BadMEM-HOWTO

Nico Schmoigl

BadMEM-HOWTO

by Nico Schmoigl

Published $Date: 2001/08/15 18:42:16 $

How to use bad memory modules in a Linux Box (still needs a special patch for the kernel)

Table of Contents

1. About this document **1**
1.1. New versions of this document 1
1.2. Who maintains this? 1
1.3. Topic of this document 1
1.4. ”I found a bug in this document!” 1
1.5. Copyright 1
1.6. Thanks 1
2. History of BadRAM/BadMEM **2**
3. Basics / Concepts **3**
3.1. Some words before we start 3
3.2. Requirements for the understanding of this Document 3
3.3. How does one know that you might have a bad ram module? 3
3.3.1. Lock ups 3
3.3.2. Errors at runtime 3
3.4. A bit of theory 4
3.5. Speaking conventions 4
3.6. Note on non-ix86 computers 5
3.7. File requirements 5
4. Getting it working **6**
4.1. The most important thing: Proof! 6
4.2. Make BadMEM run 7
5. Only one module – what is different? **9**
5.1. General discussion 9
5.2. Special case: Twin-Banks 9
6. The BadMEM utilities **11**
6.1. Overview 11
6.2. A small paragraph on „badmemlib” 11
7. Further hints **12**
7.1. Troubleshooting – if you have more than one module 12
7.2. Modules might change by the time 12
7.3. Advantages and disadvantages on installing Memtest86 on disk or hard disk 12
7.4. MODSYSTEM: An unsolved issue 12
8. The future **14**

Chapter 1. About this document

1.1. New versions of this document

This HOWTO is available in HTML and SGML. You can download it from http://badmem.sourceforge.net/ (http://badmem.sourceforge.net). It is part of the badmem-utils package.

1.2. Who maintains this?

It’s me ;-) Nico Schmoigl, nico@writemail.com (mailto:nico@writemail.com).

1.3. Topic of this document

This HOWTO is about how to use memory modules which have (some or even many) bad bits. It should give you partical hints. Only theory which is absolutely necessary for understanding will be mentioned.

1.4. ”I found a bug in this document!”

Errata humanum est! (latin: ”Making errors is human”) *g* But, please drop me short email so I can correct this.

Thank your very much for your support!

1.5. Copyright

1.6. Thanks

Thanks to the LDP team especially to everyone who is involved in writing the LDP-Author-Guide as well as the HOWTO-HOWTO. Without their help, this document would not be what it is now.

Chapter 2. History of BadRAM/BadMEM

Well, this chapter is very short as BadRAM/BadMEM is very young:

Being developed as additional patch for the 2.2.x kernel series, BadRAM was programmed by Rick van Rein in early 2000. His efforts aim towards mathematic simplicity by using high efficient code. Stability is the aim. Support for all platforms (ix86, Motorala, Alpha, etc.) is one of the main issues.

During this development in 2001, I decided that quick development is only possible with special tools and addons for the patch, to enable a quick debugging. What follows is an increase in complexity of the patch. Simplicity will not be hold. So, the two ways went from each other, resulting in two (almost) independend patches:

- BadRAM-patch: the „father", programmed and still maintained by Rick van Rein, and
- BadMEM-patch: the „child", developed by myself.

Please note this major difference between these two attempts.

Chapter 3. Basics / Concepts

3.1. Some words before we start

As you know from the chapter before, BadMEM is still very young. So, it is very likely that many things will change very much in the future. Therefore, please keep up to date and *ALWAYS* use the latest version of this Document.

3.2. Requirements for the understanding of this Document

The driver is still in its experimental phase. So, you need a certain knowledge on the kernel to get it working. As this implies that you know some standard things like how to

- download, configure, compile and setup a specific Linux Kernel,
- activate the Kernel via LILO,
- run a patch on top of the Kernel,

I will not go into these details. If you do not know them, then this HOWTO will not help you much. Under this condition I will only advise you to read the Kernel-HOWTO and the TIPS-HOWTO. So this HOWTO is not directed towards the beginner, but towards the novice/professional. Please keep this in mind while reading this Document.

