

Libraries_Manual

COLLABORATORS

	<i>TITLE :</i> Libraries_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries_Manual	1
1.1	Amiga® RKM Libraries: D Troubleshooting Guide	1
1.2	D Troubleshooting Guide / Errors	1
1.3	D / Errors / Audio--Corrupted Samples	2
1.4	D / Errors / Character Input/Output Problems	2
1.5	D / Errors / CLI Error Message Problems	2
1.6	D / Errors / CLI Won't Close on RUN	2
1.7	D / Errors / Crashes and Memory Corruption	3
1.8	D / Errors / Crashes--After Exit	3
1.9	D / Errors / Crashes--Only on 68000 and 68010	4
1.10	D / Errors / Crashes--Only on 68040	4
1.11	D / Errors / Crashes--Subtasks, Interrupts	4
1.12	D / Errors / Crashes--Window Related	4
1.13	D / Errors / Crashes--Workbench Only	5
1.14	D / Errors / Device-related Problems	5
1.15	D / Errors / Disk Icon Won't Go Away	5
1.16	D / Errors / DOS-related Problems	5
1.17	D / Errors / Fails only on 68020/30	5
1.18	D / Errors / Fails only on 68000	6
1.19	D / Errors / Fails only on Older ROMs or Older WB	6
1.20	D / Errors / Fails only on Newer ROMs or Newer WB	6
1.21	D / Errors / Fails only on Chip-RAM-Only Machines	7
1.22	D / Errors / Fails only on machines with Fast RAM	7
1.23	D / Errors / Fails only with Enhanced Chips	7
1.24	D / Errors / Fireworks	7
1.25	D / Errors / Graphics--Corrupted Images	7
1.26	D / Errors / Hang--One Program Only	8
1.27	D / Errors / Hang--Whole System	8
1.28	D / Errors / Memory Loss	8
1.29	D / Errors / Memory Loss--CLI Only	9

1.30 D / Errors / Memory Loss--Ctrl-C Exit Only	9
1.31 D / Errors / Memory Loss--During Execution	9
1.32 D / Errors / Memory Loss--Workbench Only	9
1.33 D / Errors / Menu Problems	10
1.34 D / Errors / Out-of-Sync Response to Input	10
1.35 D / Errors / Performance Loss in Other Processes	10
1.36 D / Errors / Performance Loss--On A3000	10
1.37 D / Errors / Trackdisk Data not Transferred	10
1.38 D / Errors / Windows--Borders Flicker after Resize	10
1.39 D / Errors / Windows--Visual Problems	11
1.40 D Troubleshooting Guide / General Debugging Techniques	11
1.41 D Troubleshooting Guide / A Final Word About Testing	11

Chapter 1

Libraries_Manual

1.1 Amiga® RKM Libraries: D Troubleshooting Guide

Many Amiga programming errors have classic symptoms. This guide will help you to eliminate or avoid these problems in your software.

Errors General Debugging Techniques A Final Word About Testing

1.2 D Troubleshooting Guide / Errors

Audio--Corrupted Samples
Character Input/Output Problems
CLI Error Message Problems
CLI Won't Close on RUN
Crashes and Memory Corruption
Crashes--After Exit
Crashes--Only on 68000 and 68010
Crashes--Only on 68040
Crashes--Subtasks, Interrupts
Crashes--Window Related
Crashes--Workbench Only
Device-related Problems
Disk Icon Won't Go Away
DOS-related Problems
Fails only on 68020/30
Fails only on 68000
Fails only on Older ROMs or Older WB
Fails only on Newer ROMs or Newer WB
Fails only on Chip-RAM-Only Machines
Fails only on machines with Fast RAM
Fails only with Enhanced Chips
Fireworks
Graphics--Corrupted Images
Hang--One Program Only
Hang--Whole System
Memory Loss
Memory Loss--CLI Only
Memory Loss--Ctrl-C Exit Only

- Memory Loss--During Execution
- Memory Loss--Workbench Only
- Menu Problems
- Out-of-Sync Response to Input
- Performance Loss in Other Processes
- Performance Loss--On A3000
- Trackdisk Data not Transferred
- Windows--Borders Flicker after Resize
- Windows--Visual Problems

1.3 D / Errors / Audio--Corrupted Samples

The bit data for audio samples must be in Chip RAM. Check your compiler manual for directives or flags which will place your audio sample data in Chip RAM. Or dynamically allocate Chip RAM and copy or load the audio sample there.

