MuGuardianAngel

Thomas Richter

COLLABORATORS

	TITLE : MuGuardianAngel	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Thomas Richter	July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 MuGuardianAngel **1**

1.1 MuGuardianAngel Guide . . . 1
1.2 The THOR-Software Licence . . . 2
1.3 What's the MMU.library? . . . 3
1.4 What's the job of MuGuardianAngel? . . . 3
1.5 Installation of MuGuardianAngel . . . 4
1.6 Command line options and tooltypes . . . 4
1.7 Credits : Thank you goes to... . . . 5
1.8 Correct memory handling: Avoiding hits . . . 5
1.9 System patches installed by MuGuardianAngel . . . 10
1.10 MuGuardianAngel restrictions . . . 11
1.11 MuGuardianAngel error messages . . . 11
1.12 Common problems . . . 12
1.13 Option: QUIT . . . 13
1.14 Option: MUNGLIST . . . 13
1.15 Option: PRIORITY . . . 13
1.16 Option: TASK . . . 13
1.17 Option: WAIT . . . 14
1.18 Option: SNOOP . . . 14
1.19 Option: MIN . . . 14
1.20 Option: MAX . . . 14
1.21 Option: DEBUG . . . 14
1.22 Option: SHOWFAIL . . . 14
1.23 Option: SHOWSTACK . . . 15
1.24 Option: SHOWPC . . . 15
1.25 Option: SHOWHUNK . . . 15
1.26 Option: STACKLINES . . . 15
1.27 Option: NAMETAG . . . 15
1.28 Option: DATESTAMP . . . 15
1.29 Option: STACKSNOOP . . . 16

1.30 Option: CONSISTENCY . . . 16
1.31 Option: PRESIZE . . . 16
1.32 Option: POSTSIZE . . . 16
1.33 Option: FILLCHAR . . . 17
1.34 Option: LED . . . 17
1.35 Option: INTRO . . . 17
1.36 Option: DISABLEBELL . . . 17
1.37 Option: TINY . . . 17
1.38 Option: AREGCHECK . . . 17
1.39 Option: DREGCHECK . . . 18
1.40 Option: STACKCHECK . . . 18
1.41 Option: TOMUFORCE . . . 18
1.42 Option: DISPC . . . 18
1.43 Option: DISRANGE . . . 18
1.44 Option: NOMMU . . . 18
1.45 Option: ALLOWREUSE . . . 19
1.46 Hit: Clear memory cookie at .. was found defective . . . 19
1.47 Hit: Release of ..: alloc. length .. <> released length 19
1.48 Hit: Release of .. :allocation pool .. <> release pool 19
1.49 Hit: Mung-wall at .. is damaged . . . 20
1.50 Hit: MemHeader of chunk .. is not available . . . 20
1.51 Hit: Memory at .. released twice . . . 20
1.52 Hit: Invalid FreeMem of memory at 21
1.53 Hit: Mem header ..: free counter .. <> .. in chunks . . . 21
1.54 Hit: Null-sized allocation, ignored . . . 21
1.55 Hit: FreeMem of NULL pointer attempted, ignored . . . 21
1.56 Hit: Misaligned release of .., size .. performed . . . 22
1.57 Hit: Allocation of .. bytes failed . . . 22
1.58 Hit: Task ..: nearly out of stack . . . 22
1.59 Hit: Task ..: stack overflow . . . 23
1.60 Hit: Task ..: stack underflow . . . 23
1.61 Hit: Invalid memory list . . . 23
1.62 Hit: Mem header ..: node type is not NT_MEMORY . . . 23
1.63 Hit: Mem header ..: lower is larger than upper . . . 24
1.64 Hit: Mem header ..: chunk .. <lower or >upper . . . 24
1.65 Hit: Mem header ..: chunk .. order broken . . . 24
1.66 Hit: Mem header ..: chunk after .. misaligned . . . 25
1.67 Hit: MuGuardianAngel patches have been overwritten . . . 25
1.68 Hit: .. called from interrupt or supervisor mode . . . 25

1.69 Hit: CreatePool puddleSize .., thresSize .. invalid . . . 25
1.70 Hit: CreatePool failed . . . 26
1.71 Hit: DeletePool of NULL pointer attempted, ignored . . . 26
1.72 Hit: Alloc/FreePooled from NULL pool attempted . . . 26
1.73 Hit: Memory pool .. garbled . . . 26
1.74 Example MuGuardianAngel output . . . 27
1.75 Detail Example Hit . . . 28
1.76 Output: Intro string . . . 28
1.77 Output: Date Stamp . . . 29
1.78 Output: Hit . . . 29
1.79 Output: Program Counter . . . 29
1.80 Output: User stack pointer . . . 29
1.81 Output: Forbid/Permit indicator . . . 29
1.82 Output: Disable/Enable indicator . . . 29
1.83 Output: Task name . . . 29
1.84 Output: SegTracker . . . 30
1.85 Output: SegTracker Address . . . 30
1.86 Output: SegTracker Name . . . 30
1.87 Output: Hunk . . . 30
1.88 Output: Offset . . . 30
1.89 Output: Return PC of the allocating function . . . 30
1.90 Output: Name of the allocating task . . . 30
1.91 Output: Data Register Dump . . . 31
1.92 Output: D0 Register . . . 31
1.93 Output: D1 Register . . . 31
1.94 Output: D2 Register . . . 31
1.95 Output: D3 Register . . . 31
1.96 Output: D4 Register . . . 31
1.97 Output: D5 Register . . . 31
1.98 Output: D6 Register . . . 31
1.99 Output: D7 Register . . . 32
1.100 Output: Address Register Dump . . . 32
1.101 Output: A0 Register . . . 32
1.102 Output: A1 Register . . . 32
1.103 Output: A2 Register . . . 32
1.104 Output: A3 Register . . . 32
1.105 Output: A4 Register . . . 32
1.106 Output: A5 Register . . . 32
1.107 Output: A6 Register . . . 33

1.108 Output: A7 Register . . . 33
1.109 Output: Stack Dump . . . 33
1.110 Output: Stack Word . . . 33
1.111 Output: Show PC . . . 33
1.112 Output: Show PC-$20 . . . 33
1.113 Output: Show PC-$1C . . . 33
1.114 Output: Show PC-$18 . . . 33
1.115 Output: Show PC-$14 . . . 34
1.116 Output: Show PC-$10 . . . 34
1.117 Output: Show PC-$0C . . . 34
1.118 Output: Show PC-$08 . . . 34
1.119 Output: Show PC-$04 . . . 34
1.120 Output: Show PC+$00 . . . 34
1.121 Output: Show PC+$04 . . . 34
1.122 Output: Show PC+$08 . . . 34
1.123 Output: Show PC+$0C . . . 34
1.124 Output: Show PC+$10 . . . 34
1.125 Output: Show PC+$14 . . . 35
1.126 Output: Show PC+$18 . . . 35
1.127 Output: Show PC+$1C . . . 35
1.128 Output: Disassembler output . . . 35
1.129 Output: Disassembler output with PC . . . 35
1.130 Index . . . 35
1.131 History . . . 37

Chapter 1

MuGuardianAngel

1.1 MuGuardianAngel Guide

´### ,#. ,#### ,### ####’ #### ,#### ,###’ ####’ #### _____ ,#### ,###’ || ####’ #### | ___ ___ | ___ ,#### ,###’ ---- | | | | | | |___ ####’ #### | | | | | | | ,#####. ,###’ . | |___| |___| |_ ___| ####’##. ,#### ,## ,#### ########’ # ,##’ ####’ `####’ `###’ ,#### ##### © 1999 THOR - Software, ####’ Thomas Richter `##’

MuGuardianAngel Guide

Guide Version 1.08 MuGuardianAngel Version 40.21

MuGuardianAngel is based on the Guardian Angel © CBM, MungWall © 1992-93 Commodore-Amiga, MemSniff © 1999 Thomas Richter.

The Licence : Legal restrictions

MuTools : What is this all about, and what’s the MMU library?

Credits : Thanks goes to...

What is it : Overview

Installation : How to install MuGuardianAngel

Synopsis : The command line options and tool types

Developers : How to allocate and release memory correctly

Patches : System patches installed by MuGuardianAngel

Restrictions : What can it do, and what not?

Output : A tiny example output

Detailed Ex. : A very detailed example output

Hits : The meaning of the MuGuardianAngel error messages

Problems : Problematic programs and how to work around

History : What happened before

MuGuardianAngel is a debugging tool. Software that causes MuGuardianAngel hits will not run stable on a system without it. MuGuardianAngel helps you to detect violations of the memory management of software, and it will even protect memory up

to a certain degree. However, MuGuardianAngel is NOT a full automatic memory protector, which is likely impossible in the current system. As a side effect, it may well keep a system from crashing when a hit happens, but it may just as well make the software crash earlier. Due to the way how it works, it will slow down your system to a certain degree. MuGuardianAngel is mainly a development and testing tool.

MuGuardianAngel causes no ill effects with correctly working software. If a program fails to work while MuGuardianAngel is active, you should contact the developer of that program.

When an access violation happens, a report such as the following is output:

Output Example Detail Example

Thomas Richter

Rühmkorffstraße 10A

12209 Berlin

Germany

EMail: thor@einstein.math.tu-berlin.de

WWW: http://www.math.tu-berlin.de/~thor/thor/index.html

1.2 The THOR-Software Licence

The THOR-Software Licence (v2, 24th June 1998)

This License applies to the computer programs known as "MuGuardianAngel" and the "MuGuardianAngel.guide". The "Program", below, refers to such program. The "Archive" refers to the package of distribution, as prepared by the author of the Program, Thomas Richter. Each licensee is addressed as "you".

The Program and the data in the archive are freely distributable under the restrictions stated below, but are also Copyright (c) Thomas Richter.

Distribution of the Program, the Archive and the data in the Archive by a commercial organization without written permission from the author to any third party is prohibited if any payment is made in connection with such distribution, whether directly (as in payment for a copy of the Program) or indirectly (as in payment for some service related to the Program, or payment for some product or service that includes a copy of the Program "without charge"; these are only examples, and not an exhaustive enumeration of prohibited activities).

However, the following methods of distribution involving payment shall not in and of themselves be a violation of this restriction:

(i) Posting the Program on a public access information storage and retrieval service for which a fee is received for retrieving information (such as an on-line service), provided that the fee is not content-dependent (i.e., the fee would be the same for retrieving the same volume of information consisting of random data).

(ii) Distributing the Program on a CD-ROM, provided that

a) the Archive is reproduced entirely and verbatim on such CD-ROM, including especially this licence agreement;

b) the CD-ROM is made available to the public for a nominal fee only,

c) a copy of the CD is made available to the author for free except for shipment costs, and

d) provided further that all information on such CD-ROM is re-distributable for non-commercial purposes without charge.

Redistribution of a modified version of the Archive, the Program or the contents of the Archive is prohibited in any way, by any organization, regardless whether commercial or non-commercial. Everything must be kept together, in original and unmodified form.

Limitations.

Thomas Richter

1.3 What's the MMU.library?

All "modern" Amiga computers come with a special hardware component called the "MMU" for short, "Memory Management Unit". The MMU is a very powerful piece of hardware that can be seen as a translator between the CPU that carries out the actual calculation, and the surrounding hardware: Memory and IO devices. Each external access of the CPU is filtered by the MMU, checked whether the memory region is available, write protected, can be hold in the CPU internal cache and more. The MMU can be told to translate the addresses as seen from the CPU to different addresses, hence it can be used to "re-map" parts of the memory without actually touching the memory itself.

A series of programs is available that make use of the MMU: First of all, it's needed by the operating system to tell the CPU not to hold "chip memory", used by the Amiga custom chips, in its cache; second, several tools re-map the Kickstart ROM to faster 32Bit RAM by using the MMU to translate the ROM addresses - as seen from the CPU - to the RAM addresses where the image of the ROM is kept. Third, a number of debugging tools make use of it to detect accesses to physically unavailable memory regions, and hence to find bugs in programs; amongst them is the "Enforcer" by Michael Sinz. Fourth, the MMU can be used to create the illusion of "almost infinite memory", with so-called "virtual memory systems". Last but not least, a number of miscellaneous applications have been found for the MMU as well, for example for display drivers of emulators.

Unfortunately, the Amiga Os does not provide ANY interface to the MMU, everything boils down to hardware hacking and every program hacks the MMU table as it wishes. Needless to say this prevents program A from working nicely together with program B, Enforcer with FastROM or VMM, and other combinations have been impossible up to now.

