Improve your software quality by using ABAP checkpoint statements

by Gerd Kluger and Wolf Hagen Thümmel

Gerd Kluger
SAP NetWeaver
Foundation ABAP,
SAP AG

Wolf Hagen Thümmel
SAP NetWeaver
Foundation ABAP,
SAP AG

(Full bios appear on page 94.)

Program developers make certain assumptions about a program's state at various points during its execution (e.g., that the values of certain variables stay within specific ranges). Decent developers will document these assumptions using *comments* interspersed throughout the program code — on the one hand to understand *what* they did (and perhaps *why*), even years later, and on the other hand to enable other people involved in the software lifecycle process to understand, extend, and maintain the coding.

Although these comments play an important role for all participants of the software lifecycle process, the effect on the quality of the software is only indirect. Once a change is made to the software, there is no guarantee that the previous assumptions are still fulfilled; in other words, there's no guarantee that program correctness has been preserved. How can this be improved? How can we leverage these informal statements to ensure program correctness and increased maintainability? The answer lies in ABAP checkpoints — a type of statement introduced in SAP Web Application Server (SAP Web AS) 6.20[1] that is solely dedicated to ensuring program correctness and maintainability. The most important of these statements, `ASSERT`, makes programmer assumptions explicit and testable by the runtime system. The other two help to investigate program behavior in case of problems, either interactively (`BREAK-POINT`) or offline (`LOG-POINT`). All three checkpoint statements can be controlled in a way that minimizes their impact on running applications.

This article is for developers and quality experts who want to improve the quality of software written in ABAP. Through step-by-step examples, we explain the ABAP checkpoints concept and show you how to use these statements efficiently, allowing you to easily gain hands-on experience. We start with a description of assertions and show you how they can improve

[1] Support Package 29 is required for SAP Web AS 6.20.

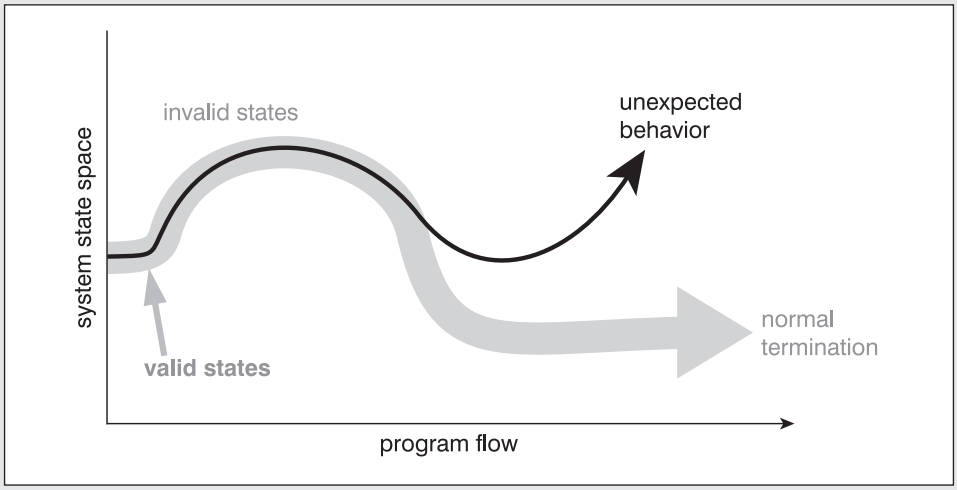


Figure 1 Errors cause unexpected behavior in programs

the quality of your software. After introducing basic assertions, we move on to *activatable* assertions. Being able to activate a statement is the key capability of all ABAP checkpoints — only active checkpoint statements are considered in program execution — and therefore we spend considerable time casting light on this important aspect. Finally, we round out the discussion by taking a look at the types of checkpoints that are used in more specialized scenarios — breakpoints and logpoints — as well as some of the more advanced features of assertions.

ABAP checkpoints

ABAP statements are used to declare program variables (declarative statements) or to build up the program logic (operative statements). While operative statements have a certain position within the program and are executed in a certain order, declarative statements are never reached during program execution. They declare properties of the program (mostly variables), which are to some extent independent from the current execution position.

ABAP checkpoints are a bit different here. Technically, they're operative statements. They are executed when reached during program flow. Unlike normal operative statements, they are not part of the program logic. Programmers may insert them into their code to check the validity of their assumptions (`ASSERT`), to record that a specific portion of their code has been reached, and to probe current variable values (`LOG-POINT`), or to allow for debugging (`BREAK-POINT`). The first one will help to ensure a higher level of program correctness, while the latter two will help to understand and maintain the code. Let's start by concentrating on program correctness and how this can be achieved using assertions. We will come back to breakpoints and logpoints later.

Assertions

Why is it so important for programmers to make their assumptions explicit? Let's have a look at **Figure 1**.

Usually there is a set of states that is considered valid during the execution of a program. If your program is erroneous, sooner or later it will leave this set of valid states, and the result will be some unexpected behavior. For example, in a banking application, if two accounts are involved in a money transfer, the sum of both balances must be equal before and after the transaction. If the balance is

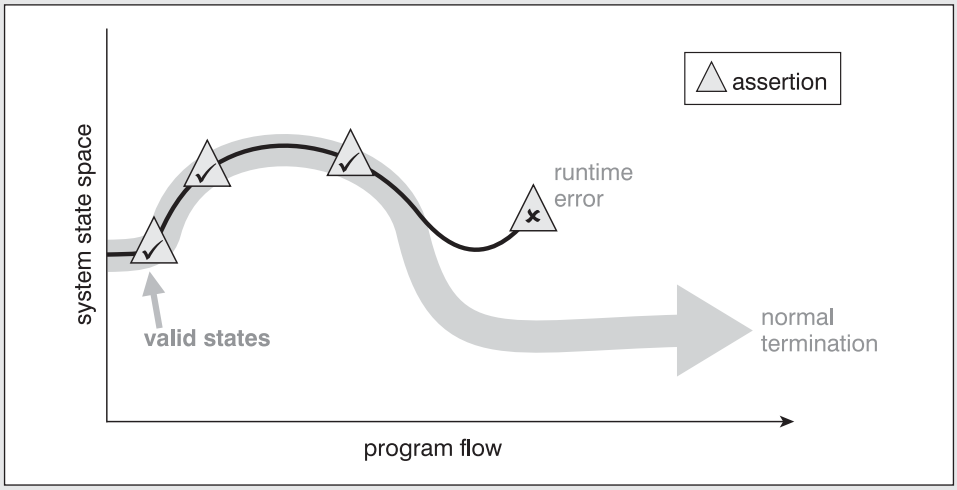


Figure 2 Assertions allow you to detect errors as soon as they occur

unequal, you want to detect this situation as early as possible in order to limit the damage the program can do, such as propagating the out-of-balance condition to other associated accounts. The risk is always high that one error will lead to successive errors.

An *assertion* is a statement placed at some point in the program describing a specific set of conditions that the programmer expects to be met at that point. **Figure 2** shows how assertions can help.

By checking the correctness of assertions during the execution of a program, execution can be stopped as soon as an incorrect state is detected. The effect of bugs therefore will be limited and program correctness enhanced.

While assertions help ensure the operation of your software, they also play an important role when software is modified — e.g., during maintenance or when the software is extended for additional functionality. Using assertions, expected system behavior is clearly stated, so the developer has a clearer understanding of the existing code. In addition, the assumptions are checked automatically when the program is tested and hence, to a large extent, assertions serve to increase program maintainability.