3.3. How does one know that you might have a bad ram module?

There are two main classes of appearence:

3.3.1. Lock ups

You can be very happy, if you encounter such a clear situation. If your Kernel completely locks up with an Ooops (and you did not make any „bad stuff' ;-)), you can be quite sure that there is a hardware failure somehow. Often the problem is caused by the ram modules. Some very clear signs are

- if your kernel halts with an Ooops and an error reference to a NULL pointer assignment
- if you can start up your system but „randomly" processes are dying, or suddenly you cannot start new programs.

If you then repeat your last commands (for example, rebooting) and everything seems to be fine again, then it is a quite sure sign that you memory modules are buggy...

3.3.2. Errors at runtime

Surprisingly many problems which are very very hard to track down are memory problems. This is often caused by the fact that the memory modules are good at first, but as the time goes by, the modules get warmer and warmer. Then suddenly a chip on the module gets too hot, and it swaps one or more bits. If you do not have allocated this memory area, it will not hurt you at all, but if this page (=sepcific area of memory) is used for example by the kernel... well, I will leave off what will happen - you can guess it. So, typical signs are

- if suddenly your compiler stops during runtime
- if your programs are receiving unwanted TERM or code 15 (sometimes code 11, too) signals and therefore exit at once
- if you encounter memory „randomization" at any kind (for example if you writing a program in C, compile it and you are very sure that this variable *MUST* have that value, but it does not have it!

Oh, just some sentences about the usage of „random": I am no friend of this word especially, if we are discussing about computer science. Nothing really happens „at random" and after all if you look hard enough, you will always find some good (but mainly only bad) reasons and explainations why something happens or not. So I please do not understand my usage of *random* as totally „random", but as „appearing as if it happens randomly".

3.4. A bit of theory

First of all, a bit of theory. What is this all about?

As all your data is normally stored in RAM first, it harms your system if one ore more modules are broken. Imagine the situation if you have a very sophisticated program in your memory and one or another byte just changes without that you notice. Now the CPU runs over this data and interprates this totally differently. What must happen: The program does not do what it should do or even crashes.

What is the solution then? Well, there are even two solutions:

- You can use 100%-always-ok chips and memory modules (GoodRAM)
- You can use not-so-ok chips and memory modules, but then you must take care that you do not use such areas which are problematic

Normally you do the first alternative. And therefore you pay a very high price at your local dealer. Production of those 100% chips is very expensive and a vast amount of bad chips are produced before you get a 100%-ok-chip.

The BadRAM/BadMEM system hooks in there. It uses the second alternative already mentioned. During boot up phase of your Linux kernel, it locks certain areas of RAM. By doing that it ensures that the kernel never will use this area and therefore will never trap in such bad memory. During allocation it just skips those bad areas.

3.5. Speaking conventions

For a better understanding I now want to introduce the following expressions:

- *BadRAM* (sometimes *BadMEM*, too) is a memory module which has one or more bad memory areas.
- *GoodRAM* the opposite of BadRAM, that is your ram you normally get, if you go to your local dealer (or at least: you should get from him/her).

If you are interested in a more detailed specification of these words, please have a look at the BadRAM-4096 specification which is downloadable from my page (URL is above).

3.6. Note on non-ix86 computers

NOTICE: BadMEM currently does only work with i386 processors! Alphas/Motorals etc. are not supported yet (at least be BadMEM). The main aim of BadMEM is not the port towards other system types. If you can do testing on non-ix86 machines, please contact Rick (BadRAM), who is very interested on your aid.

3.7. File requirements

Now let's start with practice!

You need the following files from the Internet:

- *The BadMEM patch*: You can get it from http://badmem.sourceforge.net. The BadRAM patch by Rick van Rein can be downloaded from http://home.zonnet.nl/vanrein/badram. Please note that you must use the correct patch version for your kernel! Anway, normally an elderly patch will work for a newer version as long as there is no major change in the memory mapping routines of the kernel.
- *The BadMEM utility package* (optional, only needed if you have badmem-patch-v4.6 or above): Can be downloaded from http://badmem.sourceforge.net, too. Besides it is needed for the compilation of the kernel, it prvides several useful small utilities which could make your life easier once you are a bit more experienced in BadMEM.
- *The Linux Kernel*: You can get it from http://www.kernel.org or any mirror of that. Please note that the kernel must fit to the patch (see above)!
- *memtest86*: This is the memory testing program. You can download it from http://www.memtest86.com. As of writing, the current version is 2.7 is.