THIS HAS TO CHANGE! There has to be a documented interface to the MMU that makes accesses transparent, easy and compatible. This is the goal of the "mmu.library". In one word, COMPATIBILITY.

Unfortunately, old programs won't use this library automatically, so things have to be rewritten. The "MuTools" are a collection of programs that take over the job of older applications that hit the hardware directly. The result are programs that operate hardware independent, without any CPU or MMU specific parts, no matter what kind of MMU is available, and programs that nicely co-exist with each other.

I hope other program authors choose to make use of the library in the future and provide powerful tools without the compatibility headache. The MuTools are just a tiny start, more has to follow.

1.4 What's the job of MuGuardianAngel?

MuGuardianAngel is an extension to the "MuForce" program. Where MuForce detects only accesses to un-available address regions, MuGuardianAngel will also protect "free" memory from getting overwritten or accessed at all. In case a program accesses memory without allocating it properly from the system, MuGuardianAngel will report that hit by the magic of the "MuForce" program.

MuGuardianAngel makes use of the mmu.library to setup the MMU to watch illegal accesses to pages of non-allocated, but in principle available memory. This is something different to MuForce which will only catch accesses to unavailable memory regions.

MuGuardianAngel will, additionally, provide all the functionality of the "MungWall" and "MungList" tools. All memory allocations are "munged" and watched for "out of range" accesses or illegal memory releases.

MuGuardianAngel can, optionally, snoop memory allocations and report them over the serial port. It may also check the memory list for consistency on a regular basis or report which memory regions are allocated by which task.

If SegTracker is running in the system when MuGuardianAngel is started, it will use the public SegTracker seglist tracking for identifying the hits.

For short, MuGuardianAngel replaces four tools in one:

The original "Guardian Angel" which never really worked. "MuGuardianAngel" will. The "MungWall" tool, except that the "MuGuardianAngel" dumps more information. The "MungList" tool. The "MemSniff" tool by the same author.

When an access violation happens, a report such as the following is output:

Output Example Detail Example

1.5 Installation of MuGuardianAngel

Installation is pretty simple:

- First, install the "mmu.library": Copy this library to your LIBS: drawer if you haven't installed it yet. It's contained in this archive.

- Install "MuForce" first. It is required by MuGuardianAngel.

- Copy "MuGuardianAngel" wherever you want.

- Remove other Enforcer tools like the "Enforcer" or the "CyberGuard", remove "MungWall", "MungList" and "MemSniff" as well. Detach MuGuardianAngel *after* MuForce, and possibly after SegTracker. Just start it like:

run >NIL: <NIL: MuGuardianAngel

plus all the options you wish.

Since "MuGuardianAngel" dumps it output over the serial port by default, it is possibly a smart idea to install "Sushi" or "Sashimi", too, in case no serial terminal is available. This tool will redirect the "MuGuardianAngel" output to a console window. Alternatively, use the TOMUFORCE option to redirect the output to the MuForce stream, wherever this goes.

You can also run "MuGuardianAngel" from the workbench, it will read its options from the tooltypes in this case.

IMPORTANT: It is *very* important that you do not install *ANY* patch on top of MuGuardianAngel. MuGuardianAngel will be no longer able to detect the return PC of the calling code; moreover, this will break the kludge for the layers.library and will therefore cause system crashes.

One exception: "ShowMem" releases 2.06 or better can be used safely because they jump into the allocation routine and do not call them.

IMPORTANT: Do *NOT* quit MuForce as long as MuGuardianAngel is running! This will most likely crash your machine!

IMPORTANT: It might be possible that you have to run certain "fixes" to work-around some "features" of certain programs and to keep MuGuardianAngel happy. They are intentionally not included in MuGuardianAngel itself.

More details about this

1.6 Command line options and tooltypes

MuGuardianAngel can be started either from the workbench or from the shell. In the first case, it reads its arguments from the "tooltypes" of its icon; you may alter these settings by selecting the "MuForce" icon and choosing "Information..." from the workbench "Icon" menu. In the second case, the arguments are taken from the command line. No matter how the program is run, the arguments are identically.

The options for MuGuardianAngel are as follows:

QUIT MUNGLIST PRI=PRIORITY CLIENT=TASK WAIT SNOOP MIN MAX DEBUG SHOWFAIL SHOWSTACK SHOWPC SHOWHUNK STACKLINES NAMETAG DATESTAMP STACKSNOOP CONSISTENCY PRESIZE POSTSIZE FILLCHAR

LED INTRO DISABLEBELL TINY DREGCHECK AREGCHECK STACKCHECK TOMUFORCE DISPC DISRANGE NOMMU ALLOWREUSE

When started from the workbench, MuGuardianAngel knows one additional tooltype, namely:

WINDOW=<path>

where <path> is a file name path where the program should print its output. This should be a console window specification, i.e. something like

CON:0/0/640/100/MuGuardianAngel

This argument defaults to NIL:, i.e. all output will be thrown away.

It is possibly a good idea, in case no serial terminal is available, to install a tool like "Sushi" or "Sashimi" to catch the MuGuardianAngel hits and print them to a console window instead over the serial line.

1.7 Credits : Thank you goes to...

Michael Sinz (a real *BIG* thank you!) for discussing a lot of details of CachePreDMA/CachePostDMA, for sending me the sources of these functions in his 68040, and especially - and that's really great - for making the Enforcer sources available.

Ralph Babel for giving information about the CachePreDMA/CachePostDMA functions and for taking the time and helping me a bit with the messy details.

Bjoern Schmidt and Oliver Spaniol for allowing me to run some tests on their 060 based machines.

Werner Müller for his 040 based system.

All the testers for running tests and sending me detailed information about their systems. Thank you to all of you, this project won't clearly possible without your support!

1.8 Correct memory handling: Avoiding hits

In case you wonder how MuGuardianAngel hits can be avoided, here are the rules how memory should be allocated and released correctly.

The following rules apply to all programs that are supposed to run in an OS friendly way. I didn't make them up myself. What you find here is more or less a copy of the rules taken from the ROM Kernel reference manual, the official Amiga developer documentation.

Breaking these rules will result in unstable programs, with or without any additional memory tools. A program that causes hits with MuGuardianAngel but runs fine otherwise is, nevertheless, unstable and might crash in certain situations.

Allocation of memory:

o) The MEMF_PUBLIC bit: Set the MEMF_PUBLIC bit (exec/memory.h). You usually want it!

NOT setting this bit results in memory that is a) *private* to your task, i.e. can't be read from any other task b) and can't be read safely within a Forbid()/Permit() or Enable()/Disable() pair.

The current Os DOES NOT implement any checks for this rule, neither does MuGuardianAngel. Since private memory is not under direct control of MuGuardianAngel, this type of memory is never made unavailable or touched otherwise by this debugging tool. Future memory managers might see this bit as a hint to assign "virtual memory" to the allocation, i.e. memory that can be swapped out to disk.

As an example, VMM requires correct usage of this bit.

All data that is supposed to hold Os structures MUST BE ALLOCATED WITH THE MEMF_PUBLIC flag set, any memory that is passed to other tasks, interrupts, exceptions, I/O buffers MUST BE ALLOCATED WITH THIS BIT SET.

The only exception are private structures that are only read or written to by your task, that are never passed nor read or written to by other processes or Os functions and that are not accessed with multitasking disabled.

o) Memory flushes:

Be prepared that a memory allocation might flush unused libraries, fonts and devices from memory. In special, DO NOT USE CLOSED RESOURCES. Using a "FindName()" on the exec resource lists IS NOT ENOUGH to use a resource.

If you DO NOT want that resources get flushed, set the MEMF_NO_EXPUNGE flag as memory attribute. See exec/memory.h.

The following is a safe memory flush:

AllocMem(0x7ffffff0,MEMF_PUBLIC);

(The flush used by the "avail flush" command).

MuGuardianAngel will never report a memory flush as a failed memory allocation. This is different to "MungWall".

o) Memory and custom chips:

Memory that should be read by the Amiga custom chip set MUST BE ALLOCATED with the MEMF_CHIP attribute set or the custom chips won't be able to address this memory. That goes for:

o) display buffers (native bitmaps) o) hardware copper lists (but not their gfx abstractions for the CMove(),CWait() etc... family) o) image bitmaps (struct IntuiImage->ImageData) o) floppy hardware buffers (but since V37 not required for the trackdisk.device I/O buffer) o) hardware audio buffers o) hardware sprites and image datas of "Bobs" o) everything else the custom chip set might access

o) Order of memory blocks:

Do not make *any* assumptions about the order in which you get memory. The second allocation is not necessarily the higher address!

o) The MEMF_FAST bit:

Do not use the MEMF_FAST bit unnecessary if chip mem would be O.K. for you, too. The operating system is smart enough to allocate fast memory for you if that is available. It will fall back to chip mem if fast mem is not available. There's usually no reason to ask for fast mem explicitly.

o) Alignment:

It is guaranteed that all memory allocated by AllocMem() is aligned to two long word boundaries, i.e. the bits 0 to 2 of the address will always be zero. NOT MORE! If you need more alignment, see the kludge below.

o) Size of buffers:

Make sure you allocate enough memory even for the worst case. A C style string needs n+1 bytes memory to hold a string of length n. Some Os functions require, due to bugs, a slightly larger buffer than you might think, check the "BUGS" section of the autodocs. (Mostly dos functions suffer from this bug, but some intuition functions require this as well).

MuGuardianAngel is able to detect *some* out-of-range accesses of memory, but usually only at the time the memory is released again.

o) Memory attributes:

Do NOT set ANY undocumented bits for the memory attributes of AllocMem(). They *might* be ignored for this version of the Os, but probably won't by the next version. Check the exec/memory.h file for valid flags. As for the current (V40) version of the Os, the following flags have been defined:

#define MEMF_ANY (0L) /* Any type of memory will do */ #define MEMF_PUBLIC (1L<<0) /* D*mn important, see caveats above !*/ #define MEMF_CHIP (1L<<1) /* for custom chips */ #define MEMF_FAST (1L<<2) /* explicitly fast mem, see caveats! */ #define MEMF_LOCAL (1L<<8) /* Memory that does not go away at RESET */ #define MEMF_24BITDMA (1L<<9) /* DMAable memory within 24 bits of address */ #define MEMF_KICK (1L<<10) /* Memory that can be used for KickTags */

#define MEMF_CLEAR (1L<<16) /* AllocMem: NULL out area before return */ #define MEMF_LARGEST (1L<<17) /* AvailMem: return the largest chunk size */ #define MEMF_REVERSE (1L<<18) /* AllocMem: allocate from the top down */ #define MEMF_TOTAL (1L<<19) /* AvailMem: return total size of memory */

o) Memory contents:

Do NOT MAKE any assumption about the contents of the memory block unless you specified the MEMF_CLEAR attribute to erase the memory block. Not setting this bit is a bit faster, but results in a memory block with whatever contents you might dream of.

MuGuardianAngel tries to be a bit nasty to programs that do not allocate memory with MEMF_CLEAR. Instead leaving the memory like it is, it will fill the memory region with a special non-zero pattern.

o) Self modifying code:

Self modifying is discouraged.

If you absolutely MUST play with this and can't go 'round this, use the following Os call to flush the CPU caches once you've placed your code in memory and need to run it:

ClearCacheU()

Do NOT expect that it is there BEFORE you called this routine. This is even more important to routines like interrupts that are called asynchronously.

o) Failures:

Feel prepared that your memory request might fail. An explicit check is REQUIRED after an AllocMem() call. Just "going guru" in this case *IS NOT ENOUGH*. Print a warning message, abort your program safely, CHECK YOUR CODE!

Assembly language authors: NO, IT'S NOT DOCUMENTED THAT AllocMem() SETS THE ZERO BIT IF THE ALLOCATION FAILED. YOU'VE TO TEST THAT YOURSELF.

If your calling task is indeed a process, OS versions V37 and above guarantee to set the result code for IoErr() to ERROR_NO_FREE_STO
(=103L).

MuGuardianAngel will show failed memory allocations if the SHOWFAIL option was turned on. Furthermore, it tries to be nasty to programs and messes the scratch registers d1/a0-a1 or d0-d1/a0-a1 before leaving the memory handling routines. It will also set the Z flag to fool assembly programs failing to test d0 correctly.

o) AllocMem() and context switches:

Neither AllocMem() nor FreeMem() break a "Forbid" state. This is important because it's the only way to "print" a list thru the dos.library and other functions that is access protected via Forbid().

