reaches screen 21 and turn it off again after the target message is displayed.

You can restrict a trace with a trace variant, which you specify in the start screen of ABAP Runtime Analysis. You define a specific program or function module on which to run the trace. In addition, you can explicitly turn on the trace during a running transaction. You use the *Particular units* checkbox for this purpose, which you find on the maintenance screen for a trace variant (as shown in **Figure 26**).

If you select this checkbox and use this trace variant to run your ABAP trace, the trace does not automatically start from the beginning of the program. Instead, you can trace only a specific portion of the application. You must explicitly turn on the trace by entering */ron* in the *Command* field or following the menu path *System → Utilities → Runtime Analysis → Switch on*. After the trace passes the point of interest, turn it off again with the */roff* command or via the menu path *System → Utilities → Runtime Analysis → Switch off*.

By combining this feature with the debugger, you

can precisely specify which part of an application you want to trace. Set one breakpoint in the ABAP source code where you want to start the trace, and a second breakpoint where you want to stop the trace. Run an ABAP trace with the *Particular units* restriction turned on. When you reach the first breakpoint, turn the trace on. When you reach the second breakpoint, turn it off again.

You can also use the following ABAP statements for turning the ABAP trace on and off:

```
SET RUN TIME ANALYZER ON.  
SET RUN TIME ANALYZER OFF.
```

Insert these statements in the source code of a program where you cannot turn the trace on yourself, such as in a background job.

When You Need the Power of the ABAP Debugger

In this article, you have seen how a combination of troubleshooting tools, such as the system log, ABAP Dump Analysis, and the ABAP trace functionality of ABAP Runtime Analysis, can help you solve complicated ABAP error scenarios without turning to the ABAP Debugger.

The main problem with the debugger is that you are digging at the source code level (sometimes for unfamiliar applications) and can easily become lost within minutes. You could waste hours debugging an application without finding a single piece of relevant information. Therefore, except when investigating very straightforward problems in your own ABAP code, you should always leverage the other troubleshooting tools before using the debugger.

Suppose you receive a runtime error in an application, and you follow the recommendations outlined here and first analyze the ABAP dump in detail. You obtain a thorough understanding of the error situation, but in order to solve the problem, you also need to

know the value of a specific variable in the crashing program. That information is not available from the ABAP dump.

Here is a scenario in which the debugger *can* be used effectively. Set a breakpoint at the source line identified in the ABAP dump. With the debugger, stop at this line and find the value of the variable you need. From this example, we can see that it only makes sense to start debugging once you have a clear plan for what you want to learn (in this case, the value of a specific variable). For example:

- What is the value of variable *XYZ* when the program reaches function *Z*?
- When is the variable *ABC* of program *X* changed for the first time?
- Which branch of a specific *IF* statement is reached?

An article in an upcoming issue of *SAP Professional Journal* will focus on opportunities for using the ABAP Debugger, so that when you must resort to the debugger, you can do so appropriately and effectively.

Conclusion

Effective troubleshooting means using the right tool at the right time, with expert know-how and — most important — a sound troubleshooting strategy.

This article showed you how to start with the system log to get a rough (chronological) understanding of the problem, and then analyze the root problem by extracting as much information as possible from the ABAP dump. Finally, I introduced you to the powerful ABAP trace functionality, an underutilized feature of ABAP Runtime Analysis that can be used to find specific messages or function modules in a complicated application, or simply to understand the structure of an application.

We use these tools and techniques every day as part of our ongoing responsibilities in ABAP development support. They have saved us an enormous amount of time and effort, and helped us avoid countless wrong turns. By leveraging our proven techniques, we hope you can realize the same benefits in your own troubleshooting efforts.

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.