# Memory Debugging for C and C++ Programs in IBM LAN NetView

March 10, 1994

Steve Hargis
Mike Skelton
LAN SYSTEMS PRODUCTS
Personal Software Products
Austin, Texas

# Trademarks

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM′s intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user′s responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not imply giving license to these patents. Written license inquiries may be sent to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

| | |
|---|---|
| IBM Corporation | C Set/2, C/C++ Tools, IBM, LAN NetView, LAN NetView Fix, LAN NetView Manage, LAN NetView Monitor, LAN NetView Scan, LAN NetView Tie, OS/2, SPM/2, Theseus2 |
| X/Open Company Ltd. | X/Open Management Protocol |

# Contents

# Abstract

This paper reviews memory leak detection and fixes as experienced on the IBM LAN NetView product with the XMP (X/Open Management Protocol) API (Application Programming Interface). Developing memory neutral (memory leak free) LAN NetView applications or debugging memory leaks in LAN NetView applications will both be aided by the information and techniques contained in the paper. The use of facilities internal to LAN NetView to detect and fix memory leaks are demonstrated. The paper also discusses why memory leaks were a concern, what was done to detect and fix them, and how to avoid memory leak problems.

Software designers, developers, testers, performance analysts or others who are responsible for producing memory neutral LAN NetView applications will benefit from the design and programming recommendations, and tools described in this paper. It is our hope that the techniques used in this paper will become part of the repertoire of every developer and that the tools covered here will become part of the toolbox of every analyst.

**IMPORTANT NOTICE**

This paper is intended to be read in conjunction with its companion paper “Memory Debugging for C and C++ Programs” by the same authors. It should be available from the same source as this paper. The companion paper reviews memory leak detection and fixes as experienced with the IBM C Set/2, IBM C/C++ Tools, and other compilers with Theseus2.

# Introduction

Creating an excellent software product includes two goals. First, is making the product excellent through good design and development. Second, is iterating through comprehensive testing until all problems are fixed. We will focus on these goals as they pertain to memory management in LAN NetView applications.

We cover the methodology and tools needed to be proactive/preventative in getting memory management right during design, code and unit test. We also address the methodology and tools needed to effectively react to memory management bugs found in the integration and system test phases.

It seems only fair that we disclose any bias that may occur as we write. This white paper is written by members of a performance team; that is not to say that some of us do not have any code in LAN NetView, we do. It does indicate that we highly value design and analysis before development in order to avoid errors and more analysis during development and test. As we look at code we consider its resource usage (i.e. CPU time, disk, memory, communication channels) and effects on other applications that will run concurrently. Thus we value low resource utilization over fast development and economy of resources over easy-to-code solutions. It is our experience that adherence to disciplined design and development will lead to increased software productivity and shorter development and test cycles.

Teachers are often told to teach by 1) telling the students what you are going to teach them; 2) teach them; and 3) tell them what you taught them. Thus, after some basic definitions (the first section), the paper lists the lessons that we learned (the second section) about memory debugging as it relates to developing LAN NetView applications. We try to be clear and explicit. Following the introductory sections are two sections concerning the XMP and XOM APIs. The first of these two sections discusses common uses of the APIs and the second section documents the XMP API trace facility.

# Definitions

**Memory Leak**

A memory leak is memory a program allocates or that is allocated on a program's behalf, that is not freed after the program is finished with it. If the same action is executed again, the program allocates more memory instead of using memory previously allocated. With repeated executions the program accumulates more memory than it needs. The accumulation of memory by the leaking program prevents other applications and system functions from using the memory, thereby interfering with the systems operation.

Memory leaks are usage dependent, like any programming bug. A program that has not leaked when running one set of test cases may leak when running a different test case.

**Memory Overwrite**

Also called a memory walker. A memory overwrite occurs when a program allocates less memory than it actually writes to a location. The effect is to overwrite and possibly wipe out data that happens to be located immediately after it in memory. Whether or not a memory overwrite occurs depends on how many additional bytes are stored beyond the amount allocated. The observable symptom is intermittent data corruption, which is dependent upon the length of data stored in memory (which is not an intuitive item to check). How the length of data is related to memory overwrites is the subject of an example in the companion paper to this one (see the reference to the companion paper in the abstract).

**Memory Neutrality**

A module, component, function, API, program, system, etc. is said to be memory neutral if it

1. deallocates all memory for which it is responsible (this is usually the same memory that it allocates, however some specification standards specify that function *A* is supposed to deallocate memory allocated by function *B*); and
2. does not attempt to deallocate more memory than that for which it is responsible.