The following code sequence is legal for this purpose, and should stay legal:

- call Forbid() first, - make a copy of that list element by element, using AllocMem() - call Permit(). - print the copy of the list - deallocate the copy.

Running into a Wait(), like using a semaphore for access protection of the memory list memory would be fatal here.

o) AllocMem,FreeMem,AllocAbs and interrupts:

NONE of these functions can be called from interrupts or in the supervisor mode.

Remember, however, that "input handlers" of the input.device are not run as interrupts but in the context of the input.device task, even though they are build on top of an interrupt structure. Thus, calling AllocMem() here to make a copy of an input event IS LEGAL.

---

o) Size of deallocation:

Deallocation size MUST MATCH ALLOCATION SIZE PRECISELY. It is BY NO MEANS ALLOWED to

- round the size because the rounding algorithm of the operating system might change in future to support special hardware (e.g. PowerPC cache lines which are 32 bytes wide)

Now to another rule that hasn't been formulated in the RKRMs:

- free a partial memory block, i.e. parts of an array. THIS IS DEFINITELY ILLEGAL, NO EXCEPTIONS, NO EXCUSES. Free ALL OR NOTHING.

Freeing a partial part of a memory block requires knowledge of the alignment rules of the Os and may break code if these rules change in future versions.

I would therefore strongly recommend NOT to use this technique.

MuGuardianAngel will "hit" in case a program tries to release a memory block by a different size than it was allocated with. It will furthermore hit if a program tries to release "unaligned" memory, i.e. a memory block not aligned to an eight-byte border. (However, see below for an important exception.)

The current implementation of the layers.library does not obey the rule formulated above (yuck!) and is therefore explicitly excluded from memory snooping and munging. It releases, unfortunately, partial memory blocks.

o) Access to deallocated memory:

Do not touch deallocated memory. If it's gone it's gone and you're no longer allowed to use it, address it, read it or write data to it. Another task might want it.

MuGuardianAngel will mark released memory as unavailable as soon as possible. In case you try to access this memory, you'll see a hit.

A tiny exception that hasn't been formulated in the RKRMs, but is unfortunately widely used:

Deallocation of memory WITHIN a Forbid()/Permit() pair. The memory, EXCEPT FOR THE FIRST EIGHT BYTES WHICH ARE USED FOR ADMINISTRATION, is guaranteed to stay unmodified and ready for use as long as the multitasking is disabled. Running into a Wait(), directly or indirectly, will break the Forbid() state and will therefore make the memory unusable.

Be warned! Even though this access is sort of legal, hence tolerated by MuGuardianAngel, is IMHO still ugly and therefore highly discouraged. One of the very few exceptions where this feature might be helpful is the following code segment that unloads the segment of a "load- and stay resident" program:

move.l SysBase(a4),a6 jsr _LVOForbid(a6) move.l DOSBase(a4),a6 move.l Segment(a4),d1 jsr _LVOUnloadSeg(a6) ;Unload own code move.l a6,a1 ;THIS CODE STAYS LEGAL because move.l SysBase(a4),a6 ;of the Forbid() jsr CloseLibrary(a6) ;close dos moveq #0,d0 rts ;exit.

Note that you must be definitely positively sure that the segment is not an overlayed segment because UnloadSeg() WILL break the Forbid() state in this case. However, this doesn't work for load- and stay-resident programs anyways.

MuGuardianAngel does, currently, guarantee that memory will be not made unavailable if it is released within a Forbid(). It will be made unavailable as soon as the Forbid() is broken. UNLIKE MungWall, MuGuardianAngel *does* munge this memory and will, too, check it for consistency as soon as it is allowed to do. It will, actually, check the memory twice. Once FreeMem() is called, and once again MuGuardianAngel is allowed to make it unavailable.

o) Chip memory and blitter access.

The custom "blitter logic" uses DMA and accesses the chip memory independent of the CPU. If you use a temporary buffer for the blitter, make sure the blitter does no longer access this buffer before you deallocate it. To be on the safe side, call WaitBlit() before de- allocating memory that has been used as blitter buffer.

MuGuardianAngel cannot, unfortunately, detect illegal chip ram accesses by the blitter because this is out of control of the MMU.

o) Memory and hardware DMA access:

Modern hard disk interfaces might access memory by DMA, parallel to the CPU. If you're planning to use this hardware DMA directly because you're writing a device driver for this hardware, be prepared to flush the CPU caches properly. Especially, call

CachePreDMA(...) prior the DMA operation

CachePostDMA(...) afterwards.

Check the autodocs for details about these functions and their parameters. Read them, then think about them, then read them again. It is important that you understand these correctly!

o) Return value:

FreeMem() DOES NOT return any useful value, nor does it set any condition codes.

As always, MuGuardianAngel will try to be nasty to programs and will invalidate the scratch registers and the condition codes.

AllocAbs and other weirdos:

AllocAbs is for specialized usage of allocating memory from a predefined location. DO NOT USE IT WITHOUT GOOD REASON.

o) Range of allocated memory:

AllocAbs performs some rounding. Be prepared that the memory block you get is not identical to the memory block you requested. However, IF the memory allocation could be satisfied, the requested memory block is guaranteed to be contained in the returned memory block. Feel prepared that the memory request cannot be satisfied because the requested memory is already in use by a different task.

AllocAbs() returns NULL in this case. You've to check for this explicitly! It does NOT set any condition codes.

AllocAbs() WILL NOT set the ERROR_NO_FREE_STORE return code for IoErr().

MuGuardianAngel will, again, fill the scratch registers with dummy values and set the Z flag to fool badly written programs of a failure condition.

o) Contents of allocated memory:

Do not make any assumptions about the contents of the allocated memory block. The OS uses parts of the free memory blocks for administrator purposes and might have been trashed parts of memory block.

That means especially for reset resident programs - whose memory is allocated this way by the exec KickMemPtr mechanism - that the first eight bytes will be trashed. Be prepared for that feature!

MuGuardianAngel will trash the allocated memory explicitly, but it will not snoop AllocAbs() allocations.

o) Deallocation of AllocAbs()-ed memory:

To be sure that the allocated memory is really deallocated completely, call FreeMem with the memory address and size you REQUESTED, NOT with the return value of AllocAbs(). This might sound strange indeed, but the FreeMem() logic performs the same rounding of size and address as the AllocAbs() logic. If, however, you pass in a different address, as the return value instead of the requested address, it is not guaranteed that really all memory is deallocated.

A tiny example might be helpful (assuming the the current rounding algorithm):

AllocAbs(0x07,0x300007);

allocates 16 bytes and returns 0x300000. Calling now

FreeMem(0x300000,0x07);

will only free EIGHT bytes starting from 0x300000 instead of 16 bytes. However,

FreeMem(0x300007,0x07);

will work as required.

MuGuardianAngel will detect the last FreeMem() as misaligned, though. You should ignore its hit in this case.

o) Using AllocAbs() for aligned memory allocation:

The following code segment is a kludge for allocating memory aligned to a boundary:

void *AllocAligned(ULONG bytesize,ULONG attributes,ULONG alignment) { UBYTE *mem,*res;

alignment--; if (mem=AllocMem(bytesize+alignment,attributes & (~MEMF_CLEAR))) { Forbid(); FreeMem(mem,bytesize+alignment mem = (mem + alignment)&(~alignment); res = AllocAbs(bytesize,mem); Permit(); if (res) { if (attributes & MEMF_CLEAR) memset(mem,0,bytesize); } else mem = NULL; } return mem; }

I.e, call this routine with "alignment" set to 16 for an alignment to a sixteen byte boundary. Calling this routine with anything but a power of two for the alignment doesn't make much sense and is illegal.

Note that the memory is cleared MANUALLY if MEMF_CLEAR is set. This MUST be done since AllocAbs() does not guarantee the contents of the memory, even if the former AllocMem() already cleared the memory.

Any program that obeys these rules won't cause *any* problems with MuGuardianAngel!

Debugging tools:

In case you write a debugger and you really, really *MUST* read non-allocated memory, then go to supervisor mode to read it. MuGuardianAngel will only detect memory accesses to "free" memory from user mode.

The following debugging tools are recommended:

-MuForce: Detects memory accesses to the vector base and to unmapped memory regions.

-MuGuardianAngel (-: : Detects memory accesses to "free" memory, illegal and out of range accesses, and the problems mentioned above.

-SegTracker: Keeps program names together with their loaded segments for easy identification of code. Used by the programs above.

-PatchWork: Detects invalid parameters to Os calls.

-Sashimi: Redirects MuGuardianAngel hits to a console window.

-SaferPatches: Detects illegal function patches. If this one crashes with a guru, something is wrong.

Even a program that runs without problems with these tools is not necessarily bug free!

Another set of weirdos for the "enlighted". (-:

The following is a list of "OS features" you should be aware of if you consider writing your own memory tool. I found them when writing PoolMem, so they are here for your information. However, DO NOT USE THESE TECHNIQUES in own code.

Even though the above rules have been setup for the developer, that doesn't mean that the Os respects these rules ("Quod licet Iovi non licet bovi.").

I found the following "OS features":

- The FFS (all versions V37 thru V43) expect a return value of "-1" for FreeMem(). This has been fixed for release 43.20.

MuGuardianAngel will therefore cause a strange result code for Close() for FFS releases below 43.20.

- The layers.library allocates memory in large blocks, but deallocates this large block of memory in a series of small deallocations. In other words, it breaks up large memory blocks in smaller ones.

The current version of MuGuardianAngel include a special kludge to allow this EXCLUSIVELY for the layers.library. However, THIS HAS BEEN ILLEGAL, IS ILLEGAL AND WILL CONTINUE TO BE ILLEGAL. I hope that this mess will be cleaned up in a future Os revision.

- Some programs expect the Z (zero) flag of the CPU after an AllocMem() call to be set on failure and to be cleared if the allocation worked. THIS IS UNDOCUMENTED.

MuGuardianAngel will break these programs on purpose.

1.9 System patches installed by MuGuardianAngel

MuGuardianAngel replaces the following system functions completely - this means that all other patches installed here will be no longer called:

AllocMem: replaced completely. The replacement function includes memory snooping, munging and marks the allocated memory valid. It contains, too, a consistency check.

FreeMem: replaced completely. Adds consistency checks for the deallocated memory and for the mungwall. Furthermore, memory is made unavailable and filled with a special cookie to detect memory modifications that somehow slipped thru the protection mechanism.

AllocAbs: replaced completely. It will, too, make the allocated memory available and fill the memory with a dummy pattern.

AvailMem: replaced completely. The calculation algorithm is modified to include the size of the mungwalls.

AllocPooled: replaced completey by a procedure that includes the memory munge.

FreePooled: replaced completely.

CreatePool: replaced completely, includes sanety checks of the arguments and an adjustment of the puddle and thressizes for the pool to include the mung wall.

DeletePool: replaced completely.

The replacement functions will never call the exec Alert() function to create a guru. Instead, all possible failures will be passed over to the MuGuardianAngel error handler and will be printed "in text".

MuGuardianAngel installs furthermore a "Page Access Handler" into the public MMU context, in order to be notified in case some other program tries to modify the MMU tree. The handler won't touch any page except those consisting completely of free memory, and will mark these pages as non-available. It remains therefore compatible to MuFastRom and others.

MuGuardianAngel modifies the MMU pages at the lower of the two available levels, its function is therefore transparent to programs making use of the high-level functions.

IMPORTANT: It is *very* important that you do not install *ANY* patch on top of MuGuardianAngel. MuGuardianAngel will be no longer able to detect the return PC of the calling code; moreover, this will break the kludge for the layers.library and will therefore cause system crashes.

One exception: "ShowMem" releases 2.06 or better can be used safely because they jump into the allocation routine and do not call them.

1.10 MuGuardianAngel restrictions

MuGuardianAngel is a powerful tool, but it has its quirks and its price:

- MuGuardianAngel will make your system slow. Since all memory allocations are watched and munged, every allocation will take much more time than it used to.

- The current version MuGuardianAngel does not work in case the exec free memory list contains re-mapped memory. This will change in a future version.

- MuGuardianAngel tries to detect a lot of hits, but it won't be able to get all bad memory accesses. It cannot detect an access to memory owned by a different task because it is not clear whether this access is intended or not. The AmigaOs memory model is completely shared, so there's no way to find this out. It might even fail to detect some accesses into "free" memory due to the page size granularity. Write accesses into this free memory will be found most likely, but only as soon as some other task tries to allocate the damaged memory. It is then, unfortunately, too late to find the task that was responsible for the problem. However, this is not the "typical" case anyways.