1.4 D / Errors / Character Input/Output Problems

RAWKEY users must be aware that RAWKEY codes can be different letters or symbols on national keyboards. If you need to use RAWKEY, run the codes through RawKeyConvert() (see the "Intuition Input and Output Methods" chapter) to get proper translation to correct ASCII codes. Improper display or processing of high-ASCII international characters can be caused by incorrect tolower()/toupper(), or by sign extension of character values when switched on or assigned into larger size variables. Use unsigned variables such as UBYTE (not char) for strings and characters whenever possible. Internationally correct string functions are provided in the 2.0 utility.library.

1.5 D / Errors / CLI Error Message Problems

Improper error messages are caused by calling exit(n) with an invalid or missing return value n. Assembler programmers using startup code should jump to the startup code's _exit with a valid return value on the stack. Programs without startup code should return with a valid value in D0. Valid return values such as RETURN_OK, RETURN_WARN, RETURN_FAIL are defined in <dos/dos.h> and <dos/dos.i>. Values outside of these ranges (-1 for instance) can cause invalid CLI error messages such as "not an object module". Useful hint--if your program is called from a script, your valid return value can be conditionally branched on in the script (i.e., call program, then perform actions based on IF WARN or IF NOT WARN). RETURN_FAIL will cause the script to stop if a normal FAILAT value is being used in script.

1.6 D / Errors / CLI Won't Close on RUN

A CLI can't close if a program has a Lock() on the CLI input or output stream ("*"). If your program is RUN >NIL: from a CLI, that CLI should be able to close unless your code or your compiler's startup code explicitly opens "*".

1.7 D / Errors / Crashes and Memory Corruption

Memory corruption, address errors, and illegal instruction errors are generally caused by use of an uninitialized, incorrectly initialized, or already freed/closed pointer or memory. You may be using the pointer directly, or it may be one that you placed (or forgot to place) in a structure passed to system calls. Or you may be overwriting one of your arrays, or accidentally modifying or incrementing a pointer later used in a free/close. Be sure to test the return of all open/allocation type functions before using the result, and only close/free things that you successfully opened/allocated. Use watchdog/torture utilities such as Enforcer and MungWall in combination to catch use of uninitialized pointers or freed memory, and other memory misuse problems. Use the debugging tool TNT to get additional debugging information instead of a Software Error requester. You may also be overflowing your stack--your compiler's stack checking option may be able to catch this. Cut stack usage by dynamically allocating large structures, buffers, and arrays which are currently defined inside your functions.

Corruption or crashes can also be caused by passing wrong or missing arguments to a system call (for example SetAPen(3) or SetAPen(win,3), instead of SetAPen(rp,3)). C programmers should use function prototypes to catch such errors. If using short integers be sure to explicitly type long constants as long (e.g., 42L). (For example, with short ints, 1 << 17 may become zero). If corruption is occurring during exit, use printf() (or KPrintf(), etc.) with Delay(n) to slow down your cleanup and broadcast each step. A bad pointer that causes a system crash will often be reported as an standard 680x0 processor exception \$00000003 or 4, or less often a number in the range of \$00000006-B. Or an Amiga-specific alert number may result. See <exec/alerts.h> for Amiga-specific alert numbers. Also see "Crashes--After Exit" below.

1.8 D / Errors / Crashes--After Exit

If this only happens when you start your program from Workbench, then you are probably UnLock()ing one of the WBStartup message wa_Locks, or UnLock()ing the Lock() returned from an initial CurrentDir() call. If you CurrentDir(), save the lock returned initially, and CurrentDir() back to it before you exit. Only UnLock() locks that you created.

If you are crashing from both Workbench and CLI, and you are only crashing after exit, then you are probably either freeing/closing something twice, or freeing/closing something you did not actually allocate/open, or you may be leaving an outstanding device I/O request or other wakeup request. You must abort and WaitIO() any outstanding I/O requests before you free things and exit (see the Autodocs for your device, and for Exec AbortIO())

and `WaitIO()`). Similar problems can be caused by deleting a subtask that might be in a `WaitTOF()`. Only delete subtasks when you are sure they are in a safe state such as `Wait(OL)`.