Let's have a look at how assertions are expressed in ABAP. We will start off with the most basic form of an `ASSERT` statement:

```
ASSERT logExpr.
```

When an `ASSERT` statement (in its simplest form) is executed, the logical expression `logExpr` is evaluated. If the expression is true, processing continues with the next statement. If the expression is false, program execution is aborted by issuing the runtime error `ASSERTION_FAILED`.

Assertions are well established in a variety of programming languages. There is, however, a major difference between assertions in ABAP and assertions in other languages, such as Java or C/C++. In other languages, assertions are normally not executed unless the runtime environment is started with a special option or the executable has been compiled with some debug option. In ABAP, the `ASSERT` statement in its basic form is *always active* — there is no way to disable it.

As a result, the statement:

```
ASSERT logExpr.
```

is functionally equivalent to the following code snippet, which you may have used in the past:

```
IF NOT logExpr.
  MESSAGE ´Error occurred´ TYPE ´X´.
ENDIF.
```

In both cases, program execution is aborted and a short dump is written if the logical expression does not evaluate to true.

Of course, this constant checking of assertions puts a burden on the program's performance. It would be better to be able to switch checking on and off in a flexible way, without having to stop the system. This leads us to the notion of *activatable assertions*.

Activatable assertions

An *activatable assertion* is, like the assertion in its most basic form, a conditional checkpoint. That is, at runtime the condition specified in the assertion can be checked, and if the condition is not met a runtime error is generated. Unlike the always-active basic assertion, however, whether the runtime condition is checked at all can be controlled externally. Checking can be switched on and off on the fly and in a fine granularity. Turning on the checking of an assertion condition is called "activating" the checkpoint. Note that in contrast to other programming languages, in ABAP, recompilation or restarting the server in a dedicated mode is not required to activate assertions.

Assertion checking is typically activated during the development, testing, and maintenance of a program, but in a production system, checking might be considered too expensive in terms of performance. In that case, because assertions can reside in the production coding, you can switch checking on to investigate complex problems that occur in production environments and subsequently switch the assertion checking back off. However, because problem analysis should not normally interfere with productive work in the system, it would be useful to restrict the activation of assertions to the context of explicitly specified users or application servers. In addition, because an application normally consists of a larger number of programs, function groups, and classes, it would be useful to have a way to define and activate a set of assertions that are spread over a specific set of such compilation units. And finally, it would be nice to be able to group certain assertions that are closely related.

The solution to these problems of grouping lies in the use of *checkpoint groups*, which we will describe next.

Checkpoint groups

The *checkpoint group* is a new kind of ABAP repository object. The activation state of activatable checkpoints is controlled via the checkpoint group. To this end, all activatable checkpoint statements subscribe to a checkpoint group. This is expressed by the `ID` addition in the `ASSERT` statement, followed by the name of a checkpoint group, as follows:

```
ASSERT ID CheckpointGroup CONDITION logExpr.
```

If the `ID` addition is specified, the `ASSERT` statement is activatable. Otherwise, the statement is always active. Note that when the `ASSERT` statement is used in a form other than its basic form, the logical expression to be evaluated must be separated by the keyword `CONDITION`.

Having introduced the concept of checkpoint groups, we have already met a large portion of the requirements for activatable assertions:

- Because all activatable checkpoints subscribe to a checkpoint group that controls the activation, we can activate a set of checkpoints that may reside in different compilation units.
- Because the checkpoint group is a repository object that is embedded in the ABAP Workbench's cross-reference and navigation services, all checkpoint statements subscribing to a specific checkpoint group can be easily found.

What's missing is a way to restrict the activation to individual users or specific servers. This leads us to the concept of *activation settings*. Let's first gain hands-on experience by creating a checkpoint group.

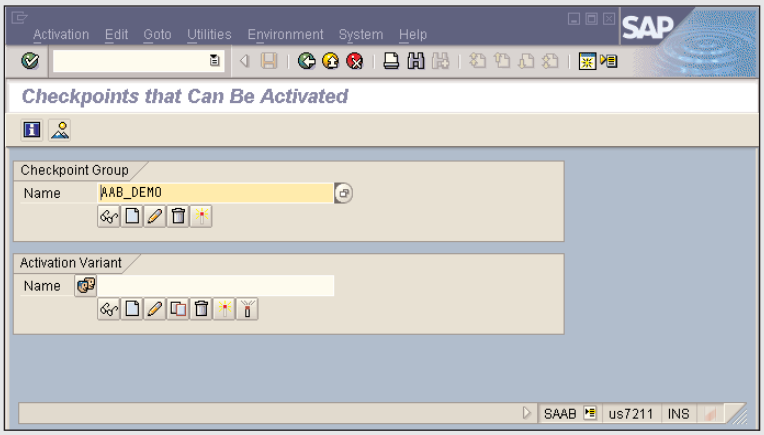


Figure 3 Initial screen of transaction SAAB

Creating a checkpoint group

Checkpoint groups are created using transaction SAAB. On the initial screen, shown in **Figure 3**, enter a name for your checkpoint group in the Checkpoint Group frame and click on the Create button. As with program names, the group name is limited to 30 characters. In the following pop-up window, enter a description and click on Save. After assigning a package, you're done. The checkpoint group is now ready for use.

Now create a simple test program that contains an activatable assertion subscribing to your new checkpoint group. Here's a sample of such a test program:

```
DATA i TYPE i.
ASSERT ID AAB_DEMO CONDITION i IS NOT INITIAL.
WRITE i.
```

Lacking any other actions, if you execute this program, the value zero (which is the initial value for variables of type i) will be displayed and no short dump will be generated. The `ASSERT` statement will have been ignored, because *by default, activatable checkpoints are always inactive*.

To activate the assertion, click on the Activate button on the SAAB initial screen. On the Activation tab on the resulting main Activate Checkpoint Group screen, shown in **Figure 4** on the next page, there's a frame labeled Personal Activation with a subframe called Assertions. This is where you can set the activation for the current checkpoint group. Switch the radio buttons from Inactive to Abort and click on the Save button. Now execute the test program again.

The program will terminate with the runtime error `ASSERTION_FAILED` and a short dump will show the position where the assertion was violated. You may ask different users to execute your test program but the runtime error will not occur in their case because you activated the assertion in your test program exclusively for you (by using the Personal Activation settings), without affecting other users.

So what's happening in the background when you activate a checkpoint group? An *activation setting* is created, which is analyzed each time a program is started. We'll look at this next.

Activation settings

Activating a checkpoint group technically results in the creation of an *activation setting*. When an activatable checkpoint statement is processed, the ABAP runtime environment looks for a matching activation setting; that is, an activation setting whose attributes are applicable to the current checkpoint statement and execution context. If one is found, the checkpoint is deemed active and the statement is processed. If none