- MuGuardianAngel tries to give you as much information as possible, this is in most cases more than - for example - MungWall could. But it is still unable to give you a task name or a stack dump for certain hits. These are mainly limitations that arise in the construction of the AmigaOs, there's no easy way to fix it.

- MuGuardianAngel will eat some memory. The program itself is small, but since each allocation is enlarged to keep a "mung wall", programs will require more memory than before.

- The "STACKSNOOP" option is most un-useful.)-:

- MuGuardianAngel requires a MMU - unlike MungWall or MemSniff. But it is able to detect a lot more hits than MungWall and is faster than MemSniff.

1.11 MuGuardianAngel error messages

MuGuardianAngel may produce the following types of "hits":

Clear memory cookie at .. was found defective Release of .. :alloc. length .. <> released length ..

Release of .. :allocation pool .. <> release pool .. Mung-wall at .. is damaged

MemHeader of chunk .. is not available Memory at .. released twice

Invalid FreeMem of memory at .. Hit: Mem header ..: free counter .. <> .. in chunks

Null-sized allocation, ignored FreeMem of NULL pointer attempted, ignored

Misaligned release of .., size .. performed Allocation of .. bytes failed

Task ..: nearly out of stack Task ..: stack overflow

Task ..: stack underflow Invalid memory list

Mem header ..: node type is not NT_MEMORY Mem header ..: lower is larger than upper

Mem header ..: chunk .. <lower or >upper Mem header ..: chunk .. order broken

Mem header ..: chunk after .. misaligned MuGuardianAngel patches have been overwritten

.. called from interrupt or supervisor mode CreatePool puddleSize .., thresSize .. invalid

CreatePool failed DeletePool of NULL pointer attempted, ignored

Alloc/FreePooled from NULL pool attempted Memory pool .. garbled

However, since MuGuardianAngel makes certain memory regions unavailable by the MMU, programs accessing these regions will not be handled by MuGuardianAngel but by "MuForce". For details about the output of this program, check the "MuForce.guide".

1.12 Common problems

Here's a list of problematic programs:

The "cybscsi.device":

Due to a firmware "feature", this program causes a hit every second on each disk access. The background is that the device driver accesses a chip memory location to slow down the code by a well-defined amount of time. However, since that memory is not allocated, MuGuardianAngel will "hit" totally justified. To work around this problem, run the "FixCybAccess" program before running the MuGuardianAngel.

The "Ram-Handler":

Yes, you read that correctly, the Commodore RAM: disk is buggy. As soon as you try to read and write to the RAM disk simulatenously, strange and wonderful things may happen.

Futhermore, the RAM handler is setup with a too small stack. This should be fixed by running "PatchRAM" somewhere in the startup-sequence, before the RAM disk gets used.

The "console.device":

And yes, yet another ROM bug. Due to a bug in the console.device supervisor task, an illegal memory location *might* be accessed as soon as a console window gets resized. This is fixed by the "ConsoleFix" patch in this archive.

The "FFS":

Not really a problem of the FFS, but for some installations. Sometimes the FFS has been setup with too little stack size, so MuGuardianAngel will complain about a possible stack overflow of the filing system. This should be fixed manually by increasing the stack size in the RDB block. This step is left to the expert as an easy exercise. (-:

The "narrator.device":

This is another example of a system program that is too low on stack space. A patch for the 37.7 edition of the device is available in the "Fixes" directory of the "MMULib" distribution. It should be applied during installation of the MMU.library.

Executive:

Due to some reason not yet known by the author, Executive seems to be incompatible to MuGuardianAngel.

PatchWork:

In case you run PatchWork in conjunction with MuGuardianAngel, please ensure that you run PatchWork first and MuGuardianAngel afterwards.

In case nothing helps:

Here are a few additional tips and hints how to make MuGuardianAngel working if everything else fails:

o) Remove system patches. MuGuardianAngel is very critical.

o) As a workaround, try the options

ALLOWREUSE and PRI=-1

o) If even that doesn't help, try

NOMMU

1.13 Option: QUIT

QUIT=OFF/S

Tells MuGuardianAngel to quit. MuGuardianAngel can also be stopped by sending a CTRL-C to its process.

1.14 Option: MUNGLIST

MUNGLIST/S

This option is only available if another copy of MuGuardianAngel is already running. In this case, it will dump all the memory that has been allocated by programs thru MuGuardianAngel. The options MIN and MAX should be used to specify a region of interest, only allocations of sizes between these borders will be printed.

You may also specify the name of a task using the TASK option to dump only allocations performed by a certain task. However, TASK works only if the former copy of MuGuardianAngel was told to keep task names, by specifying the NAMETAG option at startup time.

1.15 Option: PRIORITY

PRI=PRIORITY/K/N

Specifies the priority of the MuGuardianAngel process. It defaults to "80" and should be rather high to be effective. There is little reason to adjust this.

The MuGuardianAngel process is required to update the DATESTAMP frequently and to release memory that was FreeMem()'ed in the FORBID state.

1.16 Option: TASK

CLIENT=TASK/K

If used with the MUNGLIST option, specifies the name of a task whose allocations should be dumped. The TASK option works, however, only if MuGuardianAngel has been run with the NAMETAG before.

If used during startup in conjunction with SNOOP, this is the name of the task whose memory allocations and releases should be dumped over the serial line.

1.17 Option: WAIT

WAIT/S

If this is set, a task causing a MuGuardianAngel hit will be suspended. It will run into a

Wait(SIGF_SINGLE);

where it could be captured by a debugger.

1.18 Option: SNOOP

SNOOP/S

Turns on memory snooping: Memory allocations and releases are printed over the serial port. It is recommended that you do not snoop all allocations but limit snooping by using the MIN and MAX options to define a region of interest. Furthermore, you may define the name of the task that should be snooped, using the TASK option. If you do not use any of these restrictions, your system will become *very* slow.

1.19 Option: MIN

MIN/K

Used to define a lower limit for SNOOP and MUNGLIST. Memory allocations below this size will not be snooped or dumped. MIN defaults to 0.

This option accepts numbers in hex notation as well, by placing a "0x" or "$" in front of the number.

1.20 Option: MAX

MAX/K

Used to define an upper limit for SNOOP and MUNGLIST. Memory allocations above this size will not be snooped or dumped. MAX defaults to 0xffffffff.

This option accepts numbers in hex notation as well, by placing a "0x" or "$" in front of the number.

1.21 Option: DEBUG

DEBUG/S

If this is set, MuGuardianAngel will run the system debugger in case a task hits. Register a0 will contain the return PC of the program, register a1 the description of the failure as C style string. MuGuardianAngel loads the registers and runs into a

jsr Debug(a6)

to call the system debugger. This is by default either the RomWack for releases V38 and below, or SAD for V39 and above.

IMPORTANT: It is highly recommended that you install a system monitor like COP before trying to use this option.

1.22 Option: SHOWFAIL

SHOWFAIL/S

Tells MuGuardianAngel to tread failed memory allocations as errors and to dump them as "hits". Memory flushes are *NOT* considered to be failed allocations.

1.23 Option: SHOWSTACK

SHOWSTACK/S

In case of a hit, MuGuardianAngel is told to dump STACKLINES of the user stack of the calling program, provided the stack pointer is available.

This option will not do anything if the TINY output mode is selected.

1.24 Option: SHOWPC

SHOWPC/S

In case of a hit, MuGuardianAngel dumps the memory around the return PC of the faulty call if available. This can be used to identify part of the code that was responsible for the hit.

This option will not do anything if the TINY output mode is selected.

1.25 Option: SHOWHUNK

SHOWHUNK/S:

Tell MuGuardianAngel to try to identify the hunk, hunk offset and name of the program that caused the hit.

This option requires the SegTracker running or it will do nothing.

1.26 Option: STACKLINES

STACKLINES/K/N

The number of stack lines to dump if the SHOWSTACK is used. Each stack line consists of 16 bytes, and the number of lines defaults to two.

This option does nothing if either the TINY output mode is selected, the stack pointer is not available or the SHOWSTACK option is not used.

1.27 Option: NAMETAG

NAMETAG/S

Tells MuGuardianAngel to keep the name of the allocating task as well. This option requires more memory, but is required to show the allocating task in the MUNGLIST, and to enable the TASK option for this list.

MuGuardianAngel will also make use of the name tag in case of a faulty FreeMem() to dump the name of allocator of the memory block - note that this need not to be identically.

1.28 Option: DATESTAMP

DATESTAMP=TIMESTAMP/S

This makes MuGuardianAngel output a date and time with each hit. Due to the nature of the way MuGuardianAngel must work, the time can not be read during the hit itself so the time output will be the last time value the main MuGuardianAngel task set up. MuGuardianAngel will update this value every second as to try to not use any real CPU time. The time displayed in the hit will thus be exact. (Assuming the system clock is correct.) The date is output before anything from the hit other than the optional introduction string.

1.29 Option: STACKSNOOP

STACKSNOOP/S

Turns on periodic checks for stack overflow and underflow. MuGuardianAngel checks then in each vertical blank the system tasks for out-of-range stack pointers. Due to the way how this is done, no fault PC nor a stack dump is available.

Moreover, it has been found that some tasks switch their stack without using the recommended system function StackSwitch(), making this option more or less unreliable and un-useful. Especially, the "SetENV" command as well as the "FastFilingSystem" break these rules and will therefore cause a lot of hits if this option is turned on.

For these reasons, just leave it off.

1.30 Option: CONSISTENCY

CON=CONSISTENCY/S

Turns on periodic consistency checks of the memory lists.

If this option is used, MuGuardianAngel will check the system memory lists each vertical blank for

- correct number of free bytes, - well sorted memory chunks, - out of range memory chunks - misaligned memory chunks

This may take quite a lot of CPU time, especially on "low-end" machines.

MuGuardianAngel is unfortunately not able to find the name of the task that possibly caused the mess in these lists, nor does it anything to fix the lists. If you see a hit from this option, you should better reboot soon.

1.31 Option: PRESIZE

PRESIZE/K

Set the size of the front "mung wall" in bytes.

This wall around the memory block is allocated additionally to the memory requested by the caller. It is then filled with a predefined pattern - see the FILLCHAR option. Typical "out of range" conditions will (hopefully) only damage the wall and nothing else.

On a FreeMem(), MuGuardianAngel checks the wall for this pattern and "hits" in case it was overwritten by the program.

The size of the front wall defaults to 32 bytes, it can be enlarged or made smaller by this option, but the size passed in is always rounded to the next larger number divisible by eight.

This option accepts hex numbers as well, provided they are given in C or assembler notation with leading "0x" or "$", respectively.

1.32 Option: POSTSIZE

POSTSIZE/K

Set the size of the rear "mung wall" in bytes.

This wall around the memory block is allocated additionally to the memory requested by the caller. It is then filled with a predefined pattern - see the FILLCHAR option. Typical "out of range" conditions will (hopefully) only damage the wall and nothing else.

On a FreeMem(), MuGuardianAngel checks the wall for this pattern and "hits" in case it was overwritten by the program.

The size of the rear wall defaults to 32 bytes, it can be enlarged or made smaller by this option, but the size passed in is always rounded to the next larger number divisible by eight.

This option accepts hex numbers as well, provided they are given in C or assembler notation with leading "0x" or "$", respectively.

1.33 Option: FILLCHAR

FILLCHAR/K

Selects the fill pattern for the "mung wall".

This wall around the memory block is allocated additionally to the memory requested by the caller. It is then filled with a predefined byte value, setup by this option. Typical "out of range" conditions will (hopefully) only damage the wall and nothing else.

On a FreeMem(), MuGuardianAngel checks the wall for this pattern and "hits" in case it was overwritten by the program.

This option accepts hex numbers as well, provided they are given in C or assembler notation with leading "0x" or "$", respectively.

This option defaults to "-1" which will choose the "rolling wall": MuGuardianAngel will change the wall for each allocation. Each other value is used directly as fill pattern.

1.34 Option: LED

LED/K/N

This option lets you specify the speed at which the LED will be toggled for each MuGuardianAngel hit. The default is 0 which means that the LED will not be touched. Using a larger value will make the flash take longer (such that it can be noticed when doing I/O models other than the default serial output) The time that the flash will take is a bit more than 1.3 microseconds times the number. So 1000 will be a bit more than 1.3 milliseconds. (Or 1000000 is a bit more than 1.3 seconds.)

1.35 Option: INTRO

INTRO/K

This optional introduction string will be output at the start of every hit. For example: INTRO="*NBad Program!" The default is no string.