1.9 D / Errors / Crashes--Only on 68000 and 68010

This can be caused by illegal instructions (80000000.00000004) such as new 68020/30/40 instructions or inline 68881/882 code. But this is usually caused by a word or longword access at an odd address. This is legal on the 68020 and above, but will generate an Address Error (80000000.00000003) on a 68000 or 68010. This can be caused by using uninitialized pointers, using freed memory, or using system structures improperly (for example, referencing into `IntuiMessage->IAddress` as a `struct Gadget *` on a non-Gadget message).

1.10 D / Errors / Crashes--Only on 68040

Because of the instruction pipelining of the 68040, it is very difficult to recover from a bus error. If your program has an "Enforcer hit" (i.e., an illegal reference to memory), the resulting 68040 processor bus error will probably crash the machine. Use Enforcer (on an '030) to track down your problems, then correct them.

1.11 D / Errors / Crashes--Subtasks, Interrupts

If part of your code runs on a different stack or the system stack, you must turn off compiler stack-checking options. If part of your code is called directly by the system or by other tasks, you must use long code/long data or use special compiler flags or options to assure that the correct base registers are set up for your subtask or interrupt code.

1.12 D / Errors / Crashes--Window Related

Be careful not to `CloseWindow()` a window during a `while(msg=GetMsg(...))` loop on that window's port (next `GetMsg()` would be on freed pointer). Also, use `ModifyIDCMP(NULL)` with care, especially if using one port with multiple windows. Be sure to `ClearMenuStrip()` any menus before closing a window, and do not free items such as dynamically allocated gadgets and menus while they are attached to a window. Do not reference an `IntuiMessage`'s `IAddress` field as a structure pointer of any kind before determining it is a structure pointer (this depends on the Class of the `IntuiMessage`). If a crash or problem only occurs when opening a window after extended use of your program, check to make sure that your program is properly freeing up signals allocated indirectly by `CreatePort()`, `OpenWindow()` or `ModifyIDCMP()`.

1.13 D / Errors / Crashes--Workbench Only

If you are crashing near the first DOS call, either your stack is too small or your startup code did not GetMsg() the WBStartup message from the process message port. If your program crashes during execution or during your exit procedure only when started from Workbench, and your startup opens no stdio window or NIL: file handles for WB programs, then make sure you are not writing anything to stdout (printf(), etc.) when started from WB (argc==0). See also "Crashes--After Exit".

1.14 D / Errors / Device-related Problems

Device-related problems may be caused by: improperly initialized port or I/O request structures (use CreatePort() and CreateExtIO()); use of a too-small I/O request (see the device's <.h> files and Autodocs for information on the required type of I/O request); re-use of an I/O request before it has returned from the device (use the debugging tool IO_Torture to catch this); failure to abort and wait for an outstanding device request before exiting; waiting on a signal/port/message allocated by a different task.

1.15 D / Errors / Disk Icon Won't Go Away

This occurs when a program leaves a Lock() on one or more of a disk's files or directories. A memory loss of exactly 24 bytes is usually Lock() which has not been UnLock()ed.

1.16 D / Errors / DOS-related Problems

In general, any dos.library function which fills in a structure for you (for example, Examine()), requires that the structure be longword aligned. In most cases, the only way to insure longword alignment in C is to dynamically allocate the structure. Unless documented otherwise, dos.library functions may only be called from a process, not from a task. Also note that a process's pr_MsgPort is intended for the exclusive use of dos.library. (The port may be used to receive a WBStartup message as long as the message is GetMsg()'d from the port before DOS is used.

1.17 D / Errors / Fails only on 68020/30

The following programming practices can cause this problem: using the upper bytes of addresses as flags; doing signed math on addresses; self-modifying code; using the MOVE SR assembler instruction (use Exec GetCC() instead); software delay loops; assumptions about the order in which asynchronous tasks will finish. The following differences in 68020/30 can cause problems: data and/or instruction caches must be flushed if data or code is changed by DMA or other non-processor

modification; different exception stack frame; interrupt autovectors may be moved by VBR; 68020/30 CLR instruction does a single write access unlike the 68000 CLR instruction which does a separate read and write access (this might affect a read-triggered register in I/O space--use MOVE instead).