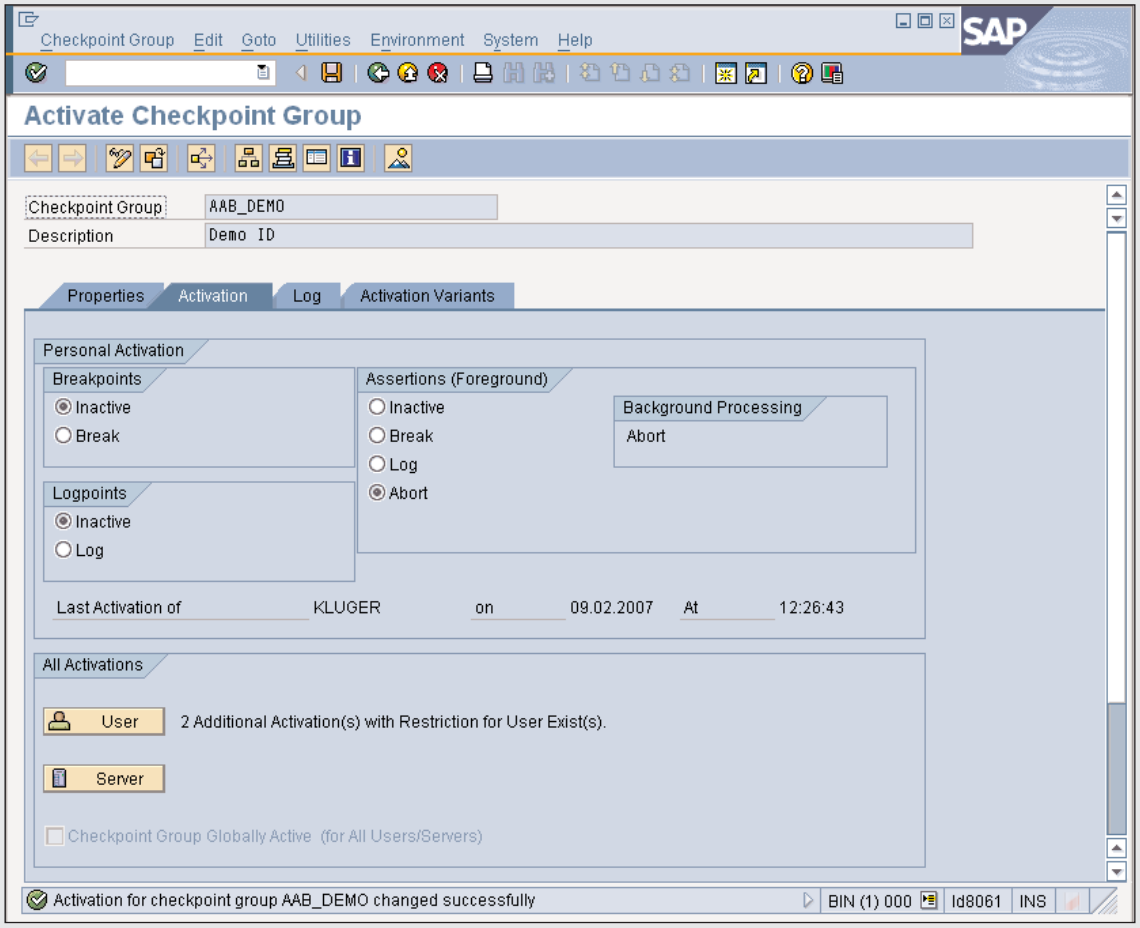


Figure 4 The main Activate Checkpoint Group screen

is found, the checkpoint is considered inactive and the statement is skipped.

Let's have a closer look at these activation settings to understand when such a setting will match (and thus make a checkpoint active) and when it will not. Activation settings are created, modified, or deleted automatically once you change the activation state of a checkpoint group using transaction SAAB. You don't manipulate them manually. Understanding in detail the structure of activation settings will help you better understand your options when activating a checkpoint group.

An activation setting consists of three parts: scope description, context description, and mode settings. Only the mode settings part differentiates between the different kinds of checkpoint statements (e.g., assertions, as in our example, versus breakpoints or logpoints). To have a matching activation setting, all three parts must match individually. As soon as one of the three parts doesn't match, the activation setting as a whole does not match for the checkpoint statement in question.

Scope description

The scope description is used to determine whether an activation setting is applicable to a specific checkpoint statement. It contains the name of a checkpoint group or the name of a compilation unit. An activation related

to the name of a checkpoint group is a *group-specific activation*. An activation related to a compilation unit is called a *program-specific activation* (more on this type of activation in an upcoming section).

When an activatable checkpoint statement is reached, the runtime system checks to see if there is a matching activation setting, either for the checkpoint group named in the `ID` addition or for the compilation unit in which the statement is located. If no such activation setting exists, the checkpoint is considered inactive and the statement is skipped. If one or more activation settings are found, the runtime environment continues and analyzes the context description.

Context description

The context description is used to determine if the activation setting is applicable for the current execution context. Execution context in this sense consists of the current user[2] and the application server on which the program is executed. A context description may specify a user name, a server name, or neither. A specified user name will restrict the activation setting to this user. If the program runs in the context of some other user, the activation setting will not match. Similarly, a specified server name will restrict the activation setting to the given application server. If the program is running on a different application server, the setting won't match either. If the context description specifies no restriction for user or server at all, the activation setting is applicable to all user and server contexts.

As a result, we can distinguish three kinds of activation settings with respect to the context:

- **Global activation:** The setting applies to all users on all application servers as long as the client matches.
- **User-specific activation:** The setting applies only to the specified user, regardless of the server he or she is working on.
- **Server-specific activation:** The setting applies to all users working in the specified client on the specified application server.

If the activation settings do not pass the context check, the checkpoint statement is considered inactive in the current context and is skipped. Otherwise, the runtime environment will analyze the mode settings, which are described next.

Mode settings

The mode settings define whether a checkpoint statement of a specific kind is active and what action should be performed if it's active and the condition expressed in the assertion has not been met. So far we have been dealing with only one kind of checkpoint statement: the `ASSERT` statement. But as we introduced earlier, there are two other types of statements: `BREAK-POINT` and `LOG-POINT`. A given checkpoint group can be used with all three kinds of checkpoint statements. To control the behavior of `ASSERT`, `BREAK-POINT`, and `LOG-POINT` individually, there are three individual settings, which correspond to each of the different kinds of checkpoints. If the mode setting part for the kind of checkpoint in question is active, the activatable statement is executed. Otherwise, it is skipped.

With an understanding of all three parts of an activation setting, we now clearly see the different options for activating a checkpoint group: activation

> ***Note!***
>
> While checkpoint groups are repository objects that are transported between systems, the corresponding activation settings are only local data. The settings are never transported, neither together with a checkpoint group nor alone. Creating a checkpoint group also does not imply activating it. In other words, checkpoint groups, like the example we created earlier, are inactive by default. You must activate it manually, as we did in the example, to have the assertion executed.

[2] The context is client independent for SAP Web AS 6.20 and 6.40. Because SAP NetWeaver 2004s settings are client dependent, the same user may have different settings under different client identifications.

can be restricted to individual users or servers and to specific kinds of checkpoint statements. We'll provide examples on how this can be controlled using transaction SAAB later.

Having learned the meaning of activation settings, we can now understand in detail what happened in our first example when we created the checkpoint group and used it in a test program. Since creating the checkpoint group does not imply the creation of a corresponding activation setting, the ASSERT statement in the test program was inactive and skipped during execution. It is important to understand that if the statement is skipped, the logical expression is not evaluated. No matter how time consuming your assertion conditions are, they won't hurt the system's performance as long as these checkpoints are not activated.

In the next step in the example, we activated the checkpoint group using transaction SAAB, resulting in the creation of an activation setting. We chose the *personal activation* option, which is a special case of user-specific activation where the activation is set for the current user. We ended up with an activation setting carrying the new checkpoint group in the scope description, specifying your user name in the context description and an active mode setting for assertions. When we executed the program a second time, the runtime environment found this matching activation setting and executed the ASSERT statement. The condition was checked and because it evaluated to *false*, the action specified in the mode settings for assertions (to abort, in this case) was performed.

When other users execute the program, the context description won't match because it specifies your user name. Thus the ASSERT statement is ignored, unless they have activated it for themselves.

Multiple activation settings and context precedence

The scope and context descriptions of an activation setting contain only single values for any of the setting's attributes (i.e., user, server, compilation unit, checkpoint group, modes). It may also contain no specific value, as we saw for the user and server attributes in the case of global (not restricted to a user or server, for example) activations. If the activation state that we want to create comprises more than one value for any attribute, a set of activation settings has to be created rather than a single activation setting.