Chapter 4. Getting it working

There is a way of enabling BadMEM correctly without the need and use of a screw driver, but I do not recommend this way because it is likely that you will make this or that mistake which could leave your system in a unstable condition. As I expect you not to be very familiar with BadRAM, I will now describe a more secure way:

4.1. The most important thing: Proof!

There are two principles which you must always have in mind while doing anything at all with BadMEM:

- Proof everthing
- Work thoroughly

From now on, I expect that you have at least two memory modules: one which is ok (called the good module) and a supicious module. If you just have one module, please read the "Only one module" section below later on.

The first thing you should do, is to remove the memory module you expect to be faulty. Then start your system normally. Do some work (approx. 10 minutes) and see if you encounter any traps, errors or bugs which could be related to bad ram modules. If this happens, it is likely that you have either

- taken the wrong module out of your PC (most likely) - alignment of the memory modules vary heavy from board to board - or
- you have two BadMEM modules (not very likely) or
- you have a different problem as BadMEM! Possibly this could be related to a bad driver, misconfiguration, bug in your kernel/application, other faulty hardware or whatever.

If you do not encounter any problems, you can ”mark” the memory module in the PC as good for a while.

Now, download memtest86 (if you have not done this, yet), unpack it and compile it. Have a least a short look into documentation. Although often stated contrarily, I recommend to install memtest86 to disk. Advantages and disadvantages will be discussed later in the ”Further hints” section.

Now start your PC with memtest86 and let it run at least through the first three tests on your good RAM (from now on I will call this a ”quick memtest” as time consumption is relative low). If it finds some bad areas there (the error number to the right top increases), you have a problem. Then you *must* regard this module as bad! Try to use the memory module you took off you PC shortly and redo the memtest on this module. If both modules are bad, put aside one of the modules and go on with only one module. Be sure to have read the ”Only one module” section, then!

As soon as one module has passed the quick memtest successfully, you can call it GoodRAM. For everything we will do now, this module will remain in the first bank of you motherboard (see your vendor’s manual to see which is it - mostly the first bank is called „Bank 0”). Do not remove it for any test as you normally need at least some good area of RAM for loading the Linux Kernel and memtest!

Now add the bad module to your PC putting in a higher bank. Naturally your total RAM size will increase - but note that the (maybe) faulty RAM is now located *above* the good RAM. If you now start memtest, you can just run the tests on the module you expect to be faulty. Configuration is done by

pressing the ’c’ key. The configuration window there will help you. Please note to switch to the BADRAM command line output by selecting (6) (2) from the configuration menu. Now, take another cup of coffee or go sleeping, depending on what memory speed and processor type you have. I recommend to pass at least the first six tests to be sure what is up with that memory module. While scanning the memory module memtest collects information what is wrong. The badram line which is prompted to your screen lists a combination what has to be locked by BadMEM, if you want use this module. Short tip: The more zeros are in every second hexadecimal number, the worse is your memory module.

As soon as the tests are through, copy the last badram command line somewhere to a (analogous *g*) sheet of paper. We will need the parameters later for locking the bad memory areas. If you do not have at least one line, all your RAM is ok. You have then no need to install the BadRAM. The problems you might have encountered before are most likely to be not memory related. Although you could rerun the long test on the first memory module, you thought was ok. Perhaps you can find a bad bit there...

4.2. Make BadMEM run

First, take the bad module off the PC - BadRAM is not enabled yet and therefore data will be written on bad areas. This would crash your system or do other harming things which are a bad-thing (tm) for the moment.. At least, it might be quite problematic compiling the kernel - and this is the next aim.

If you downloaded a BadMEM patch (only version 4.6 or above), you need to download (if not already done), unpack and install the badmem-utils package. From now on, I assume that you have installed this package into /usr/src/badmem. If you have chosen another directory, simply replace any occurance of this directory with that you have created.

Anyway, download (if you have not done this), unpack, configure your kernel as you are used to do it. Then apply the BadMEM patch on top of this kernel, by using the command

```
bash$ patch -p0 < patch-name.diff
```

Have a look on the kernel configuration (*make menuconfig* or *make xconfig* or something similar) again. Note, that there are several new options in the Gerneral Setup area, controlling the BadMEM options. I recommend the following configuration for BadMEM:

```
[*] Work around bad spots in RAM (BadMEM-patch)
        path to the BadMEM utilities package: "/usr/src/badmem"
[ ]   Enable BadRAM debug messages during kernel boot
[*]   /proc fs support
[*]   Extended Module support.
        Configuration file name: "/etc/badmem.conf"
```

The most important option is the last one - you must check it to remain synchronous to this HOWTO!