Both of these items must be true for memory neutrality. The basic definition is straight forward: Deallocate all, and only the memory that you allocate. However, when specifications call for different modules to delete memory than allocated the memory, the definition gets a little fuzzy in order to include such cases. The "memory than that for which it is responsible" wording takes care of unusual specifications.

# Design and Programming Recommendations

Our experience with LAN NetView motivated creation of a set of design and programming recommendations that focus specifically on memory management. Their purpose is to be proactive during design, code, unit test and functional test so that discovery of memory problems, in complex system tests, is limited to genuinely complex problems that could not reasonably be found in simple test environments. The earlier in the product development that these recommendations are applied the smoother (and faster) development will go.

**Know the memory usage of each component within the product.**

A development approach that has gained much popularity is prototyping code. A quick prototype is used to prove that a concept will work, with the expectation that the lessons learned from the prototype will be kept but the prototype itself will be discarded and "real," quality code will be written for the product. However, schedules are often aggressively short and the prototype becomes the product. We have noticed that a major short cut taken in prototypes is in memory management. And why not? The prototype runs for a short time using whatever memory is needed and ends, thus freeing any memory used. As the accumulation of prototypes comes together as a product there has been neither a design for memory management nor a prototype for memory management.

It is difficult to diagnose memory problems when no one has the big picture of how the product is supposed to use memory. The logical person to know what is supposed to happen is the developer, but he may need assistance in discovering what is really happening. Diagnosing memory problems usually requires an analyst using special software tools to provide an "under the covers" look at program behavior AND the developer to provide knowledge about what is supposed to be happening. The analyst provides expertise on what is actually occurring and the developer provides expertise on what is supposed to be occurring. Together their expertise can be used to make the product behave as it was designed. Ideally, the analyst and the developer are the same person.

**Do not rely on the operating system to delete memory when the process dies.**

This is a common practice with, what would seem, no ill effects. After all, if you need the memory until the process dies, why not let the operating system get rid of it for you? That the operating system will deallocate resources when the process dies is true; however, counting on that only works for relatively short-lived processes. A short running utility can count on that, but major processes that stay up for days, weeks or months should not be implemented that way. There are four main reasons 24/7 (24 hours a day, 7 days a week) processes (like LAN NetView applications), and arguably even shorter lived processes, should not rely on the operating system to delete resources.

The first is that resource consumption will grow over time; if the time is short-lived and the resource consumption is small it may be OK, but if the time is long then the resources consumed will to begin reach the maximum of the machine. For example, if memory is allocated to store information until a process dies and the process does not die for 3 weeks, the information stored will consume a lot of memory. Consuming lots of memory will cause some memory somewhere (wherever it is used the least) to be swapped. It will not take very long for the swap file to be so large that it adversely affects the performance of everything that runs on the machine - all because memory usage was not tracked when allocations began so the memory could be freed or reused when previous use has ended.

The second reason is that code may be ported or expected to run on a different operating system/platform. The new platform may not clean up resources as nicely as OS/2. Or perhaps the other platform does clean up nicely, today; but it may not in the next release.

A third reason to not rely on the operating system to delete memory when a process dies is that the program′s design may change. Code may be written to run in a process of it′s own, however the process/thread model of products are subject to change. Code may be consolidated from several processes into a smaller number of longer lived processes in order to avoid process overhead and process creation, deletion overhead. Therefore, code that started out in a process of its own may be consolidated to run on a thread(s) of a process containing other threads. Resources like memory,

semaphores, files etc., are owned by the process, not by the thread. So, when the code now terminates the thread, the process remains - along with all of the resources accumulated by the thread but never freed.

The last reason for not relying on the operating system to delete resources is that debugging is greatly complicated by it. We strongly advocate use of the C Set/2 and C/C++ debug memory functions; when using them we have noticed that many programs are not memory neutral at the instruction immediately prior to returning control to the operating system. So if we are trying to fix a memory leak we must first decipher through the "noise" of memory that is being leaked "on purpose" and memory that is being leaked accidentally. This as a major hindrance to debugging.

Let us close this piece of advice by saying that sound coding practices include allocating machine resources as needed, tracking those resources and deallocating them at the earliest reasonable time. Relying on the operating system to clean up a process′ use of resources is dangerous in todays cross-platform code environment.