Consider the following example: You created a set of checkpoint groups and instrumented your code with activatable checkpoints, each subscribing to one of the checkpoint groups. If you want to have all these checkpoints active, you will have to activate each of your checkpoint groups separately using transaction SAAB. (There is a way to activate multiple checkpoint groups at once, which we will describe later in the discussion of *activation variants*.) This will give you a set of activation settings, one for each checkpoint group. When the program is executed and one of the checkpoint statements is reached, the activation setting corresponding to the specified checkpoint group will be used, so this is straightforward.

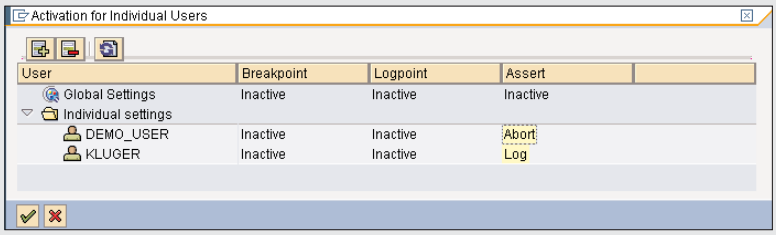
Besides having multiple activation settings that differ in scope description, you can also have multiple activation settings with differing context descriptions. For example, suppose you have an activation setting for a checkpoint group with your user name in it, as in the example described earlier. Let's say that a second activation setting specifies the same checkpoint group and a server "S" as context, and a third setting exists for the same checkpoint group, but has a global context, i.e., no user and no server specified (see the sidebar on the next page for more on how to create these additional types of activation settings). When you now execute your program on server S, the runtime environment will find three matching activation settings — which one is considered? Since the mode settings may be different for all three activation settings, this question really matters.

A precedence rule has been defined to make clear what happens in such a situation: User-specific activation settings supersede server-specific activation settings, and user- and server-specific activation settings supersede global activation settings. In the example, your personal activation setting would win.

Creating non-personal activation settings

When we created the example checkpoint group, you saw how to create a personal activation setting. Let's take a look at how you might create additional kinds of activation settings.

Take a look at the Activation tab on the main screen of transaction SAAB shown back in **Figure 4**. Below the Personal Activation frame, there is a second frame called All Activations with two buttons: User and Server. Click on the User button. In the pop-up screen, shown below, you see the global settings and a list of individual settings (if any). There are three columns for the mode settings for each kind of checkpoint statement.*



The global settings may show "inactive" in all three columns, meaning that no global settings exist for the checkpoint group in question. Remember that a global activation would be achieved by an activation setting where the user and server names in the context description are left blank. You can change the mode setting of the corresponding type of checkpoint statement by right-clicking in one of the three columns. (Remember that you must return to the main SAAB screen and click on the Save button to save any changes to the settings in the pop-up screen.) To activate a checkpoint group for a different user, click on the Add User button ( ) in the pop-up screen. Don't forget to modify the inactive mode settings for this new user before you save; otherwise, the entry in the list will silently disappear. Remember that an inactive checkpoint group for a given context has no activation setting at all.

Clicking on the Server button in the All Activations frame presents a similar pop-up screen in which you can add server-specific activations. The global settings here are identical to the global settings in the user screen. From what we have learned about the context description, it is clear that both entries technically refer to the same activation setting.

Note that if there is no personal activation setting for a checkpoint, but there is a global or server-specific setting, the radio buttons in the Personal Activation frame are set to Inactive and the Checkpoint Group Globally Active checkbox in the All Activations frame is selected, so that you are aware there are non-personal settings in place that may affect your program.

* For SAP Web AS 6.20/6.40, the mode settings are represented by a set of checkboxes instead.

Program-specific activation and scope precedence

In addition to checkpoint groups, there is another way to define the scope of activation settings. Alternatively, an activation setting may refer to the name of a compilation unit (i.e., a program, function group, or class pool). In this case, this setting controls all activatable checkpoints within this compilation unit, regardless of the checkpoint group to which the

individual checkpoint subscribes. This is called a *program-specific activation*.

Because both group-specific activation settings and program-specific activation settings may exist at the same time, we need a precedence rule to define what will happen in the case of conflicting settings. The rule is pretty simple: program-specific activation settings supersede group-specific settings for a given program. Note that group-specific settings are not affected with respect to other programs, where no corresponding program-specific setting exists. In other words, every time a program is called, the runtime environment first looks for a matching program-specific setting. If one is found, the corresponding mode settings are used for all activatable checkpoints within this program. Otherwise, the matching group-specific activations are used, if any.

Program-specific activation can only be achieved by means of an activation variant, which we look at next.

Activation variants

If you need to activate multiple checkpoint groups, you can create an activation variant to simplify this task. An activation variant is a collection of checkpoint groups and/or compilation units with a corresponding mode specification (which kinds of checkpoint statements will be active) for each collection item. The activation context (user-specific, server-specific, or global, as with activating a single checkpoint group) is not stored in the activation variant but must be supplied when *activating the variant*. All items in the collection are then activated (in the case of compilation units, this means that a corresponding program-specific activation is performed). As a result, a variant can be used to create group-specific activations, program-specific activations, or both.

Technically, an activation variant can be seen as a template for a set of activation settings. The difference between an activation variant and a real set of activation settings is the missing context description. The context must be defined when using the activation variant for creating the set of activation settings.

Activation variants come in two flavors*: global* and *local*. Global variants are repository objects that can be transported between systems. Local variants, on the other hand, can't be transported. They are intended for personal use (they are also usable by other users, though).

Note that activation variants are only *templates* for creating a set of activation settings — they don't represent activation settings themselves. This means that no checkpoints are activated simply by the creation of a global activation variant or because such a variant was transported into an SAP system. You always have to activate explicitly.

Let's take a look at how to create these different kinds of activation settings.

Maintaining activation variants

On the initial screen of transaction SAAB (**Figure 3**), there's a second frame called Activation Variant. Click on the button in front of the variant name input field to toggle between local and global (transportable) variants. Insert a variant name and click on the Create button. On the following screen, shown in **Figure 5**, the content of your variant is presented in a table. In the Object Type column, you can choose between Checkpoint Group for a group-specific activation and Program, Class, or Function Group for program-specific activations. The corresponding object name is specified in the Object Name column. The activation mode is then set in the remaining columns.

The variant is activated by clicking on the Activate button, either on the initial screen or in the variant display. When activating a variant, the missing context description has to be specified. Again, you can choose between Personal Activation, User-specific Activation, and Server-specific Activation. The system will then create an activation setting for each variant member using the corresponding mode settings as specified in the variant and assigning the context information that you supplied.