1.18 D / Errors / Fails only on 68000

The following programming practices can be the cause of this problem: software delay loops; word or longword access of an odd address (illegal on the 68000). Note that this can occur under 2.0 if you reference `IntuiMessage->IAddress` as a structure pointer without first determining that the `IntuiMessage`'s Class is defined as having a structure pointer in its `IAddress`; use of the assembler CLR instruction on a hardware register which is triggered by any access. The 68000 CLR instruction performs two accesses (read and write) while 68020/30 CLR does a single write access. Use MOVE instead; assumptions about the order in which asynchronous tasks will finish; use of compiler flags which have generated inline 68881/68882 math coprocessor instructions or 68020/30 specific code.

1.19 D / Errors / Fails only on Older ROMs or Older WB

This can be caused by asking for a library version higher than you need (Do not use the `#define LIBRARY_VERSION` when compiling!). Can also be caused by calling functions or using structures which do not exist in the older version of the operating system. Ask for the lowest version which provides the functions you need (usually 33), and exit gracefully and informatively if an `OpenLibrary()` fails (returns NULL). Or code conditionally to only use new functions and structures if the available Library's `lib_Version` supports them.

1.20 D / Errors / Fails only on Newer ROMs or Newer WB

This should not happen with proper programming. Possible causes include: running too close to your stack limits or the memory limits of a base machine (newer versions of the operating system may use slightly more stack in system calls, and usually use more free memory); using system functions improperly; not testing function return values; improper register or condition code handling in assembler code. Remember that result, if any, is returned in D0, and condition codes and D1/A0/A1 are undefined after a system call; using improperly initialized pointers; trashing memory; assuming something (such as a flag) is B if it is not A; failing to initialize formerly reserved structure fields to zero; violating Amiga programming guidelines (for example: depending on or poking private system structures, jumping into ROM, depending on undocumented or unsupported behaviors); failure to read the function Autodocs.

See Appendix E, "Release 2 Compatibility", for more information on 2.0 compatibility problem areas.

1.21 D / Errors / Fails only on Chip-RAM-Only Machines

Caused by specifically asking for or requiring MEMF_FAST memory. If you don't need Chip RAM, ask for memory type 0L, or MEMF_CLEAR, or MEMF_PUBLIC|MEMF_CLEAR as applicable. If there is Fast memory available, you will be given Fast memory. If not, you will get Chip RAM. May also be caused by trackdisk-level loading of code or data over important system memory or structures which might reside in low Chip memory on a Chip-RAM-Only machine.

1.22 D / Errors / Fails only on machines with Fast RAM

Data and buffers which will be accessed directly by the custom chips must be in Chip RAM. This includes bitplanes (use OpenScreen() or AllocRaster()), audio samples, trackdisk buffers, and the graphic image data for sprites, pointers, bobs, images, gadgets, etc. Use compiler or linker flags to force Chip RAM loading of any initialized data needing to be in Chip RAM, or dynamically allocate Chip RAM and copy any initialization data there.

1.23 D / Errors / Fails only with Enhanced Chips

Usually caused by writing or reading addresses past the end of older custom chips, or writing something other than 0 (zero) to bits which are undefined in older chip registers, or failing to mask out undefined bits when interpreting the value read from a chip register. Note that system copper lists are different under 2.0 when ECS chips are present. See "Fails only on Chip-RAM-Only Machines".

1.24 D / Errors / Fireworks

A dazzling pyrotechnic video display is caused by trashing or freeing a copper list which is in use, or trashing the pointers to the copper list. If you aren't messing with copper lists, see above section called "Crashes and Memory Corruption".

1.25 D / Errors / Graphics--Corrupted Images

The bit data for graphic images such as sprites, pointers, bobs, and gadgets must be in Chip RAM. Check your compiler manual for directives or flags which will place your graphic image data in Chip RAM. Or dynamically allocate Chip RAM and copy them there.

