

# VAssert Programming Guide

Technical Note for Workstation 6.5 RC

---

This document introduces VAssert, an easy-to-use API that helps you investigate the behavior of applications running on virtual machines, after the applications run.

VAssert allows you to insert replay-only code into your applications, incurring minimal performance overhead at runtime. After you record a virtual machine session, you can replay selected portions while performing data consistency checks and logging. VAssert code can be shipped in released software, where it normally remains inactive and unobtrusive. After you employ VMware Record and Replay technology to capture an application failure, you can activate VAssert code to provide root cause analysis.



**CAUTION** The VAssert API is experimental in this release. Interfaces might change in subsequent releases, and backward compatibility is not guaranteed.

---

## About VAssert

In this release, the VAssert SDK is released for Windows guests only. It supports C++ and C programming. Tested guest operating systems include Windows 2000, Windows XP, and Windows Vista, 32-bit and 64-bit. Windows 2003 Server and Windows 2008 Server were not verified.

VAssert depends on the Record and Replay facility developed for VMware Workstation 6. In the 6.5 release, you can insert markers while recording or replaying a session and quickly navigate to the markers during replay. You can also browse through a recording and choose the starting point for replay.

In addition to marker navigation, in this release you can insert program statements that initialize the VAssert API, perform consistency checks, and log messages to a file, all at replay time only. The performance penalty of using VAssert is encountered almost entirely at replay time. Replay is deterministic (predictably the same) whether or not VAssert statements are present and enabled.

## Hypothetical Use Case

Here is a scenario where VAssert could be useful.

- 1 A customer reports that an application fails, but only after an extended period of heavy use. You cannot reproduce the failure on your development system.
- 2 You add checking and logging statements, as described in [“Using VAssert”](#) on page 2. You recompile the application and deliver it to the customer for test debugging.
- 3 The customer enables VAssert and begins recording on Monday morning, a time of heavy use. Because VAssert recording is extremely lightweight at runtime, users are unlikely to notice the replacement application. Similar failures are likely to reoccur. The application does fail again, but the customer quickly restarts it so as not to interrupt service, and turns off recording.
- 4 The customer sends you Friday’s recording. Because the recording was made with VAssert enabled, extra checking occurs during replay, and a session log is available. You are able to diagnose the problem. (The recording must come with packaged-up virtual machine, and hardware configuration must be similar.)

VAssert-enabled applications can even run on native hardware (a physical machine without any virtual machines involved). This is not useful for debugging, but it is advantageous that the same binary works in both physical and virtual environments.

## Traditional Assertions and Logging

Two time-honored methods of debugging are to insert print statements, and to add assertions that verify programming assumptions. The VAssert API provides both methods.

The traditional `assert()` statement is a preprocessor macro defined by including `assert.h` in a C program. If its contained expression evaluates false, `assert()` writes the expression, source filename, and line number to standard error, then calls `abort()` to end the process and possibly produce a memory image. If disabled by `NDEBUG`, `assert()` has no effect.

The `VAssert_Assert()` statement is similar, but takes effect deterministically at replay time, unless you specify otherwise. Like traditional `assert()`, `VAssert_Assert()` is a macro.

Most system software sends activity and security-related messages to a log file in a well-known location, and application software often does the same. The `VAssert_Log()` statement sends messages to the `vlog.txt` file in the virtual machine's directory at replay time, unless you specify otherwise. `VAssert_Log()` is also a macro.

---

**NOTE** VAssert statements must be enabled at record time. See [“Enabling VAssert Recording”](#) on page 3.

---

## Best Practices

Although VAssert was designed with debugging in mind, you can also use it to study performance and use patterns after the fact. The logging facility is especially useful for this.

### Offline Testing

You could use VAssert to improve the efficiency of automated offline testing. Software developers can include elaborate assertion checking and extensive logging in their code. Normally, all the checking and logging I/O would alter application behavior due to its performance overhead. But with VAssert, in-house QA engineers can record testing sessions without suffering from slow performance. Later, perhaps that night, the recorded sessions can be replayed with expensive checking and logging activated. Application replay can be completely automated. The next morning, the developers can investigate and debug failures, if any.