1.36 Option: DISABLEBELL

DISABLEBELL/S

Disables printing of the ASCII bell (BEL 0x07) that beeps the display on external terminals or on Sushi output.

1.37 Option: TINY

TINY/S

This tells MuGuardianAngel to output a minimal hit. The output is basically the first line of the hit.

1.38 Option: AREGCHECK

AREGCHECK/S

This option tells MuGuardianAngel that you wish all of the values in the Address Registers checked via "SegTracker", much like STACKCHECK.

1.39 Option: DREGCHECK

DREGCHECK/S

This option tells MuGuardianAngel that you wish all of the values in the Data Registers checked via "SegTracker", much like STACKCHECK.

1.40 Option: STACKCHECK

STACKCHECK/S

This option tells MuGuardianAngel that you wish all of the long words displayed in the stack to be checked against the global seglists via "SegTracker". This will tell you what seglist various return addresses are on the stack. If you are not displaying stack information in the hit or MuGuardianAngel is not able to provide stack information, then STACKCHECK will have nothing to check. If you are displaying stack information, then each long word will be check and only those which are in one of the tracked seglists will be displayed in a "SegTracker" line. The output will show the PC address first and then work its way back on the stack such that you can read it from bottom up as the order of calling or from top down as the stack-frame back-trace.

1.41 Option: TOMUFORCE

TOMUFORCE/S

This will merge the MuGuardianAngel output with the MuForce output stream. If, for example, the MuForce output goes to a console window, the MuGuardianAngel hits will show up in the same window.

If this option is not given, MuGuardianAngel dumps its hits over the serial port. A tool like Sushi or Sashimi must then be used to redirect this stream to a console window.

1.42 Option: DISPC

DISPC/S

Enables disassembly of the code around the fault, which is most useful for debugging and for having an immediate impression what the bug might be.

This option requires the disassembler.library which is included in the distribution.

1.43 Option: DISRANGE

DISRANGE/K/N

Specifies the approximate number of bytes to disassemble around the fault if the DISPC option is turned on. This number is by default 32, i.e. MuGuardianAngel will disassemble at least 32 bytes ahead and below of the fault.

1.44 Option: NOMMU

NOMMU/S

Tells MuGuardianAngel not to use the MMU magic for protection of non-allocated memory. This option is basically here as workaround for questionable software that hacks on the MMU itself and therefore conflicts with MuGuardianAngel. Note that using this option heavely degrades MuGuardianAngel to "MungWall" as it will no longer be able to detect at least some "out of bounds" accesses. This option shouldn't be used unless it really can't make to work otherwise.

1.45 Option: ALLOWREUSE

ALLOWREUSE/S

Makes MuGuardianAngel a bit less picky about memory accesses. Without this option, MuGuardianAngel will mark released memory as non-available immediately, unless the calling task was in Forbid() or Disable() state. If this option is found, MuGuardianAngel will leave this step to its supervisor task and will, hence, delay this operation to the next task switch. It is therefore recommended that you use this option together with PRI=-1 to delay the task switches as well.

Note that this option is not required to keep correctly written software working. This option is a work-around to keep the system alive even with partially broken software installed. It will reduce the ability of MuGuardianAngel to detect software faults and should therefore be used only in the case MuGuardianAngel can't be made to work otherwise.

1.46 Hit: Clear memory cookie at .. was found defective

Cause of the fault:

When allocating some memory, MuGuardianAngel found that the cookie free memory is usually filled with is damaged. Hence, some program managed to write to the free memory without causing a hit. This might happen because MuGuardianAngel isn't able to mark all free memory as invalid due to the page granularity of the MMU. Since it is not clear when and by whom this hit happened, MuGuardianAngel will not be able to display any useful information about it. No caller PC nor a stack dump is possible. The task finding this hit is, however, not necessarily the task that made the mess. The printed address is the address where the defective cookie was found.

How to fix this:

Tough! Try to increase the mung wall with the PRESIZE or POSTSIZE a bit to get the mung wall hit instead of free memory. Try to reproduce this bug, i.e. whether it happens if a specific program is in use. Try to step thru the program until the hit happens, possibly watching parts of the free memory.

1.47 Hit: Release of ..: alloc. length .. <> released length ..

Cause of the fault:

MuGuardianAngel found a FreeMem() or a FreePooled() of a memory block whose size does not match the corresponding AllocMem() resp. AllocPooled(). Hence, a program tried to release a memory block only partially, or tried to release more memory than allocated.

How to fix this:

MuGuardianAngel will be able to show you the caller PC and task of the FreeMem() as the PC of the corresponding AllocMem(). In case you need the name of the allocating task as well, specify the NAMETAG option when starting MuGuardianAngel.

Check your program around the allocation and the memory release and make sure you're passing the correct size, or you're really not releasing the memory block you intended to release. Check for invalid pointers as well.

NOTE: In case you see a return PC somewhere in the layers.library, make sure you haven't installed a patch "on top" of MuGuardianAngel. This WON'T WORK! MuGuardianAngel has to know the right caller PC for a special kludge to fix a design bug in the layers.library.

1.48 Hit: Release of .. :allocation pool .. <> release pool ..

Cause of the fault:

MuGuardianAngel found a FreePooled() of a memory chunk which was allocated from a different memory pool, or which wasn't allocated from a memory pool at all. Alternatively, a memory chunk allocated with AllocPooled() was tried to be released with FreeMem(). Non-pooled allocations/releases correspond to the memory pool indicated by 00000000.

How to fix this:

MuGuardianAngel will be able to show you the caller PC and task of the FreeMem() as the PC of the corresponding AllocMem() or AllocPooled(). In case you need the name of the allocating task as well, specify the NAMETAG option when starting MuGuardianAngel.

Check the program for consistent use of memory pools, specifically, make sure that you allocate memory for Os structures correctly. Some libraries or functions will try to release allocated memory on their own, possibly expecting the memory to be "public", or to be allocated with AllocMem() instead being taken from a pool.

1.49 Hit: Mung-wall at .. is damaged

Cause of the fault:

On the FreeMem() of a memory chunk, MuGuardianAngel found the mung wall of the memory chunk defective. The number printed in the header is the memory block whose mung wall was found broken.

How to fix this:

This fault is typical for "out of range" accesses. Most likely, you allocated "too little" memory to hold an array or a structure, or a C string overrun the buffer. Remember that an array like

int i[5]

will contain of the FIVE elements i[0] to i[4] and not i[1] to i[5]. Check the Os documentation, especially the BUGS section of the autodocs, some functions require a larger buffer than you might think. Check your strings always for their size - a character array of 10 items can only hold strings of nine character length. Check strcpy() for overruns, this function does not check sizes at all.

1.50 Hit: MemHeader of chunk .. is not available

Cause of the fault:

A program tried to release a memory block of a location that is nowhere in the memory lists of exec.

How to fix this:

This ranges from easy to tough. Easiest possible cause is passing an invalid pointer to FreeMem(). In this case, check pointers for correctness, set unused pointers to NULL as soon as you don't need them anymore. Possibly, the pointer for an object was pointing somehow in the "wilderness".

Another reason for this fault could be that a second program damaged the exec memory list. In this case, restart MuGuardianAngel with the CONSISTENCY option to enable the periodic memory check. As soon as a program messes with the exec memory lists, you'll see a hit. Unfortunately, MuGuardianAngel won't be able to tell you the task or the instruction that caused the mess, but you might be able to find the problem by stepping thru the program that was running at the time the fault appeared.

If this is the cause of the problem, the task mentioned in the MuGuardianAngel hit is not necessarily the task causing the problem.

1.51 Hit: Memory at .. released twice

Cause of the fault:

The calling program tried to release a memory block that has been released already before.

How to fix this:

Check the FreeMem() that caused the problem: You passed in a pointer to something you already released before. As soon as you release an object, try to set all pointers to this object to NULL to avoid this situation.

1.52 Hit: Invalid FreeMem of memory at ..

Cause of the fault:

MuGuardianAngel found that the memory chunk that is to be released is already partially, but not completely released. Parts of it have been found in the exec free list.

How to fix this:

This hit means in most cases that you passed an invalid pointer to FreeMem(), or the pointer to something that has been already released before. This specific error happens then because a second program allocated already parts of the object you released before, and you're now trying to release again. As a rule of thumb, set all pointers to objects to NULL as soon as you release them, this helps to detect the problem in most cases.

1.53 Hit: Mem header ..: free counter .. <> .. in chunks

Cause of the fault:

During a consistency check, MuGuardianAngel found that the number of free bytes noted in the memory header whose address is printed in the hit is not the same as the number of free bytes in the chunk list of this header. The number left to the "<>" is the number of bytes as indicated by the memory header, whereas the right hand side is the number of bytes which are really available in the memory chunks.

How to fix this:

Tough. This means most likely that either the exec memory header was damaged, or the chunk list is broken. In the first case, a program used an illegal pointer that pointed "by accident" into the memory header. In the second case a memory chunk could have been damaged because a program tried to access it after it has been released. Typically, the chunk linkage pointer MC_NEXT is set to NULL, terminating the chunk list before its "true" end. This hit is typical for pointers that went mad or access of released memory. In most cases, the program detecting this problem is NOT the program responsible for the fault. Try to enable the CONSISTENCY option to check the memory periodically and try to reproduce the problem.

The task shown in the MuGuardianAngel hit is not necessarily the task causing the problem.

1.54 Hit: Null-sized allocation, ignored

Cause of the fault:

A program called AllocMem() with the size parameter set to zero. MuGuardianAngel did not perform any allocation at all and passed a NULL pointer as result.

How to fix this:

This bug happens typically if you want to allocate a dynamically sized object you don't need in this specific situation. It is in most cases enough to check for the NULL-sized case explicitly in your program and to fill in a NULL pointer by hand. Possibly, you failed to calculate the size of an object correctly.

1.55 Hit: FreeMem of NULL pointer attempted, ignored

Cause of the fault:

FreeMem() was called with a NULL pointer as memory argument. MuGuardianAngel ignored the attempt completely and did nothing.

How to fix this:

In most cases, an explicit check for NULL in your program should be enough to fix it - you're trying to release an object which does not, or no longer, exist, possibly because an allocation failed somewhere above. In this case, add a check for NULL behind corresponding the AllocMem().

1.56 Hit: Misaligned release of .., size .. performed

Cause of the fault:

MuGuardianAngel found that a misaligned pointer was passed into FreeMem() or FreePooled(). Memory blocks are aligned to eight-byte boundaries, and so should be the pointer passed in.

How to fix this:

This hit happens most likely because you passed an illegal and invalid pointer to FreeMem(), possibly reading it from an object that does no longer exist or that was partially overwritten by faulty code. Try to single-step your code around the fault PC to see were the pointer was taken from.

Another cause of this fault could be - even though this is unlikely - that you allocated non-aligned memory with AllocAbs(). In that case, it is CORRECT to pass the non-aligned pointer back to FreeMem(), and the hit should be ignored. Check the developer information for details about this special case.

1.57 Hit: Allocation of .. bytes failed

Cause of the fault:

This hit shows only up if you told MuGuardianAngel with the SHOWFAIL to do so. It means that a memory allocation by AllocMem() or AllocPooled() could not performed because either not enough memory is available, or the requested type of memory is not available.

How to fix this:

This is not really a hit, just a warning. The calling program will be informed about the out-of-memory condition. However, it *might* mean that the system run out of memory because your code never released the memory it allocated before. In this case, restart MuGuardianAngel with the NAMETAG option, give the name of your task as argument to TASK and enable snooping with SNOOP. Possibly, specify MIN and MAX to show only allocations and memory releases of certain sizes.

Now type "avail flush" in the shell and start your program. Watch the output carefully: A line starting with "A" is a memory allocation, the address of the memory is found behind the "@" sign. Each allocation should be matched by one and exactly one line starting with "F", showing the corresponding memory release. If you quit your program, type "avail flush again" and check the output: Every "A" line should be matched by one "F" line. If you find some "A"s without an "F", you failed to release some memory.

1.58 Hit: Task ..: nearly out of stack

Cause of the fault:

The MuGuardianAngel memory handling functions or the STACKSNOOP option found that the given task is about to run out of stack space.

How to fix this:

It is usually enough to increase the stack size. However, if you see this warning, it is most likely already too late.

WARNING: This warning message is actually generated in two situations:

First, the continously running vertical blank handler of MuGuardianAngel detected a near stack overflow. It will be most likely too late to do anything against it if this happens.