**Avoid designs where one module allocates memory and another module deallocates the memory.**

Unless of course, the design specification comes from a standards institute and you have no control over it (like the XMP APIs). This sort of design can lead to problems such as:

1. Poor communications between memory allocation and deallocation modules, or programmers will lead to memory leaks.
2. When more than one C runtime exists, each runtime will need APIs to free memory and run _heapmin. Then memory allocated by a module compiled with one runtime can be called, by a module compiled with another runtime, to deallocate the memory. (_heapmin is a C runtime function that attempts to return unused heap memory to the operating system. It is an expensive call, thus it should not be run frequently. The algorithm for when it successfully returns memory to the operating system is complex. We recommend running _heapmin after a significant amount of processing in memory has completed and a lull of activity in the system is expected; this way the time to execute _heapmin may go unnoticed by the system users.)
3. Very few people, if anyone, will really know who is allocating what, for what, and who can deallocate or reuse what, under what conditions; and communicating that to fellow team members will be extremely difficult, not to mention the difficulties in answering/debugging memory usage questions.

**Minimize the number of different compilers used in the product.**

Each compiler (and usually each version of a single compiler) has it own runtime library and runtime libraries do not communicate with each other. Inevitably, a module compiled with compiler *x* will want to know about memory structures compiled with compiler *y*. They will not be able to communicate unless you code that ability; that seems like a waste of good time. Multiple compilers will also complicate your build environment and your builds.

**Set the criteria for determining memory neutrality early in the development cycle.**

Testing for memory neutrality can consume some time to create a build with memory debug features in it, execute test cases and analyze the data, but the results are worth the effort. If the memory usage is correct this analysis goes quickly. Our strong recommendation is that you use the debug memory features available in C Set/2 and C/C++ Tools in order to have an authoritative (the compiler knows best) source certify that your code is memory neutral. We recommend that memory neutrality be a test exit criteria beginning with functional verification test. If code has memory problems, finding and fixing them in a small environment is much easier than waiting until tests in a large, complex environment. Since memory use bugs are usage dependent, like any other code bug, some memory usage bugs may not show up until the code is used in a large, complex environment.

# Use of the XMP Interface

A memory neutral use of the XMP APIs can be difficult unless one clearly understands the relationships between each API and its use of memory. This section explicitly relates each API and its memory usage; it also notes eight miscellaneous facts that are useful when using the APIs. Below are the miscellaneous notes about the use/misuse of the XMP APIs.

- XMP runs on the callers thread.
- XMP is written in C and uses C memory functions. These memory functions allocate memory on the heap. The heap is from the users thread, therefore this memory usage is owned by the users process.
- XMP never calls _heapmin, but at_heapmin can be used to run _heapmin in the C runtime that XMP uses.
- om_create allocates memory to do the create. When the application or agent is through using the object created, the created object needs to be deleted (om_delete) to free the memory.
- om_get makes a copy (service generated public object) of the private object requested. When the application or agent is through using the public object, then the public object needs to be deleted (om_delete).
- Convenience routines **may** allocate memory which may later need to be deleted. For example at_oid_to_str needs an at_free.
- mp_shutdown frees only service generated private objects. It does not free memory used by public objects or convenience routines. Delete will still function after mp_shutdown.
- om_put - The default usage is UNSAFE. This means that if something is wrong with the data passed, an error will be returned but the object is not returned to the state it was in before the om_put that caused the error. A convenience routine exists to make a put safe in the sense that a copy of the object is made before the om_put is attempted. If an error occurs the object can be returned to the state it was in before the om_put. The safe function uses more memory and time than the default unsafe.

These are the rules to follow relating to cleanup after calling various XMP, XOM and convenience routines:

***Convenience Routines***

| | |
|---|---|
| at_activate_all_packages: | at_free(pfeature) |
| at_get_token_length: | (none) |
| at_oid_match: | (none) |
| at_oid_to_label: | at_free(label)<br>at_free(package) |
| at_oid_to_str: | at_free(string) |
| at_str_to_oid: | at_free(oid.elements) |
| at_type_to_label: | at_free(label) |

***XMP Functions***

| | |
|---|---|
| mp_bind: | mp_unbind session<br>(also see mp_status) |
| mp_action_req: | if mp_status OK<br>om_delete (action_reply)<br>(also see mp_status) |

| | |
|---|---|
| mp_create_req: | if mp_status OK<br>om_delete (create_reply)<br>(also see mp_status) |
| mp_delete_req: | if mp_status OK<br>om_delete (delete_reply)<br>(also see mp_status) |
| mp_error_message: | (none) |
| mp_event_report_req: | if mp_status OK<br>om_delete (action_reply)<br>(also see mp_status) |
| mp_initialize: | mp_shutdown (workspace) |
| mp_receive: | if mp_status OK<br>om_delete (received_thing)<br>(also see mp_status) |
| mp_status: | if ( (mp_status < MP_SUCCESS) \|\|<br>(mp_status > MP_INSUFFICIENT_RESOURCES) )<br>om_delete (mp_status) |
| mp_version: | (see mp_status) |
| mp_wait: | (see mp_status) |
| ***XOM Functions*** | |
| om_create: | om_delete(whatever_you_create) |
| om_decode: | om_delete(private_form_decoded_thing) |
| om_encode: | om_delete(private_form_encoding_object) |
| om_get: | om_delete(public_form_whatever_you_got) |
| om_instance: | (none) |
| om_put: | (none for sure, but you might be able to delete the source) |