The next to do is to supply BadMEM with the layout of you memory modules. This is done with the help of the configuration file */etc/badmem.conf*. There you need the data you just got to know by the tests of memtest. The configuration language is documented in linux/Documentation/badmem_conf.txt. From now on I call the first memory module which is good, the module with name "good" and the bad one with the name "bad". I recommend to use the same names in /etc/badmem.conf. Please make sure to have read the note on the MODSYSTEM below, if you make changes by your own which are beyond the scope of this HOWTO.

As soon as you wrote your BadMEM configuration file, you can compile the kernel as normal. Install it as you like it. During compilation, the /etc/badmem.conf is read and parsed. All your data will be *hard-coded* into the kernel, so if you make any change to your configuration file, you need to recompile your kernel! Please append the following command line to the kernel by adding it to the append line of LILO:

```
badmem=good,bad
```

This tells the driver to first lock the "good" module (well - it simply does not have anything to lock) and then take the "bad" module. Do not forget to run lilo before rebooting!

Reboot the system and put your broken RAM module into your PC. Boot the new badram kernel and see rushing by the boot messages. Login as usual. If you have selected /proc fs support above, you can have a look into the file */proc/badmem*. It lists you all pages (area of 4096 bytes of RAM) which are disable by BadRAM.

As long as you do not change anything on your memory configuration, you can run your system as normal. If memtest found all the bad areas, they are marked as bad and will not be used by neither the kernel nor any application. The system can be used at full load.

Chapter 5. Only one module – what is different?

5.1. General discussion

Well, you might encounter severe problems. If there are bad areas in the low regions of your module, it is sure that Linux will have severe problems booting up. Even memtest might have problems although it only uses around 26k of RAM for itself. On rare situations it is even possible that your BIOS does not work properly.

The situation may cause that the kernel might be loaded into a bad area page. It is not possible to lock a memory page which is already in use - and if the kernel is there, it is not even in use, it is unreallocable because it is blocked and reserved by the kernel. The kernel is the only piece of work which must have linear and therefore error-free memory (in future version, there might be a certain chance that there will be a work-around for this - but this is still a theoretical issue).

The only advice I can give to you, is to check whether you have a bad area in that low region or not. If that happens, you will know it very soon. And if you boot up with the BadMEM driver, the Kernels stops with an error message telling you that you wanted to lock a kernel page, you know where the problem is.

If this happens you may only have one chance: Make the built-in part of the kernel as small as possible. All other drivers may be loadable with the help of the loadable module support. With this trick, you may be lucky...

In any case: I everybody advice to have at least one GoodRAM module in Bank 0. It is the best and cleanest solution. This module need not be huge, but the (uncompressed) Kernel should fit in there at least. So 4-8MB should be the minimum.

5.2. Special case: Twin-Banks

If you encounter a bad area problem in the lower Kernel region and are using Twin-banking modules (for example PS/2 EDO, PS/2 Fastpage Mode or SIMM - the latter are even Quad-banking), then you are in a better situation than those who only have a single-banking system: You can try to exchange the modules so that the buggy module gets to the upper end of this (logical) bank. Here is an example for this:

Imagine the following setup:

- Bank 0: Module A (BadRAM, hole in the lower region)
- Bank 1: Module B (almost clear, hole in the higher region)

The result will be that this combination is not bootable. Now, if you exchange these modules, you will get:

- Bank 0: Module B (with hole in the upper part)
- Bank 1: Module A (BadRAM)

The result is clear: With this setup you might have a better chance that it is bootable. Even better it is, if your „Module B” is even a GoodRAM; then, all your problems will be solved.

Please note: If changing this way, you *must* rerun memtest over the new setup. If using the MODSYSTEM, please read the question about the MODSYSTEM in the section „Futher Hints” below.