### Performance Analysis

To analyze performance of a database buffer pool, VAssert code could be added to collect an access history log of the database buffer pool. This logging overhead would normally skew the results if performed live online. With VAssert, you collect the log afterwards, without disturbing the access history itself. Using that data, the access history could be analyzed to design a better buffer pool replacement algorithm.

## Using VAssert

This section describes how to install, locate, and enable VAssert.

### Installing VMware Tools

When you install VMware Workstation, a VMware Tools directory is created on the host. It contains an upgrader application to create a complete VMware Tools package on guest virtual machines.

#### To Install the VAssert SDK

After installing a Windows guest operating system, click **VM > Install VMware Tools** and follow documented procedures to install useful software in the VMware tools directory on the guest. On Windows guests, this folder is `C:\Program Files\VMware\VMware Tools` by default.

Navigate to the `VMware Tools` directory and double-click `VMwareToolsUpgrader.exe` to run the application. Select **Modify > VAssert SDK** to install the VAssert library on the guest virtual machine.

## Locating the VAssert SDK

With VMware Tools installed, find the VMware Tools directory on your guest operating system. Locate the `VAssert\lib32` or `VAssert\lib64` subdirectory, whichever is appropriate, to find the following SDK files:

- `libVAssert.dll` – dynamic link library for VAssert run-time.
- `libVAssert.lib` – link-time library for VAssert programs.
- `libVAssertTest.c` – sample application with command-line options for VAssert statements.
- `vassert.h` – header include file for VAssert programs.
- `vassertShared.h` – internal file used by `vassert.h`, not important for programmers.

The VAssert SDK was tested with Microsoft Visual C++ 6.0, although later versions of Visual C++ are known to work. You need the SDK to modify your programs for debugging with VAssert. Add the header files to your project, and link with the `libVAssert.lib` file.

## Enabling VAssert Recording

VAssert functionality is disabled by default, because it incurs a small amount of overhead in the guest operating system, whether or not applications are using VAssert features.

### To Enable VAssert

- 1 In the Workstation GUI, select **VM > Settings > Options > Snapshot/Replay**.
- 2 Click **Enable execution record and replay**, if not already enabled.
- 3 Click **Enable VAssert**, and click **OK** to dismiss the dialog box.

After enabling, any recorded application that uses the VAssert SDK will execute replay-time code when replayed, regardless of the check box setting at the time of replay. Recordings created when the check box was disabled can never run replay-time code, whether or not applications used the VAssert SDK.

Internally, enabling the VAssert check box creates these two settings in the guest virtual machine's `*.vmx` file. The first turns on VAssert features, while the second permits internal communication.

```
vassert.enabled = "TRUE"
isolation.tools.vassert.disable = "FALSE"
```

One related setting is not check box enabled in this release. It permits use of the VAssert statements to start and stop recording. See [“Using VAssert\\_StartRecording\(\)”](#) on page 4 and [“Using VAssert\\_StopRecording\(\)”](#) on page 5.

```
isolation.tools.stateLoggerControl.disable = "FALSE"
```

## Programming VAssert

This section describes how to call statements in the VAssert API.

### Initialization

On a supported Windows guest virtual machine with VMware Tools installed, and `vassert.h` header file included in a C or C++ program, `VAssert_Init()` loads the DLL and initializes the library:

```
#include "vassert.h"
...
if (!VAssert_Init()) {
    cout << "Warning: cannot initialize VAssert library." << endl;
    goto continue_without_vassert;
}
```

This function fails if `isolation.tools.vassert.disable` is set TRUE. If you want to leave VAsserts in production code and enable them without recompiling, your code can ignore the return code of `VAssert_Init()`.

## Using VAssert\_StartRecording()

You can use the regular Record and Stop facility in the Workstation UI to record all or part of a VAssert-enabled application running on a guest virtual machine. However you can get finer-grained control by placing a `VAssert_StartRecording()` statement exactly where you want it in your program.

```
if (!VAssert_StartRecording()) {
    cout << "Warning: cannot start VAssert recording." << endl;
}
```

A virtual machine must be powered on to record. As you would expect, it is an error to start recording when a recording is already underway, or during replay.

You probably want to match `VAssert_StartRecording()` with a `VAssert_StopRecording()` statement.