1.26 D / Errors / Hang--One Program Only

Program hangs are generally caused by `Wait()`ing on the wrong signal bits, on the wrong port, on the wrong message, or on some other event that will never occur. This can occur if the event you are waiting on is not coming, or if one task tries to `Wait()`, `WaitPort()`, or `WaitIO()` on a signal, port, or window that was created by a different task. Both `WaitIO()` and `WaitPort()` can call `Wait()`, and you cannot `Wait()` on another task's signals. Hangs can also be caused by verify deadlocks. Be sure to turn off all Intuition verify messages (such as `MENUVERIFY`) before calling `AutoRequest()` or doing disk access.

1.27 D / Errors / Hang--Whole System

This is generally caused by a `Disable()` without a corresponding `Enable()`. It can also be caused by memory corruption, especially corruption of low memory. See "Crashes and Memory Corruption".

1.28 D / Errors / Memory Loss

First determine that your program is actually causing a memory loss. It is important to boot with a standard Workbench because a number of third party items such as some background utilities, shells, and network handlers dynamically allocate and free pieces of memory. Open a Shell for memory checking, and a Shell or Workbench drawer for starting your program. Arrange windows so that all are accessible, and so that no window rearrangement will be needed to run your program.

In the Shell, type `Avail FLUSH<RET>` several times (2.0 option). This will flush all non-open disk-loaded fonts, devices, etc., from memory. Note the amount of free memory. Now without rearranging any windows, start your program and use all of your program features. Exit your program, wait a few seconds, then type `Avail FLUSH<RET>` several times. Note the amount of free memory. If this matches the first value you noted, your program is fine, and is not causing a memory loss.

If memory was actually lost, and your program can be run from CLI or Workbench, then try the above procedure with both methods of starting your program. Note that under 2.0, there will be a slight permanent (until reboot) memory usage of about 672 bytes when the `audio.device` or `narrator.device` is first opened. See "Memory Loss--CLI Only" and "Memory Loss--WorkBench Only" if appropriate. If you lose memory from both WB and CLI, then check all of the `open/alloc/get/create/lock` type calls in your code, and make sure that there is a matching `close/free/delete/unlock` type call for each of them (note--there are a few system calls that have or require no corresponding free--check the Autodocs). Generally, the `close/free/delete/unlock` calls should be in opposite order of the allocations.

If you are losing a fixed small amount of memory, look for a structure of that size in the Structure Offsets listing in the Amiga ROM Kernel Reference Manual: Includes and Autodocs. For example, a loss of exactly

24 bytes is probably a Lock() which has not been UnLock()ed. If you are using ScrollRaster(), be aware that ScrollRaster() left or right in a Superbitmap window with no TmpRas will lose memory under 1.3 (workaround--attach a TmpRas). If you lose much more memory when started from Workbench, make sure your program is not using Exit(n). This would bypass startup code cleanups and prevent a Workbench-loaded program from being unloaded. Use exit(n) instead.

1.29 D / Errors / Memory Loss--CLI Only

Make sure you are testing in a standard environment. Some third-party shells dynamically allocate history buffers, or cause other memory fluctuations. Also, if your program executes different code when started from CLI, check that code and its cleanup. And check your startup.asm if you wrote your own.

1.30 D / Errors / Memory Loss--Ctrl-C Exit Only

You have Amiga-specific resources opened or allocated and you have not disabled your compiler's automatic Ctrl-C handling (causing all of your program cleanups to be skipped). Disable the compiler's Ctrl-C handling and handle Ctrl-C (SIGBREAKF_CTRL_C) yourself.

1.31 D / Errors / Memory Loss--During Execution

A continuing memory loss during execution can be caused by failure to keep up with voluminous IDCMP messages such as MOUSEMOVE messages. Intuition cannot re-use IDCMP message blocks until you ReplyMsg() them. If your window's allotted message blocks are all in use, new sets will be allocated and not freed till the window is closed. Continuing memory losses can also be caused by a program loop containing an allocation-type call without a corresponding free.

1.32 D / Errors / Memory Loss--Workbench Only

Commonly, this is caused by a failure of your code to unload after you exit. Make sure that your code is being linked with a standard correct startup module, and do not use the Exit(n) function to exit your program. This function will bypass your startup code's cleanup, including its ReplyMsg() of the WBStartup message (which would signal Workbench to unload your program from memory). You should exit via either exit(n) where n is a valid DOS error code such as RETURN_OK (<dos/dos.h>), or via final "}" or return. Assembler programmers using startup code can JMP to _exit with a long return value on stack, or use the RTS instruction.