Second, a program run into the memory handling functions of MuGuardianAngel with a nearly overrun stack. MuGuardianAngel will be able to give you detailed stack information only in the latter case, though. In this specific case, MuGuardianAngel will provide an "emergency stack" to aid the task and to allow it to run on without further problems. This situation will generate a warning message, regardless of whether STACKSNOOP was activated or not.

In any case, you should note that MuGuardianAngel does not check the stack continuously. Therefore, MuGuardianAngel might easily overlook an out-of-stack condition. Moreover, some tasks switch the stack without informing the system correctly about this step, causing this hit without any danger. The "FastFilingSystem" is one example. Therefore, the STACKSNOOP option is not recommended because it might print a lot of "bogus" warnings.

1.59 Hit: Task ..: stack overflow

Cause of the fault:

The STACKSNOOP option found that the given task run out of stack space.

How to fix this:

It is usually enough to increase the stack size. However, if you see this warning, it is already too late and the task overwrote innocent memory. MuGuardianAngel will not be able to help you, tough luck!

This warning is generated for one of two situations: Either, the MuGuardianAngel vertical blank handler detected an overrun stack on its duty. Or, the memory handling functions detected the problem. MuGuardianAngel will only be able to print detailed stack information in the latter case.

WARNING: The STACKSNOOP does not check the stack continuously, it is only watched every vertical blank and on memory allocations and releases. Therefore, MuGuardianAngel might easily overlook an out-of-stack condition. Moreover, some tasks switch stack without informing the system correctly about this step, causing this hit without any danger. The "FastFilingSystem" is one example. Therefore, STACKSNOOPing is not recommended.

1.60 Hit: Task ..: stack underflow

Cause of the fault:

The STACKSNOOP option found a task under-running its stack space, i.e. it pulled more data from its stack than was available.

How to fix this:

This is a fatal error condition, but see below for the most common reason for this hit. In case this is a true under-run, your code went crazy and jumped into the "wilderness". It's best to step thru the problematic part with a debugger and see what happens.

This warning is generated for one of two situations: Either, the MuGuardianAngel vertical blank handler detected an underrun stack at its duty. Or, the memory handling functions detected the problem. MuGuardianAngel will only be able to print detailed stack information in the latter case.

WARNING: The STACKSNOOP does not check the stack continuously, it is only watched every vertical blank and on memory allocations and releases. Therefore, MuGuardianAngel might easily overlook an out-of-stack condition. Moreover, some tasks switch stack without informing the system correctly about this step, causing this hit without any danger. The "FastFilingSystem" is one example. Therefore, STACKSNOOPing is not recommended.

1.61 Hit: Invalid memory list

Cause of the fault:

During a consistency check, MuGuardianAngel found the exec memory list in a desolate state. If you don't see a complete crash after this, you're lucky!

How to fix this:

Though! This hit happens only if a faulty program pokes into ExecBase and hits the memory list. MuGuardianAngel will not be able to give you the name of the task nor the PC of the faulty code. You're on your own here! Try to reproduce the bug, using a debugger.

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.62 Hit: Mem header ..: node type is not NT_MEMORY

Cause of the fault:

MuGuardianAngel found that the given memory header in the exec free memory list is not of the correct type. Even though the node type is ignored by the memory handling functions, this means that something is wrong in your system.

How to fix this:

Though! Either you're using some utility that hacks the exec free memory list and adds memory without setting the node type correctly, or some program went mad and poked into the exec free list. In the last case, it might be hard to identify the program because MuGuardianAngel can't give you the return PC nor the faulty task. Try to reproduce the bug and... good luck!

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.63 Hit: Mem header ..: lower is larger than upper

Cause of the fault:

MuGuardianAngel found a memory header in the exec free memory list whose lower address is larger than its upper end address. Hence, this header is messed up. If you don't see a total crash after this hit you're lucky and should take the chance to reboot before worse happens.

How to fix this:

Though! Looks like some program messed with the exec free list, possibly by a pointer into the "wilderness". MuGuardianAngel can't help you much here because it doesn't know the return PC nor the task the caused the mess. Try to reproduce the bug and... good luck!

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.64 Hit: Mem header ..: chunk .. <lower or >upper

Cause of the fault:

MuGuardianAngel found a memory chunk in a memory list that doesn't belong there. Its address is either lower than the lower bound of the memory list, or higher than the end address of the list.

How to fix this:

Though! Looks like some program messed with the exec free list, possibly by a pointer into the "wilderness". MuGuardianAngel can't help you much here because it doesn't know the return PC nor the task the caused the mess. Try to reproduce the bug and... good luck!

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.65 Hit: Mem header ..: chunk .. order broken

Cause of the fault:

MuGuardianAngel found the free memory list of the given memory header in a desolate state, the chunks are no longer in their correct order and are no longer sorted by address. Someone hits the memory list.

How to fix this:

Tough. A memory chunk could have been damaged because a program tried to access it after it has been released. Typically, the chunk linkage pointer MC_NEXT is set to some messy address, causing the illusion of an unsorted but de-facto invalid list. This hit is typical for pointers that went mad or access of released memory. MuGuardianAngel won't be able to give you the name of the task that caused the mess, nor the return PC. Try to reproduce the problem and... good luck. It might sometimes help to increase the POSTSIZE and the PRESIZE to give a faulty task the chance to hit the mung-wall instead of something more important.

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.66 Hit: Mem header ..: chunk after .. misaligned

Cause of the fault:

MuGuardianAngel found a memory chunk in the free memory list that is not correctly aligned - and most likely invalid. Since the fault is usually a bad pointer of the chunk in front of the "misaligned" chunk, not the misaligned address itself but the chunk pointing to this address is printed. If you don't see a total crash after this hit, you're lucky. Take the chance and reboot!

How to fix this:

Tough. A memory chunk could have been damaged because a program tried to access it after it has been released. This hit is typical for pointers that went mad or access of released memory. MuGuardianAngel won't be able to give you the name of the task that caused the mess, nor the return PC. Try to reproduce the problem and... good luck. It might sometimes help to increase the POSTSIZE and the PRESIZE to give a faulty task the chance to hit the mung-wall instead of something more important.

MuGuardianAngel is not able to give you the name of the task, nor the return PC nor a stack dump if this problem shows up.

1.67 Hit: MuGuardianAngel patches have been overwritten

Cause of the fault:

Some program installed its own patches on top of the MuGuardianAngel installations. This means asking for trouble! The MuGuardianAngel patches *MUST BE* topmost in the patch list or the layers.library kludge won't work anymore, causing a lot of problems on window re-arrangement.

How to fix this:

Try to locate the program that installed the patches. Running a tool like "SaferPatches REMEMBER" might help you here a lot in detecting the program. MuGuardianAngel will not be able to give you this information yourself, unfortunately.

1.68 Hit: .. called from interrupt or supervisor mode

Cause of the fault:

Some program called the mentioned Os function - AllocMem(), FreeMem(), AvailMem(), AllocAbs(), AllocPooled(), FreePooled(), CreatePool() or DeletePool() from interrupt or supervisor code. Note that this is illegal.

How to fix this:

Try to find the program that installed the interrupt vectors calling the Os function. This need not to be the task that shows up in the hit output. Remove the offending code or de-install the faulty software. Note that only very few functions can be called from interrupt code at all, as for example PutMsg() and Signal(). In case you need memory in an interrupt handler, build a private memory management, allocate the memory from a task, put it in a list, ready available for the interrupt and use Signal() to tell the master task to allocate more memory in case the memory list is nearly run out. Don't forget to "bracket" your memory management with Disable() or any other method to avoid simultaneous access from the interrupt and the master code.

1.69 Hit: CreatePool puddleSize .., thresSize .. invalid

Cause of the fault:

The calling program tried to build a memory pool, but specified illegal parameters for the puddleSize and thresSize arguments. MuGuardianAngel will complain if either of them is null, or the threshold size is larger than the puddle size.

How to fix this:

Check the source code around the lines where the memory pools are allocated and correct the parameters.

1.70 Hit: CreatePool failed

Cause of the fault:

This hit shows only up if you told MuGuardianAngel with the SHOWFAIL to do so. It means that the system run out of memory when trying to build a new memory pool.

How to fix this:

This is not really a hit, just a warning. The calling program will be informed about the out-of-memory condition. However, it *might* mean that the system run out of memory because your code never released the memory it allocated before. In this case, restart MuGuardianAngel with the NAMETAG option, give the name of your task as argument to TASK and enable snooping with SNOOP. Possibly, specify MIN and MAX to show only allocations and memory releases of certain sizes.

Now type "avail flush" in the shell and start your program. Watch the output carefully: A line starting with "A" is a memory allocation, the address of the memory is found behind the "@" sign. Each allocation should be matched by one and exactly one line starting with "F", showing the corresponding memory release. If you quit your program, type "avail flush again" and check the output: Every "A" line should be matched by one "F" line. If you find some "A"s without an "F", you failed to release some memory.

1.71 Hit: DeletePool of NULL pointer attempted, ignored

Cause of the fault:

A program called DeletePool() with a NULL pool argument, hence tried to delete a non-existent pool.

How to fix this:

Check the pool management of the code. Most likely, you forgot to check for a NULL pointer, or tried to release memory pools before having tried to allocate them.

1.72 Hit: Alloc/FreePooled from NULL pool attempted

Cause of the fault:

The code called AllocPooled() or FreePooled() with a NULL pool argument, hence failed to supply a memory pool for pooled allocations.

How to fix this:

Check your pool management. Most likely, you failed to call CreatePool() before trying a pooled allocation, or you forgot to check for a NULL result code of CreatePool(). Another possibility might be that you already deleted the pool and tried to access it afterwards. Note that the system will already clean up all memory in the pool when DeletePool() is called, you do not need to release all pooled memory manually - even though this is a matter of good style to do so.

1.73 Hit: Memory pool .. garbled

Cause of the fault:

A program called FreePooled() with a memory chunk that was most likely allocated correctly from the memory pool that was passed in as argument to FreePooled(), but MuGuardianAngel failed to find the puddle of the pool the chunk was taken from.

How to fix this:

This ranges from easy to tough. Easiest possible cause is passing an invalid pointer to FreePooled(), which, due to some coincidence looked reasonable for the first sanity check MuGuardianAngel performed. In this case, check pointers for correctness, set unused pointers to NULL as soon as you don't need them anymore. Possibly, the pointer for an object was pointing somehow in the "wilderness".

Another reason for this fault could be that a second program damaged the puddle list of the pool. This is nearly impossible to check. Unfortunately, MuGuardianAngel won't be able to tell you the task or the instruction that caused the mess, but you might be able to find the problem by stepping thru the program that was running at the time the fault appeared.

If this is the cause of the problem, the task mentioned in the MuGuardianAngel hit is not necessarily the task causing the problem.

1.74 Example MuGuardianAngel output

An example MuGuardianAngel output:

Bad program! 03-Jul-99 14:26:14 Mung-wall at 01083bc8 is damaged. PC : 0105bd6a USP: 010d6434 ( )( ) Name: memorydevil ----> 0105bd6a - "memorydevil" Hunk 0 Offset 00000032 PCa : 0105bd5a Name: memorydevil ----> 0105bd5a - "memorydevil" Hunk 0 Offset 00000022 Data: 0000002a 00000000 00001000 0101b744 00000001 0041ac8b 0040f69f 0105bd34 Addr: 010d6430 01083bc8 0101a62c 0105bd34 010d6438 00f9feaa SysBase -------- Stck: 00fa06d6 00001000 0101b2ac a3a3a3a3 a3a3a3a3 a3a3a3a3 a3a3a3a3 a3a3a3a3 Stck: a3a3a3a3 a3a3a3a3 a3a3a3a3 01f80000 00e94c58 010d646c 010d6470 010d6474 PC-8: 4eaeff76 4a806718 702a7201 4eaeff3a 4a80670c 22404229 ffff702a 4eaeff2e PC *: 4e759595 95959595 95959595 95959595 95959595 95959595 95959595 95959595 0105bd5a : 4a80 tst.l d0 0105bd5c : 670c beq.s $105bd6a 0105bd5e : 2240 movea.l d0,a1 0105bd60 : 4229 ffff clr.b -$1(a1) 0105bd64 : 702a moveq.l #$2a,d0 0105bd66 : 4eae ff2e jsr -$d2(a6) 0105bd6a : *4e75 rts 0105bd6c : 9595 sub.l d2,(a5) 0105bd6e : 9595 sub.l d2,(a5) 0105bd70 : 9595 sub.l d2,(a5) 0105bd72 : 9595 sub.l d2,(a5) 0105bd74 : 9595 sub.l d2,(a5) 0105bd76 : 9595 sub.l d2,(a5) 0105bd78 : 9595 sub.l d2,(a5)

Here is a breakdown of what these reports are saying:

The first line is a user-configurable intro string. It is empty by default but can be setup with the INTRO option.