Deactivating variants

When a variant is used for activation, it is expanded into individual activation settings. Technically, there's no difference between this technique and creating all of these settings individually. This has implications

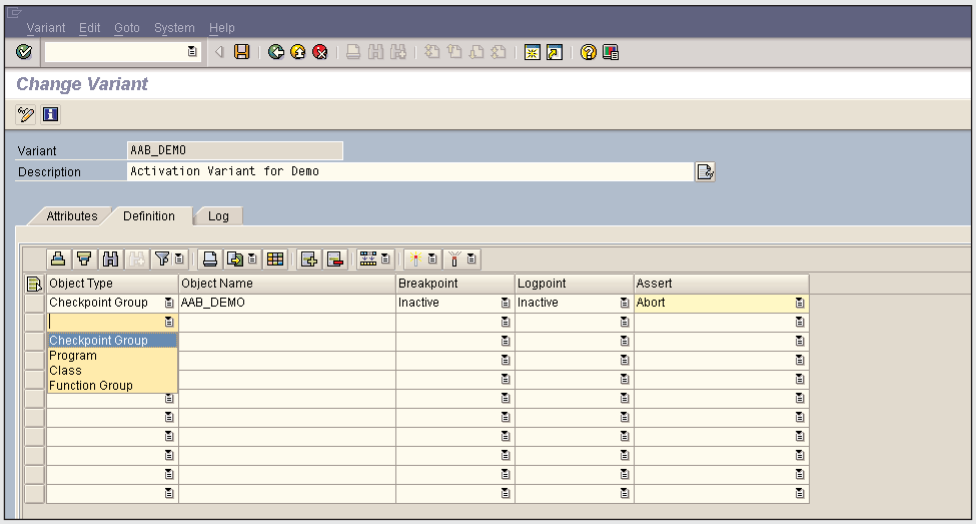


Figure 5 The variant display

when you want to get rid of such settings later. An activation setting has no memory as to whether it was created manually or via a variant. As a result, you're free to deactivate the individual checkpoint groups manually, or to use the activation variant for this purpose. If you decide on the latter, you have to specify the same context for this operation as you did when activating the variant. Keep in mind that program-specific settings can only be maintained using an activation variant (for both activation and deactivation).

In the case of personal settings, there is a third way to deactivate. You may use the menu entry Activation/Delete from the initial screen of transaction SAAB. Consider this as a last resort when the variant used for activating is unknown or no longer exists.

Displaying an overview of all activation settings

With global and personal activation settings, which can be either group-specific or program-specific and which can stem from the activation of a checkpoint group or from the activation of a variant, we need some way to obtain an overview of all activation settings that are currently in the system. Call transaction SAAB and select Display from the Activations menu. **Figure 6** on the next page shows an example overview display. You can display all settings that potentially apply to you (your personal settings and global settings), or all settings in the system (in both cases, only settings for the current client are displayed[3]).

In the right-most columns, you can see who created the individual activation settings and when this happened.

Having focused solely on assertions so far, let's move on to the other kinds of checkpoint statements, the `BREAK-POINT` statement and the `LOG-POINT` statement. We'll look at breakpoints first.

Breakpoints

In its simplest, most well-known form, a *breakpoint* consists of just the keyword itself:

```
BREAK-POINT.
```

[3] As mentioned earlier, the activation settings are client-independent for SAP Web AS 6.20/6.40.

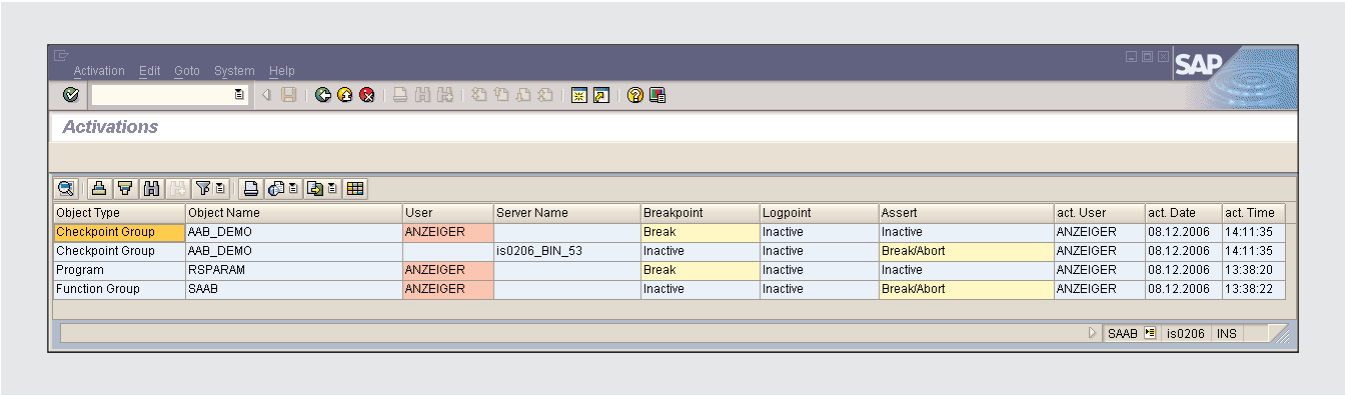


Figure 6 Overview of current activation settings in the system

If program flow reaches this statement during dialog processing, program execution is interrupted and the system switches to ABAP Debugger.[4] During background processing or inside an update task, the execution of the program is not interrupted. Instead, processing continues with the following statement.

The `BREAK-POINT` statement is extremely handy during program development. On the other hand, in production systems the user does not expect ABAP Debugger to pop up during his or her daily work. As a result, all always-active `BREAK-POINT` statements should be removed before the program is transported into a production system.

From another point of view, however, eliminating all breakpoints once the code is ready for shipment represents some loss of information. The developers may have carefully defined a set of breakpoints to further analyze the state of the application and to explore the program's behavior. These breakpoints may be needed again when it comes to maintaining the code or searching for errors in a production system. By using the same infrastructure as for activatable assertions, we can avoid the problem. This leads us directly to *activatable breakpoints*.

[4] For more on ABAP Debugger, see the *SAP Professional Journal* articles, "An Integrated Approach to Troubleshooting Your ABAP Programs: Expert Tips for Making the Most of the ABAP Debugger" (July/August 2004) and "Introducing the next generation of ABAP debugging — the New ABAP Debugger" (January/February 2006).

Activatable breakpoints

An *activatable breakpoint* is, like the basic `BREAK-POINT` statement, a hard-coded breakpoint that resides in the program code (in contrast to a transient breakpoint that can be set dynamically in ABAP Editor or Debugger). Like other activatable checkpoints, an activatable breakpoint is "switched-off" by default. This is to prevent users, especially in production systems, from being disturbed. Once a problem needs to be investigated, the breakpoint can be simply switched on in the same manner as activatable assertions.

To gain external control over a breakpoint, it must subscribe to a checkpoint group. This is necessary even for program-specific activation. Subscription to a checkpoint group is achieved by the `ID` addition followed by the name of the checkpoint group, in exactly the same way as for an activatable assertion:

```
BREAK-POINT ID CheckpointGroup.
```

An activatable breakpoint that has been activated, either by group-specific activation or program-specific activation, behaves almost exactly the same as an always-active breakpoint. The only notable difference is in background processing (batch job processing or non-local update task). Program execution is not interrupted under any circumstances. But whereas reaching an always-active breakpoint causes a system log entry stating "Breakpoint reached," no such logging happens when reaching an active activatable breakpoint. The reason for this is that always-active breakpoints must be omitted from production code

and thus reaching them during background processing can be considered an abnormal event that must be logged. In contrast, the use of activatable breakpoints is perfectly legal, so there's no reason for logging the occurrence of such a breakpoint — activatable breakpoints that are executed during background processing are simply ignored.

> ***Note!***
>
> The same restrictions apply for activatable breakpoints in dialog processing as for always-active breakpoints. This means that just as you have to switch on update debugging to debug an update task, you have to switch on system debugging to make breakpoints work in system programs, and you have to activate external debugging to debug HTTP requests.[5] None of these settings are done automatically by activating a checkpoint group using transaction SAAB.

Logpoints

During program execution, situations may arise that must be recorded. These can be erroneous states, critical conditions, or other information that helps to identify or analyze the system behavior. Such pieces of information should be collected and saved in a log for later use. This is nothing new, and well-established logging infrastructures exist in the ABAP environment, such as the System Log for system-specific information and the Application Log for application-specific information.[6]

[5] For further details concerning debugging in these situations, please refer to the online ABAP documentation for the `BREAK-POINT` statement.