## Using VAssert\_Assert()

If VAssert was enabled during recording, at replay time the `VAssert_Assert()` macro evaluates its expression, and terminates the application if the expression is false.

For example, before calling functions that expect a pointer to actual data, and might crash given a null pointer, you can assert that the pointer is non-null:

```
VAssert_Assert(ptr != 0);
```

As another example, you could check for memory problems inside the long-running program mentioned in [“Hypothetical Use Case”](#) on page 1 by asserting that memory allocation is within range after `malloc()`:

```
VAssert_Assert(((char*)p) >= min_address);
VAssert_Assert(((char*)p) <= max_address);
```

In some cases you might want to replace the underlying `abort()` routine so the program does not really fail, but instead provides helpful diagnostic information. See the define option `VASSERT_CUSTOM_ABORT` in [“Compiler Flags”](#) on page 6.

## Going Live During Replay

The Record and Replay facility allows you to interrupt an in-progress replay at any time and begin interacting with it. You can control this with the **Go Live** button in the user interface, or programmatically with VAssert. At the point in your code where you want the application to go live, insert the following statement:

```
VAssert_Assert(FALSE);
```

The `vassert.h` include file defines `VAssert_GoLiveMain()`, which `VAssert_Assert()` calls internally.

## Advanced Use – Return to Replay

You can return to replay with the `VAssert_ReturnToReplayMain()` statement, if inside a VAssert code block. First check to make sure the application is in replay state, otherwise this statement might crash the application. Your program does not have to clean up program state, which VAssert restores before returning to replay, thereby eliminating side-effects from the `VAssert_Assert()` code block.

```
if (vassert_state.inReplay)
    VAssert_ReturnToReplayMain();
```

One possible use for this statement would be to exit early from a VAssert block. For example, in the code sequence below, `VAssert_ReturnToReplayMain()` exits both `firstFunction()` and `secondFunction()` without returning back through the calling code path.

```
VAssert_Assert(firstFunction()); // calls
→ firstFunction(); // which calls
  → secondFunction(); // which calls
    → VAssert_ReturnToReplayMain(); // end current VAssert and resume replaying
```

## Using VAssert\_Log()

If VAssert was enabled during recording, at replay time the VAssert\_Log() macro sends printf() style output to the vlog.txt file on the VMware host for its respective virtual machine. A Windows XP guest on a Windows host would send output to C:\My Documents\My Virtual Machines\WindowsXP\vlog.txt or some similar path name. On Windows VAssert\_Log() calls \_vsprintf() internally, which is slightly different from the POSIX standard printf(). See the MSDN for reference information:

[http://msdn2.microsoft.com/en-us/library/56e442dc\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/56e442dc(VS.71).aspx)

Because VAssert\_Log() is implemented as a preprocessor macro, you must place the printf() style string inside double parentheses:

```
VAssert_Log(("Customer transaction number %u at %s\n", transactNum, ctime(&time) ));
```

This logging statement is useful for instrumenting your application without causing it to exit. For example, the long-running program in “Hypothetical Use Case” on page 1 could write a log message before every operation with high overhead, giving you an idea where it failed.

See the limitation “Floating Point Numbers” on page 8 about printing %f and %g format numbers.

## Using VAssert\_StopRecording()

If you are interested in only a certain section of your application, you can stop the recording in progress with a VAssert\_StopRecording() statement. Otherwise recording starts from the VAssert\_StartRecording() statement and continues to the end of program execution.

```
if (!VAssert_StartRecording()) {
    cout << "Warning: cannot stop VAssert recording." << endl;
}
```

## VAssert Reference

This reference includes the following sections:

- “API Statements” on page 5
- “Compiler Flags” on page 6
- “Configuration Options” on page 7
- “Limitations” on page 7

## API Statements

This release has three documented APIs:

### Initialize

The initialization function returns TRUE if it was able to initialize the VAssert library, otherwise FALSE.

```
#include "vassert.h"
extern char VAssert_Init(void);
```

### Record

This statement begins the underlying system recording facility, sending to a snapshot object.

```
#define VAssert_StartRecording() VAssert_SetRecordingMain(1)
```

This function is effective only when the **VAssert** check box is enabled in the Workstation options dialog box.