1.33 D / Errors / Menu Problems

A flickering menu is caused by leaving a pixel or more space between menu subitems when designing your menu. Crashing after browsing a menu (looking at menu without selecting any items) is caused by not properly handling `MENUNULL` select messages. Multiple selection not working is caused by not handling `NextSelect` properly. See the "Intuition Menus" chapter.

1.34 D / Errors / Out-of-Sync Response to Input

Caused by failing to handle all received signals or all possible messages after a `Wait()` or `WaitPort()` call. More than one event or message may have caused your program to awakened. Check the signals returned by `Wait()` and act on every one that is set. At ports which may have more than one message (for instance, a window's `IDCMP` port), you must handle the messages in a `while(msg=GetMsg(...))` loop.

1.35 D / Errors / Performance Loss in Other Processes

This is often caused by a one program doing one or more of the following: busy waiting or polling; running at a higher priority; doing lengthy `Forbid()`s, `Disable()`s, or interrupts.

1.36 D / Errors / Performance Loss--On A3000

If your program has "Enforcer hits" (i.e., illegal references to memory caused by improperly initialized pointers), this will cause Bus Errors. The A3000 bus error handler contains a built-in delay to let the bus settle. If you have many enforcer hits, this could slow your program down substantially.

1.37 D / Errors / Trackdisk Data not Transferred

Make sure your trackdisk buffers are in Chip RAM under 1.3 and lower versions of the operating system.

1.38 D / Errors / Windows--Borders Flicker after Resize

Set the `NOCAREREFRESH` flag. Even `SMART_REFRESH` windows may generate refresh events if there is a sizing gadget. If you don't have specific code to handle this, you must set the `NOCAREREFRESH` flag. If you do have refresh code, be sure to use the `Begin/EndRefresh()` calls. Failure to do one or the other will leave Intuition in an intermediate state, and slow down operation for all windows on the screen.

1.39 D / Errors / Windows--Visual Problems

Many visual problems in windows can be caused by improper font specification or improper setting of gadget flags. See the Appendix E on "Release 2 Compatibility" for detailed information on common problems.

1.40 D Troubleshooting Guide / General Debugging Techniques

Narrow the search

Use methodical testing procedures, and debugging messages if necessary, to locate the problem area. Low level code can be debugged using KPrintf() serial (or dprintf() parallel) messages. Check the initial values, allocation, use, and freeing of all pointers and structures used in the problem area. Check that all of your system and internal function calls pass correct initialized arguments, and that all possible error returns are checked for and handled.

Isolate the problem

If errors cannot be found, simplify your code to the smallest possible example that still functions. Often you will find that this smallest example will not have the problem. If so, add back the other features of your code until the problem reappears, then debug that section.

Use debugging tools

A variety of debugging tools are available to help locate faulty code. Some of these are source level and other debuggers, crash interceptors, vital watchdog and memory invalidation tools like Enforcer and MungWall.

1.41 D Troubleshooting Guide / A Final Word About Testing

Test your program with memory watchdog and invalidation tools on a wide variety of systems and configurations. Programs with coding errors may appear to work properly on one or more configurations, but may fail or cause fatal problems on another. Make sure that your code is tested on both a 68000 and a 68020/30, on machines with and without Fast RAM, and on machines with and without enhanced chips. Test all of your program functions on every machine.

Test all error and abort code. A program with missing error checks or unsafe cleanup might work fine when all of the items it opens or allocates are available, but may fail fatally when an error or problem is encountered. Try your code with missing files, filenames with spaces, incorrect filenames, cancelled requesters, Ctrl-C, missing libraries or devices, low memory, missing hardware, etc.

Test all of your text input functions with high-ASCII characters (such as the character produced by pressing Alt-F then "A"). Note that RAWKEY codes can be different keyboard characters on national keyboards (higher levels of keyboard input are automatically translated to the proper

characters). If your program will be distributed internationally, support and take advantage of the additional screen lines available on a PAL system. Enhanced Agnus chip machines may be switched to be PAL or NTSC via motherboard jumper J102 in A2000s and jumper J200 in A3000s. Note that a base PAL machine will have less memory free due to the larger display size.

Write good code. Test it. Then make it great.