[6] For more on the System Log, see the article, "An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP Investigative Tools for Production Problems" (*SAP Professional Journal*, May/June 2004). For more on the Application Log, see the article, "Decrease Problem Determination Costs and Increase User Satisfaction with SAP Logging Solutions" (*SAP Professional Journal*, September/October 2002).

The *logpoint* is an additional concept that aims to fill some gaps here, specifically the logging of low-level program-specific information. The logpoint focuses on the convenient logging of variable values that are to be analyzed by the program developer, rather than the logging of high-level messages for the system administrator or application user. The logpoint is the youngest member of the ABAP checkpoint family and became available in SAP NetWeaver 2004s.

Logpoints are activatable

In contrast to the assertion and the breakpoint, the logpoint exists *only* as an activatable checkpoint — there is no always-active form of the `LOG-POINT` statement. In other words, the addition of an `ID` is mandatory in the `LOG-POINT` statement, and a `LOG-POINT` statement that doesn't include an `ID` will trigger a syntax error. This is because it is always a bad idea to store a wealth of logging data without someone wanting to analyze it, or even without anyone knowing that this is happening. As a result, you will have to activate the corresponding checkpoint groups for the `LOG-POINT` statement, if you want this kind of logging.

On the other hand, there is also something that the logpoint and the breakpoint *do* have in common: They're both *unconditional* checkpoints. You can therefore probably guess the syntax for the `LOG-POINT` statement:

```
LOG-POINT ID CheckpointGroup.
```

If no matching activation setting exists, the statement is skipped. Otherwise, the information that this logpoint was reached is stored in a dedicated log. This log is completely different from, for example, the Application Log or System Log:

- To conserve space, all entries are aggregated on the level of the checkpoint statements by which they were produced. This means that you will have at most one entry for each individual checkpoint statement in your code. This helps to prevent running out of memory for checkpoints situated inside of loops.
- If a logpoint is reached multiple times, a counter is

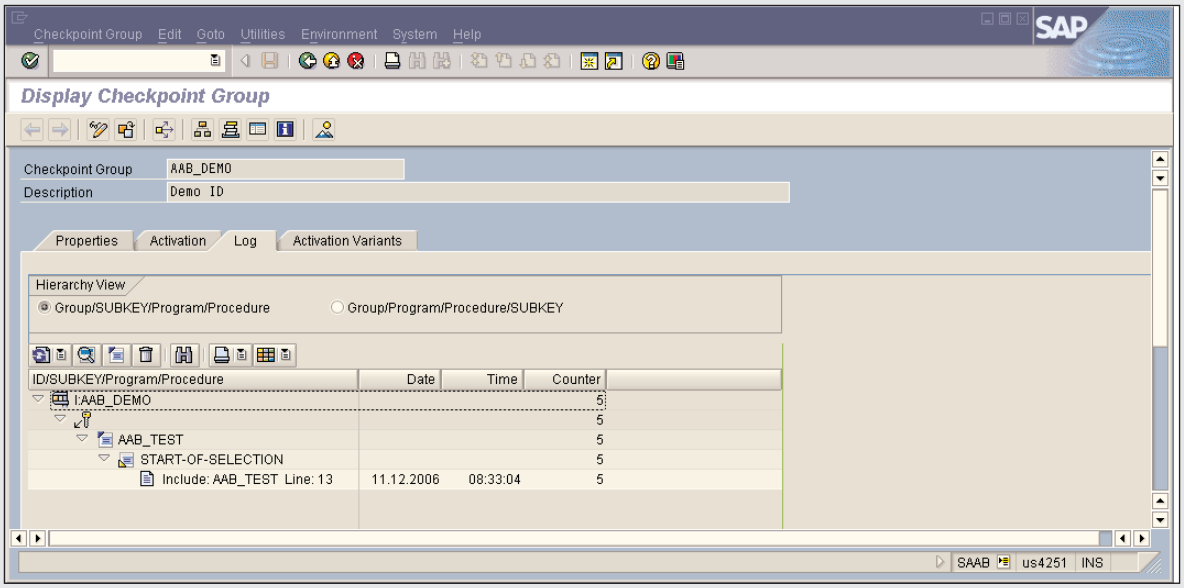


Figure 7 The SAAB log display

incremented to see how often the statement was reached.

- Since the low-level log entries are tightly connected to the program source, entries will become obsolete once the program is modified and will be deleted periodically.
- Log entries are viewed using transaction SAAB. The log display supports navigation to the source code.

Let's look at an example. Modify your test program so that it contains a logpoint and use transaction SAAB to activate the corresponding checkpoint group for the logpoint mode. Execute your program and afterward go to the Log tab on the SAAB main screen. You will be presented with a hierarchical view of all log entries belonging to the current checkpoint group, as shown in **Figure 7**. After drilling down, you can display detailed information or navigate to the checkpoint statement that issued this log entry (using the tree's button bar or by right-clicking on the checkpoint statement).

The information regarding how many times the logpoint was reached and the time of the last access may help you analyze your program's behavior. However, you probably want to store additional information in the log — this is covered next.

Storing data using the FIELDS addition

The `LOG-POINT` statement offers the `FIELDS` addition, which allows you to store additional information when a log entry is written:

```
LOG-POINT ID  CheckpointGroup  FIELDS
field1 … fieldN .
```

The `field1` … `fieldN` are variable names from your program; they can be of any type, except for reference variables or types containing references. Up to 32 fields are considered. The names and values of these variables are stored in the log. When you view the log, the values of the specified fields are rendered in the detail display, shown in **Figure 8**. Note that you can only see the details of the latest entry for each checkpoint statement since previous entries are overwritten when a new one is stored.

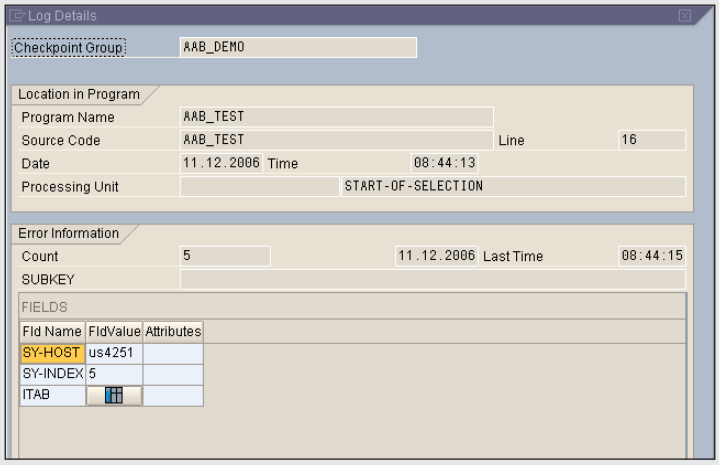


Figure 8 The SAAB log detail display

A number of logging fields may represent a considerable amount of data, especially when internal tables or strings are involved. There is an upper limit for the number of bytes of data stored with a single logging entry. If the actual data exceeds this limit, the data values are truncated. Truncated values are marked in the Attributes column of the `FIELDS` display. The default limit is 1024 bytes; if this is too restrictive for your purposes, it can be modified by changing the SAP profile parameter abap/aab_log_field_size_limit. The parameter value is given in bytes.

When you think of logging large internal tables or strings, please keep in mind that the only way to analyze this data is by browsing within the log display of transaction SAAB — this may not be feasible for large tables or strings.

Don't mistake `LOG-POINT` with the `FIELDS` addition as a way to store data for arbitrary purposes. Instead, it's a dedicated mechanism for logging low-level information for the programmer, for purposes of problem analysis.