Chapter 6. The BadMEM utilities

6.1. Overview

First of all, I must confess that these utilties are not necessary for a working BadMEM setup. They are „just another aid” for you, either during debugging or for information purposes. So, if you are just interested getting your setup working, you need not read this section (although it might inform you about some handy features) and you can totally skip this section.

As of writing (4/2001) the utility package consists of the following parts:

- badmemlib: The main routine library with many handy routines about altering any BadMEM pattern line.
- badmemcmp: Compare two badmem patterns about their differences (something similar to the unix-command „diff”).
- badmemmerge: Merge two BadMEM pattern lines to one
- badmemshift: Shifts a given BadMEM pattern line by a given offset
- badmemtype: Returns the type of a BadMEM module defined by its BadMEM pattern line.

All parts have a short documentation on how to use them. As these commands are very simple, atomic and unix-standard, I do not want to give hundreds of examples which will not be read anyway.

6.2. A small paragraph on „badmemlib”

badmemlib is the central library with many functions to alter, patch and create pattern lines. It can be installed both as shared library and statically. Please note that it is necessary for all the programs above. After its installation, a new header file in /usr/local/include is created with all the routines which may be used externally. Please make sure that - if you use this library - that you both have linked *badmem* and *m* (the GNU standard mathematical library) with your program, for example:

```
bash$ gcc -o someprogram someprogram.c -lbadmem -lm
```

The library internally uses the math library, so you must insert it into your program, too.

Chapter 7. Further hints

This section is a sort extract of my knowledge I learnt during the programming of BadRAM. If you have different or new experience which should be stated here, please email me (mailto:nico@writemail.com).

7.1. Troubleshooting – if you have more than one module

During a test, I had two bad modules and two good ones. I installed them randomly into the banks (my motherboard can take up to four SDRAM-DIMMs) and ran Memtest over them. Then I reinstalled them in another combination and Memtest then found totally different areas of bad RAM - not just the badram command line was naturally different, but locking was different, too. Pages which were bad, had been suddenly good and vice versa - not just by mistake but by usage. I put it to some different timing modes which the memory chip on my motherboard used during tests, although I disabled all my automatic routines in my BIOS.

So, be careful with changing memory modules in the banks. And if you do, only trust those reports you get in this combination!

This is still an unsolved issue. Sadly, my memory knowledge is not big enough to go into that area of memory usage.

7.2. Modules might change by the time

I have a very strange module here: It is a 64MB SDRAM-DIMM RAM, PC-100, 8ns. This is not very strange by itself, but when I got it from my local dealer, it had around 25 percent of bad pages. I ran memtest over it several times (different tests) and suddenly all the bad pages disappeared! Until today, it runs without any badram parameter - even if I disable BadRAM completely, it works. And Wind*ws can use it without any crash! This is very amazing - I cannot guess what is going on.

7.3. Advantages and disadvantages on installing Memtest86 on disk or hard disk

The main advantages of installing Memtest86 to a disk are

- You can carry it to any place you like and run it on any PC you desire
- The binary files do not eat up much; so copying times are quite low
- If you do not like the program (which I want to doubt), you can really "trash" it ;-))

The main disadvantages are:

- If not formatted correctly, a disk can loose data
- You might dislike the usage of diskettes

7.4. MODSYSTEM: An unsolved issue

The original idea of the new MODSYSTEM (Extended Module support) is - on the one hand - to make bad modules relocatable without the need of rechecking it by memtest. On the other hand it enables you to parse huge numbers of patterns which need to be locked.

Although almost all motherboards and memory management chips on them are designed to use a linear mapping method, this is not always reality. Rick has proven (!) many differences. In practise this means: Only insert your modules in that way, you have a complete memtested BadRAM pattern line. If you change it and enter a different way of module insertion in the badmem command line, there is *NO garantee* that all your bad areas are locked!

This still is an unsolved issue. I am working on this problem, but debugging on this is

- very time-consuming and
- problematic

So, please be patient. I will notify you on the case of any new discovery.

Chapter 8. The future

My next aims for BadMEM are:

- Find bugs, if possible
- Seek integration into the standard kernel code serie
- Find testing support and more BadRAM modules to enhance our working conditions and surrounding
- Add other features like
  - small locking page sized with the new stubs code in the Kernel
  - enhancing this (and other) documents to help you getting along with BadMEM
- Make everything more handy for a usage by end-users and distributions in the future