## Assert

If `<expression>` evaluates `FALSE`, this statement calls the system `abort()` routine to terminate the process, which you can attach with a debugger to determine the filename and line number of the failing assertion.

```
#define VAssert_Assert(<expression>) ...
```

It is possible to replace both the `assert()` routine and the `abort()` routine it calls; see [“Compiler Flags”](#) on page 6. You could for instance replace the `abort()` routine with one that calls another routine to emit `__FILE__` and `__LINE__` variables from the preprocessor.

## Log

The log statement sends formatted output to the `vlog.txt` file of its respective virtual machine on the host.

```
#define VAssert_Log((<printf-string>)) ...
```

See the limitation [“Floating Point Numbers”](#) on page 8 about printing `%f` and `%g` format numbers.

## Stop

This statement ends the underlying system recording facility, closing the snapshot object.

```
#define VAssert_StopRecording() VAssert_SetRecordingMain(0)
```

This function is effective only when the **VAssert** check box is enabled in the Workstation options dialog box.

## Compiler Flags

In the SDK, compiling with the `VASSERT_ALWAYS_EXECUTE` flag causes the `VAssert_Assert()` and `VAssert_Log()` calls to be treated as normal `assert` and `log` statements at runtime. In other words, they execute all the time, like standard `assert()`, not merely at replay time.

Within the SDK, the programmer can define the following macros before including the `vassert.h` header file in a program. These macros specify replacement `abort`, `assert`, and `log` statements:

- `VASSERT_CUSTOM_ABORT` – When an assertion fails, VAssert ends replay and makes the virtual machine go live, so it begins accepting new input from the user. When this occurs, `VAssert_Assert()` inserts the `VASSERT_CUSTOM_ABORT` macro to halt the application and report an error. This macro by default calls the standard library `abort()`, which prints a generic error message and allows you to attach a debugger to get details about the failure. You can alter this macro to provide more information, such as the `__FILE__` and `__LINE__` number of the failed assertion. Because `VASSERT_CUSTOM_ABORT` is inserted after the end of replay, I/O restrictions given in [“Limitations”](#) on page 7 do not apply. This macro can print output.
- `VASSERT_CUSTOM_ASSERT` – With the `VASSERT_ALWAYS_EXECUTE` flag defined at compile time, `VAssert_Assert()` is defined as a wrapper around `VASSERT_CUSTOM_ASSERT`, whose default definition is to call the standard library `assert()`.
- `VASSERT_CUSTOM_LOG` – With the `VASSERT_ALWAYS_EXECUTE` flag defined at compile time, `VAssert_Log()` is defined as a wrapper around `VASSERT_CUSTOM_LOG`, whose default definition is to call the standard library `printf()`.

## Configuration Options

Table 1 shows configuration options supported in the \*.vmx file to control the behavior of VAssert. Options to control the logging behavior of VAssert\_Log() are especially useful for customizing replay.

**Table 1.** VAssert Configuration Options

Option Name	Settings	Default	Description
vassert.enabled	TRUE FALSE	FALSE	Determines whether VAssert functionality is enabled for new recordings. The default FALSE means that VAssert is not enabled.
isolation.tools.vassert.disable	TRUE FALSE	TRUE	Determines whether VAssert functionality is exposed to guests. Use this only in conjunction with vassert.enabled. The default TRUE means VAssert functionality is not exposed to guests.
vassert.logFile	<filename>	vlog.txt	Destination log filename for VAssert_Log() output, created in the guest virtual machine's directory on the host.
vassert.logMode	"prompt" "append" "replace" "discard"	"prompt"	Indicates what to do when a VAssert_Log() produces output, and the log file already exists. "append" means always append to the log file. "replace" means always replace the log file. "discard" means always preserve the old log file and discard any new log output. "prompt" means always prompt with a dialog box so the user can select one of the above options.
vassert.maxLogFileSize	<number>	-1	Limits the log output file to a maximum file size in bytes. The default -1 means no maximum limit, so the log file can grow as much as needed.
vassert.maxTime	<number>	10	Limits the amount of time spent executing a VAssert_Assert() invocation to the given number of seconds. The default limit is 10 seconds. Any invocation lasting longer will be terminated. Specifying zero means that no limit is enforced. The imposed limit is not exact. It is never less than the specified duration, although it could be more.