Controlling aggregation using the SUBKEY addition

As stated earlier, each processing of a `LOG-POINT` statement may not result in a separate log record. Instead, log entries are aggregated in some way to prevent producing huge amounts of data, when a `LOG-POINT` statement happens to be inside of a loop, for example. The way aggregation is performed is based on a composite key. All occurrences with the same key will produce only one log record. Only the last occurrence is stored, but a counter is incremented each time. The key consists of the modification date of the corresponding program and the position of the corresponding checkpoint statement within that program. There may be situations where you want to keep multiple log entries for one checkpoint statement. This can be achieved by specifying a subkey. Entries are then only aggregated if the subkeys are equal. By giving different subkey values, you are able to circumvent the standard aggregation.

Subkey values are specified using the `LOG-POINT` statement addition `SUBKEY`. The corresponding variable is expected to be character-like. Up to 200 characters are considered. In conjunction with the `FIELDS` addition, the `LOG-POINT` statement then looks as follows (the `SUBKEY` addition can also be used with or without the `FIELDS` addition):

```
LOG-POINT ID  CheckpointGroup  SUBKEY
sub-key   FIELDS field1 … fieldN .
```

```
DATA n TYPE n.

DO 50 TIMES.
  LOG-POINT ID AAB_DEMO          FIELDS sy-index.
ENDDO.

DO 50 TIMES.
  n = sy-index MOD 10.
  LOG-POINT ID AAB_DEMO SUBKEY n FIELDS sy-index.
ENDDO.
```

Figure 9 Two logpoints inside of loops

To better understand how aggregation works, take a look at the two logpoints inside of the loops shown in **Figure 9**.

The `LOG-POINT` statement in the first loop will produce exactly one record because the statement's position is the only key component. The counter, however, will show that the statement was reached 50 times. The field value stored in the log record will be 50, because only the last record will be kept.

In the second loop, the situation is quite different. Here, 10 different log records will be produced with subkeys ranging from one to nine and an additional zero. The corresponding field values are 41 to 50.

When do the assertion, breakpoint, and logpoint activation settings you've put in place for your program become effective? What happens if you make changes to these settings when the program is running? We'll take a look at this next.

When do activation changes become effective?

When activation settings are created using transaction SAAB, they are stored in the database. After an internal mode[7] is started for program execution, the settings are loaded from the database and stored within the internal mode. For the sake of runtime performance, these settings are kept for the entire lifetime of the internal mode. As a result, any changes to the activation settings will not become effective for internal modes that are already running.

Controlling activation via ABAP Debugger

Consider that you have an activated breakpoint inside of a loop. At some point, you may want to resume processing without stopping at every iteration. Using transaction SAAB, the only option is to deactivate the corresponding checkpoint group and to restart the program. This is far from convenient. For this reason, the Breakpoint Tool in the new ABAP Debugger, shown in **Figure 10**, allows you to modify the loaded activation settings for the running program.[8] The Checkpoint Activations tab gives you a list of all currently loaded activation settings.

A checkbox for every item in the list indicates whether this setting applies to the current program (i.e., whether the checkpoint group is used by checkpoint statements within the current program). You can change the activation setting for an individual checkpoint group by right-clicking on it.

The loading of a particular activation setting may be delayed until the first program that actually makes use of this setting is loaded. Consequently, this list may grow as you step through your application. As a result,

[7] An internal mode is the execution context that is usually created when a program is started. The most important part of the internal mode is the roll memory, in which all program-related data is stored.

[8] The classical ABAP Debugger does not offer this functionality, so this is not an option for SAP Web AS 6.20, in which the new ABAP Debugger is not available.

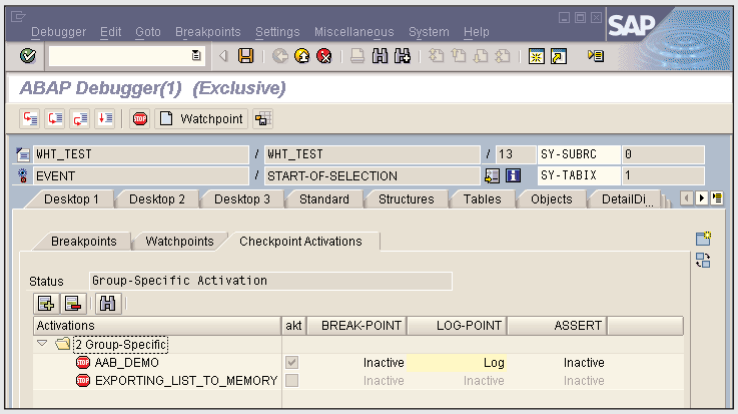


Figure 10 Checkpoint Activations tab in the ABAP Debugger Breakpoint Tool

the setting that you want to modify may not yet be loaded. In this case, you can create the activation setting manually using the Add Activation button ( ).

Such modified settings are kept as long as the external mode[9] runs under the control of the ABAP Debugger; when the Debugger is closed, your modified settings are lost.

Even if you don't want to modify activation settings, the Breakpoint Tool gives you valuable information regarding the checkpoint behavior within the current program. The Status line provides a description of the current main operation mode, which may be one of the following:

- No checkpoints contained that can be activated
- Checkpoint Groups Not Active
- Program-Specific Activation
- Group-Specific Activation
- Group-Independent Activation (more on this concept in the upcoming discussion on advanced ASSERT features)

If multiple activation settings apply for the current program and you don't remember the precedence rules, you may find this information helpful for understanding the system behavior.

[9] The *external mode* is the execution context that is related to a user session. When a new main window is opened (e.g., by starting a transaction with "/o" in the ok-code field) a new external mode is opened for this window.

Now that you have a fundamental understanding of assertions, breakpoints, and logpoints, how to use them in your ABAP programs, and how they become effective, let's take some steps toward building on your newfound knowledge by looking at a few of the more advanced features of the ASSERT statement.

Advanced features of the ASSERT statement

In the next sections, we take you through some of the more advanced assertion features that are available, including mode settings for different behaviors, statement additions for logging, and system variants for group-independent activation.

Mode settings

As explained earlier, the mode setting is part of an activation setting and describes what should happen once a corresponding checkpoint has been reached.

For breakpoints and logpoint, the choice is simple: they are either ignored or not; in the latter case, they simply accomplish the actual purpose of the statement:

- Breakpoints interrupt the current computation and the ABAP Debugger is called.

- Logpoints log the information that this point was reached during execution, storing the information passed by the FIELDS and SUBKEY additions, if supplied.

There is not much more useful behavior that you can expect for those two checkpoint types. With assertions, however, things are different. The mode setting for assertions provides for different behaviors for "active" assertions — once an assert condition has been violated, the system can behave in differing ways, depending on the corresponding mode settings:

- **ABORT:** The program execution is aborted with runtime error ASSERTION_FAILED.
- **BREAK:** The program execution is interrupted and ABAP Debugger is called.
- **LOG:** A log entry is created and program execution resumes with the next statement.

As you can see, the activatable assertion combines the features of all of the available checkpoint statements. Like the activatable breakpoint, it can be used to jump into debugging mode. Or like the logpoint, it can be used to store information for later analysis. And like the always-active assertion, it can abort the execution with runtime error ASSERTION_FAILED.

While the abort option is quite obvious, you may be wondering what the other two assertion modes are useful for. Consider these two additional modes as a way to activate assertions gracefully. In BREAK mode, you are free to continue the execution under control of ABAP Debugger. This may be helpful if the implications of a failed assertion will become clear only if you proceed a bit further in the code. Or you may want to use ABAP Debugger's call-stack navigation to try analyzing where things started going wrong. With the LOG mode selected, users may not even notice that assertions have been switched on (except for potential performance degradation, if the assertion conditions imply time-consuming checks). They can proceed with their daily work while you watch out for abnormal system behavior by viewing the assertion log offline.