The second line is the date stamp.

The third line is the cause for the fault. A detailed description about what this error message means can be found in this guide. It, too, gives you the address of the where the fault was detected.

The fourth line shows the program counter of the task that caused the hit, the user stack pointer, two flags that indicate whether exec is currently in "Forbid" or "Disable" state and the name of the task causing the hit. If the state was "forbidden", the first bracket would show an "(F)". In disable state, the second bracket would say "(D)". The shown PC is the so called "return PC", i.e. where program flow will continue after execution of the Os call. It is not the address of the subroutine call itself. Unlike MungWall, this return PC is automatically the correct one, but it might well be that the allocation or release is done as a side effect by some Os function.

The next line is only available if the SegTracker was installed and the SHOWHUNK option was activated. They give detailed information about the name of the segment, the hunk and the segment offset of the return PC shown in the line above.

The seventh line is special: In case the fault happened in a memory release function, i.e. FreeMem(), MuGuardianAngel will dump here the return PC of the corresponding allocation call, if available. This is most useful to see what was allocated and by whom. The name of the allocating task is shown as well, but information is only available if MuGuardianAngel was run with the NAMETAG option enabled. If not, the task name will be "????". It is possibly important to remember that the task allocating the memory is *usually* the same task that releases the memory, but AmigaOs does not enforce this.

The line below is again a SegTracker dump of the return PC above.

The next two lines contain the data and address register dumps from when the access fault happened. Note that A6 and A7 are not listed here. A6 has always to be the base address of the exec library, or the program would have crashed earlier. A7 is the stack pointer and is listed as USP: in the line above.

Then come the lines of stack back-trace. These lines show the data on the stack. This dump is activated with the SHOWSTACK option.

If "SegTracker" was installed before MuGuardianAngel, the "---->" lines will display in which seglist the given addresses are in based on the global tracking that "SegTracker" does. If no seglist match is found, no lines will be displayed. One line will be displayed for each of the stack longwords asked for (see the STACKCHECK option) and one line for the PC address of the hit. The lines are in order: hit, first stack find, second stack find, etc. This is useful for tracking down who called the routine that caused the hit.

Next, optionally, comes the data around the program counter when the access fault happened. It is enabled with the SHOWPC option. The first line (PC-8:) is the 8 long-words before the program counter. The second line starts at the program counter - the "return PC" and goes for 8 long words.

The last paragraph is a disassembly of the user code calling an Os function. The PC where program execution will continue is marked by an asterisk. This IS NOT the PC of the faulty instruction that caused the fault - which is usually not available. MuGuardianAngel is usually only able to detect these faults when an appropriate Os function is called, which will return to the code segment shown in the disassembly.

To make this output available, the disassembler.library must be installed and the DISPC option must have been selected.

Let's look at the output of the SNOOP option:

A: # ssssssss @ aaaaaaaa attr: tttttttt PC: pppppppp - MyTask F: # ssssssss @ aaaaaaaa PC: pppppppp - MyTask

The first letter on a line indicates whether the performed operation is an allocation - "A" - or a memory release - "F". The hex number behind "#" is the size of the allocation, the number of bytes the task requested or the number of bytes it releases. The hex number behind "@" is the address of the memory block allocated or to be released.

On allocation, the "attr" field indicates the memory attributes, i.e. the second argument to AllocMem(). The "PC" field is the return PC where program flow will continue. It is *not* the PC where the Os call was performed. The last field is the name of the task performing the operation.

See also the Detail Example for more information.

1.75 Detail Example Hit

Example "MuGuardianAngel" Hit: Click on the field for explanation.

Bad Program!

24-May-99 12:53:02 Mung-wall at 01083bc8 is damaged.

PC : 0105bd6a USP: 013edbbc ( ) ( ) Name: MemoryDevil ----> 013e496a - "MemoryDevil" Hunk 0 Offset 00000032 PCa : 0105bd5a Name: MemoryDevil ----> 013e495a - "MemoryDevil" Hunk 0 Offset 00000022 Data: 0000002a 00000000 00001000 0130fad4 00000001 004f90d7 004bce9f 013e4934

Addr: 013edbb8 013e2858 0101a62c 013e4934 013edbc0 00f9feaa SysBase -------- Stck: 00fa06d6 00001000 0130f634 adadadad adadadad adadadad adadadad adadadad

Stck: adadadad adadadad adadadad 013f3a28 00005da0 013edbf4 013edbf8 013edbfc ----> 00fa06d6 - "ROM - dos 40.3 (1.4.93)" Hunk 0 Offset 00000b2e PC-8: 4eaeff76 4a806718 702a7201 4eaeff3a 4a80670c 22404229 ffff702a 4eaeff2e

PC *: 4e759595 95959595 95959595 9f9f9f9f 9f9f9f9f 9f9f9f9f 9f9f9f9f 9f9f9f9f 0105bd5a : 4a80 tst.l d0

0105bd5c : 670c beq.s $105bd6a 0105bd5e : 2240 movea.l d0,a1

0105bd60 : 4229 ffff clr.b -$1(a1) 0105bd64 : 702a moveq.l #$2a,d0

0105bd66 : 4eae ff2e jsr -$d2(a6) 0105bd6a : *4e75 rts

0105bd6c : 9595 sub.l d2,(a5) 0105bd6e : 9595 sub.l d2,(a5)

0105bd70 : 9595 sub.l d2,(a5) 0105bd72 : 9595 sub.l d2,(a5)

0105bd74 : 9595 sub.l d2,(a5) 0105bd76 : 9595 sub.l d2,(a5)

0105bd78 : 9595 sub.l d2,(a5)

Note that "MuGuardianAngel" hit output is very configurable. The above example hit was produced with the following options:

SHOWSTACK SHOWPC SHOWHUNK NAMETAG DATESTAMP INTRO=Bad_Program! DISPC DISRANGE=16

1.76 Output: Intro string

This is a user defined intro string that gets printed in front of every hit. It is setup with the INTRO option.

1.77 Output: Date Stamp

The date stamp field, if enabled with the DATESTAMP, is at the start of the "MuGuardianAngel" hit. The time is only exact to +/- 1 second.

1.78 Output: Hit

This field describes the type of the problem. In this example, the mung wall of the memory block at 01347000 was overwritten, i.e. the program touched memory out of the allocated range.

See also: MuGuardianAngel error messages for a list of all possible hits.

1.79 Output: Program Counter

This field displays the program counter the code will return to as soon as the Os call finishes. It is not the program counter of the instruction that called the Os function.

Note that this PC need not to be in your program in case an Os function allocates or releases memory. More useful information is then available thru the stack-dump.

On some hits, for example for a failed consistency check, MuGuardianAngel is not able to provide this information. The program counter field will then show up as "00000000".

1.80 Output: User stack pointer

This field gives the contents of the user stack pointer before the Os function was called. The additional data pushed onto the stack by the Os function itself is not included.

On some hits, for example for a failed consistency check, MuGuardianAngel is not able to provide this information. The stack pointer field will then show up as "00000000".

1.81 Output: Forbid/Permit indicator

This field contains status information of the exec system library. It will say "(F)" if the task is in FORBID state, i.e. multitasking is disabled.

1.82 Output: Disable/Enable indicator

This field contains status information about the exec library. It will say "(D)" if the task is in DISABLED state, i.e. interrupts - and hence multitasking - is disabled.

1.83 Output: Task name

This field is the name of the task that called the Os function, causing the hit. In case the task was a CLI task, this will be the name of the CLI module. MuGuardianAngel will always cut down the name to 34 characters, due to space limitations.

In case the task name is invalid, MuGuardianAngel will print four question marks here: "????".

Note, however, that this need not to be the task that was really responsible for the problem. In some cases, it might only be the task detecting the problem. More detailed information about this is in the list of all error messages.

1.84 Output: SegTracker

This symbol "---->" identifies a line produced via the "SegTracker" utility.

This information is therefore only available if the "SegTracker" was started before "MuGuardianAngel" was run.

See the "FindHit" documentation for details as to how to use this information.

1.85 Output: SegTracker Address

This is the address that the hunk/offset describes. This is here such that you can cross-reference it with a value on the stack, in a register, or the program counter. The hunk/offset on the same line are produced when this address is processed via "SegTracker".

This information is therefore only available if the "SegTracker" was started before "MuGuardianAngel" was run.

See the "FindHit" documentation for details as to how to use this information.

1.86 Output: SegTracker Name

This is the name of the file, as passed to LoadSeg, which was found to be loaded around the address given, made available by the "SegTracker" utility. This information is therefore only available if the "SegTracker" was started before "MuGuardianAngel" was run.

See the "FindHit" documentation for details as to how to use this information.

1.87 Output: Hunk

This is the hunk in the load file that was loaded around the given address, made available by the "SegTracker"

This information is therefore only available if the "SegTracker" was started before "MuGuardianAngel" was run.

See the "FindHit" documentation for details as to how to use this information.

1.88 Output: Offset

This is the offset from the start of the hunk that this address is at within the given load file. This information is therefore only available if the "SegTracker" was started before "MuGuardianAngel" was run.

See the "FindHit" documentation for details as to how to use this information.

1.89 Output: Return PC of the allocating function

This information is only available if a problem during a FreeMem()/FreeVec() was detected. It displays the return PC of the corresponding memory allocation function, i.e. the address of the code the allocating Os function returned to at the time the memory was allocated.

1.90 Output: Name of the allocating task

This information is only available if a problem during a FreeMem()/FreeVec() was detected. It displays the name of the task that made the corresponding allocation, if available. Note that this need not to be the same task as the current task, making the deallocation.

If MuGuardianAngel cannot provide this name, this field will say "????". The NAMETAG option is required to enable this extended information.

1.91 Output: Data Register Dump

This line contains a dump of the data registers right before the failing Os function was called, if available. All parameters for the Os call will be included in this list.

In some situations, for example for a failed consistency check, MuGuardianAngel will not be able to provide this information.

1.92 Output: D0 Register

The D0 register of the 680x0 CPU.

See Data:

1.93 Output: D1 Register

The D1 register of the 680x0 CPU.

See Data:

1.94 Output: D2 Register

The D2 register of the 680x0 CPU.

See Data:

1.95 Output: D3 Register

The D3 register of the 680x0 CPU.

See Data:

1.96 Output: D4 Register

The D4 register of the 680x0 CPU.

See Data:

1.97 Output: D5 Register

The D5 register of the 680x0 CPU.

See Data:

1.98 Output: D6 Register

The D6 register of the 680x0 CPU.

See Data:

1.99 Output: D7 Register

The D7 register of the 680x0 CPU.

See Data:

1.100 Output: Address Register Dump

This line contains a dump of the data registers right before the failing Os function was called, if available. All parameters for the Os call will be included in this list.

In some situations, for example for a failed consistency check, MuGuardianAngel will not be able to provide this information.

Address register A7 is the user stack pointer and therefore available elsewhere. Register A6 is always a pointer to the exec.library and therefore not included.

1.101 Output: A0 Register

The A0 register of the 680x0 CPU.

See Addr:

1.102 Output: A1 Register

The A1 register of the 680x0 CPU.

See Addr:

1.103 Output: A2 Register

The A2 register of the 680x0 CPU.

See Addr:

1.104 Output: A3 Register

The A3 register of the 680x0 CPU.

See Addr:

1.105 Output: A4 Register

The A4 register of the 680x0 CPU.

See Addr:

1.106 Output: A5 Register

The A5 register of the 680x0 CPU.

See Addr:

1.107 Output: A6 Register

The A6 register of the 680x0 CPU. Since this address register is fixed to the base address of the exec.library for the Os functions that MuGuardianAngel watches, it is not included in the dump.

See Addr:

1.108 Output: A7 Register

The A7 register of the 680x0 CPU is always the user stack pointer. It is shown in the USP field of the MuGuardianAngel hit and not included here.

See Addr:

1.109 Output: Stack Dump

These lines contain stack dumps from the task that caused the "MuGuardianAngel" hit. It can be used to figure out what the program was doing and what routines called the current routine by looking at the values on the stack.

This output is enabled with the SHOWSTACK option, but even if it is enabled, MuGuardianAngel will not be able to give a stack dump in some situations. This happens, for example, if a memory consistency check failed.

1.110 Output: Stack Word

This is a longword on the stack of the task that caused the hit. Stack dumping is enabled by the SHOWSTACK option.