## IBM LAN NetView XMP Trace Facility

The XMP trace facility can be used to debug the use of the XMP, XOM and convenience routine APIs in LAN NetView. This includes all APIs that begin with _mp, _om or _at. The trace is turned on by defining an environment variable in the session where you will start LAN NetView before any programs are executed, or you can use special APIs to control the trace levels. The environment variable is TRACE_MASK and it can have several values depending on the level of trace that you want. TRACE_PATH is an environment variable used to set the directory where trace files will be saved. If this variable is not used then the current directory will contain the trace files. Unless overridden, the trace output files are named according to the following pattern: *pid_tid*.TRC. “Using the XMP Trace Facility” on page 7 contains a reference guide to the XMP trace facility.

The XMP trace facility yields information for each function called. It also nests the calls so that you can easily see the hierarchical nature of the calls. Pairing up calls which allocate memory and calls which deallocate memory is greatly aided by the lists in “Use of the XMP Interface” on page 5. Turning this trace on will require a lot of disk space to store the trace information; make sure that the disk where you execute LAN NetView has lots of room.

# Using the XMP Trace Facility

XMP provides a mechanism to trace through every internal call that XMP makes. This can be a great aid in debugging a management program.

The XMP trace facility provides two different methods of use:

**Setting trace environment variables**
This method allows you simply to toggle tracing on or off.

**Using trace function calls and macros**
This method allows you to change dynamically from your management program the trace environment and the amount of trace data that is returned.

## Trace File Names

Trace files typically use the naming convention that identifies the process and thread that is being traced. For example, a trace of the second thread of process 51 would generate a trace file named 51_2.TRC. To determine the name of the executable executing as process 51 execute the OS/2 command PSTAT and look under the label "Process ID" for 0051. The process number is a hexadecimal number. This default file naming convention can be overridden as described in "Using the Trace Function Calls and Macro" on page 8.

# Using the Trace Environment Variables

To enable or disable tracing and create a directory for trace data files, use the following environment variables:

*TRACE_MASK*
This environment variable is used to enable or disable tracing and request different levels of tracing. Valid values for this variable are the following:

*LEVEL_ALL*
Generates trace data for all routines.

*LEVEL_CONV*
Generates trace data for convenience routines only.

*LEVEL_MP*
Generates trace data for mp routines only.

*LEVEL_NONE*
Disables all tracing.

*LEVEL_USER*
Generates trace data for user defined and validation routines only.

*LEVEL_XOM*
Generates trace data for XOM routines only.

*TRACE_PATH*
This environment variable is used to set the directory path for trace files. If TRACE_PATH is not used, trace files are placed in the current directory.

The following example sets on *all* tracing and places the trace files in the temporary directory on the C: drive.

```
set TRACE_MASK=LEVEL_ALL
set TRACE_PATH=c:\tmp
```

## Using the Trace Function Calls and Macro

A trace debug API is provided by the XMP tracing facility and includes two function calls and one macro:

**mp_debug_register()**
Used to register the name of the trace file.

**mp_debug_level()**
Used to define the level of tracing.

**imp_debugn()**
Used to specify user defined tracing.

### Registering the Trace File

To register the name of a trace file for a specific process thread, use the mp_debug_register() function call. The format for this function call is the following:

```
mp_debug_register ("char");
```

The character string parameter is entered without an extension. For example, the following statement generates a trace file named TEST.TRC:

```
mp_debug_register("TEST");
```

If no value is entered in the parameter to this function call, the result is written to standard out (stdout).

### Setting the Level of Tracing

The mp_debug_level function call is used to turn tracing on and off and to set the level of tracing. The format of the mp_debug_level function call is shown below:

```
mp_debug_level(int level, int depth);
```

The mp_debug_level function call parameters are described in the following sections.