## Limitations

Replay-time code running with VAssert has the following limitations:

- Such code cannot perform I/O or cause I/O to be performed. It cannot execute `in` and `out` assembly instructions, nor can it read or write memory-mapped regions. Implicit I/O might result from swapping, so do not run VAssert code near physical memory limits. When the VAssert library detects I/O, it terminates and skips the offending replay-time code. Subsequent VAssert execution remains undisturbed.

Attempting to display output with a `VAssert_Assert()` does not work, because calling `printf()` or popping up a dialog box inherently produces I/O. The only way to produce user-visible output is through `VAssert_Log()`, or by defining `VASSERT_CUSTOM_ABORT`, which is allowed to produce I/O because it executes after an assertion has failed and replay has stopped.

- Replay-time code cannot depend on the delivery of interrupts. For example, timer interrupts and I/O interrupts are not delivered during replay-time execution.
- If a guest application terminates abruptly, it is possible for the VAssert DLL to de-initialize improperly. This might happen when you forcibly terminate processes with Windows Task Manager. If improper de-initialization occurs, spurious VAssert invocations might cause replay glitches. At worst, spurious invocations can cause replay to halt indefinitely, although you can manually stop replaying if you notice that progress has terminated. Normally, spurious VAssert invocations have no noticeable effect during replay, and valid VAssert invocations run undisturbed. When the VAssert DLL re-initializes because of a `VAssert_Init()` call in the recording, it prevents spurious invocations during further replay.

- Replay-time code should not modify privileged guest CPU state. This includes CR0, CR4, and Machine Specific Registers. However, modifications to normal non-privileged CPU state such as the CPU general purpose registers and flags, the segment registers, and address space modifications (that is, assignments to CR3) are permitted.
- When replaying a recording that has VAssert enabled, the **Suspend** and **Add Marker** operations are disabled in the Workstation user interface.

## Floating Point Numbers

The `VAssert_Log()` statement might fail when printing floating point numbers expressed in `%f` or `%g` format.

## Performance Implications

The VAssert APIs defer execution of some code from recording to replay. This generally makes the record-time execution of a program faster, when VAssert calls are not evaluated, and the replay-time execution slower, when VAssert calls are evaluated. To implement this facility, the VAssert library must add overhead to the replay-time execution. Testing indicates overheads of hundreds of microseconds per VAssert. Depending on the rate of VAssert invocations per second, this can make replay-time execution much slower than record-time execution. For example, if during recording, your program runs 15 million VAssert invocations in one second, replay-time execution could take substantially longer (15 million  $\times$  500 microseconds = over 2 hours).

Users should plan for these performance slow-downs either by using more complex, but more sparsely packed VAsserts, or by scheduling an appropriate amount of time to perform replay (such as overnight, or batched).

## Support and Feedback

See the “Record and Replay” chapter in the *Workstation 6.5 User’s Manual* for information about the underlying facility and its user interface.

If you encounter any issues while programming VAssert, you are welcome to post questions in the VMware Workstation 6.5 Beta forum, or in VMware communities that discuss developer products, such as this one:

<http://communities.vmware.com/community/developer>

---

If you have comments about this documentation, submit your feedback to: [docfeedback@vmware.com](mailto:docfeedback@vmware.com)

**VMware, Inc. 3401 Hillview Ave., Palo Alto, CA 94304 [www.vmware.com](http://www.vmware.com)**

Copyright © 2008 VMware, Inc. All rights reserved. Protected by one or more of U.S. Patent Nos. 6,397,242, 6,496,847, 6,704,925, 6,711,672, 6,725,289, 6,735,601, 6,785,886, 6,789,156, 6,795,966, 6,880,022, 6,944,699, 6,961,806, 6,961,941, 7,069,413, 7,082,598, 7,089,377, 7,111,086, 7,111,145, 7,117,481, 7,149,843, 7,155,558, 7,222,221, 7,260,815, 7,260,820, 7,269,683, 7,275,136, 7,277,998, 7,277,999, 7,278,030, 7,281,102, and 7,290,253; patents pending. VMware, the VMware “boxes” logo and design, Virtual SMP and VMotion are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

Revision 20080815, Item EN-000079-00

---