See Stck: for more details.

1.111 Output: Show PC

If the SHOWPC option is turned on, "MuGuardianAngel" will dump the 8 longwords before the return PC and the 8 longwords starting at the PC. The return PC is the address of the code the Os function will return to after completion, not the address of the Os call itself.

This can be used to help debug programs by being able to look at the code around the hit by disassembling it.

1.112 Output: Show PC-$20

This is the longword at the memory address (PC - $20) where PC is the return PC of the failed Os call.

1.113 Output: Show PC-$1C

This is the longword at the memory address (PC - $1C) where PC is the return PC of the failed Os call.

1.114 Output: Show PC-$18

This is the longword at the memory address (PC - $18) where PC is the return PC of the failed Os call.

1.115 Output: Show PC-$14

This is the longword at the memory address (PC - $14) where PC is the return PC of the failed Os call.

1.116 Output: Show PC-$10

This is the longword at the memory address (PC - $10) where PC is the return PC of the failed Os call.

1.117 Output: Show PC-$0C

This is the longword at the memory address (PC - $0c) where PC is the return PC of the failed Os call.

1.118 Output: Show PC-$08

This is the longword at the memory address (PC - $08) where PC is the return PC of the failed Os call.

1.119 Output: Show PC-$04

This is the longword at the memory address (PC - $04) where PC is the return PC of the failed Os call.

1.120 Output: Show PC+$00

This is the longword at the memory address (PC) where PC is the return PC of the failed Os call.

1.121 Output: Show PC+$04

This is the longword at the memory address (PC + $04) where PC is the return PC of the failed Os call.

1.122 Output: Show PC+$08

This is the longword at the memory address (PC + $08) where PC is the return PC of the failed Os call.

1.123 Output: Show PC+$0C

This is the longword at the memory address (PC + $0C) where PC is the return PC of the failed Os call.

1.124 Output: Show PC+$10

This is the longword at the memory address (PC + $10) where PC is the return PC of the failed Os call.

1.125 Output: Show PC+$14

This is the longword at the memory address (PC + $14) where PC is the return PC of the failed Os call.

1.126 Output: Show PC+$18

This is the longword at the memory address (PC + $18) where PC is the return PC of the failed Os call.

1.127 Output: Show PC+$1C

This is the longword at the memory address (PC + $1C) where PC is the return PC of the failed Os call.

1.128 Output: Disassembler output

This line is generated by the disassembler, it presents the code in Motorola opcode notation around the PC where the fault was detected.

1.129 Output: Disassembler output with PC

This line is the disassembly of the instruction where the fault was detected. It is marked by an asterisk "*" in front of the opcodes. The actual Os call that caused the MuGuardianAngel hit will be the instruction immediately above this line.

1.130 Index

A...

Output: A0 Register Output: A1 Register Output: A2 Register Output: A3 Register Output: A4 Register Output: A5 Register Output: A6 Register Output: A7 Register Output: Address Register Dump Hit: Allocation of .. bytes failed Hit: Alloc/FreePooled from NULL pool attempted Option: ALLOWREUSE Option: AREGCHECK

C...

Hit: Clear memory cookie at .. was found defective Command line options and tooltypes Option: CONSISTENCY Correct memory handling: Avoiding hits Hit: CreatePool failed Hit: CreatePool puddleSize .., thresSize .. invalid Credits : Thank you goes to...

D...

Output: D0 Register Output: D1 Register Output: D2 Register Output: D3 Register Output: D4 Register Output: D5 Register Output: D6 Register Output: D7 Register Output: Data Register Dump Output: Date Stamp Option: DATESTAMP Option: DEBUG Hit: DeletePool of NULL pointer attempted, ignored Detail Example Hit Option: DISABLEBELL Output: Disable/Enable indicator Output: Disassembler output Output: Disassembler output with PC Option: DISPC Option: DISRANGE Option: DREGCHECK

E...

MuGuardianAngel error messages Example MuGuardianAngel output

F...

Option: FILLCHAR Hit: FreeMem of NULL pointer attempted, ignored Output: Forbid/Permit indicator

H...

History Output: Hit Hit: Release of ..: alloc. length .. <> released length .. Hit: Release of ..: allocation pool .. <> release pool .. Hit: Allocation of .. bytes failed Hit: Alloc/FreePooled from NULL pool attempted Hit: Clear memory cookie at .. was found defective Hit: CreatePool puddleSize .., thresSize .. invalid Hit: CreatePool failed Hit: DeletePool of NULL pointer attempted, ignored Hit: FreeMem of NULL pointer attempted, ignored Hit: Invalid FreeMem of memory at .. Hit: Invalid memory list Hit: .. called from interrupt or supervisor mode Hit: Mem header ..: mem chunk .. <lower or >upper Hit: Mem header ..: mem chunk after .. misaligned Hit: Mem header ..: mem chunk .. order broken Hit: Mem header ..: mem lower is larger than upper Hit: MemHeader of chunk .. is not available Hit: Mem header ..: free counter .. <> .. in chunks Hit: Mem header ..: node type is not NT_MEMORY Hit: Memory pool .. garbled Hit: Memory at .. released twice Hit: Misaligned release of .., size .. performed Hit: MuGuardianAngel patches have been overwritten Hit: Mung-wall at .. is damaged Hit: Task ..: nearly out of stack Hit: Null-sized allocation, ignored Hit: Task ...: stack overflow Hit: Task ...: stack underflow Output: Hunk

I...

Index Installation of MuGuardianAngel Option: INTRO Output: Intro string Hit: Invalid FreeMem of memory at .. Hit: Invalid memory list Hit: .. called from interrupt or supervisor mode

L...

Option: LED The THOR-Software Licence

M...

Option: MAX Hit: Mem header ..: chunk .. <lower or >upper Hit: Mem header ..: chunk after .. misaligned Hit: Mem header ..: chunk .. order broken Hit: Mem header ..: lower is larger than upper Hit: MemHeader of chunk .. is not available Hit: Memory at .. released twice Hit: Mem header ..: free counter .. <> .. in chunks Hit: Memory header ..: node type is not NT_MEMORY Hit: Memory pool .. garbled Option: MIN Hit: Misaligned FreeMem of .., size .. performed What's the MMU.library? MuGuardianAngel error messages MuGuardianAngel restrictions Hit: MuGuardianAngel patches have been overwritten Option: MUNGLIS Hit: Mung-wall at .. is damaged

N...

Output: Name of the allocating task Option: NAMETAG Hit: Task ..: nearly out of stack Option: NOMMU Hit: Null-sized allocation, ignored

O...

Output: Offset Option: ALLOWREUSE Option: AREGCHECK Option: CONSISTENCY Option: DATESTAMP Option: DEBUG Option: DISABLEBELL Option: DISPC Option: DISRANGE Option: DREGCHECK Option: FILLCHAR Option: INTRO Option: LED Option: MAX Option: MIN Option: MUNGLIST Option: NAMETAG Option: NOMMU Option: POSTSIZE Option: PRESIZE Option: PRIORITY Option: QUIT Option: SHOWFAIL Option: SHOWHUNK Option: SHOWPC Option: SHOWSTACK Option: SNOOP Option: STACKCHECK Option: STACKLINES Option: STACKSNOOP Option: TASK Option: TINY Option: TOMUFORCE Option: WAIT Output: A0 Register Output: A1 Register Output: A2 Register Output: A3 Register Output: A4 Register Output: A5 Register Output: A6 Register Output: A7 Register Output: Address Register Dump Output: D0 Register Output: D1 Register Output: D2 Register Output: D3 Register Output: D4 Register Output: D5 Register Output: D6 Register Output: D7 Register Output: Data Register Dump Output: Date Stamp Output: Disable/Enable indicator Output: Disassembler output Output: Disassembler output with PC Output: Forbid/Permit indicator Output: Hit Output: Hunk Output: Intro string Output: Name of the allocating task Output: Offset Output: Program Counter Output: Return PC of the allocating function Output: SegTracker Output: SegTracker Address Output: SegTracker Name Output: Show PC Output: Show PC+$00 Output: Show PC+$04 Output: Show PC+$08 Output: Show PC+$0C Output: Show PC+$10 Output: Show PC+$14 Output: Show PC+$18 Output: Show PC+$1C Output: Show PC-$04 Output: Show PC-$08 Output: Show PC-$0C Output: Show PC-$10 Output: Show PC-$14 Output: Show PC-$18 Output: Show PC-$1C Output: Show PC-$20 Output: Stack Dump Output: Stack Word Output: Task name Output: User stack pointer Hit: Task ..: stack overflow

P...

Option: POSTSIZE Option: PRESIZE Option: PRIORITY Common Problems Output: Program Counter

Q...

Option: QUIT

R...

Hit: Release of ..: alloc. length .. <> released length .. Hit: Release of ..: allocation pool .. <> release pool .. MuGuardianAngel restrictions Output: Return PC of the allocating function

S...

Output: SegTracker Output: SegTracker Address Output: SegTracker Name Option: SHOWFAIL Option: SHOWHUNK Output: Show PC Output: Show PC+$00 Output: Show PC+$04 Output: Show PC+$08 Output: Show PC+$0C Output: Show PC+$10 Output: Show PC+$14 Output: Show PC+$18 Output: Show PC+$1C Output: Show PC-$04 Output: Show PC-$08 Output: Show PC-$0C Output: Show PC-$10 Output: Show PC-$14 Output: Show PC-$18 Output: Show PC-$1C Output: Show PC-$20 Option: SHOWPC Option: SHOWSTACK Option: SNOOP Output: Stack Dump Output: Stack Word Option: STACKCHECK Option: STACKLINES Option: STACKSNOOP System patches installed by MuGuardianAngel

T...

Option: TASK Output: Task name The THOR-Software Licence Option: TINY Option: TOMUFORCE Command line options and tooltypes

U...

Hit: Task ..: stack underflow Output: User stack pointer

W...

Option: WAIT What's the job of MuGuardianAngel? What's the MMU.library?

1.131 History

Release 40.1:

This is the first official release. MuGuardianAngel is a remake of the original CBM "GuardianAngel" program. But unlike this program, MuGuardianAngel works. (-: It combines the features of MungWall, MemSniff and GuardianAngel in a mmu.library conformal program.

Release 40.2:

Fixed a problem on installation that caused a few "bogus" exceptions, dependent on the installed software.

Release 40.3:

Changed the output style of the SegTracker lines a bit, back to the old style to be compatible with tools interpreting this automatically.

Release 40.4:

Made the error messages more verbose, added more information to the "hit" header. Fixed a tiny bug in one of the hits which could have reported a wrong stack pointer.

Release 40.10:

Fixed a bug in the SHOWSTACK option. Made the timer handling slightly more careful. Added the DISPC and DISRANGE options via the disassembler.library.

Release 40.11:

Fixed a bug in the MemHeader free count mismatch error handler, did not restore one register properly and hence caused subsequent hits.

Release 40.14:

Fixed a bug in the MemHeader free count mismatch output, the memheader argument was not forwarded correctly to the output routine and the error message was hence broken. Added explicit checks whether the MuGuardianAngel patches have been overwritten. MuGuardianAngel will warn you now in case this happens, and hence someone is asking for trouble.

Release 40.15:

I was puzzled to find that some system handlers are really low on stack and might easely cause stack overflows if run with MuGuardianAngel. The 40.15 release is now smart enough to hold and offer a tiny - but large enough - "emergency stack" which is used in case MuGuardianAngel detects this situation. Furthermore, a warning message will be printed. Problematic tasks are the FFS(!) and the system RAM-DISK(!), both because they're too low on stack. The first must be fixed by hand, by a HD

installation utility, the second by "PatchRAM", which is part of the distribution. Futhermore, stack snooping has been extended to the memory allocation functions.

Release 40.20:

Added the "ALLOWREUSE" option to keep MuGuardianAngel happy even if a program accesses memory after releasing it with FreeMem() without proper Forbid()ing before. Note that programs that attempt to do so are buggy. Added the "NOMMU" option to turn off the MMU magic, even though a working MMU is still checked on startup. Added consistency checks in front of the memory allocation and release functions for optimal safety. Added checks for the memory pool functions - these procedures are now snooped and watched as well.

Release 40.21:

The CreatePool() patch treated a puddle size equal to the thresh size as an error. I still regard this as a "not so smart" idea to make the thresh size that large, but MGA accepts this now for compatibility reasons. Made some information/debug messages more informal by including more information. Added a consistency check in front of the AllocMem()/FreeMem() and other functions to inform the user as early as possible about problems.