#### int level

This parameter defines the level of tracing. Valid tracing levels are:

**LEVEL_ALL**
Traces all routines.

**LEVEL_ATTR**
Traces attribute calls.

**LEVEL_MP**
Traces mp calls.

**LEVEL_OBJ**
Traces managed-objects calls.

**LEVEL_SYS**
Traces system routines.

**LEVEL_USER**
Traces user and validation calls.

**LEVEL_UTIL**
Traces utility routines.

**LEVEL_XOM**
Traces XOM calls.

These values may be or'd together to form combinations of trace information. The following example illustrates how to define a trace that returns all tracing *except* utility level routines.

```
LEVEL_ALL & LEVEL_UTIL
```

### int depth

This parameter defines the quantity of information returned. A depth of zero indicates only which routine was called. Typically, this parameter is set to 100 or any other arbitrary large number.

## Defining User Specific Traces

The imp_debugn macro is provided as a mechanism to dump XMP internal structures such as private objects and workspaces. The format for the imp_debugn macro is the following:

```
imp_debugn(level,type,str, arg1 ... argn)
```

The parameters to the imp_debugn macro are described in the following sections.

The macro can be used to dump a maximum of 6 arguments. You specify the number of arguments to be dumped as part of the macro name as shown in the following list:

**imp_debug0**
Dumps no arguments except the name of the routine.

**imp_debug1**
Dumps 1 argument and the name of the routine.

**imp_debug2**
Dumps 2 arguments and the name of the routine.

**imp_debug3**
Dumps 3 arguments and the name of the routine.

**imp_debug4**
Dumps 4 arguments and the name of the routine.

**imp_debug5**
Dumps 5 arguments and the name of the routine.

### level

This parameter indicates the level of the DEBUG statement. If the current trace level includes this level, the DEBUG statement writes to the TRACE file. Valid levels are the following:

**LEVEL_ATTR**
Traces attribute calls.

**LEVEL_CONV**
Traces convenience routines.

**LEVEL_MP**
Traces MP calls.

**LEVEL_OBJ**
Traces OBJ calls (object methods).

**LEVEL_SYS**
Traces system routines.

**LEVEL_USER**
Traces USER and validation.

**LEVEL_UTIL**
Traces utility routines.

**LEVEL_XOM**
Traces XOM calls.

**Note:** Typically, a management program uses LEVEL_USER.

### type

This parameter specifies the type of debug statement, and is used by the trace facility to control the indenting of trace information. Valid types are the following:

**DEBUG_ENTRY**
Indicates the entry of a routine. Subsequent traces are indented 5 spaces.

**DEBUG_ERROR**
Treated the same as DEBUG_EXIT, except that the type keyword in the trace file is ERROR.

**DEBUG_EXIT**
Indicates the exit of a routine. Subsequent traces are outdented 5 spaces.

**DEBUG_PROCESSING**
Indicates a neutral information trace. No changes in indenting.

### str

This parameter specifies the trace dump header. With entry or exit types, it usually specifies the name of the routine. With a processing type, it usually indicates a description of the information being dumped.

### argn

This parameter specifies the information to be dumped. A maximum of six user defined arguments can be entered with the imp_debugn macro. User-defined arguments are defined using an argument description macro. This description macro informs the trace facility of the syntax of information to be dumped. Valid values are the following:

**D_ADDRESS(value,depth)**
Dumps an address. The depth argument specifies the number of bytes to dump.

**D_BOOLEAN(value)**
Dumps a BOOLEAN value.

**D_DEC(value)**
Dumps a DECIMAL value.

**D_DESC(value)**
Dumps an OM_descriptor.

**D_EXCLUSIONS(value)**
Dumps an exclusive value.

**D_EXT_OBJECT(value)**
Dumps an OM Object.

**D_HEX(value)**
Dumps a HEX value.

**D_MODIFICATION(value)**
Dumps a modification value.

**D_OCT(value)**
Dumps an OCTET value.

**D_OID(value)**
Dumps an object identifier.

**D_OM_ERROR(value)**
Dumps an OM_return_code.

**D_OM_STRING(str)**
Dumps an OM_string value.

**D_ROLE(value)**
Dumps a role value.

**D_STRING(str,depth)**
Dumps a string value. Depth is unused.

**D_SYNTAX(syntax)**
Dumps a syntax value.

**D_TYPE(value)**
Dumps an OM_type value.

**D_TYPELIST(value)**
Dumps a type list.

**D_WORKSPACE(workspace)**
Dumps a workspace.

## Programming Example

The following example illustrates how to dump a workspace and an object:

```
imp_debug2(     LEVEL_USER,
                DEBUG_PROCESSING,
                "test dump",
                D_WORKSPACE(ws),
                D_EXT_OBJECT(pub_obj))
```

The above example specifies that two arguments are to be dumped. The first argument specified is the D_WORKSPACE macro. This tells the trace facility to dump the information as a workspace. The second argument specified is the D_EXT_OBJECT macro. This tells the trace facility to dump out the second argument as a public OM object.