In any case, the BREAK and LOG modes will help you gain additional information on the runtime state that you may not be able to find in the ASSERTION_FAILED short dump or that may not be so easily extracted from the short dump.

When an activatable breakpoint is reached during background processing, we just ignore it. When it comes to an activatable assertion that is activated in BREAK mode, the situation is different. Detecting an assertion failure is always a noticeable event, whether we're running in batch or dialog. For this reason, we need to specify a second activation mode for background processing, once we've selected BREAK mode. For the other two modes, LOG and ABORT, this distinction between modes for dialog and background processing is neither necessary nor supported, so we end up with the following set of activation modes for the ASSERT statement:

- Inactive
- ABORT (both in dialog and background)
- LOG (both in dialog and background)
- BREAK/ABORT (jump to ABAP Debugger during dialog processing and abort with runtime error during background processing)
- BREAK/LOG (jump to ABAP Debugger during dialog processing and create a log entry during background processing)

In transaction SAAB, you will see these five options when working with mode settings for the activatable ASSERT statement.

Statement additions for logging

The activatable ASSERT statement offers the additions SUBKEY and FIELDS, which we discussed earlier in the section on LOG-POINT statements. Despite the fact that the LOG-POINT statement wasn't introduced until ABAP 7.00 in SAP NetWeaver 2004s, the logging mode for assertions and the corresponding additions to the ASSERT statement have been available since SAP Web AS 6.20.[10]

While the SUBKEY addition applies solely to activation mode LOG and is thus only allowed for activatable assertions, the FIELDS addition can also be specified for always-active assertions. In this case, the content of the

[10] In SAP Web AS 6.20, no deep ABAP variables are supported. This means that you can't specify internal tables or strings (or structures containing internal tables or strings) as logging fields. Consequently, profile parameter abap/aab_log_field_size_limit is also not available and no limit on the amount of data is applied.


```
ASSERT ID  CheckpointGroup  SUBKEY Sub-Key  FIELDS  field1 … fieldN
  CONDITION  logExpr.

ASSERT FIELDS  field1 … fieldN  CONDITION  logExpr.
```

Figure 11 The CONDITION keyword must separate the logical expression from the field list

first eight fields will be given in the text of the short dump — only the values of flat-like (those that aren't deep) variables will be rendered. The same is true for activatable assertions with mode ABORT. Please note that also for always-active assertions, the logical expression must be separated from the field list by specifying keyword CONDITION, as shown in **Figure 11**.

System variants for group-independent activation

While it would make no sense to activate all activatable breakpoints or logpoints in the system at once, regardless of the checkpoint groups they subscribe to, the situation is different when it comes to assertions. Assertions always have a logical expression connected to them and the system behavior will only change if an assertion failure is detected. From this point of view, you may wish to activate all assertions in the system in order to see if any of them fail. Transaction SAAB offers a set of predefined, dedicated global activation variants for this purpose:

- ALL_ABORT_ABORT
- ALL_LOG_LOG
- ALL_BREAK_LOG
- ALL_BREAK_ABORT

The variant names consist of the prefix ALL_ followed by two mode names separated by an underscore. The first mode in the name always applies to dialog processing, the second mode to execution in the background. When you activate such a system variant, a special kind of activation setting is created — this special activation setting applies to all checkpoint groups, including any that are created later. This is called a *group-independent setting*.

Besides the group-specific settings and program-specific settings that we have already discussed, we now have a third kind of activation setting. So it's time for a third precedence rule: Group-independent settings supersede group-specific settings. Program-specific settings supersede both group-independent and group-specific settings.

System variants are used just like "normal" variants — you have the same context options when activating a system variant. The corresponding setting is also displayed in the activation overview list. The only difference from other variants is that you cannot modify system variants. When you copy a system variant into a normal variant, it will be expanded into a list of all checkpoint groups that currently exist in the system.

Once there is a group-independent activation setting with a matching context description, all group-specific settings will be ineffective. Transaction SAAB warns you in case there are group-independent settings, as shown in **Figure 12** on the next page.

With the foundational knowledge you now have in hand, together with the beginnings of some more advanced concepts, you can now start using assertions, breakpoints, and logpoints effectively in your programs to improve your software quality. There are a few important considerations you must always keep in mind in the case of assertions.

Some words of caution when using assertions

As we have stated from the start, checkpoints in ABAP serve to instrument the code but have no impact on the program logic. This is mandatory, because the programmer cannot know whether checkpoint state-

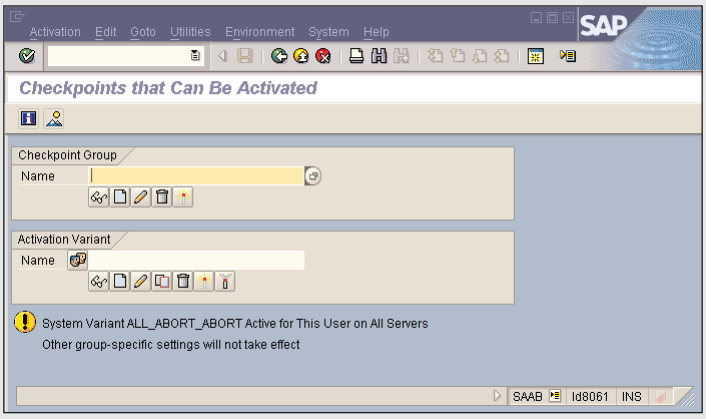


Figure 12 Group-independent activation warning

ments will be active when the program is executed in a production environment: Code must behave identically, regardless of checkpoint activation settings.

While this is obviously the case for activatable breakpoints and logpoints, assertions deserve special diligence here. For example, an assertion condition may contain the call of a method. If this method has side effects, the program behavior may be different depending on whether checking for this assertion is switched on or off.

Do not rely on an activatable assertion being executed. Do not rely on such an assertion *not* being executed, either. Also, do not rely on a specific activation mode. There is no way to restrict the mode to a given subset. The same rules should be applied to always-active assertions. Chances are good that someone will change an always-active assertion into an activatable one for some reason. If your code relies on side effects imposed by the evaluation of the assert condition, you will run into problems.

Conclusion

This article has shown you how using checkpoint statements in your ABAP code helps you write functionally correct code and facilitates code maintenance. The `ASSERT` statement enables you to check your assumptions during code execution. The `BREAK-POINT` and `LOG-POINT` statements facilitate the analysis of your program's behavior. All three checkpoint statements, when used in their activatable flavors, serve as an annotation in your code; using cross-referenced information on the different checkpoint groups may help you to find related code fragments in your application.

Gerd Kluger studied computer science at the University of Kaiserslautern, Germany. After receiving his degree, he worked for a company whose main focus was the development of programming languages for business applications. He was responsible for the development of the compiler and programming environment for the object-oriented programming language Eiffel. Gerd joined SAP in 1998, and since then has been working in the SAP NetWeaver Foundation ABAP group. His main responsibility is in the development of ABAP Objects, the new class-based exception concept, and the further development of system interfaces, especially with regard to the file system. You may reach him at gerd.kluger@sap.com.

Wolf Hagen Thümmel studied physics and received his doctorate at the University of Karlsruhe, Germany. After a year of postdoctoral research work in the field of experimental particle physics, he joined SAP in 2001 as a member of the SAP NetWeaver Foundation ABAP group. Wolf Hagen works on tools for the ABAP language in the areas of debugging, tracing, and memory analysis. You may reach him at wolf.hagen.thuemmel@sap.com.

